HAL
open science

Probabilistic relational models learning from graph databases

Marwa El Abri

To cite this version:

Marwa El Abri. Probabilistic relational models learning from graph databases. Computer Science [cs]. Université de Nantes, 2018. English. NNT : . tel-01901255

HAL Id: tel-01901255

https://theses.hal.science/tel-01901255

Submitted on 22 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE DE DOCTORAT DE

L'UNIVERSITE DE NANTES
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique, section CNU 27*

Par
Marwa El Abri

Probabilistic relational models learning from graph databases

Thèse présentée et soutenue à Nantes, le 02/10/2018
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)
Laboratoire de recherche opérationnelle, de décision et de contrôle de processus (LARODEC)
Thèse N° :

Rapporteurs avant soutenance :

M. Simon	De Givry	Chargé de Recherche	INRA	Toulouse
M. Nicolas	Lachiche	Maître de Conférences HDR	Université de Strasbourg	Strasbourg

Composition du Jury :

Président :

Mme Nahla	Ben Amor	Professeur des Universités	Université de Tunis	Tunisie

Examinateurs :

M.Emmanuel	Mazer	Directeur de Recherche	CNRS	France

Dir. de thèse :

M. Philippe	Leray	Professeur des Universités	Université de Nantes	France

Co-dir. de thèse :

Mme Nadia	Essoussi	Professeur des Universités	Université de Tunis	Tunisie

Acknowledgements

This thesis would not have been possible without the inspiration and continuous support of a number of people.

First, I thank the jury's members who honored me by judging my dissertation. I am grateful for the time they spent to do this.

Then, I want to express my deep gratitude and sincere appreciation to my two supervisors: Pr. Nadia Essoussi who often gave me the courage to move forward in my research, I am thankful for her continuous support, her precious advice and her invaluable help, Pr. Philippe Leray for the constructive discussions, continuous guidance, suggestions, and optimism. I was fortunate to benefit from his expertise in this interesting research area. Without them this dissertation would never have been achieved.

I would like to offer my heartiest thanks to my mum for her understanding, support and confidence to which I am forever indebted. Mum, I love you, you have always supported my dreams! To my father's soul. Dad, may god bless you, I wish you were here to share with us these moments of joy and success.

An honorable mention goes to my family and all my friends especially Montassar and Saif for their help, their company, their emotional support and for listening to my complaints and frustrations. I really love them and I am sure that having brought this work forward is the best reward for them.

More than anything else, I am deeply grateful to my brother Marwen and my sister Maram for their love and continuous moral support.

I would like to express my sincere gratitude to my colleagues either in LARODEC or in DUKe team, especially Rajani Chulyadyo and Thomas Vincent for always sharing their expertise and experiences, and making the development of PILGRIM fun. Their domain expertise has helped me get good insights of PILGRIM.

I would also like to thank all those who have been helped me directly or indirectly in all stages of this thesis.

Contents

I Introduction and motivations **1**

1 Context and issue **3**
1.1 Introduction 4
1.1.1 Context 4
1.1.2 Motivation and problem statement 5
1.2 Dissertation organization 7

II State of the art **9**

2 Probabilistic Graphical Models **11**
2.1 Introduction 12
2.2 Basic concepts 12
2.2.1 Concepts related to graph theory 12
2.2.2 Concepts related to probability theory 13
2.3 Bayesian Networks 15
2.3.1 Bayesian Networks formalism 15
2.3.2 Parameter learning 18
2.3.3 Structure learning 20
2.4 Markov Networks 23
2.4.1 Markov Networks formalism 23
2.4.2 Parameter learning 25
2.4.3 Structure learning 26
2.5 Inference in PGMs 27
2.6 Bayesian Networks vs Markov Networks 28
2.7 Conclusion 29

3 Probabilistic Relational Models **31**
3.1 Introduction 32
3.2 Basic concepts 32
3.2.1 Entity-Relationship model 33
3.2.2 First-Order-Logic 35
3.3 Direct Acyclic Probabilistic Entity Relationship models 36
3.3.1 DAPERs formalism 36
3.3.2 Extensions: DAPERs with structural uncertainty 39
3.3.3 Parameter learning 39
3.3.4 Structure learning 39
3.4 Markov Logic Networks 40
3.4.1 MLN formalism 40
3.4.2 Parameters learning 43

3.4.3 Structure learning . . . 44
3.5 Computing sufficient statistics for PRMs learning . . . 46
3.5.1 Definitions . . . 46
3.5.2 SQL implementation for computing contingency table . . . 47
3.6 Inference in PRMs . . . 48
3.7 Conclusion . . . 48

4 Graph database **51**
4.1 Introduction . . . 52
4.2 NoSQL databases . . . 52
4.2.1 CAP theorem . . . 54
4.2.2 BASE properties . . . 54
4.3 Basic concepts related to graph database . . . 55
4.4 Graph database manipulation . . . 57
4.4.1 Subgraph matching . . . 57
4.5 Graph database properties . . . 59
4.6 Graph database use cases . . . 61
4.7 Comparison with other database models . . . 61
4.8 Discussion . . . 62
4.9 Conclusion . . . 63

III Contributions **65**

5 DAPER learning from Graph database using PRM framework **67**
5.1 Introduction . . . 68
5.2 DAPER learning from graph database . . . 68
5.2.1 Principle . . . 69
5.2.2 ER model identification . . . 70
5.2.3 Probabilistic dependencies identification . . . 72
5.3 Experiments . . . 75
5.3.1 Experiment methodology . . . 76
5.3.2 Networks and Datasets . . . 79
5.3.3 Algorithms . . . 80
5.3.4 Evaluation metrics . . . 80
5.3.5 Results and interpretations . . . 82
5.4 Conclusion . . . 89

6 DAPER learning from Graph Database using MLN framework **91**
6.1 Introduction . . . 92
6.2 Expressing a DAPER in Markov Logic . . . 92
6.3 DAPER joint learning using MLN framework . . . 95
6.3.1 DAPER joint learning . . . 95
6.3.2 Evaluation process . . . 97
6.4 Experiments . . . 97
6.4.1 Experiment methodology . . . 97
6.4.2 Evaluation metrics . . . 99
6.4.3 Results and interpretations . . . 99
6.5 Conclusion . . . 106

7 Conclusion **107**
7.1 Summary . . . 108
7.2 Future works . . . 109

IV Appendices **111**

A PILGRIM relational **113**
A.1 Introduction . . . 113
A.2 Pilgrim project . . . 113
A.3 PILGRIM Relational . . . 114
A.3.1 Existing libraries . . . 115
A.3.2 Additional libraries . . . 115
A.3.3 Data accessibility . . . 115
A.3.4 PRM specification in PILGRIM . . . 116
A.3.5 Relational schema in PILGRIM . . . 116
A.3.6 PRM learning module . . . 116
A.3.7 PRM benchmark generation module . . . 116
A.3.8 PRM extension module . . . 117
A.3.9 Learning from graph database . . . 118
A.4 Conclusion . . . 121

B Neo4j graph database **123**
B.1 Introduction . . . 123
B.2 Neo4j graph database . . . 123
B.2.1 Connecting to Neo4J graph database . . . 124
B.2.2 Cypher query language . . . 126
B.2.3 From relational database to graph database . . . 127
B.3 Conclusion . . . 127

List of Tables

2.1 Example of CPD $P(X|Y)$ 14
2.2 Example of JPD $P(X,Y)$ 15

3.1 Example of a first-order knowledge base. Fr() is short for Friends(), Sm() for Smokes(), and Ca() for Cancer() 36

4.1 ACID vs. BASE (Brewer, 2000) 55
4.2 A Comparison of the NoSQL and relational databases 62

5.1 Characteristics of DAPER networks used for naive datasets generation. Values in parenthesis are min/max values per class. Values in the last column correspond to the maximum length of slot chain $kmax$ and the number of dependencies for each slot chain length (from 0 to k_{max}). 79
5.2 Characteristics of DAPER networks used for respectively naive and k-partite datasets generation. 80
5.3 Average $\pm$ standard deviation of reconstruction metrics (Precision, Recall, F-score, s-Precision, s-Recall and s-F-score) when executing RGS with k_{max}=1 for databases generated from DAPER1 (top) and DAPER2 (bottom). 82
5.4 Average $\pm$ standard deviation of reconstruction metrics (Precision, Recall, F-score, s-Precision, s-Recall and s-F-score) for each sample size when executing RGS with $k_{max} = 1$ (top) and k_{max}=4 (bottom) for databases generated from DAPER3. 83
5.5 Average $\pm$ standard deviation of reconstruction metrics for each sample size when executing RGS with $k_{max} = 1$ (top) and k_{max}=4 (bottom) for databases generated from DAPER'3. 84
5.6 Average $\pm$ standard deviation of reconstruction metrics for each sample size when executing RGS with $k_{max} = 1$ (top) and k_{max}=4 (bottom) for databases generated from DAPER"3. 85
5.7 Average $\pm$ standard deviation of reconstruction metrics for DAPER3 for a particular size (3000 instances) with a percentage α of exceptions in the relationship instances for λ fixed to 10% 86
5.8 Average $\pm$ standard deviation of reconstruction metrics for DAPER3 for a particular size (3000 instances) with a percentage λ of ER identification for α fixed to 50%. 86
5.9 A table of running time (in seconds) for each sample size, when executing RGS_Neo4j with naive datasets and k-partite datasets generated from each DAPER. 87

6.1 Knowledge base corresponding to the DAPER described in Figure 6.1. 93
6.2 characteristics of DAPER networks used for dataset generation. Values in parenthesis are min/max values per class. Values in parenthesis are min/max values per class. Values in the last column correspond to the maximum length of slot chain $kmax$ and the number of dependencies for each slot chain length (from 0 to k_{max}). 98
6.3 Average $\pm$ standard deviation of RSHD_ER for DAPERs for a particular size (5000 instances). 101

6.4 Average $\pm$ standard deviation of RSHD_PD for each algorithm for a particular sample size and network with a percentage α of exceptions in the relationship instances fixed to 10% . Bold values present the best values for a given model and a given sample size. 102
6.5 Average $\pm$ standard deviation of RSHD_PD for each algorithm for a particular sample size and network with a percentage α of exceptions in the relationship instances fixed to 30% . Bold values present the best values for a given model and a given sample size. 103
6.6 Average $\pm$ standard deviation of RSHD_PD for each algorithm for a particular sample size and network with a percentage α of exceptions in the relationship instances fixed to 50% . Bold values present the best values for a given model and a given sample size. 104
6.7 A comparative study of obtained optimum values of both methods using the statistical student test with a critical threshold= 5%. 106

B.1 A table contains information about the clauses in the Cypher query language. 130

List of Figures

1.1 Thesis synopsis. 7

2.1 Example of graphical models 13
2.2 Example of Bayesian Network. 16
2.3 The rules of Bayes Ball theorem 17
2.4 Markov equivalence 18
2.5 Example of Markov Network 24
2.6 (a) A DAG; The dotted red node X_8 is independent of all other nodes (black) given its Markov blanket, which includes its parents (blue), children (green) and co-parents (orange). (b) The same model represented as a MN; The red node X_8 is independent of the other black nodes given its neighbors (blue nodes) (Koller and Friedman, 2009). 28

3.1 An example of (a)ER diagram, (b) instance $\mathcal{I}$ and (c)skeleton $\sigma_{\mathcal{ER}}$ for the university domain. 34
3.2 Relation components 34
3.3 An example of DAPER model which corresponds to the ER model and Instance given in Figure 3.1. Slot chains are not represented in order to simplify the figure. 38
3.4 An example of MLN for the university domain. The pairs of formulas and weights together with the set of constants construct the ground Markov network. 42
3.5 Excerpt from the CT table for the attribute *student.ranking* and its *parent* as they are linked in the DAPER of Figure 3.3. 47

4.1 Example of Key-Value store. 53
4.2 Example of Column Family database. 53
4.3 Example of Document database. 54
4.4 The CAP theorem (Brewer, 2000) 55
4.5 The graph database components. 57
4.6 Example of graph database for the university domain. 58
4.7 Example of graph database 59
4.8 Example of subgraph matching. 59
4.9 Categorization of NOSQL databases. 62

5.1 Step I: ER schema identification from a graph database. 69
5.2 Step II: Computation of sufficient statistics and DAPER learning using the ER schema already identified and the graph database instance. 69
5.3 Pattern element. 73
5.4 Illustration of counting sufficient statistics process for the attribute *student.ranking* and its *parents*. 74
5.5 Evaluation process of a DAPER structure learning from usual relational database versus graph database. 76
5.6 Evaluation process of a DAPER structure learning from "noisy" graph database 78

5.7 Boxplot of running time (in seconds and log scale) for each sample size, when executing RGS_postgres and RGS_neo4j with k_{max}=1 and databases generated from DAPER1 (a) and DAPER2 (b). . . . 83
5.8 Boxplot of running time (in seconds and log scale) for each sample size, when executing RGS with k_{max}=1 (a) and k_{max}=4 (b) and databases generated from DAPER3. . . . 84
5.9 Boxplot of running time (in seconds and log scale) for each sample size, when executing RGS with k_{max}=1 (a) and k_{max}=4 (b) and databases generated from DAPER'3. . . . 85
5.10 Boxplot of running time (in seconds and log scale) for each sample size, when executing RGS with k_{max}=1 (a) and k_{max}=4 (b) and databases generated from DAPER"3. . . . 86
5.11 An example of screenshot of the structure of strongly connected (naive) and dispersed k-partite) graph databases. . . . 88
5.12 An illustration of normal and critical paths. . . . 89

6.1 Example of a DAPER model (inspired from (Getoor, 2001) and detailed in Chapter 3). . . . 93
6.2 Process of DAPER joint learning using MLN framework. . . . 96
6.3 The Average values of RSHD_ER with respect to λ_{er} when executing approach 1 and 2 with (a) α=30% and (b) α=50% for a particular size (5000 instances) of DAPER1. . . . 100
6.4 The Average values of RSHD_PD with respect to λ_{pd} when executing approach 1 and 2 with (a) Data size =1000 and (b) Data size =3000 and (c) Data size =5000 for DAPER1 with a percentage α of exceptions in the relationship instances fixed to 10%. . . . 101
6.5 The Average values of RSHD_PD with respect to λ_{pd} when executing approach 1 and 2 with (a) Data size =1000 and (b) Data size =3000 and (c) Data size =5000 for DAPER1 with a percentage α of exceptions in the relationship instances fixed to 30%. . . . 102
6.6 The Average values of RSHD_PD with respect to λ_{pd} when executing approach 1 and 2 with (a) Data size =1000 and (b) Data size =3000 and (c) Data size =5000 for DAPER2 with a percentage α of exceptions in the relationship instances fixed to 30%. . . . 103
6.7 The Average values of RSHD_PD with respect to λ_{pd} when executing approach 1 and 2 with (a) Data size =1000 and (b) Data size =3000 and (c) Data size =5000 for DAPER2 with a percentage α of exceptions in the relationship instances fixed to 50%. . . . 104
6.8 Boxplot of running time (in seconds and log scale) for each sample size of each DAPER. . . 105

7.1 A new definition for DAPER model in context of graph databases. . . . 109

A.1 Class diagram for the *RelationalSchema*, *Class*, *Attribute* and *Domain* classes. . . . 117
A.2 Implementation environment . . . 118
A.3 Class diagram for *Neo4jGenerator* and *Neo4jRequest*. . . . 119
A.4 Class diagram for *Neo4jInstance* . . . 120

B.1 Neo4j graph database model. . . . 124
B.2 Neo4j graph database features. . . . 125
B.3 The Neo4j browser. . . . 125
B.4 A HTTP request example which executes Cypher query to create a Person. . . . 126
B.5 A "Hello world" example shows the minimal configuration necessary to interact with Neo4j through a driver designed for C language. . . . 127
B.6 A social graph describing the relationship between three friends. . . . 128
B.7 A Cypher pattern example. . . . 128
B.8 A Cypher query example contains MATCH and RETURN clauses. . . . 128
B.9 Cypher versus SQL language. . . . 129
B.10 The three most common paradigms for deploying relational and graph databases (Hunger et al., 2016). . . . 129
B.11 An example of importing a CSV file into Neo4j using the LOAD CSV Cypher command. . 131

List of Algorithms

1 Relational Greedy Search (Friedman et al., 1999a) 40
2 Generate_Neighbors 41
3 MLN structure learning (Kok and Domingos, 2005) 45
4 Find_Best_Clause (Kok and Domingos, 2005) 45

5 ER_Model_identification 70
6 Get_Filtered_Relationship_Classes 72
7 Get_Counts 76
8 Create_Slot_chain_Traversals 76
9 Create_SubGraphs 77

I

Introduction and motivations

1

Context and issue

Contents

1.1 Introduction **4**

1.1.1 Context 4

1.1.2 Motivation and problem statement 5

1.2 Dissertation organization **7**

1.1 Introduction

1.1.1 Context

Machine Learning (Bishop, 2006) is a field of artificial intelligence aimed at defining efficient methods for data mining and knowledge extraction. This knowledge is generally represented in the form of functions, mathematical or statistical models. Learning is divided into deductive learning and inductive one. The first one consists of starting from generalizations (rules) to specific examples. Inductive learning consists of learning from examples, to induce patterns or hypotheses from the given input data for predicting new data (predictive modeling) or for describing the input data (descriptive modeling).

Historically, the vast majority of techniques in Machine Learning has focused on flat data also called propositional data or attribute-value representation. Flat data is a single-table or single-tuple format where each row represents a data instance, and each column represents an attribute.

A critical assumption made by these techniques on flat datasets is that their individuals are Independently and Identically Distributed (IID). These algorithms admit that the data instances are drawn independently from each other from a same distribution. Examples of such dataset are available on UCI Machine Learning repository [1] include well known Iris dataset [2] , Wine dataset [3] etc., which have been used in many research studies.

Many real cases involve unsure data. In fact, we are often uncertain about the real state of the data because our observations about it are partial (only some aspects of the world are observed)or noisy (even those aspects that are observed are often observed with some errors). The real state of the world is rarely determined with certainty because of our limited observations as if relationships are not deterministic. For example, there are few diseases where we have a clear and deterministic relationship between the disease and its symptoms. These domains can be characterized regarding to a set of random variables, where the value of each variable defines an essential property of the world.

In the early stage of research in Machine Learning, many techniques were proposed in many research communities. Not all techniques can support real-world uncertainty, contrariwise the framework of Probabilistic Graphical Models(PGM) (Koller and Friedman, 2009). There are two major types of PGMs, directed graphical models such as Bayesian Network (BN) (Pearl, 1988), and undirected graphical models such as Markov random fields (MRFs) or Markov Networks (MN) (Pearl, 1988). PGMs become quickly an important tool to address real-world applications where uncertainty is almost an inescapable aspect. They allow to reason probabilistically about the values of one or set of variables, given observations about some others by efficiently encoding a joint distribution over the space of possible assignments of random variables. They are designed to work with flat data using a graph-based representation.

However, many real-world data sets are innately relational. Such data consist of entities of different types, where a different set of attributes characterizes each entity type. Entities are related to each other via various kinds of links, and the link structure is an essential source of information.

Relational data can be represented as a multi-table format where each table corresponds to an entity type or a relationship type. Each row in an entity table represents an entity/object, and each row in a relationship table denotes the relationship between objects.

As objects are related to each other, the IID hypothesis of individuals in tabular data set is often unrealistic. Indeed, according to this hypothesis, no individual can influence other people. Or, individuals are generally not merely data records, but interact with each other, generating many links between them. The IID hypothesis contradicts the so-called autocorrelation phenomena often observed in the real world. For instance, the fact that the cultural tastes of a person depend on those shared by his friends. The knowledge obtained on an individual can then give information about the others, and the individuals can not, therefore, be considered as IID. Thus, the IID assumption is often violated in relational data contexts. This limits the

1. https://archive.ics.uci.edu/ml/index.php
2. https://archive.ics.uci.edu/ml/datasets/Iris
3. https://archive.ics.uci.edu/ml/datasets/Wine

application of PGMs on relational data. Consequently, Statistical Relational Learning (SRL) (Taskar et al., 2007; Neville and Jensen, 2005) has emerged as a branch of machine learning that is concerned with statistical analysis on domains with complex relations and uncertainty. The presence of multiple interrelated aspects characterizes complex systems. Or, many real cases involve both complex and uncertain relational data.

During the last decade, various approaches have been proposed to deal with relational and statistical learning. Probabilistic Relational Models (PRMs) are one of the most used models to handle uncertainty in a relational context. PRMs present a relational extension of PGMs, such as Relational Bayesian Networks (RBN) (Koller and Pfeffer, 1998; Friedman et al., 1999a), Directed Acyclic Probabilistic Entity-Relationships models (DAPER) (Heckerman and Meek, 2004) and Markov Logic Networks (MLN) (Richardson and Domingos, 2006).

PRMs have shown the importance of relaxing the IID assumption. They can handle well structured relational data stored in relational databases for which a relational schema is defined in advance.

In the following section, we will describe the motivating scenario for this thesis and state the problems we try to address.

1.1.2 Motivation and problem statement

With the rise of the Internet, numerous technological innovations and web applications are driving the dramatic increase in data. For example, Facebook, Twitter, and other forms of social media sites, location-based services, smartphones, and other consumer mobile devices including PCs, mobile phones, laptops have allowed billions of individuals around the world to contribute to the amount of the available data. Much of that data does not come only from the web and social networks, but also, from sensor networks. Indeed, technological devices are made up of enabled embedded devices connected to the Internet, including sensors, active positioning tags, and radio-frequency identification (RFID) readers. For example, today some roads have sensors and can communicate with the vehicles passing over them to determine traffic patterns and find more sustainable ways to route cars. Also, texts, pictures, log files, videos, etc. generate a tremendous amount of unstructured data. Much of them are gathered in real time and provide a unique compelling opportunity if they can be analyzed and acted in real time. All these technologies and new forms of personal communication are driving the extensive collection of unstructured and complex datasets.

Consequently, Big Data has emerged as traditional techniques of process and storage that can not deal with a large amount of various and uncertain data coming in fast streaming.

According to (Sun and Reddy, 2013), Big data consists of the major four V's namely "Volume" which means simply lots of data gathered by a company. "Variety" refers to the type of data that Big Data can comprise, data can be structured as well as unstructured. "Velocity" refers to the speed of data generation and the time in which Data Stream can be processed. "Veracity" deals with the uncertainty of data. Each one of these Vs consists of a whole issue that researchers and developers want to solve. So, the challenge for Big Data is how to deal with these issues.

Today, the most of services and sites such as Facebook, Google, Yahoo, etc. have a billion entries in their databases, as many daily visits. Accordingly, one machine cannot handle the whole database. Besides, for reasons of reliability, these databases are duplicated to do not interrupt a service in case of failure or breakdown. The method consists of adding servers to duplicate data and thus increase performance and resist to breakdown. Therefore, two problems appear. First, the storage cost is enormous because each data is present on each server. Second, the cost of insertion/ modification/ deletion is great, because we can not validate transaction if we are not certain that it was performed on all servers and system makes user waiting during this time. This leads to a difficulty of scaling. Also, certain types of queries are not optimized using a relational database, specially queries that consume a huge number of joins between tables. A join means a Cartesian product, so a request with many joins in an extensive database where data are strongly linked takes a lot of time and even can fail. Relational databases cannot cope with all these requirements. Thus, with the emergence of Big data, the need for more flexible databases is evident.

This problem has been recently addressed by a new brand category of data management systems that do not use SQL exclusively, the so-called NoSQL movement (Partner et al., 2014). NoSQL includes four significant types of databases: key/value, oriented document, oriented column, and graph databases. Recently there has been an increasing interest in graph databases to model objects and interactions. In contrast to relational databases, where join-intensive query performance deteriorates as the dataset gets more prominent, a graph database depends less on a rigid schema, and its performance tends to remain relatively constant, even as the dataset grows. This is can be explained by the fact queries are expressed in terms of graph traversal operations which are localized to a portion of the graph.

Consequently, graph databases are rapidly emerging as an effective and efficient solution to the management of extensive datasets in scenarios where data are firmly connected, and data access mainly rely on traversing theses graphs.

Graph databases are unstructured and schema-free data stores. Edges between nodes can have various signatures. In other words, some edges (relationships) that do not correspond to the ER model could be depicted without any problem in the graph database. These relationships are considered as exceptions.

As we have discussed in the previous section, PRMs can be used for learning and reasoning from relational datasets. Using PRMs in Big Data context has been little-tackled topic of research. No work has been identified for PRMs learning from partially structured graph databases.

In this thesis, we are particularly interested in the task of learning PRMs from graph databases. Our focus is on DAPER and MLN models.

This thesis addresses three main issues:

1. How to adapt DAPER learning methods to handle unstructured and various data stored in the graph database?
2. How to deal with "noisy" datasets which contain exceptions?
3. How Markov logic framework can be used for DAPER learning from partially structured graph database?

Consequently, our first contribution is the proposition of an algorithmic approach allowing to learn DAPER model from a graph database.

The originality of this process is that it allows to first identify an Entity Relationship model automatically from an existing graph database instance. As in our graph database, a same typed relationship could be used to connect pairs of vertices with different pairs of classes, our identification approach allows to manage exceptions by considering each relationship label and each corresponding signatures that are enough represented for this label in the database.

Our objective is to learn DAPERs with less structured data and to accelerate the learning process by querying graph databases. This work was presented at AICCSA 2017 and will be explained in detail in Chapter 5.

In our second contribution, we are interested by another probabilistic relation model which rely on the possible-world semantics of first-order probabilistic logic namely MLN. Our proposed method consists of DAPER learning from partially structured graph databases using MLN frameworks. MLN can help us to take into account the exceptions that are dropped during DAPER learning. We propose a joint learning from partially structured graph databases, where we want to learn at the same time the ER schema and the probabilistic dependencies. This contribution is introduced in ICDEc 2018 and will be more explained in Chapter 6.

Finally, we contribute to the development of PILGRIM (**P**robab**IL**istic **GR**aph**I**cal **M**odel) software. A tool developed by our research group DUKe. This thesis has made significant contributions in the imple-

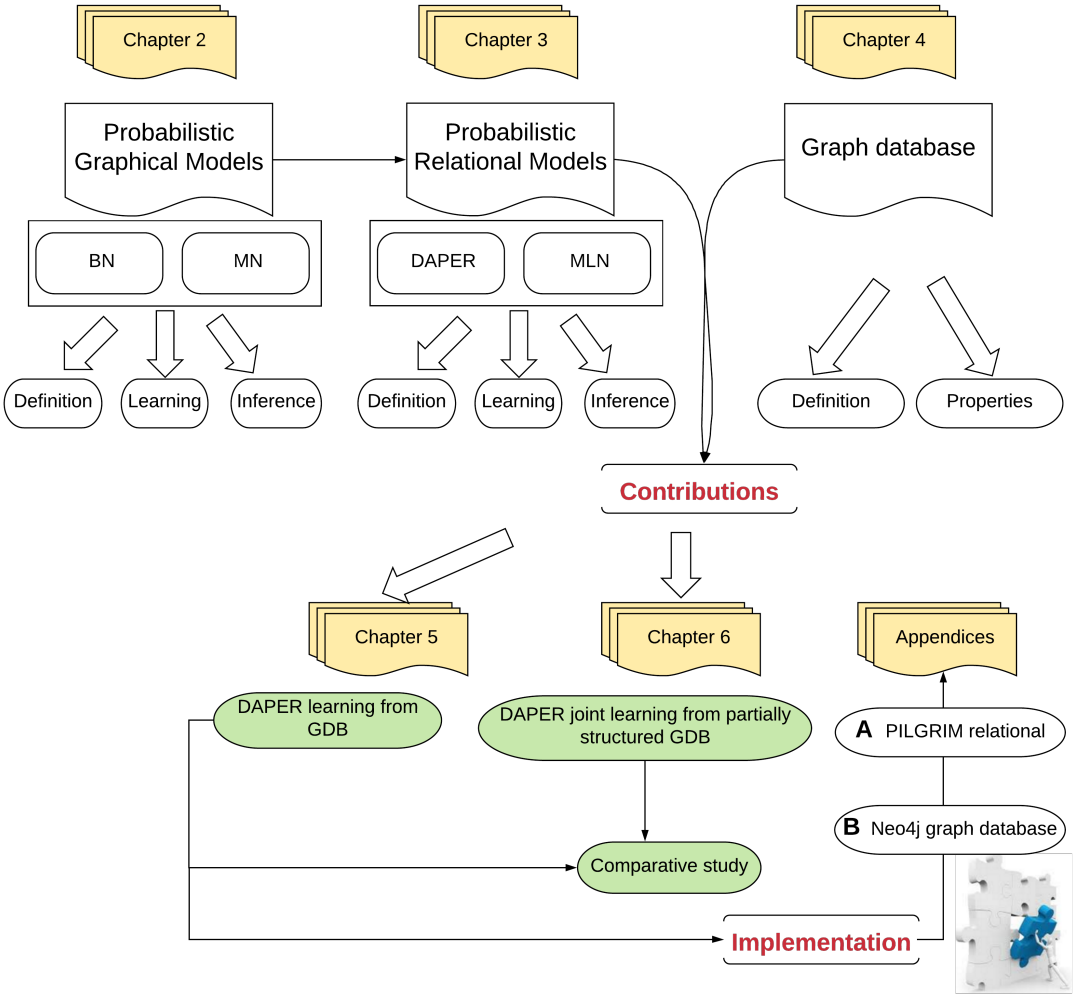


Figure 1.1 – Thesis synopsis.

mentation of a new interface to first be able to communicate with Neo4j [4] graph database as well as the development of functionalities that allow PRM learning from graph database instance.

1.2 Dissertation organization

Figure 1.1 presents the thesis organization and interdependencies between chapters.

Chapter 2 is dedicated to introducing Probabilistic Graphical Models, notably Bayesian Networks and Markov Networks and to make a survey on existing learning approaches.

Chapter 3 is dedicated to the introduction of extensions of Probabilistic Graphical Models, more precisely Direct Probabilistic Entity Relationship and Markov Logic Network models by giving their definition and their learning methods.

Chapter 4 gives an overview of graph database principles. Then, we give a formal definition of the graph database and discuss its properties. Finally, we provide a brief comparison between the graph databases and the most used databases.

4. https://neo4j.com/

Chapter 5 describes our first contribution which is divided into twofold. First, we present how to extract the underlying ER model from a partially structured graph database. Then, we describe a method to compute sufficient statistics (counts) based on graph traversal techniques.

Chapter 6 describes our second contribution which consists of DAPER joint learning from partially structured graph databases using an MLN framework, where we want to learn at the same time the ER schema and the probabilistic dependencies. The Markov Logic Network formalism seems an efficient solution for this task. Both parts of the models are described in the same logical way, and dedicated structure learning algorithm could be able to retrieve both information at the same time. The logical formulas used to describe the model can also manage exceptions, i.e., data which are not coherent with an underlying structured model, as we have to deal with when working with partially structured data.

Chapter 7 presents our conclusions, and point towards some prospects of future research in the area addressed by this thesis.

Appendices A and B presents environments and software used during the development phase and our implementation process.

II

State of the art

2

Probabilistic Graphical Models

Contents

2.1 Introduction **12**
2.2 Basic concepts **12**
2.2.1 Concepts related to graph theory 12
2.2.2 Concepts related to probability theory 13
2.3 Bayesian Networks **15**
2.3.1 Bayesian Networks formalism 15
2.3.2 Parameter learning 18
2.3.3 Structure learning 20
2.4 Markov Networks **23**
2.4.1 Markov Networks formalism 23
2.4.2 Parameter learning 25
2.4.3 Structure learning 26
2.5 Inference in PGMs **27**
2.6 Bayesian Networks vs Markov Networks **28**
2.7 Conclusion **29**

2.1 Introduction

In traditional machine learning settings, data is usually expected to consist of a single type of object, and the objects in the data are assumed to be independent and identically distributed (IID).

One of the most known techniques in model-based machine learning from flat data consists in constructing graphical models which also fits within the dependence analysis framework and corresponds to extracting dependency relations from observed data to make predictions.

Probabilistic Graphical Models (PGM) (Koller and Friedman, 2009) have become a viral tool for dealing with uncertainty through the use of probability theory, and a practical approach to coping with complexity using graph theory. They represent the probability distribution of a set of random variables in combination with dependent variables using graphs.

There are two major types of PGMs, directed graphical models such as Bayesian Network (BN) (Pearl, 1988), and undirected graphical models such as Markov Random Fields (MRFs) or a Markov Networks (MN) (Pearl, 1988).

The two main tasks of PGMs are learning and inference. Learning implies the identification of both components of the model (structure and parameters). Inference task consists of studying the impact of some partially observed variables known as evidence on remaining ones.

Parameter learning of BN and MN consists of identifying a set of conditional probabilistic distributions (CPDs) of the BN or to identify a set of parameters which define the potential functions of the MN. Structure learning of BN and MN consists of determining the set of probabilistic dependencies between discrete random variables in the network. Several approaches have been proposed to address these tasks. The inference task consists of studying the impact of some partially observed variables known as evidence on remaining ones given a specific fixed joint probability distribution, in which case the difference between BN and MN are less important. Indeed, same inference methods can be applied to BN and MN.

The remainder of this chapter is as follows: Section 2.2 recalls some basic concepts from graph and probability theories. A detailed description and definition of these two models with their graphical representation are presented in Sections 2.3 and 2.4. Sections 2.3.2 and 2.4.2 introduce parameter learning methods for BN and MN respectively. Sections 2.3.3 and 2.4.3 present BNs and MLNs structure learning approaches. Section 2.5 goes through inference task in both models. A brief comparison between both models is given in Section 2.6.

2.2 Basic concepts

Bayesian Network and Markov Network are founded on both graph and probability theories. Thus, it is useful to recall some valuable concepts from these theories before addressing these models. All these concepts and definitions are derived (Diestel, 2005) and (Durrett, 2009; Renyi, 2007).

2.2.1 Concepts related to graph theory

Definition 2.2.1. ***Graph***
A graph $G = (V, E)$ is a finite collection of nodes (or vertices) $V = \{n_1, n_2, ..., n_N\}$ and set of edges (or links) $E \subset \{(u, v) \parallel u, v \in V\}$

Definition 2.2.2. ***Neighbor***
Two nodes are neighbors (or adjacents) if an edge connects them

Definition 2.2.3. ***Directed graph***
A directed graph is a graph in which all edges are directed is a directed graph.

Definition 2.2.4. ***Undirected graph***
An undirected graph is a graph in which all edges are undirected is an undirected graph.

Definition 2.2.5. ***Moral graph***
A moral graph of a directed acyclic graph G is formed by adding edges between all pairs of nodes that have a common child and then making all edges in the graph undirected.

Definition 2.2.6. ***Complete graph***
A complete graph is a graph in which an edge connects every pair of nodes is a complete graph.

Definition 2.2.7. ***Clique***
A clique is a complete subgraph of a graph G.

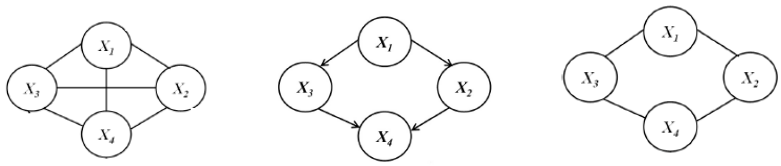
Definition 2.2.8. ***Maximal clique***
Clique with a maximal number of nodes; cannot add any other node while still retaining complete connectedness.

Definition 2.2.9. ***Path***
A path in a graph is a finite or infinite sequence of edges which connect a sequence of vertices.

Definition 2.2.10. ***Directed path***
A directed path is a path in which all edges are directed in the same direction is a directed path.

Definition 2.2.11. ***Directed cycle***
A directed cycle is a directed path starting and ending at the same vertex.

Definition 2.2.12. ***Directed Acyclic Graph****(DAG)*
A Directed Acyclic Graph is a graph that contains only directed edges and no cycles is called a directed acyclic graph (DAG). Given a graph G with four nodes X_1, X_2, X_3, and X_4,
if there is a directed edge from X_1 to X_2, then X_1 is a parent of X_2, and X_2 will be the child of X_1.
If there is a directed path from X_1 to X_4, then X_1 is an ancestor of X_2 and X_2 is a descendant of X_1.



(a) A complete graph (b) A directed acyclic graph (c) An undirected graph

Figure 2.1 – Example of graphical models

2.2.2 Concepts related to probability theory

Definition 2.2.13. ***Random variable***
A random variable is a variable whose possible values are numerical outcomes of a random phenomenon. There are two types of random variables, discrete and continuous.

Let us consider only discrete variables and denoted by upper case letters $(e.g., X, Y, Z, ...)$.

	Y_0	Y_1
X_0	0.2	0.8
X_0	0.8	0.2

Table 2.1 – Example of CPD $P(X|Y)$

Definition 2.2.14. ***Domain of values***
The Domain of values is the number of states (values) associated with random variable known as the domain of values. The finite domain of the variable X is denoted by $\mathcal{D}(X)$.

Let us denote by lower case letters $(x, y, z, ...)$ the value taken by the corresponding variable $(X, Y, Z, ...)$.

Definition 2.2.15. ***Probability distribution***
The probability distribution $P(X)$ of a discrete random variable X is a list of probabilities $P(X = x)$ associated with each of its possible values $x \in \mathcal{D}(X)$.
The probability of an event $x \in \mathcal{D}(X)$ is non-negative,

$$P(X = x) > 0$$

and the probabilities of all events sum to one,

$$\Sigma_{x_i \in \mathcal{D}(X)} P(X = x_i) = 1$$

Definition 2.2.16. ***Conditional Probability Distribution CPD***
Given two random variables X, Y, the distribution over X conditioned on Y denoted $P(X, Y)$ defines a distribution over X for each possible value of Y. (e.g., Table 2.1 is an example of CPD $P(X|Y)$.

Generally, the conditional probability of X = x under the condition Y = y (evidence) is defined as:

$$P(x|y) = \frac{P(x, y)}{P(y)} \tag{2.1}$$

Definition 2.2.17. ***Bayes theorem***

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)} \tag{2.2}$$

Definition 2.2.18. ***Conditional independence***
Let three random variables X, Y and Z. X and Y are conditionally independent to Z $(notedX \perp Y|Z)$ if and only if:

$$P(X|Y, Z) = P(X|Z)$$

or

$$P(Y|X, Z) = P(Y|Z)$$

X_0Y_0	X_0Y_1	X_1Y_0	X_1Y_1
0.3	0.5	0.1	0.1

Table 2.2 – Example of JPD $P(X, Y)$

Definition 2.2.19. ***Joint Probability Distribution (JPD)***
For a set of random variables X_i, i = 1, 2, ..., n, the joint probability distribution is defined to be the distribution over all variables $P(X_1, ..., X_n)$. To determine the joint distribution of n random variables, the cartesian product of all the random variables domains must be provided. If each variable has k states, then there are k^n combinations. This product is exponential on the number of variables. (e.g., the Table 2.2 is an example of joint distribution $P(X, Y)$ for two binary variables X, Y.
More generally, a joint distribution over n random variables may be expressed using conditional probabilities as follows:

$$\begin{aligned} P(X_1, ..., X_n) : \mathcal{D}_{X_1} \times ... \times \mathcal{D}_{X_n} &\to [0, 1] \\ (x_1, ..., x_n) &\mapsto P(x_1, ..., x_n) = P(\bigcap_{i \in \{1,...,n\}} \{X_i = x_i\}) \end{aligned} \tag{2.3}$$

where $\mathcal{D}_X$ is the domain of the random variable X, and corresponding lower-case letter x denotes the assignments or states of X.

2.3 Bayesian Networks

2.3.1 Bayesian Networks formalism

Bayesian Network (BN) (Pearl, 1988) is known as a directed acyclic graph (DAG) in which nodes are random variables, while the edges between the nodes represent probabilistic dependencies among the corresponding random variables. Nodes that are not connected by edges are variables that are conditionally independent of each other. Nodes that are connected by edges do have conditional, dependent relationships. BN enables a representation and computation of the joint probability distribution (JPD) over a set of random variables. This JPD can be determined uniquely by these local conditional probability tables (CPTs). In this section, a formal definition of the Bayesian network (Pearl, 1988) is given. Then, we recall the d-separation criterion. Finally, we present the Markov equivalence class for directed acyclic graphs.

2.3.1.1 Bayesian Network definition

Formally, Bayesian Network $\mathcal{B} =< G, \theta >$ is defined by:

— A graphical (qualitative) component consists of a directed acyclic graph (DAG) $G = (V, E)$ where V is the set of vertices (nodes) $X_1, X_2, \ldots, X_n$ represents random variables. E is the set of directed edges that describes the direct dependencies between these variables.

— A numerical (quantitative) component consists of a set of parameters $\theta = \{\theta_1, ..., \theta_n\}$ where each $\theta_i = \{P(X_i|Pa(X_i))\}$ represents the set of probabilities for each node X_i conditionally to the states of its parents $Pa(X_i)$ in G.

Example 2.3.1. *An example of a Bayesian network is shown in Figure 2.2 which illustrates four random variables "Difficulty, Intelligence, Grade and Ranking". This network demonstrates the influence between the ranking of student and its grade giving its intelligence and course' difficulty.*

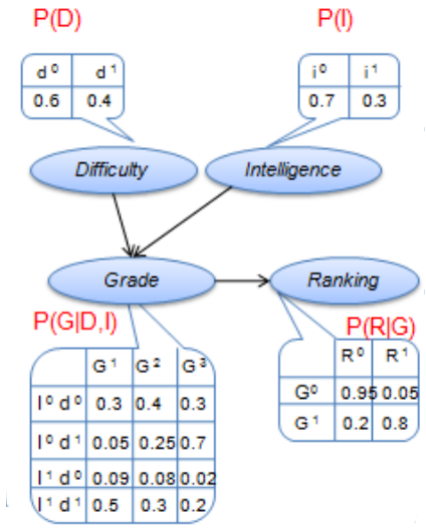


Figure 2.2 – Example of Bayesian Network.

2.3.1.2 Conditional Independence in Bayesian Networks

A BN reflects a simple conditional independence statement called local Markov property.

Property 2.3.1. ***(Local Markov Property)****: Each variable is independent of its non descendants $Nd(X_i)$ in the graph given its parents $Pa(X_i)$.*

$$X_i \perp Nd(X_i)|Pa(X_i)$$

Example 2.3.2. *In the figure 2.2 $Ranking$ is independent of $Difficulty$, $Intelligence$ given its parent $Grade$.*

The conditional independence relationships encoded by a BNs are devised into three type of connections; serial, diverging and converging connections.

The three types of connections as illustrated in Figure 2.3.

— Serial connection: consider the case in which there is one incoming and outgoing arrow to Y, $X \rightarrow Y \rightarrow Z$. It is intuitive that the nodes upstream and downstream of Y are dependent if Y is hidden.

— Diverging connection: when there are two diverging arrows from Y, $X \leftarrow Y \rightarrow Z$. Y is a "root". If Y is hidden, the children are dependent, because they have a hidden common cause, so the ball passes through. If X is observed, its children are rendered conditionally independent, so the ball does not pass through.

— Converging connection (collider or v-structure): when two arrows are converging on a node $X \rightarrow Y \leftarrow Z$ with Y is a "leaf". If Y is hidden, its parents are marginally independent, and hence the ball does not pass through (the ball being "turned around" is indicated by the curved arrows); but if Y is observed, the parents become dependent, and the ball does pass through.

Given these three type of connections, BN encodes a critical criterion called d-separation. Thanks to this criterion a joint probability distribution encoded by a Bayesian Network can be factorized into a local probability distribution.

Definition 2.3.1. *Let G = (V,E) be a DAG and let **X**, **Y**, and Z be disjoint sets of variables in V:*

— *A path from some $X \in$ **X** to some $Y \in$ **Y** is d-connected given **Z** if and only if every collider **W** on the path, or a descendant of **W**, is a member of Z and there are no non-colliders in **Z**.*

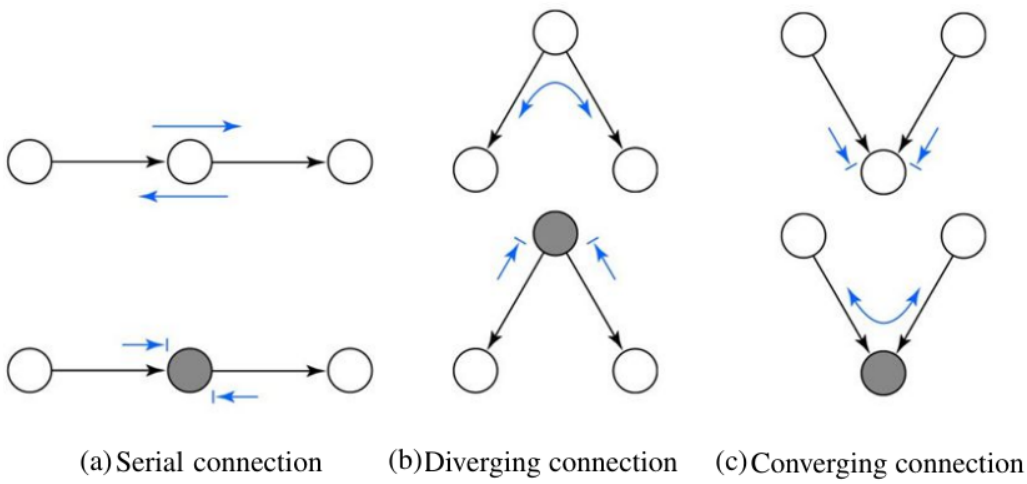


Figure 2.3 – The rules of Bayes Ball theorem

— ***X*** *and* ***Y*** *are said to be d-separated by* ***Z*** *if and only if there are no d-connecting paths between* ***X*** *and* ***Y*** *given* ***Z****.*

Example 2.3.3. *In Figure 2.2, If $Grade$ is not observed, $Difficulty$ and $Ranking$ are dependent. If $Grade$ is observed, $Difficulty$ and $Ranking$ are conditionally independent; this path is a serial connection. If $Grade$ is not observed, $Difficulty$ and $Intelligence$ are independent; this path is a converging connection having its middle node $Grade$. If $Grade$ is observed, $Difficulty$ and $Intelligence$ are conditionally dependent.*

Accordingly, a Bayesian Network $\mathcal{B}$ defines a unique join probability distribution over V, namely:

$$P_B(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P_B(X_i|Pa(X_i)) = \prod_{i=1}^{n} \theta_{X_i|Pa(X_i)} \tag{2.4}$$

This function is decomposed into a product of terms depending only on the local node and its parents in the graph that is a significant result, and very useful of the local Markov Property 2.3.1 that verifies each variable X_i is independent of its non descendants given its parents in G.

2.3.1.3 Markov equivalence class for Directed Acyclic Graphs

Different DAGs can have the same d-separations. We say that they are Markov equivalent.

Definition 2.3.2. *Let $\{G_1 = V, E_1\}$ and $G_2 = \{V, E_2\}$ be two DAGs containing the same set of nodes V. Then G_1 and G_2 are called Markov equivalent if for every three mutually disjoint subsets $A, B, C \subseteq V$, A, and B are d-separated by C in G_1 if and only if A and B are d-separated by C in G_2. That is*

$$I_{G_1}(A, B \mid C) \Leftrightarrow I_{G_2}(A, B \mid C)$$

Theorem 2.3.1. *Two DAGs are Markov equivalent if and only if, based on the Markov condition, they entail the same conditional independence.*

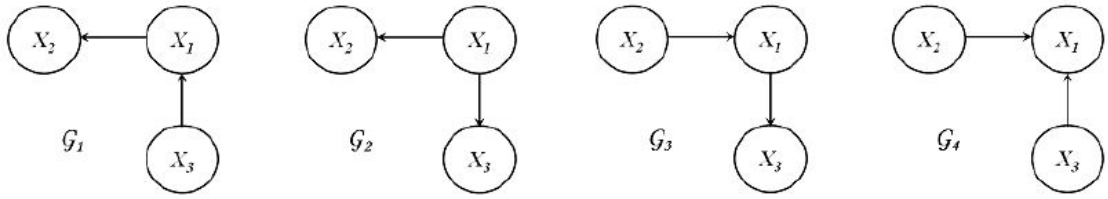


Figure 2.4 – Markov equivalence

Example 2.3.4. *Let's consider the Figure 2.4. In this example, it can be easily demonstrated that G_1 and G_2 are equivalent by the decomposition of their joint probability distributions (Naïm et al., 2011). We have:*

$$P(X_1, X_2, X_3)_{G_1} = P(X_2 \mid X_1) * P(X_1 \mid X_3) * P(X_3)$$

and

$$P(X_1, X_2, X_3)_{G_2} = P(X_2 \mid X_1) * P(X_3 \mid X_1) * P(X_1)$$

Thus

$$\begin{aligned} P(X_1, X_2, X_3)_{G_2} &= P(X_2 \mid X_1) * P(X_3 \mid X_1) * P(X_1) \\ &= P(X_2 \mid X_1) * \frac{P(X_1 \mid X_3) * P(X_3)}{P(X_1)} * P(X_1) \\ &= P(X_2 \mid X_1) * P(X_1 \mid X_3) * P(X_3) \\ &= P(X_1, X_2, X_3)_{G_1} \end{aligned}$$

Similarly, it can be demonstrated that G_3 is equivalent to G_1 and G_2. These three networks are not equivalent to the v-structure G_4. As $P(X_1, X_2, X_3)_{G_4} = P(X_1 \mid X_2, X_3) * P(X_2) * P(X_3)$ and $P(X_1 \mid X_2, X_3)$ cannot be simplified.

Markov equivalence is redefined regarding the concepts of skeleton and v-structure by the following theorem, initially introduced by (Verma and Pearl, 1990).

Theorem 2.3.2. *Two DAGs are equivalent if and only if they have the same skeletons and the same v-structures.*

The main two tasks performed by a Bayesian network are learning and inference. BN learning implies structure learning and parameter estimation from data. Parameter estimation consists of identifying the set of probability distributions using either the statistical approach or the Bayesian one. Structure learning consists of determining the set of probabilistic dependencies of the network. Once the BN is learned, the inference can be performed. In the following sections, we focus on BN parameter and structure learning from complete data.

2.3.2 Parameter learning

The joint probability is given by Equation 2.4 of a domain which can be factorized into smaller and local distribution. Each local distribution involves a node and its parents only. Parameter learning means to estimate these CPDs from data. In the parameter estimation task, we assume that the qualitative dependency structure of the graphical model is known.

Suppose that we have n discrete variables, $x_1, ..., x_n$, and n complete observations $D = \{(x_{k_1}^{(l)}, ..., x_{k_n}^{(l)}), l = 1, ..., n\}$. Each observation contains a value assignment to all the variables in the model. There are two classes of approaches for learning the CPDs of a given structure. The statistical approach consists of estimating the probability of an event by its frequency of occurrence in the database known as the maximum likelihood estimation. The Bayesian approach consists on finding the most likely parameter knowing that the data were observed using some prior.

2.3.2.1 Statistical approach

The statistical approach consists of estimating the probability of an event by calculating its occurrence frequency in the dataset. The statistical approach based on the Maximum Likelihood (MV) criterion, i.e., the probability of the data given the BN is maximal. Given the complete data D, the MV function is:

$$\hat{P}(X_i = x_k | pa(X_i) = x_j) = \hat{\theta}_{i,j,k}^{MV} = \frac{N_{i,j,k}}{\Sigma_k N_{i,j,k}} \tag{2.5}$$

where, N_{ijk} is the number of samples in the database for which X_i is on its kth configuration for parent configuration x_j.

Despite, this approach has its disadvantageous because when some configurations do not exist in the dataset, the corresponding parameter will be evaluated by a zero value. Thus, the Bayesian approach was introduced where each configuration even it does not exist in the data set will be assigned to a non null probability.

2.3.2.2 Bayesian approach

The idea in Bayesian approach consists of estimating the most likely parameter θ_i as the data were observed, by assigning a prior probability over the graph, this is advantageous when the available data are limited, and the number of parameters is significant. A prior distribution is assumed over the parameters of the local distributions before the data are used. A distribution family is called conjugate prior to a data distribution when the posterior over the parameters belong to the same family as the prior, albeit with different hyperparameters α_i (to not confuse with the parameters θ_i) (Margaritis, 2003).

For multinomial distributions, the conjugate prior comes from the Dirichlet family. Denoting the probability of each θ_{ijk}, $k = 1, ..., r_i$ in the local distribution of variable X_i for the parent configuration x_j, the Dirichlet distribution over these parameters is expressed by:

$$P(\theta_{ij1}, \theta_{ij2}, ..., \theta_{ijr_i} \mid G) = Dir(\alpha_{ij1}, \alpha_{ij2}, ..., \alpha_{ijr_i}) = \Gamma(\alpha_{ij}) \prod_{k=1}^{r_i} \frac{\theta_{ijk}^{\alpha_{ijk}-1}}{\Gamma(\alpha_{ijk})} \tag{2.6}$$

where α_{ijk} are its hyperparameters and $\alpha_{ij} = \Sigma_{k=1}^{r_i} \alpha_{ijk}$. Based on this Equation 2.6, the distribution over the set of parameters of the entire BN is

$$P(\theta \mid G) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} \Gamma(\alpha_{ij}) \prod_{k=1}^{r_i} \frac{\theta_{ijk}^{\alpha_{ijk}-1}}{\Gamma(\alpha_{ijk})} \tag{2.7}$$

The advantage of Dirichlet distribution is that it can easily express the posterior probability over the parameters conditional on the data D, since its conjugate prior to the multinomial.

$$P(\theta_{ij1}, \theta_{ij2}, ..., \theta_{ijr_i} \mid G, D) = Dir(N_{ij1} + \alpha_{ij1}, N_{ij2} + \alpha_{ij2}, ..., N_{ijr_i} + \alpha_{ijr_i}) \tag{2.8}$$

$$P(\theta \mid G, D) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} \Gamma(\alpha_{ij}) \prod_{k=1}^{r_i} \frac{\theta_k^{N_{ijk}+\alpha_{ijk}-1}}{\Gamma(N_{ijk}+\alpha_{ijk})} \tag{2.9}$$

where, N_{ijk} is the number of samples in the database for which X_i is on its kth configuration for parent configuration x_k.

Maximum A Posteriori approach:

$$\theta^{MAP} = argmax_\theta P(\theta \mid G, D) = \frac{N_{ijk}+\alpha_{ijk}-1}{\Sigma_k(N_{ijk}+\alpha_{ijk}-1)} \tag{2.10}$$

Expected A Posteriori approach:

$$\theta^{EAP} = E[P(\theta \mid G, D)] = \frac{N_{ijk}+\alpha_{ijk}}{\Sigma_k(N_{ijk}+\alpha_{ijk})} \tag{2.11}$$

2.3.3 Structure learning

Learning the Bayesian Network Structure consists of identifying the set of probabilistic dependencies between the set of random variables. There are three classes of algorithms for BN structure learning. The first class tackles this issue as a constraint satisfaction problem. Constraint-based algorithms look for independencies (dependencies) in the data, using statistical tests then, try to find the most suitable graphical structure with this information. The second class treats learning as an optimization problem. They evaluate how well the structure fits the data using a score function. So, these Score-based algorithms search for the structure that maximizes this function. The third class presents hybrid approaches that mix both of the first two ones. Before introducing the basics of these three families, some learning assumptions are cited.

2.3.3.1 Assumptions for Learning the BN Structure

Besides the Local Markov Property 2.3.1, other assumptions are also required to perform structure learning. These are:

Faithfulness assumption:A BN graph G and a probability distribution P are faithful to one another iff every one and all independence relations valid in P are those entailed by the Markov assumption on G.

Causal Sufficiency Assumption: The set of variables X is sufficient to represent all the conditional dependence relations that could be extracted from data. This means, there are no unobserved (latent) variables in the domain that are the parent of one or more observed variables.

2.3.3.2 Constraint-based methods

Constraint-based methods are based on conditional independencies obtained from statistical tests on the data. They try to test for conditional dependence and independence in the data, and then find a network that best explains these dependencies and independencies. Constraint-based methods are quite intuitive; they strictly follow the definition of Bayesian network.

Several algorithms have been proposed in the literature, e.g., IC (Verma and Pearl, 1990), SGS (Spirtes et al., 1990), PC (Spirtes et al., 2000). (Chickering, 2003) proposed a method to convert a BN DAG to CPDAG presenting its Markov equivalence class. It begins by ordering all the arcs of the originating network, then browses the set of arcs already ordered to "Simplify" reversible arcs.

Constraint-based algorithms have certain disadvantages. The most important one, they are sensitive to failures in individual independence tests. Another drawback is that most of the algorithms such as the SGS (Spirtes et al., 1990) and IC (Verma and Pearl, 1990) algorithm, are exponential in time in the number of variables of the domain (Margaritis, 2003).

2.3.3.3 Score-based methods

Score-based approaches, also known as score-and-search techniques, consist to either search the structure that maximize the score function or look for the best structures and combine their results.

The scoring functions

Score functions are often devised into two terms: the likelihood $L(D \mid \theta, \mathcal{B})$, and a second term which takes into account the complexity of the model.

Definition 2.3.3. ***Bayesian Network dimension***
Given a random vector composed of n random variables $X = (X_1, ..., X_n)$. If r_i is the modality of the variable X_i, then the number of parameters needed to represent the probability distribution $P(X_i \mid Pa(X_i) = x_j)$ is equal to $r_i - 1$. The dimension of BN $\mathcal{B}$ is defined as:

$$Dim(\mathcal{B}) = \sum_{i=1}^{n} Dim(X_i, \mathcal{B}) \tag{2.12}$$

with,

$$Dim(X_i, \mathcal{B}) = (r_1)q_i \tag{2.13}$$

where, q_i is the number of possible configurations for the parents of X_i, $q_i = \prod_{X_j \in Pa(X_i)} r_j$

To be able to use the based-score method, it is required that the score function must be decomposable. Other interesting properties, that the score function is score equivalent.

Definition 2.3.4. ***Decomposable score function***
A score S is decomposable (or local) if it can be written as the sum or the product of scores where each one of them is measured concerning one node and its parents. That is, if n is the number of nodes of the BN $\mathcal{B}$, the score function must have one of these forms:

$$S(\mathcal{B}) = \sum_{i=1}^{n} s(X_i, Pa(X_i)) \qquad or \qquad S(\mathcal{B}) = \prod_{i=1}^{n} s(X_i, Pa(X_i)) \tag{2.14}$$

Definition 2.3.5. ***Score-equivalent score function***
A scoring function is score-equivalent if it assigns the same score to similar graphs.

Several scoring functions have been developed to assess the goodness-of-fit of a particular model. In the remainder, we will focus on Bayesian scores.
For instance, (Cooper and Herskovits, 1992) proposed a score based on a Bayesian approach. This score computes the posterior probability distribution, starting from a prior probability distribution on the possible networks $P(\mathcal{B})$, conditioned to data D, that is, $P(\mathcal{B} \mid D)$.
Given a set of random variables $X = (X_1, ..., X_n)$, $n \geq 1$, with $X_i = \{x_{i1}, ..., x_{ir_i}\}$, $i = 1...n$. D is a dataset with N sample case and $\mathcal{B}$ is the structure of BN. q_i is the possible parent configurations of the variable i and N_{ijk} is the number of times where variable i took on value k with parent configuration j. The Bayesian Dirichlet (BD) score is expressed by:

$$ScoreBD(\mathcal{B}, D) = P(\mathcal{B})P(D \mid \mathcal{B}) = P(\mathcal{B}) \prod_{i=1}^{n} \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij} + \alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})} \tag{2.15}$$

Where, Γ is the Gamma function. However, BD score is unusable in practice because it is not score-equivalent.
(Heckerman et al., 1995) introduced the Bayesian Dirichlet Equivalent score known as BDe which is based

on the same formula as BD score with other interesting properties. Namely, two equivalent structures have the same score. BDe uses a priori distribution over the parameters.

$$\alpha_{ijk} = N' \times P(X_i = x_k, pa(X_i) = x_j \mid \mathcal{B}_c) \tag{2.16}$$

where $\mathcal{B}_c$ is a priori structure with no conditional independencies; a completely connected graph. N' is the number of similar examples defined by the user.
(Buntine, 1991) proposed a particular case of BDe, which is especially interesting, appears when that is, the prior network assigns a uniform probability to each configuration of X_i. It is known as BDeu.

$$\alpha_{ijk} = \frac{N'}{r_i q_i} \tag{2.17}$$

This score only depends on one parameter, the equivalent sample size N'. In practice, the BDeu score is very sensitive concerning the equivalent sample size N', and so, several values are attempted.

(Borgelt and Kruse, 2002) generalized the BD score by introducing a hyperparameter γ. This score is known as BD_γ and expressed by:

$$ScoreBD_\gamma(\mathcal{B}, D) = P(\mathcal{B}) \prod_{i=1}^{n} \prod_{j=1}^{q_i} \frac{\Gamma(\gamma N_{ij} + \alpha_{ij})}{\Gamma((\gamma+1)N_{ij} + \alpha_{ij})} \cdots \prod_{k=1}^{r_i} \frac{\Gamma(\gamma + 1N_{ijk} + \alpha_{ijk})}{\Gamma(\gamma N_{ijk} + \alpha_{ijk})} \tag{2.18}$$

The search procedure

Given training dataset, scoring function and a set of possible structures as input, the score-based methods consist of generating a network that maximizes the score as output. Some recent works deal with getting the optimal solution (Bartlett and Cussens, 2017). However, more usual methods propose to get a local optimum by exploring the solution space with various heuristics. Mainly they are classed into two main groups: heuristics on the search space and heuristics on the search method. The principal of the first class of heuristics is to reduce the search space. Namely, (Chow and Liu, 1968) proposed a search method based on the idea of searching for the Maximum Weight Spanning Tree or (MWST). In other words, the tree passing through all nodes and maximizing a score defined for all possible arcs. The other heuristic to reduce the search space is ordering the nodes to look for parents of a node only from the next nodes. Such method was proposed by (Cooper, 1992). This method called K2. It is widely used. However, it has the disadvantage of requiring an order listed as an input parameter.
For the second class of heuristic on the search method, The most straightforward search procedure is the greedy one. Greedy Search (GS) (Chickering, 2003) performs as follows: for each BN structure $\mathcal{B}$, GS moves on to the neighbor graph that has a better score, until it reaches a structure that has the highest score in the list of neighbors. (Chickering et al., 1995) use the idea of GS on the space of Bayesian networks. The algorithm consists of starting from an initial network structure $\mathcal{B}$ that may be either empty or not. Second, compute its score. Then, considering all of the neighbors of $\mathcal{B}$ in the space. The score is computed for each of them. Apply the change that leads to the best improvement in the score. This process is repeated until convergence (the score obtained will be stable). Thus GS is sensitive to initialization and converges to a local optimal. A way to reduce the risk of falling into the local optimum is to repeat the algorithm several times starting from different initializations randomly chosen. This method is known as the iterated hill climbing or random restart to discover multiple optima and therefore more likely to converge to the optimal solution.

2.3.3.4 Hybrid methods

Hybrid algorithms combine both techniques (constraint-based methods and score-based methods) by using the local conditional independence tests and the global scoring functions. Using a hybrid approach

was proposed for the first time by (Singh and Valtorta, 1993), (Singh and Valtorta, 1995), where they used conditional independence tests to construct an ordering variable. Then, they use this ordering as input to the K2 algorithm to learn the structure.

After that, several methods were proposed. (Friedman et al., 1999b) proposed the Sparse Candidate (SC) algorithm. It uses conditional independence to find suitable candidate parents and hence limit the size of the search in later stages. (Dash and Druzdzel, 1999) combine the PC algorithm with Greedy Search. (Acid and de Campos, 2003) perform an initial GS with random restart and then use conditional independence tests to add and delete arcs from the obtained DAG. (Tsamardinos et al., 2006) proposed the max-min-hill-climbing (MMHC), a hybrid approach that combines the local search technique of the constraint-based methods and the overall structure search technique of the score-based methods. This algorithm consists of two main phases. First, it searches for each node in the graph, the set of candidate nodes that can be connected to it. Second, it constructs the graph using the greedy search heuristic constrained to the set of candidate children and parents of each node in the graph as defined in the first phase.

In this section, we introduced the Bayesian Network as a powerful directed probabilistic graphical tool. Also, undirected graphical models are a robust framework for learning and reasoning under uncertainty. However, directed and undirected graphical models differ regarding their Markov properties (the relationship between graph separation and conditional independence) and their parameterization (the relationship between local numerical specifications and global joint probabilities). These differences are important in discussions of the family of joint probability distribution that a particular graph can represent.

Therefore, in the next, we turn our attention to undirected graphical models especially the Markov Network.

2.4 Markov Networks

2.4.1 Markov Networks formalism

A Markov Network (MN) is based on undirected graphical models. As in a Bayesian network, the nodes in the graph of a Markov network graph represent the variables, and the edges correspond to the probabilistic interaction between the neighboring variables. To represent the distribution, the graph structure is associated with a set of parameters, in the same way, that CPDs were used to parameterize the directed graph structure BN. In this section, a formal definition of Markov network (Pearl, 1988) is given. Then, we recall the clique factorization criterion and Markov properties.

2.4.1.1 Markov Network definition

A Markov Network (MN), also called Markov Random Field is a probabilistic graphical model that represents a joint distribution of a set of variables $X = (X_1, X_2, ..., X_n)$ (Pearl, 1988). Formally, a Markov Network $M = (G, \varphi_k)$ is defined by:

— A graphical (qualitative) component $G = (V, E)$ represented by an undirected graph where V is the set of vertices or nodes that represent discrete random variables and E a set of edges indicates a dependency relationship among these variables.

— A numerical (quantitative) component φ_k consists of the potential function for each clique (a subset of nodes in the graph in which all pairs of nodes are connected) in the graph.

Example 2.4.1. *Figure 2.5 presents the graphical component of a MN. It illustrates a MN having four random variables $X = (A, B, C, D)$. The links between nodes indicate which variables have contact with each other. This graphical structure contains four maximal cliques: $\{A, B\}$, $\{B, C\}$, $\{C, D\}$ and $\{D, A\}$.*

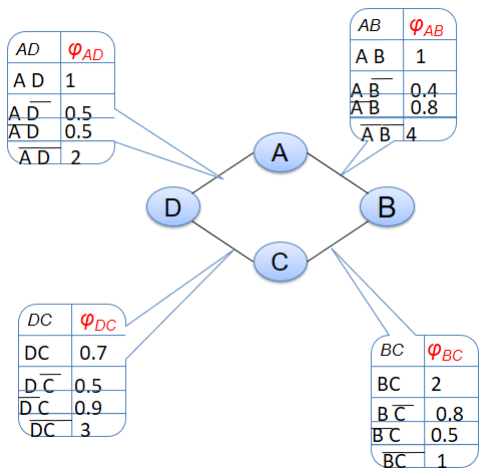


Figure 2.5 – Example of Markov Network

Definition 2.4.1. ***Markov blanket***
For each node $X \in \mathcal{X}$ in a Markov Network, the Markov blanket of X, denoted $N_H(X)$, is the set of neighbors of X in the graph (those that share an edge with X).

2.4.1.2 Conditional Independence in Markov networks

As in the case of Bayesian Networks, the graph structure of a Markov Network can be viewed as encoding a set of independence assumptions.
Given an undirected graph $G = (V, E)$, a set of random variables $X = (X_1, X_2, ..., X_n)$, a MN encodes the following conditional independence assumptions, called Markov properties:

Property 2.4.1. ***Pairwise Markov property****: Any two non-adjacent variables are conditionally independent given all other variables.*

$$X_a \perp X_b \mid V_{\setminus\{a,b\}}$$

Example 2.4.2. *In the figure 2.5, $A \perp C \mid \{D, B\}$*

A MN also encodes the Local Markov property like BN. A Local Markov property for MN is defined as follow:

Property 2.4.2. ***Local Markov property****: Every variable is conditionally independent of all other variables given its closed neighbors.*

$$X_a \perp X_{V\setminus cl_{(a)}} \mid X_{N_H}(a)$$

where $N_H(a)$ is the set of neighbours of a, and $cl(a) = \{a\} \cup ne(a)$ is the closed neighbourhood of a.

Example 2.4.3. *In the Figure 2.5, we have the following Local Markov assumptions:*
$A \perp D \mid \{B, C\}$, $B \perp C \mid \{A, D\}$, $C \perp B \mid \{A, D\}$, $D \perp A \mid \{B, C\}$

2.4.1.3 Clique factorization

As MN is an undirected graph, so we can't use the chain rule to represent $P(X)$ since there is no topological ordering associated with a MN graph. So instead of associating CPDs with each node as it is made with Bayesian Networks, MNs are factorized according to the cliques of the graph. Factor graphs are a little more complex. They provide a possible representation to describe explicitly the cliques and the corresponding potential functions. In other words, we associate potential functions or factors with each maximal clique in the graph, this makes conditional dependencies computed regarding maximal cliques. Given a set of random variables $X = (X_1, X_2, ..., X_n)$. $P(X = x)$ is the probability of finding that the random variables X take on the particular value x. This joint distribution can be factorized over the cliques of G as follow:

$$P(X = x) = \frac{1}{Z} \prod_k \varphi_k(x_{\{k\}}) \tag{2.19}$$

x_k is the state of the kth clique (i.e., the state of the variables that appear in that clique). Z, known as the partition function, is given by:

$$Z = \Sigma_{x \in X} \varphi_k(x_{\{k\}})$$

Where X denotes the set of all possible assignments of values to all the MN's random variables. Z is included to maintain the normalization condition for the distribution, $\Sigma_x P(X) = 1$.

Example 2.4.4. *In the Figure 2.5,we have four maximal cliques. Suppose that $\varphi_1(A, B)$, $\varphi_2(B, C)$ and $\varphi_3(C, D)$ and $\varphi_4(D, A)$ are four potential functions corresponding respectively to these maximal cliques and $x = (a, b, c, d) \in X$. We have:*

$$P(A = a, B = b, C = c, D = d) = \tfrac{1}{Z}\varphi_1(a, b)\varphi_2(b, c)\varphi_3(c, d)\varphi_4(d, a)$$

2.4.1.4 Logistic model

A Markov network is often written as a log-linear model with each clique potential replaced by an exponentiated weighted sum of features of the state, leading to:

$$P(X = x) = \frac{1}{Z} exp(\sum_k w_k f_k(x)) \tag{2.20}$$

With $\varphi_k(x_{\{k\}}) = exp(\sum_k w_k f_k(x))$.
A feature may be any real-valued function of the state. It is an indicator of the clique configuration. There is one feature corresponding to each possible state $x_{\{k\}}$ of each clique. Its weight corresponds to the logarithm of the corresponding clique factor with: $w_k = log\varphi(x_{\{k\}})$

2.4.2 Parameter learning

Parameter learning task consists in estimating the clique potentials, or feature weights, given the clique templates. In another world, it consists of computing the weights $\mathcal{W}$ for the potentials φ given a training set D. Weight learning are generally divided into generative and discriminative learning.

2.4.2.1 Generative weight learning

The goal of generative weight learning is to optimize the joint probability distribution of all the variables. Weights can be generatively learned by selecting feature weights that maximize the likelihood (ML)of the data as the objective function.

Consider an MN in log-linear form given by the Equation 2.20, maximizing the likelihood consists in computing:

$$\frac{\partial}{\partial w_k} log P_w(x) = n_k(x) - E_w\left[n_k(x)\right] \tag{2.21}$$

The first term $n_k(x)$ corresponds to the number of times feature k is true in data. The second term is the expected number of times feature k is true according to the model. Its computing requires inference at each step.
The derivative of the log partition function is called the expectation of the kth feature under the model. This function is convex, so it has a unique global maximum which can be found using gradient-based optimizers (Kinderman and Snell, 1980).

In a Markov network, log likelihood is a convex function of the weights, and thus weight learning can be posed as a convex optimization problem (Kinderman and Snell, 1980). However, this optimization typically requires evaluating the log likelihood, and this must be done once per gradient step. This is typically intractable to compute exactly due to the partition function. Therefore, computing the Pseudo-Likelihood (PL) was widely used instead, because it is a consistent estimator that does not require inference at each step. The PL is the product of the likelihood of each variable given its Markov blanket (its neighbors in the network) in the data. It is expressed by:

$$PL \equiv \prod_k P(x_k \mid MB(x_i)) \tag{2.22}$$

However, PL method may not work well for long inference chains. This makes MN training much slower than BN training. Thus discriminative learning was proposed using Bayesian approaches.

2.4.2.2 Discriminative weight learning

This task consists of computing the discriminative likelihood of the data $\prod_d P_w(y|x)$ assuming a prior over the weights. In other words, it consists of maximizing the Conditional Likelihood (CLL) of query y given an evidence x.

$$\frac{\partial}{\partial w_k} log P_w(y \mid x) = n_k(x, y) - E_w\left[n_k(x, y)\right] \tag{2.23}$$

where, the first term is the number of times feature k is true in data. The second term is the expected number of true groundings according to model. Computing the expected counts $E_w\left[n_k(x, y)\right]$ is intractable. However, they can be approximated by the counts $n_k(x, y_w^*)$ in the MAP state $y_w^*(x)$. Computing the gradient of the CLL now requires only MAP inference to find $y_w^*(x)$, which is much faster than the full conditional inference for $E_w\left[n_k(x, y)\right]$. This approach was used successfully by (Collins, 2002).

2.4.3 Structure learning

MLN structure learning is the task of estimating the graph structure of the graphical model from i.i.d. samples. As we mentioned in Section 2.3.3 existing work falls into two categories. The first category involves methods called "constraint based approaches" which use hypothesis testing to estimate the set of conditional independencies in the data and then determine a graph that most closely represents those independencies. Second categories involve score based approaches which involve two components:a score metric, which is typically a sum of goodness of fit measure of the graph to the data, and a search procedure which searches through the candidate space of graphs with the goal to find the graph with the highest score. However, in the case of MNs this category of approaches is intractable due to the computation of score metric of each candidate which involves first learning the optimal weights for it. Weight learning requires inference which involves computing the normalization constant. Thus the estimation of graph structures in MNs has restricted to simple graph classes such as trees (Chow and Liu, 1968), polytrees (Dasgupta, 1999)

and bounded tree-width hypertrees (Srebro, 2003). (Pietra et al., 1997) propose a standard approach which performs a top-down search. At each step, it creates candidate features by conjoining it to the variable or feature that most improves the score. It calculates the weight for each candidate feature by assuming that all other feature weights remain unchanged. It uses Gibbs sampling for inference when setting the weights.

Several other algorithms exist that perform top-down heuristic search through the space of candidate structures such as the method proposed by (McCallum, 2003). However, top-down search has the disadvantage that it tests many feature variations with no support in the data, and is highly prone to local optima.

An alternative approach is proposed by (Mihalkova and Mooney, 2007). It performs a bottom-up search. The algorithm BLM (Bottom-up Learning of Markov Networks) starts with each complete training example as a long feature and repeatedly generalizes a feature to match its k nearest examples by dropping variables.

Another class of algorithm combines parameter learning and feature induction into one step through L1 regularization (Wainwright et al., 2006). It learns the structure by trying to discover the Markov blanket of each variable. However, these approaches still require long run times for domains that have a large number of variables. Recently, (Lowd and Davis, 2014) proposed DTSL (Decision Tree Structure Learner) algorithm which is inspired from (Wainwright et al., 2006) algorithm by substituting a probabilistic decision tree learner of L1 logistic regression. Other recent methods are proposed such as (Park et al., 2017) which is a scalable method of learning the graphical structure across the variables by solving a regularized approximated maximum likelihood problem.

2.5 Inference in PGMs

BN and MN differ regarding their relationship between graph separation and conditional independence. These differences are important in learning their structures because one is directed, the other is undirected and in learning their parameters because of the family of joint probability distribution that a particular graph can represent. Or, in the inference task, we generally have a specific fixed joint probability distribution, in which case the difference between BN and MN are less important. Indeed, same inference methods can be applied to BN and MN. Inference process looks for the impact of certain information regarding some variables, known as evidence, on the remaining ones. In the inference task, there are two kinds of queries we wish to answer. The first one is the standard conditional probability query called *marginal query* :

$$P(Y \mid E = e) = \frac{P(Y, e)}{P(e)} \tag{2.24}$$

This query consists of two parts: the evidence, a subset E of random variables and an instantiation e to these variables called by evidence; and the query: a subset Y of random variables in the network. The process consists to look for the impact of a certain information regarding some evidence, on the remaining ones Y. The second type of queries is to find the most probable assignment to a subset of variables given an evidence $E = e$. This type called *Maximum A Posteriori query* (MAP) defined by:

$$P^*(Y \mid E = e) = argmax_y P(y|e) \tag{2.25}$$

A graphical model is used to answer these queries. After generating the joint distribution, the joint is summed out in the case of a conditional probability query, and the entry that maximises the probability is looked for in the case of MAP inference.

A panoply of algorithms has been proposed to perform exact inference (propagation) with PGMs. The first algorithms are based on message passing architecture and were proposed by (Pearl, 1982), and (Kim and Pearl, 1983) . An extension of these algorithms, that are limited to trees is found in several works like

the method of join-tree propagation (Lauritzen and Spiegelhalter, 1988) and the method of Belief Propagation cut-set conditioning (Yedidia et al., 2000) that provide propagation in general networks.

However, in large domains with a massive number of nodes, densely connected networks, exact inference quickly becomes intractable and approximate inference is necessary.

A set of approximate algorithms was proposed to be used in such case (Jensen and Nielsen, 2007), (Xiang, 2010) and (Darwiche, 2013).

2.6 Bayesian Networks vs Markov Networks

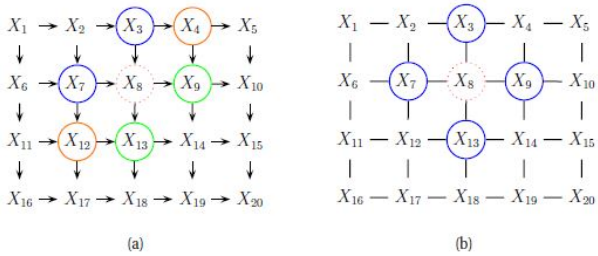


Figure 2.6 – (a) A DAG; The dotted red node X_8 is independent of all other nodes (black) given its Markov blanket, which includes its parents (blue), children (green) and co-parents (orange). (b) The same model represented as a MN; The red node X_8 is independent of the other black nodes given its neighbors (blue nodes) (Koller and Friedman, 2009).

When we compare BNs and MNs, in reality, we compare Directed Graph Models (DGM) and Undirected Graph Models (UGM) respectively. Both are probabilistic models that allow representing different sets of probabilistic distributions. This means, no one is more powerful than the other as a representation language.

In fact, the family of joint probability distributions associated with a given graph, either direct on undirected, can be expressed as a product of the potential functions associated with subsets of nodes. For BN, this subset is a single node and its corresponding parents. For MN, this subset is each clique of the graph. Given the Equation 2.4 and Equation 2.19, the first equation can be viewed as a special case of the second one if (1) a normalizing factor Z is included in Equation 2.4 with Z=1. (2) $P(X_i|Pa(X_i))$ is made as a potential function by connecting all parents of each node, because the parent of a node in a BN is not necessarily connected. Thus we obtain the moral graph G^m. So that, $P(X_i|Pa(X_i))$ is a potential function and Equation 2.4 is a special case of Equation 2.19. However, some distributions can be perfectly modeled by either a BNs or a MNs.

For example, consider modeling an image as it is shown in Figure 2.6 (Domke et al., 2008). We suppose that the intensity values of neighboring pixels are correlated, if we represent this case using a DAG as shown in Figure 2.6(a). We have to choose a direction for the edges, as required by a BN. In this case, the Markov blanket of the node X_8 in the middle is the other colored nodes (3, 4, 7, 9, 12 and 13) rather than just its 4 nearest neighbors.

An alternative is to use a MN. These do not require us to specify edge orientations and are much more natural for some problems such as image analysis. For example, in Figure 2.6 (b) now the Markov blanket of each node is just its nearest neighbors.

The main advantage of MN over BN that they are symmetric (there is no orientation) and therefore more useful for specific domains, such as spatial or relational data. Thus, conditional independence relationships in MNs is much easier than in BNs, because we do not have to worry about the directionality of the edges.

However, comparing to BNs, the parameter estimation of MNs are less interpretable and less modular, for reasons we explained in Section 2.4.2.

2.7 Conclusion

In this chapter, we introduced basic definitions and concepts related to Bayesian Networks and Markov Networks, including their representation, learning algorithms, and inference algorithms. BNs and MNs are generally learned from "flat" data set in which data consists of identically structured entities, typically assumed to be IID (Independent and Identically Distributed). However, many real-world data sets are relational. Such data consists of entities of different types, where each entity type is characterized by a different set of attributes. Entities are related to each other via different types of relationships, and relationships are an important source of information. Thus, Probabilistic Relational Models such as Directed Acyclic Probabilistic Entity Relationship and Markov Logic Networks were defined to extend PGM for relational data.

Chapter 3 will introduce Probabilistic Relational Models in the context of relational data representation, namely Directed Acyclic Probabilistic Entity Relationship and Markov Logic Networks.

3

Probabilistic Relational Models

Contents

3.1 Introduction **32**
3.2 Basic concepts **32**
3.2.1 Entity-Relationship model 33
3.2.2 First-Order-Logic 35
3.3 Direct Acyclic Probabilistic Entity Relationship models **36**
3.3.1 DAPERs formalism 36
3.3.2 Extensions: DAPERs with structural uncertainty 39
3.3.3 Parameter learning 39
3.3.4 Structure learning 39
3.4 Markov Logic Networks **40**
3.4.1 MLN formalism 40
3.4.2 Parameters learning 43
3.4.3 Structure learning 44
3.5 Computing sufficient statistics for PRMs learning **46**
3.5.1 Definitions 46
3.5.2 SQL implementation for computing contingency table 47
3.6 Inference in PRMs **48**
3.7 Conclusion **48**

3.1 Introduction

The relational representation is strictly more expressive than propositional representation assumed by almost all machine learning techniques such as Bayesian Networks and Markov Networks. The relational model principles where originally laid down by (Codd, 1970) who realized that simple propositional datasets cannot represent the majority of real-world situations. Indeed, real-world data usually come as sets of related objects which could be stored in multi-tables. As objects are related to each other, the IID assumption is often violated in relational data. This limits the application of BN and MN learning algorithms on relational data. Consequently, Relational Statistical Learning (SRL) has been emerged.

SRL (Neville and Jensen, 2005; Taskar et al., 2007) combines the descriptive power of relational learning with the flexibility of statistical learning to develop learning models and algorithms capable of representing complex relationships among entities in uncertain domains. In fact, relational learning aims to deal with the complexity of relational data, whether sequential, graphic, multi-relational, or other. Statistical learning makes it possible to take into account the notion of uncertainty using probability theory. However, many real cases involve both complex and uncertain data.

During the last decade, various approaches have been proposed to deal with relational and statistical learning. Probabilistic Relational Models (PRMs) are one of the most used models to handle uncertainty in real-world applications. PRMs present an extension of PGMs in the relational context, such as Relational Bayesian Networks (RBN) (Koller and Pfeffer, 1998; Friedman et al., 1999a), Directed Acyclic Probabilistic Entity-Relationships models (DAPER) (Heckerman and Meek, 2004) and Markov Logic Networks (MLN) (Richardson and Domingos, 2006).

DAPER models (Heckerman and Meek, 2004) are a kind of PRMS which generalize BNs to work with relational database representation rather than propositional data representation. MLNs (Richardson and Domingos, 2006) are also one of the most recent approaches to relational statistical learning. They generalize both First-Order-Logic (FOL) (Smullyan, 1968) and MNs to work with relational data represented in the knowledge database. PRMs are interested in producing and manipulating structured representations of the data, involving objects described by attributes and participating in relationships, actions, and events. The probability model specification concerns classes of objects rather than simple attributes. DAPERs and MLNs learning and inference are inspired by standard BNs and MNs learning approaches respectively. This chapter is dedicated to introduce DAPER and MLN frameworks and to make a survey on existing learning as well as inference approaches.

The remainder of this chapter is organized as follows: we recall some basic concepts from Entity-Relationship (ER) model and First-Order-Logic (FOL) in Section 3.2. Sections 3.3 and 3.4 define DAPER and MLN frameworks respectively. Existing learning DAPER and MLN parameters are introduced in Sections 3.3.3 and 3.4.2. Sections 3.3.4 and 3.4.3 present learning DAPER and MLN structure. In Section 3.5, we introduce the process of computing sufficient statistics from relational databases which are used for PRMs learning. Section 3.6 presents a list of inference approaches which can be used for both DAPER and MLN models.

3.2 Basic concepts

As the relational data representation is widely different from the propositional data representation assumed by BNs and MNs, we devote this section to present some useful concepts related to Entity-Relationship model derived from (Date, 2003; Date, 2008) and First-Order-Logic derived from (Raedt and Kersting, 2008; Dinh, 2011; Domingos and Webb, 2012).

3.2.1 Entity-Relationship model

Definition 3.2.1. ***Entity class***
An entity class $\mathcal{E} = \{E_1, ..., E_m\}$: a set of entity classes which correspond to a relation schema in a relational database.

Definition 3.2.2. ***Relationship class***
A relationship class $\mathcal{R} = \{R_1, ..., R_n\}$: a set of relationship which correspond to a specific interaction among entity classes.

Let us consider only binary relationship classes. Each binary relationship class link a couple of entity classes.

Definition 3.2.3. ***Attribute class***
$\mathcal{A}(X)$: attribute classes for each $X \in (\mathcal{E} \cup \mathcal{R})$ which correspond to variables describing some properties of an entity or relationship. Each attribute $x.A$ has domain $V(x.A)$ of possible values.

Definition 3.2.4. ***Restricted Relationships class***
A relationship class $\mathcal{R}$ in an ER model is restricted when some skeletons for the entity and relationship classes of the ER model are prohibited.

Definition 3.2.5. ***Cardinality***
Relationships are classified into four main types, namely; 1) One-to-Many relation, 2) Many-to-One, 3) One-to-One relation and 4) Many-to-Many relation.

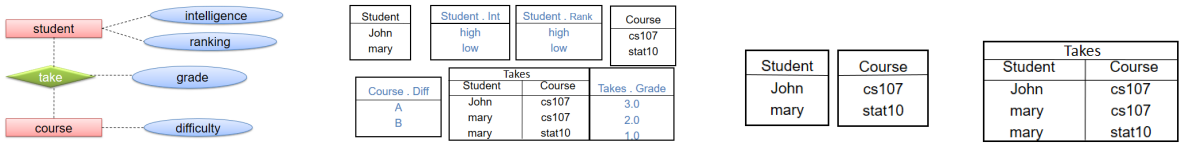
- *One-to-Many relationship type. It associates one entity with more than one entity. For instance, an order may contain many items. while an item belongs to only one order.*
- *Many-to-One relationship type. It associates more than one entity to just one entity. As relationships work both ways, the same example of Order and Item presents an example of Many-to-One relationship: There may be many items in a particular order.*
- *One-to-One relationship type. This relationship type is rarely met. It associates one entity to exactly one entity, and it is considered as a special case of a One-to-Many relationship. As an example of a One-to-One relationship is the relationship between a person and a passport in a particular country. A person has only one passport, a particular passport is delivered to only one person.*
- *Many-to-Many Relationship Type. It associates more than one entity with more than one entity. For instance, a student can take many courses, and a course can have many students.*

Definition 3.2.6. ***ER diagram***
ER diagram is a graphical representation of entities and their relationships to each other. In ER diagram, the entities classes are shown as rectangular nodes, the relationship classes are shown as a diamond-shaped node, and the attribute classes are represented using ellipses.

Definition 3.2.7. ***ER model***
An ER model is a collection of named entity classes linked by relationship classes.

Definition 3.2.8. ***Skeleton***
*A **skeleton** $\sigma_{\mathcal{ER}}$ is a collection of corresponding entity and relationship sets defined by $\sigma_{\mathcal{E}} \cup \sigma_{\mathcal{R}}$.*

Definition 3.2.9. ***Instance***
An instance of an ER model is $\mathcal{I}_{\mathcal{ERA}}$, is defined by a skeleton $\sigma_{\mathcal{ER}}$ and an assignment a valid value in $\mathcal{V}(X.A)$ to every attribute $x.A$ where $x \in \sigma_{\mathcal{ER}}$ and $A \in \mathcal{A}(X)$.



(a) An example of ER diagram.

(b) An example of instance $\mathcal{I}$.

(c) An example of skeleton $\sigma_{\mathcal{ER}}$.

Figure 3.1 – An example of (a)ER diagram, (b) instance $\mathcal{I}$ and (c)skeleton $\sigma_{\mathcal{ER}}$ for the university domain.

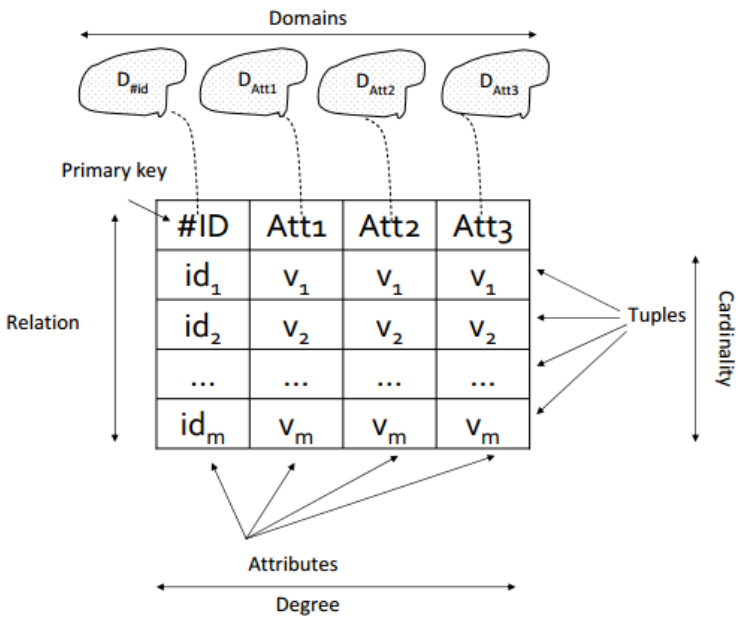


Figure 3.2 – Relation components

Example 3.2.1. *Figure 3.1(a) illustrates an example of ER diagram designed for a university model which holds information relating to Students and Student Courses that they take.*

In this example we distinguish between $\mathcal{E}$={student, course}, $\mathcal{R}$={ takes} and A(student)={student.intelligence, student.ranking}, A(course)={course.difficulty}, A(takes)={takes.grade}.

The Entity-Relationship model provides the conceptual level of relational database design.

Definition 3.2.10. ***Relational database***
A relational database is a set of relations R, with each R consists of a heading and a body:

— *The heading (=relation schema) consists of a fixed set of attributes associated to a set of domains, or more precisely $\langle$ attribute_name : domain_name $\rangle$ pairs $(A1 : D1, ..., \langle A_n : D_n \rangle)$. A subset K of the heading is a primary key of a relation R.*

— *The body consists of a set of tuples. Each tuple consists of a set of $\langle$ attribute _ name : attribute _ value $\rangle$ pairs $(A1 : Vvi_1, ...; \langle A_n : vi_n \rangle)$, where $(i = 1, ..., m)$ and m is the number of tuples in the set. In each such tuple, there is one such $\langle$ attribute_name : attribute_value $\rangle$ pair $\langle A_j; v_i j \rangle$ for each attribute A_j in the heading.*

Example 3.2.2. *Figure 3.2 summarizes all the concepts related to a relation. The relation heading is defined through 4 attributes, each of which has its one domain. The relation body is a set of m tuples.*

A relational database maintains the following properties of transactions; Atomicity, Consistency, Isolation and Durability (ACID) (Liu and Özsu, 2009; Khachana et al., 2011)

— Atomicity : All operations are performed, or none of them are. If one part of the transaction fails, then all fail.

— Consistency : Execution of transaction results in a consistent database.

— Isolation : Each transaction is independent unto itself.

— Durability : Persistence, means that once complete, the transaction cannot be undone.

3.2.2 First-Order-Logic

First Order Logic is a system of representing knowledge in deductive systems. The syntax of first-order logic is composed of user-defined term, predicates and formulas. We present in this section basic definitions in first-order logic derived from (Raedt and Kersting, 2008; Dinh, 2011; Domingos and Webb, 2012).

Definition 3.2.11. ***Variable***
A variable is a container that holds information about data, often denoted by lowercase letters.

Definition 3.2.12. ***Constant***
A constant is an object in the domain of interest (e.g., people: Anna,Bob, Chris, etc.).

Definition 3.2.13. ***Function***
A function represents a mapping from tuples of objects to objects (e.g., MotherOf).

Definition 3.2.14. ***Term***
A term is an expression representing an object in the domain. It can be a constant, a variable, or a function applied to a tuple of terms.

Example 3.2.3. *Anna, x, and GreatestCommonDivisor(x, y) are terms.*

Definition 3.2.15. ***Ground term***
A ground term is a term containing no variables.

Definition 3.2.16. ***Predicate***
A predicate represents relations among objects in the domain.

Definition 3.2.17. ***Atomic formula***
An atomic formula or atom is a predicate symbol applied to a tuple of terms.

— *A positive literal: is an atomic formula.*

— *A negative literal: is a negated atomic formula.*

Example 3.2.4. *Friends*$(x, MotherOf(Anna))$ *is an atom.*

Definition 3.2.18. ***Ground atom***
A ground atom or a ground predicate is an atomic formula, such that all of its arguments are ground terms.

Definition 3.2.19. ***Formulas***
Formulas are recursively constructed from atomic formulas using logical connectives and quantifiers.
Formulas are constructed using four types of symbols: constants which represent objects in the domain of interest (e.g., people: Anna,Bob, Chris, etc.), variables, predicates represent relations among objects in the domain (e.g., Friends), functions represent mappings from tuples of objects to objects (e.g., MotherOf).
A formula is satisfiable iff there exists at least one argument in which it is true.

Example 3.2.5. *Consider a formula Smoke(X) ⇒ Cancer(X) in which X is a variable, Smoke and Cancer are two predicate symbols, Smoke(X) and Cancer(X) are two variable literals.*

First-Order Logic	Clausal Form
"Friends of friends are friends."	
$\forall x \forall y \forall z\ \mathtt{Fr}(x,y) \wedge \mathtt{Fr}(y,z) \Rightarrow \mathtt{Fr}(x,z)$	$\neg\mathtt{Fr}(x,y) \vee \neg\mathtt{Fr}(y,z) \vee \mathtt{Fr}(x,z)$
"Friendless people smoke."	
$\forall x\ (\neg(\exists y\ \mathtt{Fr}(x,y)) \Rightarrow \mathtt{Sm}(x))$	$\mathtt{Fr}(x,g(x)) \vee \mathtt{Sm}(x)$
"Smoking causes cancer."	
$\forall x\ \mathtt{Sm}(x) \Rightarrow \mathtt{Ca}(x)$	$\neg\mathtt{Sm}(x) \vee \mathtt{Ca}(x)$
"If two people are friends, then either both smoke or neither does."	$\neg\mathtt{Fr}(x,y) \vee \mathtt{Sm}(x) \vee \neg\mathtt{Sm}(y)$,
$\forall x \forall y\ \mathtt{Fr}(x,y) \Rightarrow (\mathtt{Sm}(x) \Leftrightarrow \mathtt{Sm}(y))$	$\neg\mathtt{Fr}(x,y) \vee \neg\mathtt{Sm}(x) \vee \mathtt{Sm}(y)$

Table 3.1 – Example of a first-order knowledge base. Fr() is short for Friends(), Sm() for Smokes(), and Ca() for Cancer()

If F1 and F2 are formulas, the following are also formulas:

— $\neg F_1$(negation), which is true iff F_1 is false

— $F_1 \wedge F_2$ (conjunction), which is true iff both F_1 and F_2 are true

— $F_1 \vee F_2$ (disjunction), which is true iff F_1 or F_2 is true.

— $F_1 \Rightarrow F_2$ (implication), which is true iff F_1 is false, or F_2 is true.

— $F_1 \Leftrightarrow F_2$ (equivalence), which is true iff F_1 and F_2 have the same truth value.

— $\forall x F_1$ (universal quantification), which is true iff F_1 is true for every object x in the domain.

— $\forall x F_1$ (existential quantification), which is true iff F_1 is true for at least one object x in the domain.

Definition 3.2.20. ***First-order knowledge base***
A first-order knowledge base (KB) is a set of sentences or formulas in first-order logic. Generally, formulas in KB are converted to a more regular form, typically clausal form (also known as Conjunctive Normal Form (CNF)). A KB in clausal form is a conjunction of clauses, with a clause is a disjunction of atoms.

Example 3.2.6. *Table 3.1 shows a simple KB and its conversion to clausal form.*

3.3 Direct Acyclic Probabilistic Entity Relationship models

In this section, we introduce the Directed Acyclic Entity-Relationship (DAPER) model in detail. Originally introduced by (Heckerman and Meek, 2004), the DAPER model is semantically equivalent to the Probabilistic Relational Model (PRM) introduced by (Friedman et al., 1999a) called also RBN by (Neville and Jensen, 2005) who proposed to preserve the use of the term PRM in its more general sense to distinguish the family of PGMs that are interested in extracting statistical patterns from relational models, and to use RBN as an extension of BN used to model relational databases.

3.3.1 DAPERs formalism

DAPER (Heckerman and Meek, 2004) is a relational extension of BN. It is defined based on an ER model and its instantiation, where attribute classes are random variable classes. The probabilistic dependencies are defined between variable classes rather than on the level of instantiations.

3.3.1.1 DAPERs definition

Let ER be an Entity Relationship model, $\mathcal{E}$ be the set of Entity classes, $\mathcal{R}$ be the set of relationship classes and $\mathcal{A}(X)$ where each attribute A is associated with a class $X \in \mathcal{X} \equiv (\mathcal{E} \cup R)$. The relational structure can be defined using a set of reference slots $R(X)$ of class $X \in \mathcal{X} \equiv (\mathcal{E} \cup \mathcal{R})$ (Friedman et al., 1999a) or a set of constraints C_{AB} (Heckerman and Meek, 2004).

Definition 3.3.1. ***Reference slot***
A reference slot of class X denoted by X.ρ has X as domain type and Y as a range type, where $Y \in \mathcal{X}$. For each reference slot ρ , a reversed slot ρ^{-1} can be defined. In the context of relational databases, a reference slot leads to expression regarding the foreign keys of the entity/relationship classes that connect the parent and child attribute.

Definition 3.3.2. ***Slot chain***
A slot chain K is a sequence of (reversed or not) reference slots. $\rho_1, ..., \rho_k$, where $\forall i$, Range $[\rho_i]$=Dom $[\rho_{i+1}]$.

Definition 3.3.3. ***Constraint***
A constraint is a generalization of slot chain. C_{AB} is a first-order-expression that define a subset of $\sigma_{\mathcal{ER}}(X.A) \times \sigma_{\mathcal{ER}}(Y.B)$

Note that a constraint can be multiple paths connecting two attributes. Or, a slot chain is limited to one path. It is possible for an attribute instance $x.A$ to have multiple parents for a given dependency. This means that the local distribution is shared among all objects of attribute $X.A$. To describe the dependency of an attribute with an unknown number of parents, (Zhang and Poole, 1996) proposed to use aggregate functions.

Definition 3.3.4. ***Aggregate function***
An aggregate function γ takes a multi-set of values of some ground type and returns a summary of it.
If a dependency $X.A \leftarrow Y.B$ applied to $x.A$ returns a multiset $\{y.B\}$, then the function $\gamma : \{y.B\} \rightarrow y.B_\gamma$ returns a single valued summary of the multiset.

Example 3.3.1. *Examples of aggregate functions: maximum, minimum, mode, average, number of rows, etc.*

Definition 3.3.5. ***Direct Acyclic Probabilistic Entity Relationship model***
A DAPER Π associated with ER is composed of:

- *a qualitative dependency structure S with:*
  - *set of random variables and set of probabilistic dependencies among them.*
  - *possible slot chains/constraints and aggregation functions*
- *a quantitative component: a set of parameters θ_S representing the set of probability distributions $P(X.A \mid Pa(X.A))$.*

Example 3.3.2. *Figure 3.3 shows a DAPER model related to the ER model in Figure 3.1 where a student's grade depends both on the student's intelligence and on the difficulty of the course. Red solid arcs represent the probabilistic dependencies between attributes. An example slot chain in Figure 3.3 $[takes.student].[takes.student]^{-1}.[takes.course]$ which could be interpreted as all courses taken by a particular student. In Figure 3.3, stud.ranking depends probabilistically on takes.grade. As a student can take more than one course, stud.ranking will depend on takes.grade of more than one takes an object, and this number will not be the same for all students. So, to get a summary of such dependencies, an aggregator **MODE** is used.*

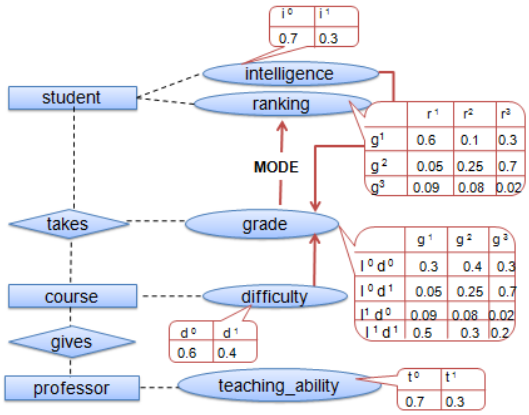


Figure 3.3 – An example of DAPER model which corresponds to the ER model and Instance given in Figure 3.1. Slot chains are not represented in order to simplify the figure.

DAPERs define a distribution over instantiations of the database that are consistent with the skeleton. Instantiating a DAPER for a particular skeleton consists of generating a Ground Bayesian Network (GBN).

Definition 3.3.6. ***Ground Bayesian Network (GBN)*** *(Taskar et al., 2007)*
Giving a DAPER and a skeleton $\sigma_{\mathcal{ER}}$, we can define a Ground Bayesian Network as follow:

- *A qualitative component:*
  - *A node is created for every attribute instance of every object $x.A \in \mathcal{A}(\sigma_{\mathcal{ER}})$.*
  - *an arc is added to every pair $x.A$ and $y.B$ (or $y.K.B$), $y.B \in \mathcal{A}(\sigma_{\mathcal{ER}})$, if $x.A$ depends probabilistically on parents $y.B$ or $y.K.B$. If K is not single valued, then the parent is the aggregate computed from the set of random variable y, $\gamma(y.K.B)$.*
- *A quantitative component: CPD for each $x.A$ is $P(X.A \mid Pa(X.A))$.*

Like a standard Bayesian network the joint distribution over the instantiations of a DAPER Π for a skeleton $\sigma_{\mathcal{ER}}$, is as follow:

$$P(\mathcal{I} \mid \sigma_{\mathcal{ER}}, \Pi) = \prod_{X \in \mathcal{X}} \prod_{A \in \mathcal{A}(X)} \prod_{x \in \sigma_{\mathcal{ER}}} P(x.A \mid Pa(x.A)) \tag{3.1}$$

The joint distribution must be coherent, i.e., the sum of the probability of all instances is 1. This requirement is satisfied if the dependency structure S of a DAPER is acyclic relative to a given skeleton. (Pfeffer and Koller, 2000) introduces the class dependency graph notion to ensure that the probabilistic dependencies defined by the RBN are acyclic.

Definition 3.3.7. ***Class dependency graph (CDG)***
The class dependency graph G_Π for a DAPER Π has a node for each descriptive attribute $X.A$, and the following edges:

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

So that, if CDG is acyclic then, the corresponding DAPER is guaranteed to be coherent.

3.3.2 Extensions: DAPERs with structural uncertainty

Regular DAPER models provide a model for domains where attribute values are uncertain. In these models, all relations between attributes are known, uncertainly exists in the descriptive attributes only. (Friedman et al., 1999a) have extended regular PRM to deal with the cases where both attributes and link structure are uncertain modeling more complex structural uncertainty namely: reference uncertainty and existence uncertainty.

PRM with Reference Uncertainty (PRM-RU) It models uncertainty over the value of reference slots. This model assumes that a partial skeleton called object skeleton is given instead of a full one which specifies only the objects in each class X but not the values of the reference slots.

PRM with Existence Uncertainty (PRM-EU) This model deals with the task of predicting links between objects. In other words, it gives entity classes, and the existence of the objects of some relationship classes is uncertain.

Like in Bayesian Networks, the two main tasks concerning DAPERs are learning and inference. Learning can be divided into structure learning and parameter learning. Structure learning consists of finding a set of probabilistic dependencies from data while parameter learning focuses on computing probability distributions from data. We detail these tasks in the following subsections.

3.3.3 Parameter learning

Given the dependency structure S, parameters learning consists of providing θ_S. If all variables have been observed, i.e., there are no latent variables and no missing data, the parameters can be learned using statistical approach; namely the maximum likelihood estimation (MLE) or Bayesian approach. The likelihood function which computes the probability of the data given the model structure (Getoor et al., 2007).

$$L(\theta_S, \sigma_{\mathcal{ER}} \mid \mathcal{I}_{\mathcal{ERA}}) = P(\mathcal{I}_{\mathcal{ERA}} \mid \theta_S, \sigma_{\mathcal{ER}}) \tag{3.2}$$

Maximizing the likelihood consists of computing the parameters that best explain the data according to our model. It is the same likelihood function of standard BNs given by Equation 2.5. The main difference is that we can find nodes having the same probability distributions induced by the DAPER given the skeleton.

3.3.4 Structure learning

The learning task for DAPER models aims at identifying the probabilistic dependencies between attributes given an ER model and its instantiation $\mathcal{I}_{\mathcal{ERA}}$. Like in the Bayesian network, structure learning is devised into three classes: constraint-based algorithms, score-base algorithms and hybrid algorithms.

(Maier et al., 2010) proposed the Relational PC, that learns causal dependencies from relational data. RPC is an extension of the PC algorithm for propositional data (Spirtes et al., 2000). RPC is a constraint-based algorithm for learning causal models of relational data, and it can be used alone or as a component of a hybrid algorithm. (Maier et al., 2013b) extended the theory of d-separation for DAPER models and proposed an intermediate representation, the abstract ground graph, and an adaptation of a usual constraint-based learning algorithm using conditional independence facts measured in the data.

(Friedman et al., 1999a) proposed Relational Greedy Search (RGS) algorithm which is a score-based algorithm. RGS determines the neighboring structure of the starting network. The procedure for generating neighboring is based on 'Add_edge', 'Delete_edge' and 'Revert_edge' operations. Then, it assigns a score to each of them, selects the best scoring one, and iterates through the same process of searching for neighbors and scoring them until a stopping criterion is met. (Friedman et al., 1999a) proposed walking through

the slot chains to discover potential parents for each attribute and applying the search procedure on this set of parents. The search for potential parents and the corresponding structure begins with the slot chains of length 0 which is increased by 1 at each phase until a predefined slot chain length limit is reached or there is no improvement in the DAPER structure.

Algorithm 1 lists their overall approach of learning the structure of DAPERs, and Algorithm 2 details the procedure for generating neighboring DAPER structures.

Algorithm 1 Relational Greedy Search (Friedman et al., 1999a)

Data: Initial dependency graph G
ER schema ER
Scoring function Score
Maximum slot chain length K_{max}
Result: Local optimal dependency graph G'
begin
  $G' \longleftarrow G$
  $S_{max} \longleftarrow Score(G')$
  $SL \longleftarrow 0$
  repeat
    repeat
      $\mathcal{N} \longleftarrow$ Generate_Neighbors(G,ER,SL)
      $N^* \longleftarrow \text{argmax}_{N' \in \mathcal{N}} Score(N')$
      $S^* \longleftarrow Score(N^*)$
      if $S^* > S_{max}$ **then**
        $G' \longleftarrow N^*$
        $S_{max} \longleftarrow S^*$
    until *No change in* G'
    $SL \longleftarrow SL + 1$
  until $SL > K_{max}$

(Ben Ishak, 2015) have proposed Relational Max-Min Hill Climbing (RMMHC) which is a hybrid method. RMMHC is an adaptation of MMHC algorithm (Tsamardinos et al., 2006).

3.4 Markov Logic Networks

Initially introduced by (Kok and Domingos, 2005; Richardson and Domingos, 2006), the MLN model is a kind of PRM that generalizes both full first-order logic and Markov networks. A MLN consists of a set of weighted clauses in which the syntax of formulas follows the standard syntax of first-order logic and weights are real numbers. In this section, we introduce the Markov Logic Network (MLN) model in detail.

3.4.1 MLN formalism

In a first-order knowledge base, each formula is a hard constraint on the state of the possible worlds. A formula is false if there exists a world that violates it (the probability of such formula becomes 0). The key idea of MLNs is to try to soften these constraints. Thus, when a world violates one formula in the KB, this formula becomes less probable, but not impossible. By consequence, its probability does not become 0. It is just less likely to occur than one that is not violated by any world. The fewer formulas a world violates, the more probable it is. Each formula has an associated weight that reflects how strong a constraint it is.

Definition 3.4.1. ***Markov Logic Network*** *(Richardson and Domingos, 2006)*
A Markov logic network L is a set of pairs (F_i, ω_i)*, where* F_i *is a formula in first-order logic and* ω_i *is a*

Algorithm 2 Generate_Neighbors

Data: A DAPER $\Pi =< ER, S >$
ER schema: ER
Available aggregators: Agg
Slot chain length: SL
Result: Neighhbors of S: $S^{'}$

```
begin
    S' ⟵ {}
    for each X.A ∈ 𝒜(X) and X ∈ ER do
        P ⟵ Find_Accessible_Classes(X.A, ER, SL)
        for each < Y, ρ > ∈ P and Y.B ∈ 𝒜(Y) do
            if X.ρ.B, X.A ∉ S then
                if ρ contains at least one reverse slot then
                    for γ ∈ AGG(Y.B) do
                        S'' ⟵ S ∪ {(γ(X.ρ.B), X.A)}
                        if S'' has no cycle then
                            S' ⟵ S' ∪ S''
                else
                    S'' ⟵ S ∪ {(X.ρ.B), X.A)}
                    if S'' has no cycle then
                        S' ⟵ S' ∪ S''
            else
                S' ⟵ S' ∪ S\{(X.ρ.B), X.A)}
                Y.ρ' ⟵ Reverse(X.ρ)
                if ρ' contains at least one reverse slot then
                    for γ ∈ AGG(X.A) do
                        S'' ⟵ S\{(X.ρ.B), X.A)} ∪ {γ(Y.ρ'.A, Y.B)}
                        if S'' has no cycle then
                            S' ⟵ S' ∪ S''
                else
                    S'' ⟵ S\{(X.ρ.B), X.A)} ∪ {(Y.ρ'.A, Y.B)}
                    if S'' has no cycle then
                        S' ⟵ S' ∪ S''
```

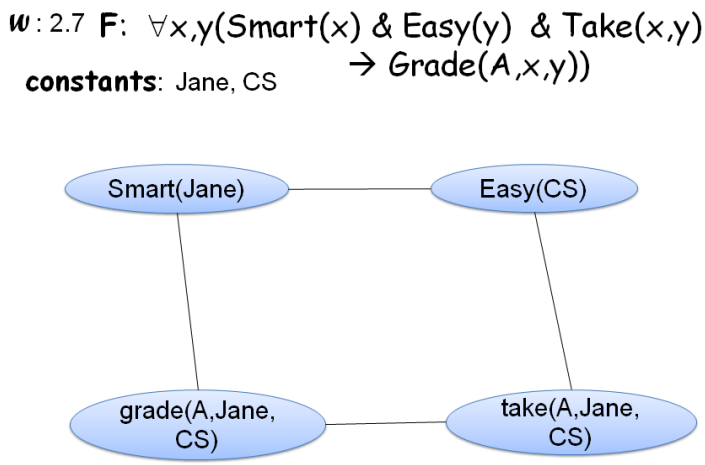


Figure 3.4 – An example of MLN for the university domain. The pairs of formulas and weights together with the set of constants construct the ground Markov network.

real number weight associated with the formula. The Markov logic network also contains a finite set of constants $C = \{c_1, c_2, ..., c_{|C|}\}$. The pairs of formulas and weights together with the set of constants are used to construct ground Markov networks as follow:

1. *$M_{L,C}$ contains one node for each possible grounding of each predicate appearing in L. The value of the node is 1 (true) if the ground predicate is true, and 0 (false) otherwise.*

2. *$M_{L,C}$ contains one feature for each possible grounding of each formula F_i in L. The value of this feature is 1 if the ground formula is true, and 0 otherwise. The weight of the feature is the weight ω_i associated with F_i in L. More precisely, each formula is first translated in its clausal form. When a formula decomposes into more than one clause, its weight is divided equally among the clauses.*

3. *If the ground predicates appear together in at least one grounding of one formula, this indicates that there is a dependency relationship between the predicates, and hence the Markov Network would have an edge to link these nodes. Therefore, each formula forms a clique in the ground Markov Network.*

From Equation 2.20, The probability distribution over a possible world x specified by the Markov network $M_{L,C}$ is given by:

$$P(X = x \mid M_{L,C}) = \frac{1}{Z} exp(\sum_{f_i \in F} \sum_{j \in G_i} \omega_j g_j(x)) \tag{3.3}$$

With Z is the normalization constant, F is the set of all first-order-formula in the MLN L, G_i and ω_i are respectively the set of groundings and weight of the ith first-order-formula. $g_j(x) = 1$ if the gth ground formula is true and $g_j(x) = 0$ otherwise.

Example 3.4.1. *Let us consider a university domain of four predicates Smart(x), Easy(y), Take(x,y) and Grade(A,x,y). Here is a possible MLN composed of one formula in this domain:*

$$F : \forall x, y Smart(x) \& \ Easy(y) \& \ Take(x, y) \rightarrow Grade(A, x, y)$$

If we also consider a set of two constants Jane and CS of type x and y respectively, we can replace variables x and y by one of these two constants, and thus we obtain a ground Markov network with four cliques shown in Figure 3.4.

Like in Markov Networks, the two main tasks concerning MLNs are learning and inference. Learning can be divided into structure learning and parameter (weight) learning; both generative and discriminative models can be applied to them. Structure learning consists of finding a set of weighted formulas from data while parameter learning focuses on computing weights for a given set of formulas. We detail these tasks in the following subsections.

3.4.2 Parameters learning

Given a set of formulas and a relational database, MLN parameter learning consists in finding formula weights that maximize either the likelihood or pseudo-likelihood measure for generative learning or either the conditional likelihood or max-margin measures for discriminative learning (Richardson and Domingos, 2006).

3.4.2.1 Generative weight learning

MLN weights can be learned generatively by maximizing the likelihood of a relational database. Generative approaches try to optimize the joint probability distribution given by Equation 3.3 of all the variables .

$$P(X = x \mid M_{L,C}) = \frac{1}{Z} exp(\sum_{f_i \in F} \sum_{j \in G_i} \omega_j g_j(x) = \frac{1}{Z} \sum_{f_i \in F} \omega_i n_i(x)) \tag{3.4}$$

where $n_i(x)$ is the number of true instantiations of F_i in x. It must be noted that the database is a set of entirely ground atoms in the domain, which is always difficult to collect. The derivative of the log-likelihood regarding its weight is:

$$\frac{\delta}{\delta \omega_i} log P_\omega(X = x) = n_i(x) - \sum_{x'} P_\omega(X = x') n_i(x') \tag{3.5}$$

where the sum is over all possible databases x', and $P_\omega(X = x')$ is $P(X = x')$ computed using the current weight vector $\omega = (\omega_1, ..., \omega_i, ...)$. In other words, the ith component of the gradient is simply the difference between the number of true groundings of the ith formula in data and its expectation according to the current model. However, as in Markov networks, this requires computing the expected number of true groundings $(n_i(x))$of each formula, which can take exponential time (Richardson and Domingos, 2006). Therefore, the pseudo-log-likelihood of data is widely used instead of the likelihood (Besag, 1975).

If x is a possible world (relational database) and x_l is the truth value of lth ground atom, the pseudo-log-likelihood of x given weights ω is:

$$log P^*(X = x) = log \prod_{i=1}^{n} P_\omega(X_l = x_l \mid MB_x(X_l)) \tag{3.6}$$

where $MB_x(X_l)$ is the state of X_l's Markov blanket in the data.

3.4.2.2 Discriminative weight learning

Discriminative learning can be applied when a priori on data is presented. Give a set of evidence atoms X and a set of query atoms Y, the discriminative weight learning consists of maximizing the Conditional Likelihood (CLL) of Y given X. The CLL of Y given X is defined by:

$$P(Y = y \mid X = x) = \frac{1}{Z_x} \sum_i \omega_i n_i(x, y) \tag{3.7}$$

where Z_x normalizes over all possible worlds consistent with the evidence x, and $n_i(x, y)$ is the number of true groundings of the ith formula in data.

As mentioned before, computing the expected counts is intractable. To resolve this problem, these expected counts are approximated by the counts $n_i(x, y^*_\omega)$ in the MAP state $y^*_\omega(x)$. So that, computing the gradient of the CLL will require only MAP inference to find $y^*_\omega(x)$, which is much faster than the full conditional inference.

3.4.3 Structure learning

Structure learning consists to identify a set of formulas or clauses. (Richardson and Domingos, 2006) have used inductive logic programming to learn an MLN structure. However, the results obtained by this method were not efficient enough due to the lack of ability to deal with uncertainty in ILP. Thus, new methods have been proposed. Generally, these methods are divided into two kinds: generative and discriminative structure learning (Lowd and Domingos, 2007).

3.4.3.1 Generative structure learning

Generative structure learning constructs an MLN from scratch with adding clause by clause to the MLN. Generally, this approach consists of assigning a score to each candidate clause. The weighted pseudo-log-likelihood (WPLL) is often used as a score to evaluate candidate clauses. Then, the clause that gives the highest WPLL is kept. The same process of searching for the best clause is repeated until finishing all clauses.

The weighted pseudo-log-likelihood (WPLL)is given by:

$$logP^*_\omega(X = x) = \sum_{r \in R} c_r \sum_{k=1}^{gr} logP_\omega(Xr, k = x_{r,k} \mid MB_x(X_{r,k})) \tag{3.8}$$

where R is the set of clauses(predicates), g_r is the number of groundings of first-order clause r, $x_{r,k}$ is the truth value (0 or 1) of the kth grounding of r. The choice of the clause weights c_r depends on the user's goals.

Algorithm 3 depicts an overall approach called MSL for MLN structure learning which is proposed by (Kok and Domingos, 2005). It uses beam search to search from all possible clauses. In each iteration, MSL uses beam search to find the best clause, and add it to the MLN: starting with all the unit clauses, it applies each possible operator (addition and deletion) to each clause, keeps the best ones, applies the operator to those, and repeats until no new clause improves the WPLL. The best clause is the one with highest WPLL score and it will be added to the MLN. MSL terminates when no more clause can be added to the MLN, and it then returns the MLN. Algorithm 4 details the procedure for finding the best clause adapted from (Kok and Domingos, 2005).

Several other methods have been proposed for generative MLN structure learning using WPLL measure. Those are Bottom-up Structure Learner (BUSL) (Mihalkova and Mooney, 2007), Iterated Local Search (ILS) (Biba et al., 2008b), Learning via Hypergraph Lifting (LHL) (Kok and Domingos, 2009), Learning using Structural Motifs (LSM) (Kok and Domingos, 2010), Moralized Bayes Net (MBN) (Khosravi et al., 2010) and Graph of Predicates (GoP) (Dinh, 2011).

3.4.3.2 Discriminative structure learning

Discriminative learning is applied if there is a specific target predicate that must be inferred given evidence data.

Discriminative approaches are extensions of ILS (Biba et al., 2008b) algorithm. (Huynh and Mooney, 2008) proposed a method which discriminatively learns both structure and parameters to optimize predic-

Algorithm 3 MLN structure learning (Kok and Domingos, 2005)

Data: R, a set of predicates
MLN, a clausal Markov logic network
DB, a relational database
Result: Modified MLN
begin
  Add all unit clauses from R to MLN
  for *each non-unit clause c in MLN* **do**
    Try all combinations of sign flips of literals in c
    keep the one that gives the highest WPLL(MLN, DB)
  $Clauses_0 \longleftarrow \{$ All clauses in MLN $\}$
  LearnWeights(MLN,DB)
  Score $\longleftarrow WPLL(MLN, DB)$
  repeat
    $Clauses \longleftarrow Find_Best_Clauses(R, MLN, Score, Clauses_0, DB)$
    if *Clauses* $\neq \emptyset$ **then**
      Add Clauses to MLN
      LearnWeights(MLN,DB)
  until *Clauses* = $\emptyset$
  for *each non-unit clause c in* MLN **do**
    Prune c from MLN unless this decreases WPLL(MLN,DB)
  return MLN

Algorithm 4 Find_Best_Clause (Kok and Domingos, 2005)

Data: R, a set of predicates
MLN, a clausal Markov logic network
Score, WPLL of MLN
$Clauses_0$, a set of clauses
DB, a relational database
Result: $BestClause$ a clause to be added to MLN
begin
  $BestClause \longleftarrow \emptyset$
  $BestGain \longleftarrow 0$
  $Beam \longleftarrow Clauses_0$
  Save the weights of the clauses in MLN
  repeat
    Candidates $\longleftarrow$ Create_Candidate_Clauses(Beam, R)
    for *each clause* $c \in$ *Candidates* **do**
      Add c to MLN
      LearnWeights(MLN, DB)
      Gain(c)$\longleftarrow$ WPLL(MLN, DB) $-$ Score
      Remove c from MLN
      Restore the weights of the clauses in MLN
      Beam $\longleftarrow \{$ The b clauses $c \in$ Candidates with highest Gain$(c) > 0$ and with Weight $(c) > \epsilon > 0\}$
    if *Gain(Best clause* c^* *in Beam)*$>$ *BestGain* **then**
      BestClause $\longleftarrow c^*$
      BestGain $\longleftarrow$ Gain(c^*)
    return BestClause
  until *Beam* = $\emptyset$ *or BestGain has not changed in two iterations*

tive accuracy for a query predicate given evidence specified by a set of defined background predicates.

(Biba et al., 2008a) proposed Iterated Local Search - Discriminative Structure Learning (ILS-DSL) approach which learns discriminatively first-order clauses and their weights. This algorithm shares the same structure with the ILS approach for generative MLN structure learning.

For both models, DAPER and MLN learning tasks are based essentially on sufficient statistics counts from data. The sufficient statistics are the counts of the number of times that we observe the child states given its parents' states performed in MLE. These counts have to be performed on a complete instance $\mathcal{I}$, using SQL queries in the case of the relational database. In the next section, we describe how to perform these sufficient statistics.

3.5 Computing sufficient statistics for PRMs learning

Databases contain information about which relationships do and do not hold among entities. To make this information accessible for PRMs learning, computing *sufficient statistics* that combine information from different database tables is required. These sufficient statistics are instantiation *counts* for conjunctive queries represented in the *contingency table*.

A key operation required by most probabilistic graphical models is counting sufficient statistics. For many parametric models, such as the exponential family, maximum likelihood estimation is done via sufficient statistics that summarize the data. For instance, (Friedman et al., 1999a) used counting for PRMs learning. Most directed models that employ a combination function such as mean or weighted mean (Kersting et al., 2009) use counts.

Similarly, MLNs use counting as their fundamental operation in computing posterior probabilities (Domingos and Lowd, 2009). Also, most inference methods (Poole, 2003), (Singla and Domingos, 2008), (Milch et al., 2008), (Kersting et al., 2009), (de Salvo Braz et al., 2009), (Ahmadi et al., 2010), (Van den Broeck et al., 2011) that aim to reason at a high-level as much as possible by exploiting symmetries also consider counts of the objects in each group to compute the final posteriors.

3.5.1 Definitions

Definition 3.5.1. ***Query***
A query is a set of $\{variable = value\}$ *pairs where each value is of a valid type for the random variable. The result set of a query in a database D is the set of instantiations of its attributes.*

Definition 3.5.2. *Count*
The count of a query is the cardinality of its result set.

Definition 3.5.3. ***Contingency Table****(Moore and Lee, 1998)*
The contingency table denoted CT *presents counts of a set of variables* $\{A_1, A_2, ..., A_n\}$.
Each row of the contingency table represents possible assignments (configurations) of values to the variables and an integer column called ***count****.*
The value of the count column in a row corresponding to $\{V_1 = v_1, ..., V_n = v_n\}$ *records the count of the corresponding query.*

Example 3.5.1. *Figure 3.5 shows one contingency table for the university database. We omit the "ranking" and "grade" attributes for simplicity.*

ranking	grade	count
C	B	20
A	A	10
⋮	⋮	⋮
B	B	24

Figure 3.5 – Excerpt from the CT table for the attribute *student.ranking* and its *parent* as they are linked in the DAPER of Figure 3.3.

3.5.2 SQL implementation for computing contingency table

Generally, for PRMs the contingency tables are computed from relational databases using SQL queries (Getoor, 2001). For a fixed set of attributes, a contingency table can be computed by an SQL *count(*)* query of the form:

```
CREATE VIEW CT-table(<VARIABLE-LIST>) AS
SELECT COUNT(*) AS count, <VARIABLE-LIST>
FROM TABLE-LIST
WHERE <Join-Conditions>
GROUP BY VARIABLE-LIST
```

As seen in the query form listed before, counting sufficient statistics consist of making a set of join operations in the database tables.

Example 3.5.2. *The query used to compute the CT table in Figure 3.5 is as follow:*

```
SELECT ranking, grade, count(*)
FROM student,takes
JOIN takes ON student.id = takes.student-id
GROUP BY ranking, grade
```

Example 3.5.3. *As a simple example, takes the case of three attributes of our university domain "grade", "intelligence" and "difficulty". The computation of the sufficient statistics of this set of attribute requires the join of three tables which correspond to three classes "student", "take" and "course". The SQL query is as follow:*

```
SELECT grade, intelligence, difficulty, count(*)
FROM takes, student, course
JOIN ON student.id = takes.student-id
JOIN ON takes.course-id=course.id
GROUP BY grade, intelligence, difficulty
```

We notice that two joins operation are obtained just for three attributes belong to three classes. Therefore, increasing the number of instances and attributes, queries become more complex because we have to join large number of entities to specify the mapping between a foreign key in one table and the associated primary key.

However, SQL joins have an exponential cost in term of time-consuming especially with complex and highly connected data. This is explained by the fact that a large increase of connected data led to a consequent growth of the number joins which impacted negatively on performances of the relational databases.

Therefore, due to the increasing volume of various data being generated, computing contingency table from relational tables is no longer feasible (Oliver and Zhensong, 2015; Das et al., 2015), (Vicknair et al., 2010; Partner et al., 2014). Also, the machine learning application saves expensive data transfer by

executing count operations in memory. For many datasets, the number of sufficient statistics runs in the millions and is too big for a main memory. Therefore, accessing sufficient statistics is often the main scalability bottleneck (Oliver and Zhensong, 2015). Thus, different database frameworks emerged in the NoSQL movement, such as graph databases.

3.6 Inference in PRMs

Like in PGMs, probabilistic inference in PRMs can be viewed as a process by which influence flows through the network. But instead of constraining that flow to be between the random variables of one instance, like in Bayesian Network and Markov Network, PRMs allow flow between interrelated objects as well.

In directed models such as DAPER, much of the existing works involve constructing a ground Bayesian Network (GBN) and performing inference in this model. Theoretically, standard inference algorithms for Bayesian Networks can be used to query the GBNs. When GBNs are small, the exact inference can be performed. However, Large and complex GBNs make exact inference very expensive.

To resolve this problem, (Kaelin and Precup, 2010) have proposed a method for performing approximate inference in DAPERs called Lazy Aggregation Block Gibbs (LABG). This method is based on the idea that a query can be answered in a Bayesian network by taking into account only the subgraph that contains all event nodes and is d-separated from the full GBN given the evidence nodes. The method constructs a partial GBN for the given query and apply Gibbs sampling approach for approximate inference.

(Koller and Pfeffer, 2000) introduce the structured variable elimination approach . It is an extension of the original variable elimination method (Zhang and Poole, 1996) to the relational domain.

In the setting of undirected models such as MLN, inference consists of translating first-order formulae into propositional ones. MLNs have the advantage that influence can flow in both directions. However, this fact increases the complexity during inference.

(de Salvo Braz et al., 2005; de Salvo Braz et al., 2006) developed a variable elimination algorithm for undirected models. But, it is extremely complex, generally does not scale to realistic domains, and has only been applied to very small artificial problems.

(Poon and Domingos, 2006) propose MC-SAT, an inference algorithm that combines ideas from MCMC (Wei et al., 2004) and satisfiability. MC-SAT is based on Markov logic, which defines Markov networks using weighted clauses in first-order logic.

(Singla and Domingos, 2008) introduce probabilistic inference in MLN, which aims at performing as much inference as possible without propositionalizing. This method is based on first constructing a lifted network, where each node represents a set of ground atoms that all pass the same messages during belief propagation. Then, it runs belief propagation on this network.

3.7 Conclusion

In this chapter, we recalled some backgrounds of ER model and First-Order-Logic. Then we introduced two probabilistic relational models namely DAPER and MLN by representing several concepts related to each one of them. Then a brief overview of methods in learning and inference is given. We have seen that DAPERs and MLNs are extensions of BNs and MNs respectively. These PRMs are generally learned from relational data in which links or relationships are an essential source of information. We finally described how to construct contingency tables to learn PRMs in the case of the relational database and we described the challenge of computing CT tables from relational data. However, due to the increasing volume of various data being generated, different database frameworks emerged in the NoSQL movement, such as graph databases.

Chapter 4 will introduce the context NoSQL especially the graph database by giving its definition, characteristics and advantages and by comparing it with other NoSQL databases.

4

Graph database

Contents

4.1 Introduction **52**
4.2 NoSQL databases **52**
4.2.1 CAP theorem 54
4.2.2 BASE properties 54
4.3 Basic concepts related to graph database **55**
4.4 Graph database manipulation **57**
4.4.1 Subgraph matching 57
4.5 Graph database properties **59**
4.6 Graph database use cases **61**
4.7 Comparison with other database models **61**
4.8 Discussion **62**
4.9 Conclusion **63**

4.1 Introduction

With the rise of the Internet, data increase both in volume and interconnections. This fact produces collection of large and complex data sets which are difficult to process using traditional database management tools and applications. With the emergence of Big data, the need for more flexible databases is evident.

According to (Sun and Reddy, 2013), Big data consists of the major four V's namely "Volume" which means simply lots of data gathered by a company. "Variety" refers to the type of data that Big Data can comprise, data can be structured as well as unstructured. "Velocity" refers to the speed of data generation and the time in which Data Stream can be processed. "Veracity" deals with the uncertainty of data. One of the important challenges in Big data is how to store and manage this new complex data.

Relational database systems are generally efficient unless the data contains many relationships requiring joins of large tables. That's why a relational database are unsuited to scale and manage such kind of data.

This problem has been recently addressed by a new brand category of data management systems that do not use SQL exclusively, the so-called NoSQL movement (Partner et al., 2014). NoSQL includes four major types of databases: key/value, oriented document, oriented column, and graph databases. Recently there has been an increasing interest in graph databases to model objects and interactions. In contrast to relational databases, where join-intensive query performance deteriorates as the dataset gets bigger, a graph database depends less on a rigid schema and its performance tends to remain relatively constant, even as the dataset grows. This is because queries are expressed regarding graph traversal operations which are localized to a portion of the graph.

Therefore, graph databases are rapidly emerging as an effective and efficient solution to the management of very large data sets in scenarios where data are strongly connected, and data access mainly rely on traversing these graphs. Data scientist queries in these scenarios are more interested on the relationships between data rather than on the nodes of the graph. Namely, find the list of friends of a Facebook user that have studied with him in the same class, find the calls of a person in a telecommunication network, find users assisting the same category of movie, find the peoples buying a specific product or mining relationships between customers in a marketing study, etc. Relationships are first-class citizens of the graph database. This is not the case in other database management systems, where we have to infer connections between entities using things like foreign keys or out-of-band processing such as map-reduce.

For this reason, graph database are often found in scenarios where the data model is considerably connected, including social, telecommunications, logistics, master data management, bioinformatics, fraud detection and system recommendation.

The rest of this chapter is structured as follows. The Section 4.2 introduces NoSQL databases and their properties. Some basic concepts and the graph database definition are given in Section 4.3. Section 4.4 presents an overview of graph databases querying and manipulation. A comparison between graph database and other database stores is given in Section 4.7. A brief discussion is given in Section 4.8. We conclude in Section 4.9.

4.2 NoSQL databases

The increase of data volume during the last decade is attributed to a variety of data sources. Data that was once considered too expensive to store can now be captured, stored and processed thanks to new innovated storage system called NoSQL (Not only SQL). NoSQL databases can support larger volumes of data by providing faster data access and cost savings. The cost savings of NoSQL databases results from a

physical architecture that includes the use of inexpensive commodity servers that leverage distributed processing. NoSQL includes four other categories of databases; Key-Value database, Column Family database and Document database and graph database.

In this section, a brief refresh on Key-Value database, Column Family database and Document database definitions are proposed, and an overview of CAP and BASE properties (Dan, 2008) supported by NoSQL frameworks is introduced. The next section focuses on introducing the graph database.

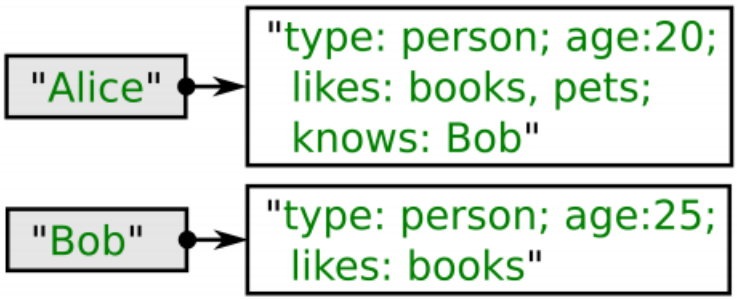


Figure 4.1 – Example of Key-Value store.

Definition 4.2.1. ***Key-Value database****(Edlich, 2011)*
Key-Value based storage systems are associative arrays, consisting of keys and values. Each key has to be unique to provide non-ambiguous identification of values. The values can be simple text or complex data types such as sets of data.

Example 4.2.1. *An example of a Key-Value database is given in Figure 4.1.*

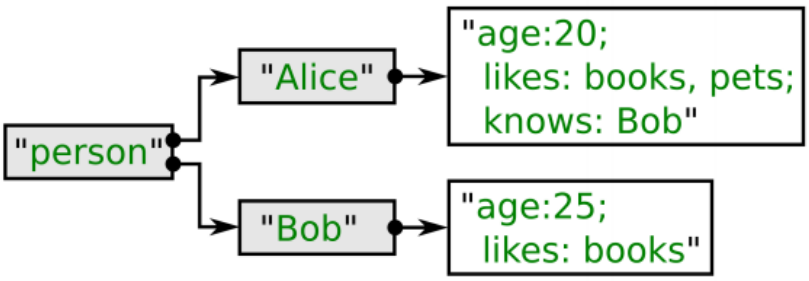


Figure 4.2 – Example of Column Family database.

Definition 4.2.2. ***Column Family database****(Cattell, 2011)*
This class of storage is an extension of Key-Value database which has a wide column store can be seen as a Key-Value database, with a two-dimensional key: A stored value is referenced by a column key and a row key. It has a format of data storage that is very similar to the relational database. They can support complex data types, unstructured text and graphics.

Example 4.2.2. *An example of Column Family database is given in Figure 4.2.*

Definition 4.2.3. ***Document database****(Huang and Luo, 2014)*
In a document store, data is stored in the format of documents which refers to arbitrary data in some structured data format. Examples of used formats are JSON, BSON and XML. While the document database typically fixes the type of data format.

```
people
{
  name:  "Alice",
  age:   20,
  likes: ["books", "pets"],
  knows: "Bob"
}
{
  name:  "Bob",
  age:   25,
  likes: ["books"]
}
```

Figure 4.3 – Example of Document database.

Example 4.2.3. *An example of Document database is given in Figure 4.3.*

To really make efficient storing of large volumes of data possible in NoSQL databases, the transactional model is usually relaxed and does not guarantee the same assurances as it is in relational databases. In NoSQL, performance and scalability are preferred over consistency. In general, scaling can be performed using two distinct techniques; vertical and horizontal scaling.

— *Horizontal scaling* is often based on the partitioning of the data i.e., each node contains only part of the data.

— *Vertical scaling* means the data resides on a single node and scaling is done through multi-core i.e., spreading the load between the CPU and RAM resources of that machine.

4.2.1 CAP theorem

While relational databases highly value data integrity, using the ACID theorem for data consistency, NoSQL databases differ significantly on their approach to maintaining data integrity and consistency. A more relaxed approach to data consistency helps NoSQL databases improve the performance of data storage. Indeed, NoSQL databases use the CAP theorem (Consistency, Availability and Partition Tolerance) for data consistency (Brewer, 2000).

— *Consistency* means that data access in a distributed database is considered to be consistent when an update operation of some writer all readers see his updates in some shared data source.

— *Availability* means that the system guarantees availability for requests even though one or more nodes are down.

— *Partition tolerance* understood as the ability of the system to continue operation in the presence of network partitions

The CAP Theorem states that of the three possible combinations of CAP, only two are available at a given point in time. In other words, NoSQL databases can have partition tolerance (in a distributed environment) and consistency or partition tolerance and availability, but not all three factors at the same time (Brewer, 2000). Figure 4.4 is an illustration of CAP theorem.

4.2.2 BASE properties

Relational databases allow manipulating any combination of rows from any table in a single (ACID) transaction (i.e., Atomic, Consistent, Isolated, and Durable). In contrast, in the NoSQL world, ACID transactions are less important as some databases have loosened the requirements for immediate consistency, data freshness and accuracy to gain other benefits, like scale and resilience.

ACID requirements were too constraining for many of the Internet's demands; some constraints had to be relaxed. With the rise of non-relational databases, a new acronym in the database transaction reliability world is emerged called *BASE* (Dan, 2008; Han et al., 2011). The BASE acronym entails:

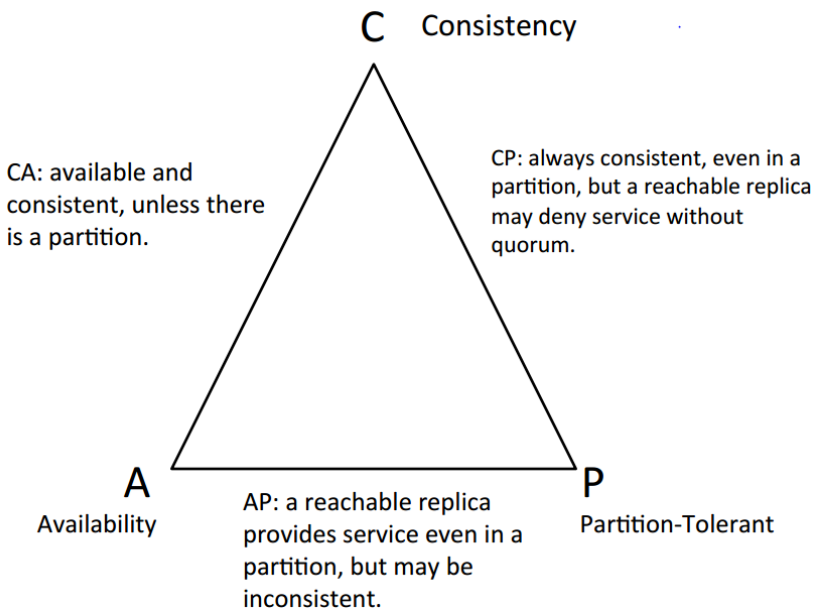


Figure 4.4 – The CAP theorem (Brewer, 2000)

— Basically Available: the system guarantees some level of availability to the data even in regards to node failures. The data may be stale, but will still give and accept responses.

— Soft State: the data is in a constant state of flux; so, while a response may be given, the freshness or consistency of the data is not guaranteed to be the most current.

— Eventual Consistency: the data will eventually be consistent through all nodes and in all databases, but not every transaction at every moment. It will reach some guaranteed state eventually.

A BASE database values availability , but it does not offer guaranteed consistency of replicated data at write time. Overall, the BASE consistency model provides a less strict assurance than ACID. BASE is still a leading innovation that is wedded to the NoSQL model, and the evolution of both together is harmonious. But that does not mean they always have to be in partnership.

The BASE consistency model is primarily used by aggregate stores, including column family, key-value and document stores. This is however not true for graph databases.

To make relation with the CAP theorem, BASE properties satisfy the Availability and Partition tolerance: BASE= A+P means that no strong consistency guarantees.

In the Table 4.1 a comparison between ACID and BASE properties is given (Brewer, 2000).

ACID	BASE
Strong consistency	Weak consistency (=> allow stale data)
Robust database	Simple database
Available and consistent	Available and partition-tolerant
Scale-up (limited)	Scale-out (unlimited)
Difficult evolution(e.g. schema)	Easier evolution
Shared-something (disk, mem, proc)	Shared-nothing (parallellizable)

Table 4.1 – ACID vs. BASE (Brewer, 2000)

4.3 Basic concepts related to graph database

In this section some basic concepts related to graph database are defined.

Definition 4.3.1. ***Node***
A node also called vertex is the fundamental unit of which graphs are formed that represents concepts, variable or classes of objects. Usually, represented by a circle with a type.

Definition 4.3.2. ***Relationship***
A relationship is a labeled edge which links the interconnected nodes in a graph. The link (n_i, n_j) is of initial extremity n_i and terminal extremity n_j. The edges may be directed or undirected.

Definition 4.3.3. ***property***
A property is presented in the form of $\langle key, value \rangle$ pairs associated with node and edge to create additional semantics.

Definition 4.3.4. ***Alphabet***
An alphabet Σ is the set of all relationships' labels.

Definition 4.3.5. ***Regular expression***
A regular expression $\Sigma^*.sub.\Sigma^*.sub.\Sigma^*$, where sub indicates subordinacy in the hierarchy. In other words, if there is a path between n_i and n_j that goes via at least two intermediaries, this will be expressed by a regular expression.

Definition 4.3.6. ***Path***
A path ρ from n_0 to n_m is a sequence $(n_0, a_0, n_1), (n_1, a_1, n_2), ..., (n_{m-1}, a_{m-1}, n_m)$, for some $m \geq 0$, where each (n_i, a_i, n_{i+1}), for $i < m$, is an edge with n_i's are nodes and all a_j's are letters in Σ. The label of ρ, denoted $\lambda(\rho)$, is the word $a_0...a_{m-1} \in \Sigma$.

Definition 4.3.7. ***Shortest path***
The shortest path is the path that relates two nodes via one relationship.

Definition 4.3.8. ***Tree navigation***
Tree navigation is set of shortest paths.

Definition 4.3.9. ***Graph database****(Cruz et al., 1987)*
A graph database is a property graph or a finite edge-labeled graph. Let Σ be a finite alphabet, and N a countably infinite set of a labeled node. Then a property graph over Σ is a pair $G = (N, E)$, where N is the set of nodes and E is the set of edges (relationships), i.e., $E \subseteq N * \Sigma * N$. That is, we view each edge as a triple $(n, a, n^{'})$ and every node $n \in N$ and every edge $e \in E$ is associated with a set of pairs $< key, value >$, called properties.

Figure 4.5 shows the components of a graph database.

Example 4.3.1. *Figure 4.6 summarizes all the concepts related to a property graph. In this graph, there are three nodes: 1 node with type "Student", a node with type "Course" and the third one has the type "Professor". Each node has a set of properties with information about it. Also, there are two relationships "Takes" and "Gives" between the nodes, with only one relationships have properties.*

Definition 4.3.10. ***Pattern element***
A Pattern element $\mathcal{P}$ can be used to match data, evaluating to a list of paths. The difference between Tree navigation and pattern element that the first one corresponds to a long path expressed in the same matching query, or in the second one we can find a simple match followed by a sequence of optional subgraph matching.

Definition 4.3.11. ***Graph pattern***
A pattern graph specifies the structural requirements that a subgraph of G must satisfy to match the pattern $\mathcal{P}$. The task is to find the set of subgraphs of G that "match" the pattern $\mathcal{P}$

Definition 4.3.12. ***Degree***
The degree of a node is the total number of nodes adjacent to it. Equivalently, we can define the degree of a node as the cardinality of its neighborhood.

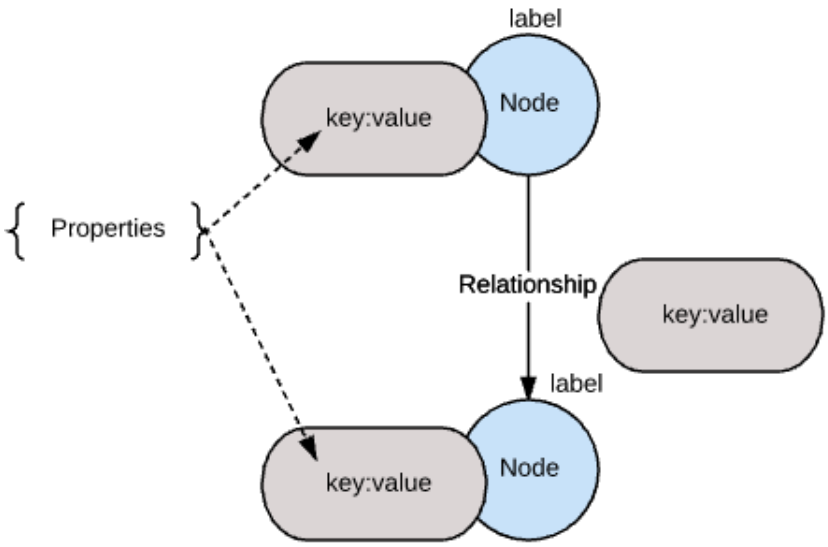


Figure 4.5 – The graph database components.

4.4 Graph database manipulation

As in traditional databases, queries in graph databases cover the transactional operations, known in traditional databases, to add, delete, update and find entities. The underline data structure of graph database allows running efficiently queries with complex joins, called patterns. Pattern matching is the most query type required on graph database applications. They are considered as analytical queries that allow online finding semantic relationships of baseline data in order to discover knowledges from the data graph. Relationships in a graph naturally form paths. Querying or traversing the graph involves following paths. Thus, the third feature that satisfies the graph database is the fact that data is queried using path traversal operations expressed in some graph-based query language based on graph theory described in Chapter 2. There are several graph database frameworks such as Neo4j [1], AllegroGraph [2], Infogrid [3], HypergraphDB [4], DEx [5] but they don't provide a universal query language such as SQL for relational databases.

There are various proposals of query languages for graph data models. However, most of them are based on subgraph matching.

(Wood, 2012) provides a survey of many of the graph query languages, focusing on the core functionality provided by subgraph matching such as finding nodes connected by paths, comparing and returning paths, aggregation, node creation, etc.

4.4.1 Subgraph matching

The goal of graph matching is to find occurrences of a specific pattern in a graph . It includes graph oriented query language, in which queries are expressed as graphs. A match or a partial match is as a set of an edge and node pairs. Each edge represents a mapping between nodes. This technique of querying specify the existence of paths between nodes, with the restriction that the labels of such path belong to regular languages. The simplest such queries are known as Regular Path Queries, or RPQs (Cruz et al., 1987),

1. http://db-engines.com/en/ranking/graph+dbms
2. https://franz.com/
3. http://infogrid.org/trac/
4. http://hypergraphdb.org/
5. http://www.sparsity-technologies.com/

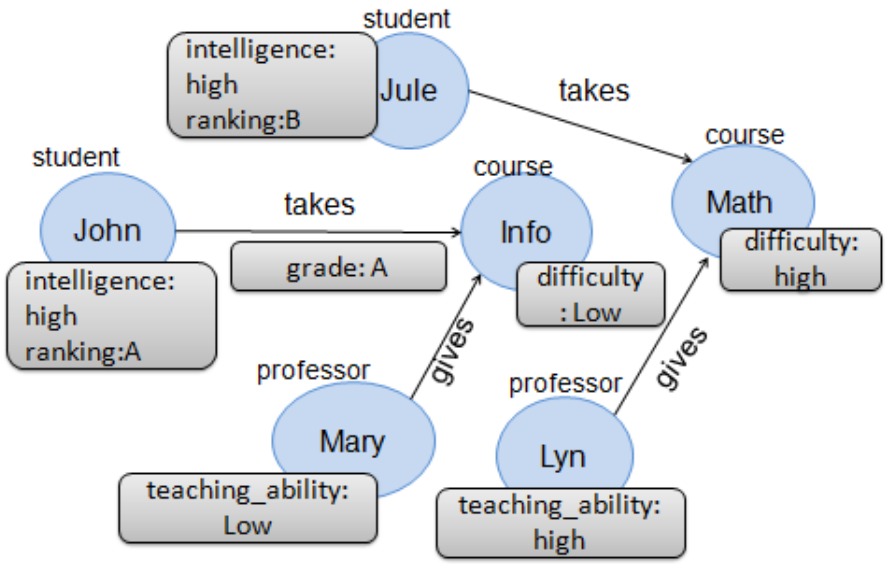


Figure 4.6 – Example of graph database for the university domain.

(Reutter, 2013); those select nodes connected by a path that belongs to a regular language. Conjunctive RPQs, or CRPQs, extend them by allowing intermediate nodes in paths. Among them, we mention Cypher, Gremlin , XPath and XQuery which are based on expressions of the form start-match-where-return. These languages, usually called traversal query based on path expressions.

Path queries are classified on Regular Path Query and Conjunctive Regular Path Queries.

Definition 4.4.1. ***Regular Path Query****(RPQ): provides queries which return all pairs of nodes in a graph connected by a path conforming to some regular expression.*

$$ans(x, y) \longleftarrow (x, r, y)$$

Where x and y are node variables, and r is a regular expression over Σ.

Definition 4.4.2. ***Conjunctive Regular Path query*** *(CRPQ)*
In such queries, multiple RPQs can be combined, and some variables can be existentially quantified. Formally, give a set of nodes $(x, y, z, ...)$, a CRPQ over a finite alphabet Σ is an expression of the form:

$$ans(z_1, ..., z_n) \leftarrow \wedge_{1 \leq i \leq m}(x_i, a_i, y_i)$$

such that $m > 0$, each x_i and y_i, is a node, each $a_i \in \Sigma(1 \leq i \leq m)$ and each z_i is some x_i or y_j with $(1 \leq i \leq n, 1 \leq j \leq m)$. The atom $ans(z_1, ..., z_n)$ is the head of the query, while the expression on the right of the arrow is its body

Example 4.4.1. *Using the example graph G from Figure 4.7, the following query finds courses by Fatim which are taken by both Mary and Lyn students:*

$$ans(x) \longleftarrow (x, takes, Mary), (x, takes, Lyn), (x, gives, Fatim)$$

Two matching subgraph are returned as described in Figure 4.8

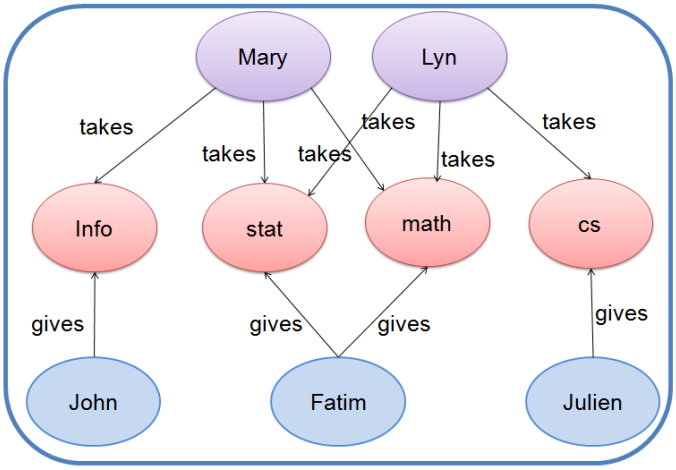


Figure 4.7 – Example of graph database

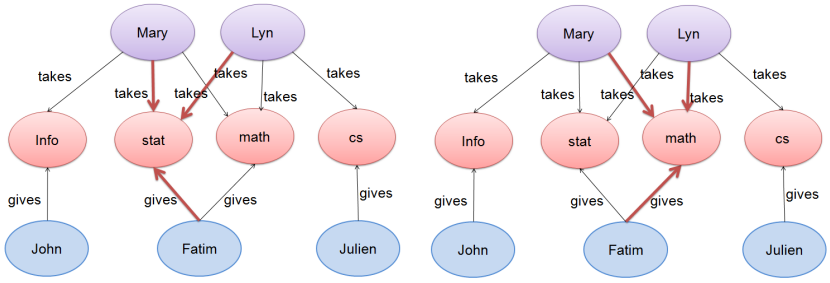


Figure 4.8 – Example of subgraph matching.

4.4.1.1 Aggregation

Determining various properties of graphs requires computation that goes beyond matching and path finding. Such properties range from simple computations to determine the degrees of nodes to more complex ones for computing the eccentricity of a node, the distance between pairs of nodes, or the diameter of a graph. To formulate queries which return the values of such properties, we need aggregation operators such as count, sum, min and max.

Definition 4.4.3. ***Aggregation***
Giving a CRPQ, the equivalent $CRPQ^{agg}$ is:

$$ans(x, \gamma) \longleftarrow \gamma(x, a_i, y)$$

With γ is an aggregation function and $a_i \in \Sigma$

4.5 Graph database properties

Graph database provides a powerful and novel data modeling techniques thanks to its properties. Like a relational database, a graph database satisfies ACID properties

The key ACID guarantee is that it provides a safe environment in which to operate on your data. In the NoSQL world, ACID transactions are less important as some databases have loosened the requirements for immediate consistency, data freshness and accuracy in order to gain other benefits, like scale and resilience.

However, graph databases use an ACID consistency model to ensure data is safe and consistently stored.

A graph database satisfies another important property called *index-free adjacency* property.

Definition 4.5.1. ***Index-free adjacency***
Index-free adjacency means that connected nodes physically "point" to each other in the database. A graph database G satisfies the index-free adjacency if the existence of an edge between two nodes n_1 and n_2 in G can be tested on those nodes and does not require to access an external or global index. No global index means each node stores information about its neighbors only.

So that every element contains a direct pointer to its adjacent elements. Specifically, each node is associated with an index as the preferred way to find start points for graph traversals. As a result, the traversal of an edge is basically independent on the size of data. This makes the graph database faster for associative data sets, and map more directly to the structure of object-oriented applications than relational database especially where data size and connections increase rapidly.

Graph databases are unstructured data stores which are *schema-free*, so the database itself is no more constrained to be an instance of a meta model (ER model or relational schema). This fact has some consequent on the database structure. The database can contain nodes with the same type but have different properties. Also, an important character that in a graph database, we can find the same edge type links between a different pair of nodes' types. By consequence, many "exceptions" relationship can be found in the database.

As we mentioned before, graph databases are *schema-free* unstructured data stores. This fact makes graph database naturally *additive* and *flexible*, meaning new kinds of relationships, new nodes, new labels, and new subgraphs can be added to an existing structure without disturbing existing queries and application functionality. This can also explain the fact that the graph database can contain "exceptions", because links can be added right and wrong as long as the database does not obey to a given meta-model defined in advance.

Also, the *schema-free* nature of the graph data model, coupled with the *testable* nature of a graph database's application programming interface (API) and query language, allow evolving an application in a controlled manner. That's why graph database supports *agility*, which is crucial in a test-driven development environment. So that, the graph database can change as the application's changing requirements.

The graph database *performance* tends to remain relatively constant, even as the dataset grows. This is because queries are localized to a portion of the graph. In fact, you find a logical starting point, and you branch out from there and identify the relationships. For instance, you might write a query that asks, *"Find all of the friends of the friends of Mary in a social network"*. Instead of having to "JOIN" many different indexes, the graph database uses pointer arithmetic that is in-memory or in a cache and performs the operation. As a result, the execution time for each query is proportional only to the size of the part of the graph traversed to satisfy that query, rather than the size of the overall graph.

Definition 4.5.2. ***Connected Query Performance***
the Connected Query Performance corresponds to the Query Response Time which is expressed in function of graph density, graph size and query degree as follow :
Query Response Time = f(graph density, graph size, query degree)

where graph density is an average number of the relationship per node, and graph size is the total number of nodes in the graph and query degree is the number of hops in one's query.

The success of graph databases is since they meet our requirements of explicitness and clarity by transforming high-dimensional domains into graphs making, thereby, entities represented simply as nodes and relationships as edges. This make graph databases more *expressive*.

4.6 Graph database use cases

Graphs are widely used to model social networks, but the use of graphs to solve real-world problems is not limited to that. Many other applications can naturally be modeled as graphs. Graph databases are very useful in understanding big data sets in scenarios as diverse as logistics route optimization (Leonid and Toby, 2012) (such as planning routes), protein interaction networks (Royer et al., 2008), where the patterns of interactions are important to understand metabolic processes.

Graph databases also out-class other database technology for connecting masses of buyer and product data making effective real-time recommendations depends on a database that understands the relationships between entities, as well as the quality and strength of those connections (Webber and Robinson, 2012). Only a graph database efficiently tracks these relationships according to user purchase, interactions, and reviews to give the most meaningful insight into customer needs and product trends. Graph-powered recommendation engines can take two major approaches: identifying resources of interest to individuals, or identifying individuals likely to be interested in a given resource.

Graph databases also uncover patterns that are difficult to detect using traditional representations such as tables. An increasing number of companies use graph databases to solve a variety of connected data problems. Namely, fraud detection (graph databases offer new methods of uncovering fraud rings and other complex scams with a high level of accuracy through advanced contextual link analysis, and they are capable of stopping advanced fraud scenarios in real time) (Webber and Robinson, 2012), financial services (for monitoring and preventing internal and external fraud and its risks), retail suggestion engines (graph database can be used for understanding purchase decisions and to provide recommendations to customers based on how different products link with each other) (Ritter, 2012).

Graph databases are a powerful tool for modeling, storing and querying network and IT operational data (EMA, 2013). They are successfully employed in the areas of telecommunications, network management, impact analysis, cloud platform management and data center and IT asset management. In fact, graph databases store configuration information to alert operators in real time to potential shared failure modes in the infrastructure and to reduce problem analysis and resolution times from hours to seconds.

4.7 Comparison with other database models

In this section, we compare the graph database against the other NoSQl and relational databases models.

Although key-value, column, and document databases have the main advantage of the performance of data processing, graph databases can solve very complex problems that other NoSQL databases would be unable to do. For example, Social networks (Facebook, Twitter, etc.) where millions of users are connected in different ways. The challenge here is not the number of elements to be managed, but the number of relationships that can exist between all of these elements and how we can extract knowledge from these relationships. While key-value, column and document databases are focusing on the number of instances that they can store. Due to the interest to graph databases which focus on the relationships between data more than on the data nodes. Moreover, graph databases represent the natural way of modelling graph-like data. Compared to the others NoSQL databases; graph databases are dedicated for managing the higher level of data complexity. (Angles and Gutierrez, 2008) has categorized the NOSQL databases as in the Figure 4.9.

The difference between relational and graph databases that the relational model is geared to handle simple record-type data, where the data structure is known in advance; namely accounting, airline reservations, accounting, inventories, etc. In the case of relational databases, the schema is fixed in advance which makes it difficult to extend these databases. It is not easy to integrate different schemas. The query language cannot explore the underlying graph of relationships among the data, such as paths, neighborhoods, patterns. Also

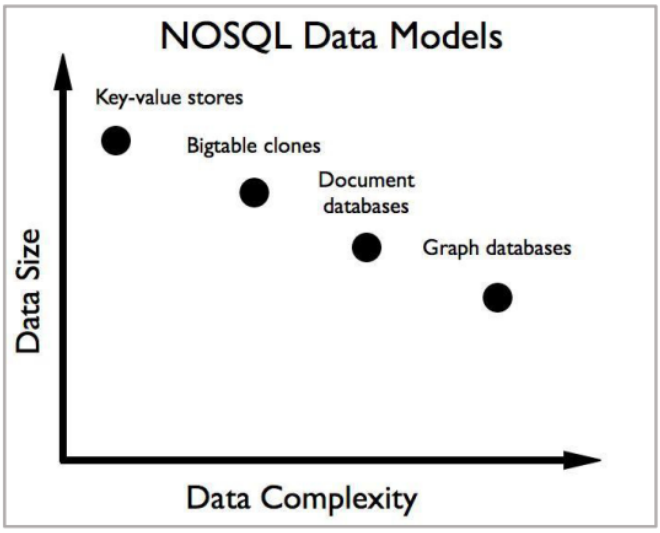


Figure 4.9 – Categorization of NOSQL databases.

Database	Abstraction level	Base data structure	Data complex	Information focus
Key-Value	logical	simple column	simple	data + attributes
Column family	logical	groups of columns	simple	data + attributes
Document	logical/user	document	medium	data + attributes
Relational	logical	relations	simple	data + attributes
Graph	logical/user	graph	complex	data+attributes + relations

Table 4.2 – A Comparison of the NoSQL and relational databases

comparing to the relational database, graph database are more expressive because it allows distinguishing between an entity (a node) and a relationship (an edge). or in the case of relational model all entities and relationships are presented as tables.

Table 4.2 presents an overview of this comparison.

4.8 Discussion

Databases contain information about which relationships do and do not hold among entities. To make this information accessible for statistical analysis requires sufficient computing statistics that combine information from different database tables. Such statistics may involve any number of table join.

Whereas SQL joins can efficiently compute sufficient statistics of existing database tables, a table join approach is not feasible for large datasets. This is because we would have to enumerate all tuples of entities that are related. The cost of the enumeration approach is close to materializing the cartesian product of entity sets, which grows exponentially with the number of entity sets involved (Oliver and Zhensong, 2015; Das et al., 2015), (Vicknair et al., 2010; Partner et al., 2014).

Alternative database models such as graph databases are more used to address this problem. Indeed, graphs can be used to model many interesting problems. As seen in Section 4.6, graphs are widely used to solve real-world problems such as social networks, Semantic Web, geographic applications, and bioinformatics. In these application domains, data have a natural representation regarding graphs, and queries mainly require to traverse those graphs. It has been observed that relational database technology is usually unsuited to manage such kind of data since they hardly capture their inherent graph structure. Besides, graph traversals over highly connected data involve complex join operations, which can make typical oper-

ations inefficient and applications hard to scale. In the opposite, with graph databases, there are no tables and columns nor "select" and "join" commands, but only graph traversing operations (Vicknair et al., 2010; Partner et al., 2014).

The traversal consists simply in visiting nodes moving across their relationships; it is strictly related to the graph model, and it makes it the most important feature. The most critical factor for the traversals is that queries are localized, and they run only on the data that is required, different from the SQL where the major part of the resulting sets are discarded , thus avoiding heavy actions and leading to a general better performance.

Therefore, there is a real proliferation of graph databases (Graves and Lawrence, 2002), (Hunger, 2012), (Jouili and Vansteenberghe, 2013), (Shin and Hiroki, 2014), (Liu et al., 2015). Besides, since graph databases do not rely on a rigid schema, they provide a more flexible solution in scenarios where the organization of data evolves rapidly.

Graph databases can also be real-time data ingest, which make their schema dynamic, and real-time querying. Also, most queries in graph databases traverse edges to fetch neighbor's information. For these reasons and because of the large size of graphs, incremental processing, and query decomposition are mainly devoted for graph pattern matching (Yang et al., 2012), (Fan et al., 2013) (Choudhury et al., 2014). Incremental processing is used as a technique to avoid running complex queries from scratch on large graph databases.

However, as graph databases are schema-free, it does not exist a standard method to generate a conceptual schema as well as the relational database. Some methods are proposed such as the dense, compact and sparse strategies.

The dense strategy (De Virgilio et al., 2014) which consist of adding as many edges as possible between nodes representing conceptual entities. This reduces the length of paths between nodes and so the number of data access operations needed at run-time. However, it requires to add edges that do not correspond to conceptual relationships in the application domain.

The compact strategy (De Virgilio et al., 2014) which consist of aggregating the same node data that are related but are stored in different nodes. This reduces the number of data accesses as well but, on the other hand, it has a negative impact on the semantic of data connectivity especially in case of one-to-many and many-to-many relationship. This causes a conflict on nodes properties, which requires a suitable renaming of keys.

Some other modeling approaches are proposed to deal with the problem of the compact approach such as the sparse strategy (Sequeda et al., 2012). In this method, the properties of an object with n properties is decomposed into a set of n different nodes, with the goal of minimizing the number of edges incident to the nodes of the graph. This avoids potential conflicts between the properties of a node, but it usually increases largely the number of nodes that need to be traversed during query execution. Moreover, all these approaches demand the intervention of a designer and an additional effort of an expert to deal with all possible data consistencies.

4.9 Conclusion

Today, traditional databases are enable to store large graphs and querying their complex relationships, which need complex joins of relational tables.The actual NoSQL databases systems, have demonstrated their efficiency to store the large volume of data. However, this efficiency decreases when querying highly interconnected data. Thus, there is a new tendency to graph databases in order to develop standard tools for

managing graph data.

In this chapter, we have reviewed the graph database' basic concepts, definition and properties. We have seen from this study that graph database is an expressive model for representing connected data for complex domains expressively and flexibly. Some graph database applications are described. Finally, we have made a brief comparative study between graph database, NoSQL and relational databases. In the next chapter, we will focus on learning Probabilistic Relational Models from graph databases.

III

Contributions

5

DAPER learning from Graph database using PRM framework

Contents

5.1 Introduction **68**
5.2 DAPER learning from graph database **68**
5.2.1 Principle 69
5.2.2 ER model identification 70
5.2.3 Probabilistic dependencies identification 72
5.3 Experiments **75**
5.3.1 Experiment methodology 76
5.3.2 Networks and Datasets 79
5.3.3 Algorithms 80
5.3.4 Evaluation metrics 80
5.3.5 Results and interpretations 82
5.4 Conclusion **89**

5.1 Introduction

From our state of art, we have seen that learning PRMs with NoSQL frameworks could have a significant impact on the performances and the scalability of the considered application. With the recent emergence of graph database implementations, there is an increasing need for using PRMs with graph databases to benefit from its characteristics as it is described in Chapter 4. Even though a panoply of works has focused on PRMs, all existing works about PRMs are dedicated to relational databases as shown in Chapter 3. No work has identified for PRMs learning from graph databases.

In this chapter, we try to fill this gap by proposing our first contribution which is based on the recent work on DAPER models (Heckerman and Meek, 2004). This proposed method describes how to learn DAPER (the ER schema and the probabilistic dependencies) from partially structured graph databases. Generally, the ER schema is defined in advance before the databases constructing. As graph databases are schema-less, an ER schema identifying from a graph database is required.

Our contribution is twofold. First, a new method to extract the underlying ER model from a partially structured graph database will be presented. Indeed, all the elements of the output ER diagram are retrieved directly from concepts appearing in the input database instance. This ensure that our ER model is semantically close to the design layer of the database. After identifying the ER model, the second phase consists to define the probabilistic dependencies of our DAPER model. As seen in Chapter 3, the learning task relies on scoring functions or independence measurements estimated from statistics (counts) extracted from the database. So one of the key operations inside most learning algorithms is counting. However, existing learning methods use a relational database for counting and do not exploit any fast counting methods. Therefore, a method to compute sufficient statistics based on graph traversal techniques will be proposed. The objectives of the proposed method is to learn DAPERs with less structured data, and to accelerate the learning process by querying graph databases. A first part of our experiments of this work are published in (El Abri et al., 2017).

This chapter is organized as follows. In Section 5.2.2 a method to identify an ER model as the metamodel of our graph database will be shown. Then, given this ER model and the graph database, we will focus in Section 5.2.3 on computing sufficient statistics. Section 5.3.5 presents experiments have been performed to evaluate the efficiency of the proposed method. Finally, Section 6.5 presents conclusion.

5.2 DAPER learning from graph database

The DAPER structure learning task comes to induce the dependency structure automatically from the complete observational training database. Databases contain information about which relationships do and do not hold among entities. To make this information accessible for learning tasks requires computing sufficient statistics. Therefore, counting is one of the key operations inside most relational probabilistic models learning process. However, most approaches use SQL queries performed in relational database and do not exploit any fast counting methods. This problem is solved with a new approach that performs traversal techniques over the data instances stored in graph databases. Our learning assumptions stipulate that:
First, the input graph database can be described via an ER schema that can be identified.
Second, the training data consists of a fully specified instance of the identified ER schema.

In this section, the ER schema identification strategy from a graph database is described. Then, the overall probabilistic dependencies learning process is presented.

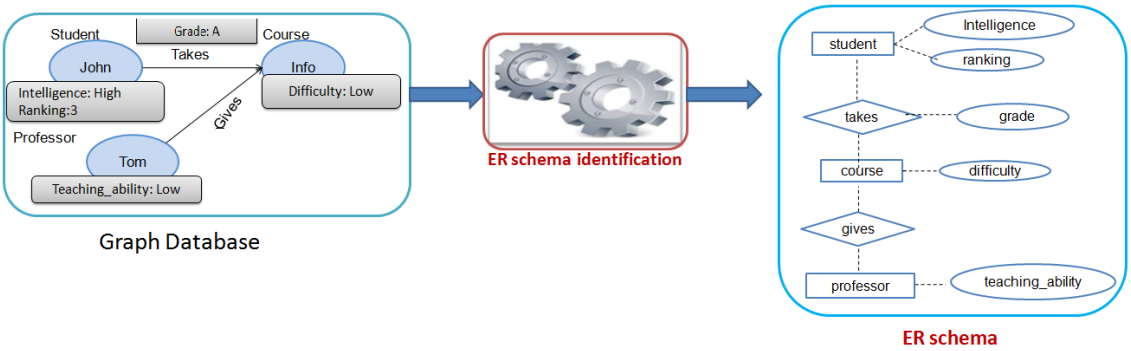


Figure 5.1 – Step I: ER schema identification from a graph database.

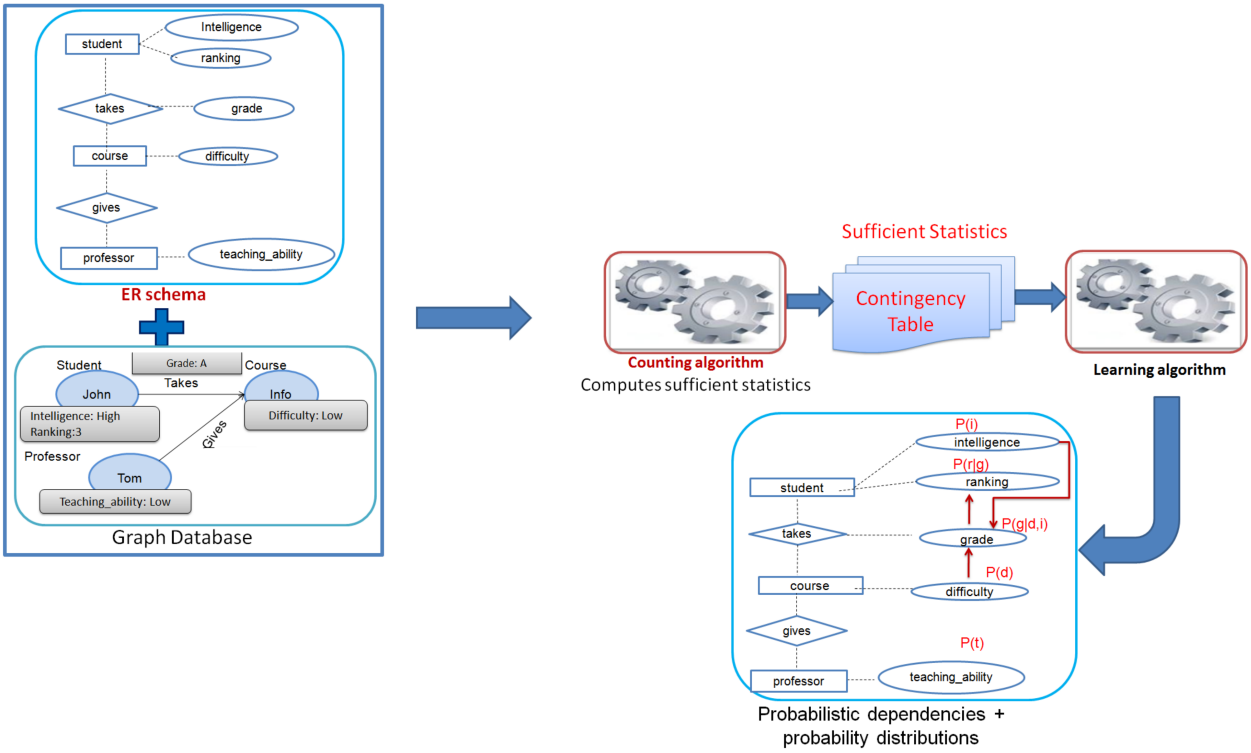


Figure 5.2 – Step II: Computation of sufficient statistics and DAPER learning using the ER schema already identified and the graph database instance.

5.2.1 Principle

A DAPER model is defined by an ER model and a probabilistic structure between its attributes. Therefore, to generate a DAPER model from graph database we have to resolve many steps. The first one corresponds to the identification of a meta-model given a graph database. The second one corresponds to the identification of the probabilistic dependencies given this meta-model and a graph database instance.

Our proposed approach involves the following components:

1. A generic algorithm for ER schema identification from an existing graph database instance as described in Figure 5.1. As graph databases are schema-free, so they can be "*clean*" databases (they do not contain exceptions) or "*noisy*" (they contain exceptions), so our approach has to be valid for both cases by managing exceptions. Or, exceptions are relationships which have the same type and connect between different types of nodes.

Algorithm 5 ER_Model_identification

Data: GDB: Graph Database instance
Result: $ER = (E, R, A, Dom(A))$
begin
  $ER \longleftarrow \{\}$
  I: $ER.E \longleftarrow$ Get_Entity_Classes(GDB)
  II: $ER.R \longleftarrow$ Get_Filtred_Relationship_Classes(GDB,λ)
  II: $ER.A \longleftarrow$ Get_Attribute_Classes_and_Domain(GDB,$ER.E, ER.R$)
  return ER

2. Counting methods to compute sufficient statistics from data stored in a (clean/noisy) graph database for probabilistic dependencies learning as described in Figure 5.2.

5.2.2 ER model identification

As we have mentioned in Chapter 4 (Section 4.8), (De Virgilio et al., 2014) and (Sequeda et al., 2012) proposed their insights to deal with schema-free property of graph database. However, these approaches demand the intervention of a designer and an additional effort of an expert to deal with all possible data consistencies. Therefore, to avoid all these problems and to reduce human intervention to make an analysis of a conceptual representation of the domain of interest, we propose our method which consists to start from a graph database instance to generate the meta-model. Our translation method between the graph database and the ER model is completely automatic and the designer does not need to introduce artificial concepts and relationships. Indeed, all the elements of the output ER diagram originate directly from concepts appearing in the input database. This ensure that our ER model is semantically close to the design layer of the database.

This section proposes an approach to identify the graph database schema from scratch, determining data connection with respect to the definition presented in Section 3.2. We need to identify the ER schema of a such graph database in order to be able to learn DAPER from graph database. However, as said in Section 4.5 , graph databases are unstructured data stores, so identifying a structured meta-model (ER model) from such a data store is a difficult task. Indeed, the database instance is not forced to conform a template in a rigid way. Several scenarios have been developed in this context.

Case I: GDB without exceptions

We will consider a first scenario where the graph database is partially structured without exceptions. We suppose that nodes are instances of entity classes and edges are instances of relationship classes, where the classes of the connected nodes are coherent with domain and range of the correspondent relationship. This class information can be modeled in graph database by assigning a type (label) to each node or edge. We have no specific constraint about the attributes defined for nodes or relationships.

We propose to define $\mathcal{E}$, set of entity classes in the ER model, as the set of unique types of the vertices in the graph databases. $\mathcal{E} = unique(\{type(v_i), \forall v_i \in \mathcal{V}\})$, each subset of nodes that have the same type will be stored in the same entity class.

In the same way, $\mathcal{R}$, set of relationship classes in the ER model, will be defined as the set of all the different edge types we met in the graph database. So that, each edge type is mapped to relationship class with referential constraints definition . $v_1 \longrightarrow v_2$ means that v_1 is the referencing relationship class and v_2 is the referenced one. thereby a pair of entity classes E_1 and E_2 for two nodes' types v_1 and v_2 are joined

in ER if v_1 and v_2 are connected in the graph database.

Finally, $\mathcal{A}(X)$, attribute classes for each entity (or relationship) class, will be defined as the set of all the different attributes we met for each node (resp. edge) of the corresponding class. We define the domain of values of each attribute by the set of values that it takes in the graph database. In the case of graph database nodes (edges) of the same type can have different properties, therefore we make the union of all properties of these nodes (edges) and we add them to the corresponding entity class (relationship class) in ER.

The proposed identification process is divided into three main steps:

1. Get_Entity_Classes returns all different node labels (types).
2. Get_Filtred_Relationship_Classes() returns relationships types.
3. Then, we get the set of attribute classes and their associated domains using Get_Attribute_Classes_and_Domain().

This process can be solved in a unique graph traversal where $\mathcal{E}$, $\mathcal{R}$ and $\mathcal{A}$ are iteratively built. The overall process is outlined in Algorithm 5.

Case II: GDB with exceptions

As graph databases are schema-free, so we can find some incoherent relationships which form exceptions in the database. Exceptions are relationships which have the same type and connect between different types of nodes. Thus the definition of the relationship classes in this case is more complex. In an usual ER model, a relationship class is defined for one unique pair of entity classes. In a graph database, a same typed relationship could be used to connect pairs of vertices with different pairs of types.

Considering this second scenario, we propose two methods to resolve this problem. The first one, we calculate the occurrence of all possible combinations for a relationship (edge). We consider that we don't know the range and domain of this relationship class. Then, we consider only the most frequent relationship by migrating it to $\mathcal{R}$ in ER. The rest of relationships are considered as exceptions. However, this solution has its limits, because it excludes some other relevant relationships that can be also considered particularly for large databases. In this case, we propose a second solution in which we set a threshold (generally given by an expert) to extract the set of relationship classes. If we have more than one accepted relationship with the same type, thus the method consists to rename the others. By inspiring ourselves from simple application of Inductive Logic Programming principles (Muggleton and Raedt, 1994), we propose to define a relationship class in $\mathcal{R}$ for each relationship label and each corresponding signature s that is enough represented for this label in the database, i.e. $N(type(v_i) - type(e_k) - type(v_j)) > \lambda N(type(e_k))$ where $N(.)$ is the number of occurrences in the graph database, and $\lambda \in [0, 1]$ is a user-defined threshold. For each signature s, we compare it with the parameter λ.

Example 5.2.1. *For instance, for 10 instances of the relation $takes$, our graph database could contain 9 instances with signature $student{-}takes{-}courses$ and 1 instance with signature $professor{-}takes{-}course$.*

With $N(takes) = 10$, $N(student - takes - course) = 9$, $N(professor - takes - course) = 1$ and $\lambda = .3$, we create one only $takes$ relationship in $\mathcal{R}$.

With $N(takes) = 10$, $N(student - takes - course) = 6$, $N(professor - takes - course) = 4$ and $\lambda = .3$, we create two relationships $takes_{sc}$ and $takes_{pc}$ in $\mathcal{R}$, one for each signature.

Step 2 of Algorithm 5 is more detailed in Algorithm 6 which corresponds to the exceptions management in order to handle the characteristics "flexibility and schema-less" of graph databases.

Algorithm 6 Get_Filtered_Relationship_Classes

Data: threshold λ:identification percentage of accepted Relationshipclass
GDB: Graph Database instance
Result: $FiltredR$: set of all accepted Relationshipclasses

```
begin
    FiltredR ⟵ {}
    RPath ⟵ Get_WiRaH_and_Occurence (GDB)  /* Get What is Related and How:
        returns for each relationship type the nodes that connect and its
        occurrence                                                        */
    Total ⟵ {}
    Nbroccurence ⟵ {}
    for each r ∈ RPath do
        Total [r.type]+= occurence(r)
        if first iter then
            Nbroccurence[r.type] ⟵ {}
        /* To initialize the list of occurrence of each relationship type
           */
        Nbroccurence[r.type] ⟵ Nbroccurence[r.type] ∪ occurence(r)
    for each r ∈ RPath do
        Max ⟵ λ
        BestR ⟵ Null
        AcceptedR ⟵ {}
        for each n ∈ [r.type] do
            Threshold r ⟵ n / Total[r.type]
            if Threshold r ≥ λ then
                if Threshold r ≥ Max then
                    Max ⟵Threshold r
                    BestR ⟵ r
                AcceptedR ⟵ AcceptedR ∪ r /* AcceptedR for each type      */
        if Threshold r =0 then
            FiltredR ⟵ FiltredR ∪ BestR
        else
            for each r ∈ AcceptedR do
                FiltredR ⟵ FiltredR ∪ Rename(r)
        return FiltredR
```

5.2.3 Probabilistic dependencies identification

5.2.3.1 Principle

As seen in Section 3.3, a DAPER model is defined by an ER model and a probabilistic structure between its attributes. By considering than we can already identify the ER model from a graph database 5.2.2, we are now interested in the identification of the probabilistic dependencies given an ER model and a graph database. As seen in Section 3.5, the learning task relies on scoring functions or independence measurements estimated from sufficient statistics extracted from the database.

All the algorithms proposed for DAPER structure learning relies on scoring function or independence measurement based on the estimation of sufficient statistics (or contingency tables) $N(X.A, pa(X.A))$,

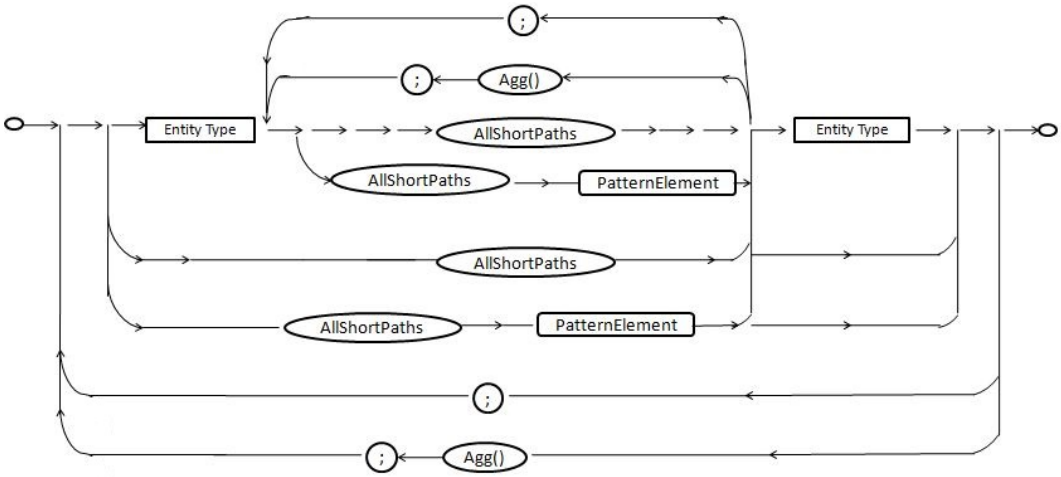


Figure 5.3 – Pattern element.

where each parent in $pa(X.A)$ have general form of $\gamma(X.K.B)$ when referencing an attribute B related to the starting class X with a slot chain K, and a possible aggregation function γ. These sufficient statistics are instantiation counts for conjunctive queries represented in contingency tables. So that one of the fundamental operations inside most PRMs is counting. For a relational database, the computation of this table can be done by executing SQL joins of the database tables (Friedman et al., 1999a). In fact, to calculate a table of conditional probability requires computing sufficient statistics of its attributes given by a list of slot chains. However, a table join is not feasible for a long length of slot chains and large collection of relationship classes. In fact, queries become more complex because we have to join large number of entities to specify the mapping between a foreign key in one table and the associated primary key.

Our aims is to resolve the problem of computing counts (stored in contingency table) using a graph traversal technique performed in graph database which takes into account only the required data, without needing to perform complex grouping operations on the entire data set.

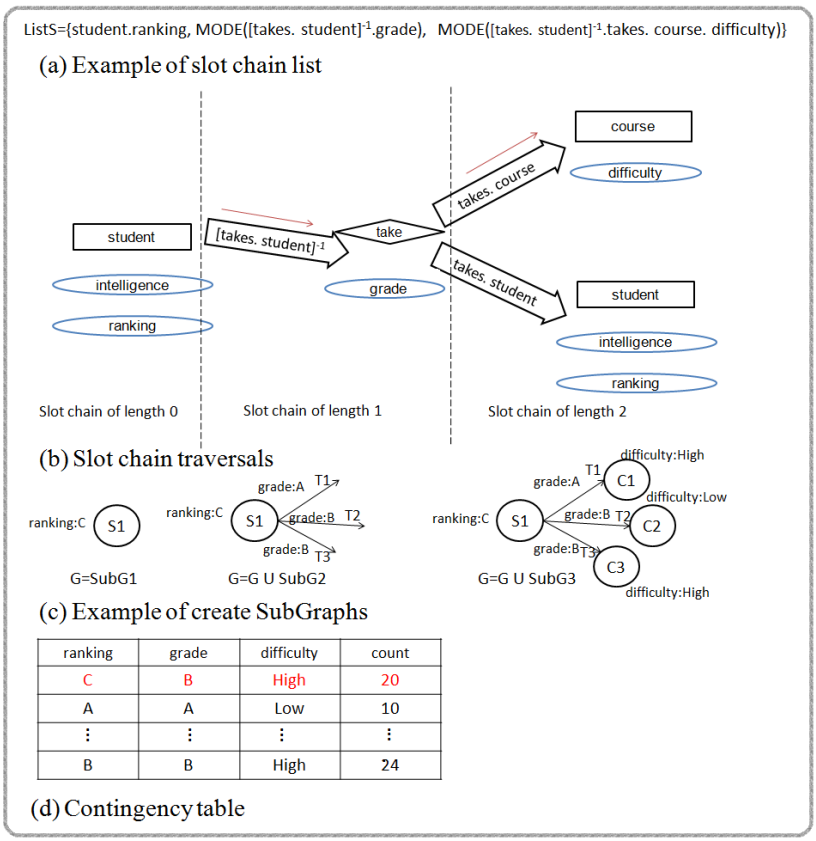
Our method consists to explore a slot chain list and construct the corresponding graph G.

5.2.3.2 Sufficient statistics computation

Sufficient statistics computation consists to perform counting given a slot chain list. We observe that this is equivalent to counting subgraphs in a network while satisfying certain configurations of attributes. The given graph structure becomes the constraint on the counting task. Thus, counting requires simply retrieving all the subgraphs matching the motif or the pattern induced by a slot chain list.

We parse a given slot chain into a query graph as shown in Figure 5.3. Our method consists to break down the slot chain, so that each pair of references slot corresponds to a simple shortest path in our graph database. As a slot chain is a set of reference slots or inverse slots $[X_1.\rho 1].[X_2.\rho 2].[X_i.\rho_i]...[X_n.\rho_n]$ and each class X_i corresponds to an entity type or a relation type in the graph database, we define the slot chain $[X_1.\rho 1].[X_2.\rho 2]$ as the shortest path between X_1 and X_2.

The structure of the shortest path starts and finishes always by a node for which we define the entity type. Or, the slot chain can start or finish by a relation type, for this case our matching path will start or finish by an empty node. on other side, we can find a slot chain in which we have duplicated classes X_i. Thus, we have to re-lookup data that we already traverse and we have additional paths. However, subgraph matching read in one direction. In other words, a single match statement that involves multiple paths will automatically filter out duplicate entities. So, to re-lookup some data, we break long match statement into



ranking	grade	difficulty	count
C	B	High	20
A	A	Low	10
⋮	⋮	⋮	⋮
B	B	High	24

Figure 5.4 – Illustration of counting sufficient statistics process for the attribute *student.ranking* and its *parents*.

multiple match statements such that not all relationships are matched in the same statement. In other words, we construct pattern elements and this does not change the performance of our methods. All these situations are shown in Figure 5.3.

Example 5.2.2. *For example, the computation of the conditional probability distribution of* $P(student.ranking \mid MODE(student.student^{-1}.grade), MODE(student.student^{-1}.course.difficulty))$ *requires the computation of the contingency table* $N(student.ranking, MODE(student.student^{-1}.grade), MODE(student.student^{-1}.course.difficultye))$ *given by the slot chain list shown in Figure 5.4(a). The corresponding contingency table shown in Figure 5.4(d).*

For a relational database, the computation of these sufficient statistics can be done by executing SQL joins of the database tables (Getoor, 2001).

```
SELECT ranking, MODE(grade), MODE(difficulty), count(*)
FROM  student, takes, course
JOIN ON student.id = takes.student-id
JOIN  ON takes.course-id=course.id
GROUP BY ranking, grade, difficulty
```

For a graph database, the same computation can be seen as an exploration of the graph database, where we count occurrences of subgraphs patterns corresponding to the variables we want to count, without performing complex grouping operations on the entire data set. In Neo4J graph database and using Cypher language the previous query will be written simpler as follow:

```
MATCH (a:student)-[b:takes] --> (c:course)
RETURN a.ranking, MODE(b.grade), MODE(c.difficulty)
count(*)
```

In this example the slot chain given in Figure 5.4(a) corresponds to a simple short path because it start and finish by two different entity classes through just one relationship class.

Example 5.2.3. *Another example in which the slot chain start by a relationship class, the computation of the conditional probability distribution of* $P(takes.grade \mid takes.student.intelligence, takes.course.difficulty)$ *requires the computation of the contingency table* $N(takes.grade, takes.course.difficulty, takes.student.intelligence)$.

For a relational database, the computation of these sufficient statistics can be done as follow:

```
SELECT grade, difficulty, intelligence , count(*)
FROM takes, course, student
JOIN ON student.id = takes.student-id
JOIN ON takes.course-id=course.id
GROUP BY grade, difficulty, intelligence
```

For a graph database, the same computation can be done as follow:

```
MATCH ()-[a:takes] --> (b:course)
OPTIONAL MATCH (c:course)<--[d:takes]-(e:student)
RETURN a.grade, b.difficulty, e.intelligence
count(*)
```

In this example the given slot chain consists to construct a pattern element.

Example 5.2.4. *As another more complex example, estimating the conditional probability distribution* $P(student.ranking \mid MODE(student.student^{-1}.grade), student.student^{-1}.course.course^{-1}.professor.teach_ability)$ *requires the computation of the contingency table* $N(student.ranking, MODE(student.student^{-1}.grade), student.student^{-1}.course.course^{-1}.professor.teach_ability)$.
For Neo4j graph database, the computation of this table can be done by executing the following Cypher query:

```
MATCH (a:student)-[b:takes] --> (c:course)
OPTIONAL MATCH (a:student)-[d:takes] --> (e:course)
OPTIONAL MATCH (e:course)<--[f:gives]-(g:professor)
RETURN  a.ranking, MODE(b.grade),g.teaching_ability
count(*)
```

However for relational database, this same query is done by executing the following SQL joins:

```
SELECT ranking, MODE(grade), teaching_ability, count(*)
FROM takes, student, course, gives, professor
JOIN takes ON student.id = takes.student-id
JOIN course ON takes.course-id=course.id
JOIN gives ON gives.course-id=course.id
JOIN professor ON professor.id=gives.professor.id
GROUP BY ranking, MODE(grade), teaching_ability
```

Algorithm 9 presents the creation of a graph G=$\{SubG_1, SubG_2, ..., SubG_i\}_{i=1}^{n}$, with n is the number of given slot chains. Algorithm 8 presents the graph G traversal. The key idea is to use this graph to create the slot chain traversals as well as paths traversed in the graph G. The execution of these paths is then performed to estimate the counts of attributes as it is detailed in Algorithm 7.

5.3 Experiments

This section is dedicated to the empirical validation of our theoretical contributions. We explain our experiment protocols, for which we will specify used algorithms for the structure learning experiments, used networks as well as the set of evaluation metrics. Then we report the experimental results and interpretations.

Algorithm 7 Get_Counts

Data: ListS: Slot chain list
FiltredIndice= array[]
Result: Counts= array[]
begin
  Configurations C ⟵ Get_All_Configurations(ListS)
  Path ⟵Create_Slot_chain_Traversals(ListS,C,FiltredIndice)
  result ⟵ Execute(Path) i ⟵ 0 **for** *each* $r \in result$ **do**
    Counts[i]⟵ r
    i ⟵ i+1
  return Counts

Algorithm 8 Create_Slot_chain_Traversals

Data: ListS: Slot chain list
C:configurations
FiltredIndice= array[]
Result: Paths: Slot chain Traversals
begin
  Paths ⟵ {}
  for *each* $s \in ListS$ **do**
    Paths ⟵ Paths ∪ Create_SubGraphs(s,C,FiltredIndice)
  return Paths

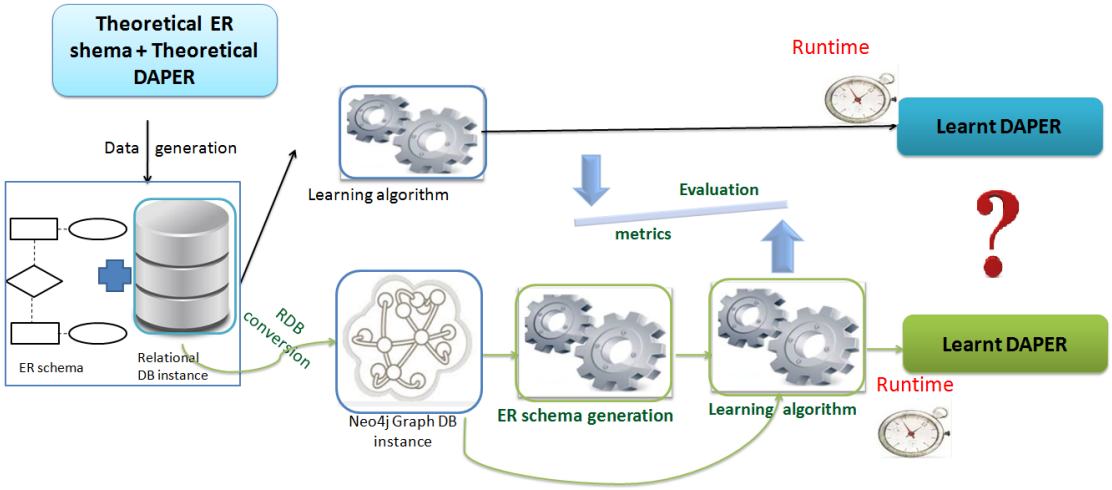


Figure 5.5 – Evaluation process of a DAPER structure learning from usual relational database versus graph database.

5.3.1 Experiment methodology

The primary objective of this experimental study is to evaluate the performance of our proposed method. We had performed an empirical study using small then large datasets to assess the scalability of our algorithm. Then we have evaluated our method by varying the length of the used slot chain k_{max} to illustrate the impact of this parameter on both the running time and the reconstruction quality. The first two experiments are performed on structured databases. However, as we have mention in Chapter 4, graph databases are generally schema-free data store and they do not have a rigid structure defined beforehand. Therefore, we consider a second scenario in which we use partially structured databases. In these experiments, we have used naive synthetic datasetes that are generated by applying the naive sampling algorithm (Ben Ishak et al.,

Algorithm 9 Create_SubGraphs

Data: S : slot chain
C:configurations
FiltredIndice.
Result: G: Graph of Slot_chain_Traversals.

```
begin
    parts ⟵ Get_Parts_of_Slot_chain(S)
    G ⟵ {}
    SubGraph SubG ⟵ {}
    for each p ∈ parts do
        StartwithEntityClass E_S ⟵ True
        if p.reversed then
            E_S ⟵ False
            Start ⟵ p.RefSlotClass /* Relationship class */
            End ⟵ p.RefSlotName /* Entity class */
        else
            End ⟵ p.RefSlotClass
            Start ⟵ p.RefSlotName
        if Start ∈ SubG.E or Start ∈ SubG.R then
            G ⟵ G ∪ SubG
            G ⟵ {}
        if E_S then
            Start Node N_S ⟵ Start
            End Node N_E ⟵ {}
            R ⟵ End
            New Triple ℑ ⟵ ⟨N_S, R, N_E⟩
            Add ℑ to SubG
        else
            N_S ⟵ {}
            N_E ⟵ End
            R ⟵ Start
            New Triple ℑ ⟵ ⟨N_S, R, N_E⟩
            Add ℑ to SubG
    if SubG is not empty then
        G ⟵ G ∪ SubG
    return G
```

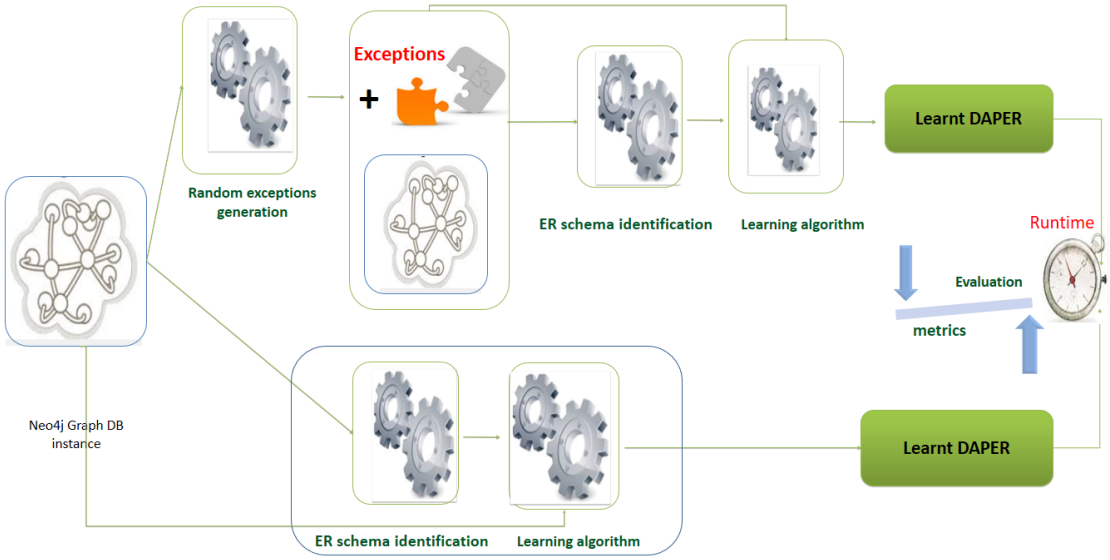


Figure 5.6 – Evaluation process of a DAPER structure learning from "noisy" graph database .

2016). In order to evaluate the performance of the proposed approach when it will be used in real world situations, we had generated more realistic datasets by applying k-partite sampling algorithm (Chulyadyo and Leray, 2018).

-Experimental protocol I: small datasets

We performed our first experiment using naive small datasets, in order to compare DAPERs learned from the same data but stored in two different ways (relational databases and graph databases) as described in Figure 5.5.

- Experimental protocol II: scalability and k_{max} for learning algorithm

We run this experiment increasing the number of instances to justify our choice of adopting graph databases for DAPER learning.

Then we have performed our experiment by varying the length of the used slot chain k_{max} to learn our DAPER models. In fact, Any learning algorithm will be unable to find some dependencies if the k_{max} value is lower than the length of slot chains of some dependencies. Therefore, we have chosen a simple value (1) and maximum value (4) of k_{max} as we have fixed the same value (4) for some generated models to illustrate the impact of this parameter on both the running time and the quality of reconstruction.

-Experimental protocol III: learning from noisy datasets

We generated random exceptions in some datasets and find the impact of these exceptions generation on the graph structure learned from these "noisy" datasets as described in Figure 5.6.

-Experimental protocol IV: learning from k-partite datasets

The benchmarking method used to generate our datasets in this experimental protocol is different to the one used in protocols I,II and III. In the first experiments naive datasets are used in which data are strongly connected but not near from the real word situations. Therefore, in order to ensure the efficiency of the proposed method under real situations, a k-partite method is then used to generate new datasets. k-partite method consists to sample more realistic and dispersed datasets in which data are not very connected between each other. Since k-partite graph is graph whose vertices can be partitioned into k disjoint sets so that there is no edge between any two vertices within the same set. The entityclass instances of the ER skeleton

generated by this method are related under control over the degree of their connectivities.

5.3.2 Networks and Datasets

Unlike standard Bayesian networks, where several benchmarks or golden networks are available to perform experiments, there is no such models defined in the context of DAPER. In the other side, comparing DAPER learned from relational databases and DAPER learned from graph databases is required. Consequently, the generating process defined by (Ben Ishak et al., 2016) has been used to generate theoretical DAPERs (ER models and probabilistic dependencies) from which naive skeletons where randomly generated. Then, we have sampled relational database instances from these probabilistic models. The 5 DAPERs have been generated whose characteristics are described in table 5.1.

	$\mid \mathcal{E} \mid$	$\mid \mathcal{R} \mid$	$\mid \mathcal{A} \mid$	$\mid \mathcal{V} \mid$	$kmax :\mid \mathcal{K} \mid$
DAPER1	2	1	8 (2-3)	2-3	1 : 2-1
DAPER2	5	4	16(2-5)	2-3	1 : 9-4
DAPER3	8	7	32 (1-5)	2-4	4 : 2-4-12-5-6
DAPER'3	8	7	52 (1-8)	2-5	4 : 3-2-19-11-18
DAPER"3	8	7	107 (1-11)	2-7	4 : 5-10-34-33-15

Table 5.1 – Characteristics of DAPER networks used for naive datasets generation. Values in parenthesis are min/max values per class. Values in the last column correspond to the maximum length of slot chain $kmax$ and the number of dependencies for each slot chain length (from 0 to k_{max}).

Then relational databases have been sampled with 500, 1000, 3000 and 5000 instances. This sampling process is repeated 10 times for each DAPER. Our relational databases are managed with PostgreSQL 9.3.

Each relational dataset already generated is converted to a graph database instance managed with Neo4J 3.1.0 following generating process resumed here:

- Each table tuple for each entity class becomes a node
  - Table name as node type
  - Attributes turn into nodes' properties
- Each table tuple for each relationship class becomes relationship
  - Join tables are transformed into relationships
  - Foreign keys are removed afterwards
  - Attributes on those tables become relationships' properties
- Clean-up rules
  - Remove technical primary keys (auto-incrementing PKs)
  - Add unique constraints for PKs
  - Add indexes for frequent lookup attributes

DAPER3 has also be used to generate some additional graph databases. Some exceptions have been added in the 10 databases of size 3000 in order to perturb the underlying ER model. The signature of relationship instances has been replaced by another signature not conform with the underlying ER model. The percentage of exceptions α is varied as 5%, 10%, 30% and 50%. A large database is also generated from DAPER3 with 500.000 instances.

Finally, (Chulyadyo and Leray, 2018) method has been used to generate 4 DAPER (ER models and probabilistic dependencies) from which 5 k-partite skeletons where generated. Then, we have sampled relational database instances from these schemas. Also, naive method is used to generate other 4 naive skeletons from the same 5 DAPERs described in table 5.2.

	$\mid \mathcal{E} \mid$	$\mid \mathcal{R} \mid$	$\mid \mathcal{A} \mid$	$\mid \mathcal{V} \mid$	$kmax : \mid \mathcal{K} \mid$
DAPER4	2	1	8	2-3	2
DAPER5	3	2	16	2-3	2
DAPER6	4	3	32	2-4	2
DAPER7	5	4	76	2-5	2

Table 5.2 – Characteristics of DAPER networks used for respectively naive and k-partite datasets generation.

Then a relational database has been sampled, with 1000, 5000, 10000 and 15000 instances. These datasets are converted to graph databases.

5.3.3 Algorithms

The problem of structure learning is considered by applying the reference algorithm, Relational Greedy Search (RGS), already implemented in PILGRIM Relational C++ Library [1]. RGS uses as an input a relational schema (directly imported from SQL database, or identified from our graph database as described in section 5.2.2). The parameters used for this process are :

- λ : identification threshold for ER identification, which is:
  - fixed to 10% for α=5%, 10%, 30% and 50%. A low value of λ has been chosen to see the impact of exceptions by easily accepting creation of relationships.
  - varied as 10%, 30% and 50% for α=50%
- k_{max} : maximum possible slot chain length during greedy search (set to 1 for experiments with databases generated from DAPER1 and DAPER2, set to 1 then 4 for DAPER3, DAPER'3 and DAPER"3). This parameter controls the complexity of the algorithm by defining "how far" (in the relational schema) can be some dependent attributes. Here, simply the same horizon has been chosen than the one used for generating the DAPERs, except for DAPER3, where a simple value (1) and an longer one (4) is used to illustrate the impact of this parameter on both the running time and the quality of reconstruction.

Let us denote RGS_postgres the usual RGS algorithm, working with postgreSQL databases (without exceptions), and RGS_neo4j the same algorithm, working with Neo4J graph databases (with or without exceptions) with an initial ER identification.

5.3.4 Evaluation metrics

Evaluation metrics of DAPER structure learning algorithms can concern either the execution speed of the proposed algorithm or the quality of the graph reconstruction.

Evaluation with respect to the execution time will further depend on the used database management system quality.

The quality of the learned graph structure is measured using Precision, Recall and F-score (Maier et al., 2013a). Let S_{True} be the dependency structure of the theoretical standard PRM and $S_{Learned}$ be

1. http://pilgrim.univ-nantes.fr

the dependency structure of the learned PRM.

Precision: The ratio of the number of relevant dependencies retrieved to the total number of relevant and irrelevant dependencies retrieved in the learned PRM dependency structure $S_{Learned}$. Relevant dependencies are those that are present in the true model: same edge, same slot chain and same aggregator.

$$Precision = \frac{N_r}{N_l} \tag{5.1}$$

Recall: The ratio of the number of relevant dependencies retrieved to the total number of relevant dependencies in the true PRM dependency structure S_{True}, which is generated using the random generation process.

$$Recall = \frac{N_r}{N_t} \tag{5.2}$$

where N_t the number of dependencies in S_{True}, N_l the number of dependencies in $S_{Learned}$, and N_r is the number of relevant dependencies retrieved in $S_{Learned}$.
F-score: is used to provide a weighted average of both precision and recall.

$$F_score = \frac{2 * Precision * Recall}{Precision + Recall} \tag{5.3}$$

A Soft_Precision and Soft_Recall metrics are used, soft versions for precision and recall, refined by (Ben Ishak, 2015) by adding penalization for wrong slot chains and wrong aggregators that can be found in a discovered dependency.

Let Nb_{true} be the number of dependencies in S_{True}, and $Nb_{learned}$ be the number of dependencies in $S_{Learned}$.

$$Soft_Precision = \frac{\sum_{i=0}^{Nb_{learned}} \omega_i}{Nb_{learned}} \tag{5.4}$$

$$Soft_Recall = \frac{\sum_{i=0}^{Nb_{learned}} \omega_i}{Nb_{true}} \tag{5.5}$$

where relevant dependencies are the ones that have the same edge, slot chain and aggregator as in the theoretical standard model.

$$\omega_i = \begin{cases} 1; & for\ relevant\ dependencies \\ 0; & for\ reversed\ edges\ and\ the\ edges\ not\ present\ in\ the\ theoretical\ standard\ model \\ 1-\psi; & when\ the\ edges\ match\ but\ slot\ chains\ and/or\ aggregators\ do\ not \end{cases}$$

ψ is the arithmetic mean of the penalization for wrong slot chains (α) and that for wrong aggregators (β) in true and learned dependencies.

$$\psi = \frac{\alpha + \beta}{2} \tag{5.6}$$

$$\alpha = 1 - \frac{MaxL_c}{Max(L_t, L_l)} \tag{5.7}$$

where $MaxL_c$ is the longest common sub slot chain in true and learned dependencies, L_t is the length of the true dependency, L_l is the length of the learned dependency and $\beta \in [0, 1]$ is the user-defined cost for penalizing wrong aggregators.

When the graph databases include exceptions, the quality of the ER schema identification is estimated by computing Precision$_{ER}$ and Recall$_{ER}$ between the identified ER schema and the theoretical one. For this purpose, precision and recall for ER schema comparison are defined in the following way. Let ER_{True} and $ER_{Learned}$ be the ER schema of the theoretical DAPER and the learned DAPER respectively, which are obtained considering the direction of relationship classes which link between entity classes. Let NR_{True} and $NR_{Learned}$ be the number of relationship classes in ER_{True} and $ER_{Learned}$ respectively.

$$Precision_{ER} = \frac{Relevant\ NR_{Learned}\ in\ ER_{Learned}}{NR_{Learned}\ in\ ER_{Learned}} \tag{5.8}$$

$$Recall_{ER} = \frac{Relevant\ NR_{Learned}\ in\ ER_{Learned}}{NR\ in\ ER_{True}} \tag{5.9}$$

A relationship class in $ER_{Learned}$ is relevant if it links between the same entity classes in ER_{True}, with considering the direction of the relationship class (from $entityclass_i$ to $entityclass_j$).

5.3.5 Results and interpretations

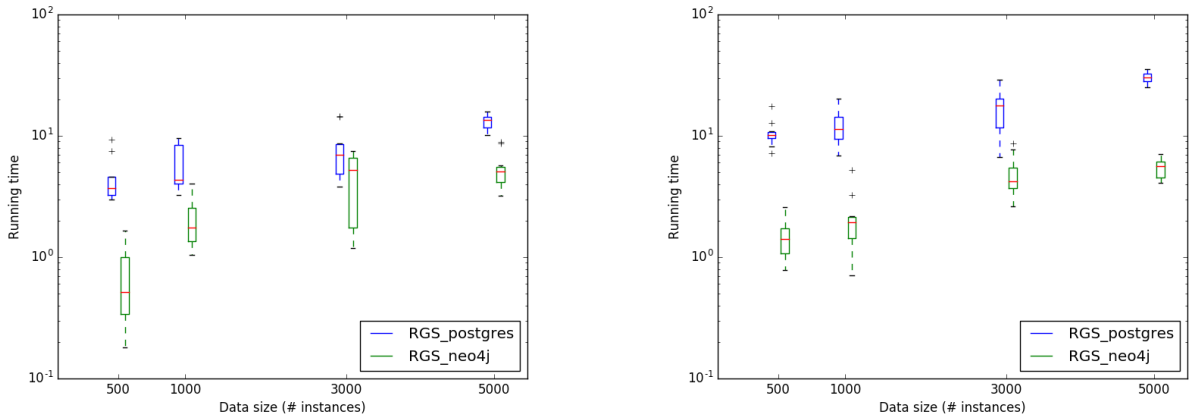
5.3.5.1 Experimental protocol I: Results and interpretation

As an initial result not described here, the RGS_postgres and RGS_neo4j converged to the same learned ER schema and probabilistic dependency structure (for databases without exceptions) are successfully checked. This is logical because both compute the same sufficient statistics from the same data, but stored in two different ways.

Data size	500	1000	3000	5000
Precision	0.22±0.11	0.28±0.1	0.35±0.11	0.49±0.09
Recall	0.29±0.13	0.38±0.1	0.40±0.05	0.56±0.07
F-score	0.27±0.11	0.30±0.08	0.37±0.13	0.51±0.07
s-Precision	0.17±0.04	0.23±0.06	0.29±0.12	0.31±0.04
s-Recall	0.19±0.03	0.26±0.10	0.32±0.06	0.44±0.04
s-F-score	0.18±0.04	0.25±0.08	0.31±0.03	0.35±0.06

Data size	500	1000	3000	5000
Precision	0.42±0.08	0.49±0.1	0.76±0.06	0.84±0.05
Recall	0.36±0.1	0.43±0.09	0.61±0.04	0.71±0.06
F-score	0.38±0.1	0.45±0.07	0.70±0.06	0.81±0.08
s-Precision	0.31±0.03	0.39±0.03	0.69±0.08	0.74±0.03
s-Recall	0.38±0.07	0.44±0.06	0.74±0.06	0.79±0.06
s-F-score	0.34±0.04	0.43±0.11	0.70±0.05	0.75±0.05

Table 5.3 – Average ± standard deviation of reconstruction metrics (Precision, Recall, F-score, s-Precision, s-Recall and s-F-score) when executing RGS with k_{max}=1 for databases generated from DAPER1 (top) and DAPER2 (bottom).



(a) DAPER1 (b) DAPER2

Figure 5.7 – Boxplot of running time (in seconds and log scale) for each sample size, when executing RGS_postgres and RGS_neo4j with k_{max}=1 and databases generated from DAPER1 (a) and DAPER2 (b).

Table 5.3 depicts the quality reconstruction (identical for RGS_postgres and RGS_neo4j) for our 2 first DAPERs, when executing RGS with k_{max}=1. We can observe that increasing the number of instances of datasets improves the quality of reconstruction of our structures.
Figure 5.7 represents running time in the same context. With a very small model such as DAPER1, the running time is almost divided by 2 when accessing the graph database. With a small model such as DAPER2, the running time is almost divided by 10.

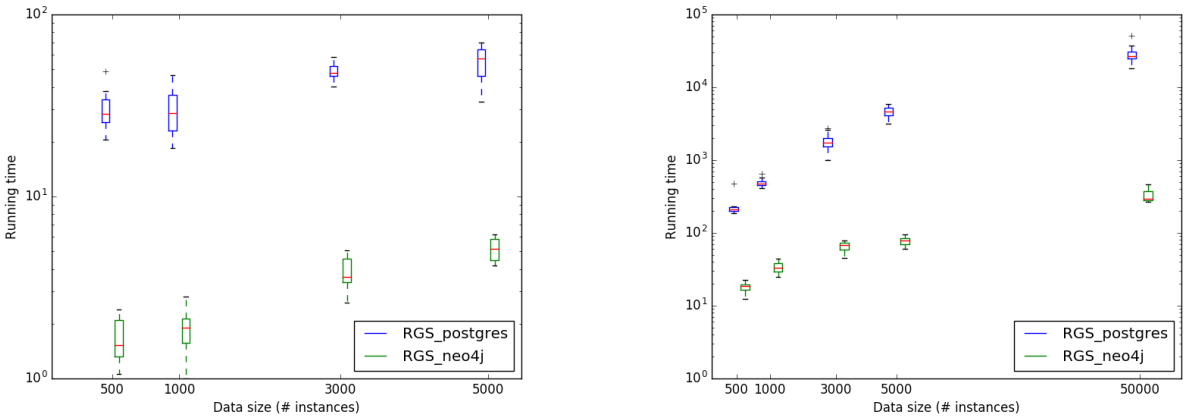
5.3.5.2 Experimental protocol II: Results and interpretation

Data size	500	1000	3000	5000
Precision	0.39±0.11	0.36±0.10	0.47±0.08	0.52±0.10
Recall	0.44±0.08	0.42±0.06	0.57±0.09	0.58±0.08
F-score	0.40±0.05	0.39±0.12	0.52±0.10	0.55±0.08
s-Precision	0.22±0.02	0.26±0.03	0.27±0.10	0.29±0.06
s-Recall	0.28±0.05	0.29±0.13	0.30±0.04	0.36±0.05
s-F-score	0.24±0.06	0.27±0.10	0.29±0.05	0.35±0.08

Data size	500	1000	3000	5000	500000
Precision	0.45±0.05	0.47±0.07	0.56±0.10	0.62±0.08	0.68±0.05
Recall	0.48±0.11	0.53±0.07	0.64±0.07	0.73±0.06	0.78±0.04
F-score	0.46±0.12	0.50±0.10	0.58±0.06	0.68±0.09	0.72±0.05
s-Precision	0.39±0.06	0.42±0.10	0.51±0.13	0.60±0.02	0.62±0.06
s-Recall	0.41±0.04	0.47±0.10	0.59±0.08	0.63±0.06	0.66±0.05
s-F-score	0.40±0.13	0.46±0.08	0.53±0.06	0.61±0.12	0.64±0.04

Table 5.4 – Average ± standard deviation of reconstruction metrics (Precision, Recall, F-score, s-Precision, s-Recall and s-F-score) for each sample size when executing RGS with $k_{max} = 1$ (top) and k_{max}=4 (bottom) for databases generated from DAPER3.

Table 5.4 depicts the quality reconstruction (identical for RGS_postgres and RGS_neo4j) for DAPER3, when executing RGS with k_{max}=1 and with a bigger search space (k_{max}=4). Figures 5.8 represents



(a) k_{max}=1 (b) k_{max}=4

Figure 5.8 – Boxplot of running time (in seconds and log scale) for each sample size, when executing RGS with k_{max}=1 (a) and k_{max}=4 (b) and databases generated from DAPER3.

running time in the same context, in log scale. We can observe that increasing the slot chain length improves the quality of reconstruction, but increases the running time. The deeper we go, the better results we get in term of Precision, Recall and F-score. By increasing the search space, we increase the number of times the data is accessed, and Figure 5.8 shows us again the interest of using a graph database instead of a relational one, with a running time divided by a factor greater than 10. This is even more impressive in Figure 5.8(b) with 500.000 instances, where the running time drops drastically transforming the structure learning process into a feasible task.

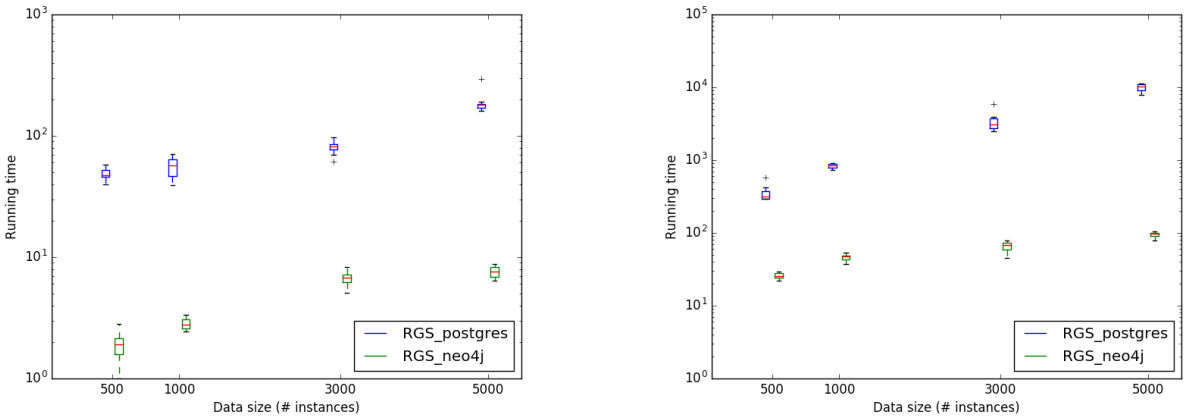
Data size	500	1000	3000	5000
Precision	0.41±0.08	0.45±0.04	0.49±0.03	0.54±0.08
Recall	0.47±0.06	0.51±0.07	0.55±0.07	0.57±0.10
F-score	0.42±0.11	0.46±0.12	0.50±0.11	0.55±0.08
s-Precision	0.29±0.03	0.29±0.13	0.33±0.02	0.34±0.05
s-Recall	0.30±0.04	0.33±0.09	0.37±0.08	0.39±0.05
s-F-score	0.31±0.03	0.31±0.11	0.35±0.07	0.36±0.05

Data size	500	1000	3000	5000
Precision	0.47±0.08	0.51±0.03	0.63±0.10	0.67±0.04
Recall	0.48±0.12	0.53±0.05	0.65±0.10	0.71±0.10
F-score	0.47±0.09	0.50±0.12	0.61±0.11	0.69±0.08
s-Precision	0.39±0.02	0.44±0.06	0.58±0.12	0.61±0.07
s-Recall	0.43±0.03	0.47±0.12	0.59±0.05	0.66±0.03
s-F-score	0.41±0.02	0.45±0.11	0.57±0.07	0.64±0.11

Table 5.5 – Average ± standard deviation of reconstruction metrics for each sample size when executing RGS with $k_{max} = 1$ (top) and k_{max}=4 (bottom) for databases generated from DAPER'3.

Tables 5.5 and 5.6 depict the quality reconstruction for DAPER'3 and DAPER"3 respectively with kmax=1 and kmax=4.

We notice from Tables 5.4, 5.5 and 5.6 that for kmax=1 the precision values are about 50%. This is explained by the fact that DAPER3, DAPER'3 and DAPER"3 contain a most number of dependencies



(a) k_{max}=1 (b) k_{max}=4

Figure 5.9 – Boxplot of running time (in seconds and log scale) for each sample size, when executing RGS with k_{max}=1 (a) and k_{max}=4 (b) and databases generated from DAPER'3.

for slot chain length=2 (12, 19 and 34 respectively, of Table 6.2).

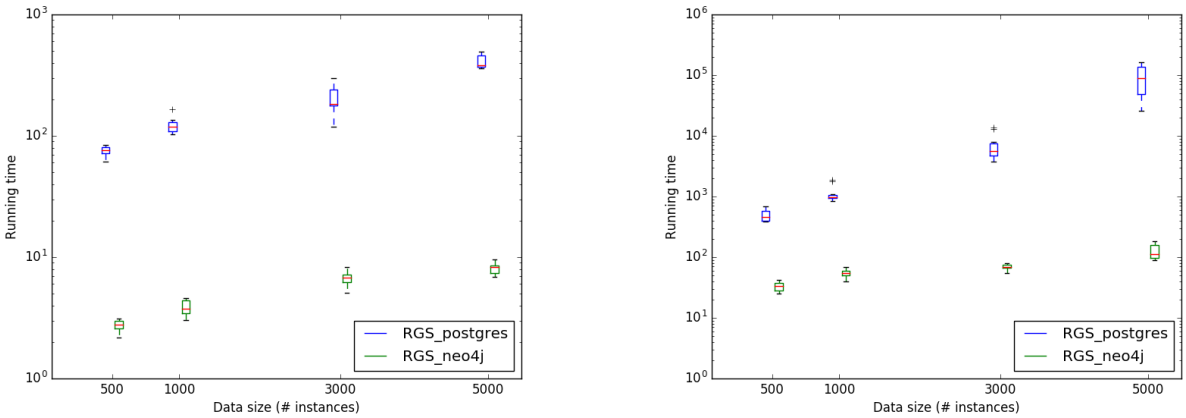
We observe also that increasing the number of attributes for DAPER'3 and DAPER"3 (52 and 107 respectively, of Table 6.2) improves the quality reconstruction.

Data size	500	1000	3000	5000
Precision	0.43±0.05	0.39±0.10	0.47±0.10	0.51±0.05
Recall	0.49±0.07	0.46±0.06	0.53±0.07	0.57±0.05
F-score	0.46±0.06	0.42±0.09	0.49±0.03	0.54±0.03
s-Precision	0.29±0.12	0.27±0.04	0.33±0.10	0.37±0.03
s-Recall	0.34±0.08	0.33±0.02	0.36±0.12	0.40±0.04
s-F-score	0.30±0.06	0.29±0.05	0.35±0.10	0.38±0.05

Data size	500	1000	3000	5000
Precision	0.47±0.06	0.48±0.11	0.66±0.10	0.70±0.05
Recall	0.50±0.04	0.54±0.13	0.69±0.08	0.73±0.05
F-score	0.48±0.05	0.50±0.09	0.63±0.03	0.71±0.03
s-Precision	0.41±0.10	0.43±0.07	0.53±0.09	0.68±0.03
s-Recall	0.42±0.06	0.46±0.02	0.56±0.13	0.70±0.10
s-F-score	0.41±0.02	0.44±0.01	0.54±0.06	0.68±0.05

Table 5.6 – Average ± standard deviation of reconstruction metrics for each sample size when executing RGS with $k_{max} = 1$ (top) and k_{max}=4 (bottom) for databases generated from DAPER"3.

Figures 5.9 and 5.10 represent running time for DAPER'3 and DAPER"3 respectively in the same context, in log scale time. We notice that increasing the number of attributes of our DAPER does not have a bad impact when we use a graph database. However, with a relational database the running time is multiplied by a factor greater than 10. This is more impressive in Figure 5.10(b) in which the running time for RGS_postgres exceeds 10^5 seconds. Or for RGS_neo4j always it never exceeds 10^2 seconds. This proves once again the interest of using graph database for PRMs learning.



(a) k_{max}=1 (b) k_{max}=4

Figure 5.10 – Boxplot of running time (in seconds and log scale) for each sample size, when executing RGS with k_{max}=1 (a) and k_{max}=4 (b) and databases generated from DAPER"3.

5.3.5.3 Experimental protocol III: Results and interpretation

α	0%	5%	10%	30%	50%
Prec$_{ER}$	1.00±0.00	1.00±0.00	1.00±0.00	0.87±0.03	0.67±0.05
Rec$_{ER}$	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00
Prec	0.47±0.08	0.47±0.08	0.47±0.08	0.44±0.05	0.41±0.06
Rec	0.62±0.09	0.62±0.09	0.62±0.09	0.59±0.08	0.57±0.07
F-score	0.57±0.10	0.57±0.10	0.57±0.10	0.55±0.04	0.49±0.09

Table 5.7 – Average ± standard deviation of reconstruction metrics for DAPER3 for a particular size (3000 instances) with a percentage α of exceptions in the relationship instances for λ fixed to 10% .

λ	0%	10%	30%	50%
Prec$_{ER}$	0.27±0.11	0.67±0.05	0.73±0.04	0.81±0.06
Rec$_{ER}$	1.00±0.00	1.00±0.00	1.00±0.00	0.91±0.06
Prec	0.19±0.12	0.39±0.06	0.41±0.03	0.43±0.02
Rec	0.29±0.08	0.52±0.06	0.57±0.04	0.51±0.05
F-score	0.22±0.13	0.49±0.09	0.49±0.03	0.47±0.05

Table 5.8 – Average ± standard deviation of reconstruction metrics for DAPER3 for a particular size (3000 instances) with a percentage λ of ER identification for α fixed to 50%.

Table 5.7 depicts the quality reconstruction (RGS_neo4j) for DAPER3 when dealing with partially structured data by varying the α parameter of exceptions generating. We notice that increasing the percentage of exceptions in the relationship instances do not impact the quality reconstruction of the DAPER model with small α. The relationship signatures are not enough perturbed so the recall is perfect, and the exceptions added in the database are not sufficient to be identify as new (and false) relationships, so the precision is also perfect. In this context, precision, recall and F-score corresponding to the reconstruction of the dependency structure remain constant.

DAPER	Data size			
	1000	5000	10000	15000
DAPER4_naive	**545**	**942**	**1885**	**2563**
DAPER4_k-partite	1160	1665	3308	4975
DAPER5_naive	**1202**	**1263**	**2808**	**4323**
DAPER5_k-partite	2915	8350	15630	18793
DAPER6_naive	**3702**	**4263**	**8808**	**9323**
DAPER6_k-partite	4915	7350	18660	21793
DAPER7_naive	**2702**	**3457**	**7908**	**8930**
DAPER7_k-partite	5214	8270	25137	43765

Table 5.9 – A table of running time (in seconds) for each sample size, when executing RGS_Neo4j with naive datasets and k-partite datasets generated from each DAPER.

For $\alpha = 30\%$, all existing relationships still well represented in the database and discovered in the ER schema so the recall$_{ER}$ is yet perfect. The number of perturbed relationships increases and generate false relationships so the precision$_{ER}$ decreases.

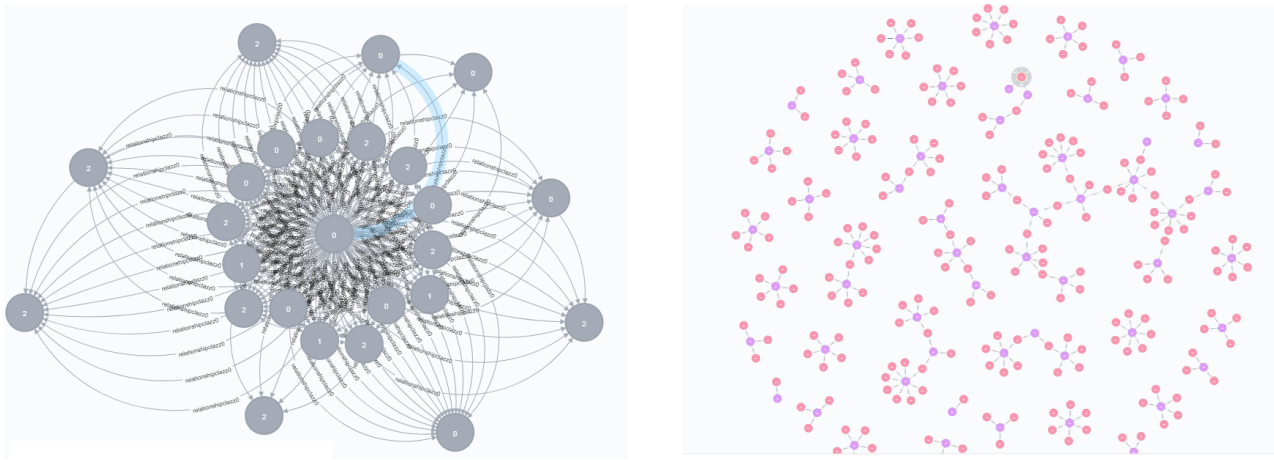
By increasing α, we always kept existing relationships represented in the graph database because λ value is very small so the recall$_{ER}$ is yet perfect. The number of perturbed relationships for one given signature is sufficient to generate false relationships so the precision$_{ER}$ decreases. Some probabilistic dependencies can't be discovered because of new attributes (and classes) no related to the theoretical ER model are added. For this reason the corresponding quality reconstruction metrics progressively decrease.

Table 5.8 shows the quality reconstruction (RGS_neo4j) for DAPER3 when dealing with partially structured data by varying the λ parameter of ER schema identification. (λ=0%) means that we accept all relationships even if there are exceptions. In this case all exact relationship are represented in the database so the recall is perfect. The number of exceptions is sufficient to generate a big number of false relationships so the precision has a too small value. Some probabilistic dependencies can't be discovered because new relationship classes and their corresponding attribute classes are discovered which are not related to the theoretical ER model. For this reason the corresponding reconstruction metrics also decrease.

By increasing λ, the number of perturbed relationships for one given signature which generate false relationships decreases so the precision$_{ER}$ increase. On the other hand and for λ=50%, existing relationships in the DAPER are no more discovered so recall$_{ER}$ decreases. Some probabilistic dependencies can't be discovered because the corresponding attributes (and classes) are no more related in the ER model . For this reason the corresponding reconstruction metrics progressively decrease.

5.3.5.4 Experimental protocol IV: Results and interpretation

Table 5.9 depicts the execution time of our contribution comparing naive and k-partite graph databases. We notice that using k-partite datasets the running time increases. It is even multiplied by 4 for DAPER7. The k-partite data structure of a graph database has a negative impact on traversal graph performance. This is can be explained by the used database structure by the fact that the relationships of graph databases are not indexed.



(a) A screen shot of a strongly connected sub-graph database.

(b) A screen shot of a dispersed sub-graph database.

Figure 5.11 – An example of screenshot of the structure of strongly connected (naive) and dispersed k-partite) graph databases.

In fact, naive structure is a strongly connected database, as shown in Figure 5.11.(a), which is usually associated with directed graph (one way edges) in which there is a path between every two nodes. This type of database can be seen as a one large component that fills most of the stored objects.

However, k-partite structure is a dispersed database in which most nodes are not neighbors, as shown in Figure 5.11.(b). In this type of structure, each node could be reached by a small number of steps. Since, the average diameter of each connected component from the graph is small. Therefore, with this type of datasets, parsing long paths to reach a big number of nodes can be a limit in term of queries time execution.

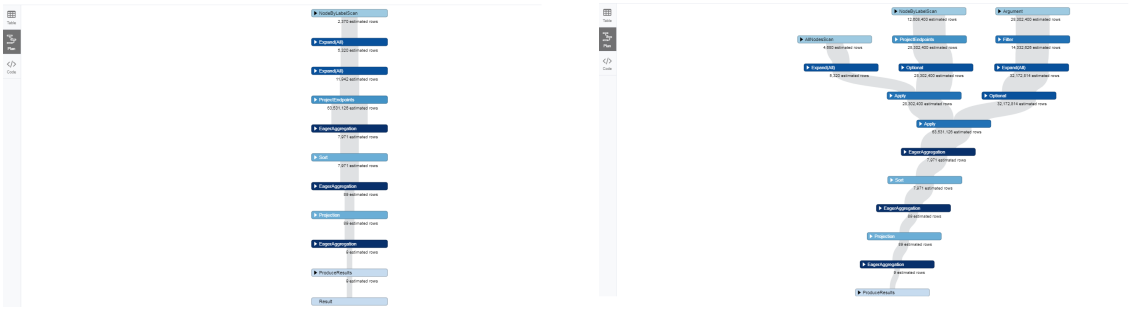
In naive graph database, graph traversal performance is independent of the size of the graph especially with data flows that start from nodes. However, its performance tends to be slower with k-partite datasets in which pair of nodes that are totally disconnected could be found.

In another hand, the obtained results can be interpreted according to the specific representation of a path in the graph database which must start by a node going through a relationship to reach a target node (N)-[R]-(N). In fact, given a set of **starting nodes**, traversal techniques consist to explore recursively the neighborhood of those nodes until a terminating condition, such as the depth or visiting of a target node, is fulfilled. In learning DAPER algorithm there are often slot chains that start directly from a relationship. In this case, a path that starts by an empty node ()-[R]-(N) is created. In order to search for this type of slot chains in our graph database, critical paths that start by an empty node ()-[R]-(N) will be created. A critical path in our case, is a path which starts by a labeled relationship and an empty node.

Therefore, time execution of the data flow queries that start by nodes is reduced. However, their performances tend to be slower with critical paths that start directly by relationships and not by nodes. This is explained by the fact that relationships in graph database are not indexed.

Example 5.3.1. *The following example describes two types of queries where the first type creates a normal path in the graph database and the second one creates a critical path.*

1. *Students who take the same course taken by a particular student.*
2. *All courses taken by a particular student.*



(a) A normal path starting by a labeled node.

(b) A critical path which starting by an empty node and a labeled relationship.

Figure 5.12 – An illustration of normal and critical paths.

These queries correspond respectively to these two slot chains:

1. $[takes.student]^{-1}.[takes.course].\ [takes.course]^{-1}.[takes.student]$
2. $[takes.student].[takes.student]^{-1}.[takes.course]$

By executing the first slot chain in the graph database this path will be created:

```
MATCH (a:student)-[b:takes] --> (c:course)
OPTIONAL MATCH (c:course)<--[d:takes]-(e:student)
```

Once the second query is executed this critical path will be created:

```
MATCH ()<--[a:takes]-(b:student)
OPTIONAL MATCH (b:student)<--[c:takes]-(d:course)
```

The Figure 5.12.(a) is an illustration of the first query which starts by a node to reach the relationship. The second one, Figure 5.12.(b), is an illustration of the critical path which starts by an empty node. We notice that in the critical path all nodes of the graph are scanned in order to match the right relationship. However, in the first path only the right labeled nodes that are mentioned in the query are scanned. This can be explained by the fact that relationships in graph database are not indexed.

5.4 Conclusion

In this chapter, our contribution of DAPER learning from graph databases is presented.

Our work builds on the recent work on DAPER models (where data come from well structured relational databases), and describes how to use them with graph databases (which can be schema-free). A first solution using an intermediate scenario where the graph database is partially structured is proposed. This solution is improved by making the assumption that one meta-model can be extracted even if some parts of the instance do not obey to the meta-model (exceptions). A new solution is proposed to resolve the problem of having exceptions in the graph database in order to deal with lesser structured databases. A new method to identify an ER model as the meta-model of our graph database is presented. Then, given this ER model and the graph database, we focus on counting method for probabilistic dependencies identification. As this learning task relies on numerous database queries involving potential long slot chain dependencies, experiments showed that querying our graph database is more efficient than a relational one without decreasing the quality of the obtained structures. However, by using more realistic k-partite graph databases the running time increases as well as the number of stored instances.

As conclusion, the type of computations and queries commonly performed on graph databases is various as the variety of data structures. To effectively distinguish between the different types of access patterns performed on graph databases, depending on domain or use case, it is important to understand the building blocks that graph database operations and queries are built up from.

6

DAPER learning from Graph Database using MLN framework

Contents

6.1 Introduction **92**
6.2 Expressing a DAPER in Markov Logic **92**
6.3 DAPER joint learning using MLN framework **95**
6.3.1 DAPER joint learning 95
6.3.2 Evaluation process 97
6.4 Experiments **97**
6.4.1 Experiment methodology 97
6.4.2 Evaluation metrics 99
6.4.3 Results and interpretations 99
6.5 Conclusion **106**

6.1 Introduction

We have demonstrated that we can learn DAPERs even with partially structured data using graph databases (which can be schema-free) using our method introduced in Chapter 5. However, we are shown that our proposed solution to resolve the problem of having exceptions in the graph database is sensitive to the parameter λ (for ER schema identification) which is a user-defined threshold. Experiments have shown that for a low value of λ and a high value of α (percentage of exceptions in relationships' instances), the quality of the reconstructed structures gradually decreases.
In another hand, the ER schema and the set of probabilistic dependencies are generally learned separately.
Therefore, we are now interested in other probabilistic and relational frameworks derived from Logic to learn both components simultaneously (ER and CPDs) and to improve the quality reconstruction of the model.

In this chapter, we are interested in learning DAPER models from graph databases using MLN frameworks. MLN can help us to take into account the exceptions that are dropped during DAPER learning. We propose in this chapter a joint learning from partially structured graph databases, where we want to learn at the same time the ER schema and the probabilistic dependencies. The Markov Logic Network formalism is an efficient solution for this task. Both parts of the models are described in the same logical way, and dedicated structure learning algorithm could be able to retrieve both information at the same time. The logical formulas used to describe the model can also manage exceptions, i.e., data which are not coherent with an underlying structured model, as we have to deal with when working with partially structured data. This work was introduced in (El Abri et al., 2018).

This chapter is organized as follows. Section 6.2 describes how a DAPER can be expressed with Markov Logical framework concerning their representation ability and learning methods already introduced in Chapter 3. In Section 6.3, we describe our main contribution, i.e., how to learn simultaneously both part of a DAPER model by using an MLN. Section 6.3.2 presents a way to evaluate the efficiency of the proposed method. Some conclusions are drawn in Section 6.5.

6.2 Expressing a DAPER in Markov Logic

By inspiring ourselves from (Domingos and Richardson, 2004), we describe in this section how to define a DAPER in the Markov Logic framework.
A DAPER model is composed of the ER schema plus the set of probabilistic dependencies. Recall the components of a DAPER given in Section 3.3; we have a set of entity classes $\mathcal{E}$ connected by a set of relationship classes $\mathcal{R}$, a set of attribute classes $\mathcal{A}(X)$ which can be associated to $\mathcal{E} \cup \mathcal{R}$ and a set of probabilistic dependencies between $\mathcal{A}(X)$.

We have first to define $predicates_{\mathcal{E}}$, the set of predicates corresponding to the set of entity classes $\mathcal{E}$ of the ER schema, where each entity class E will correspond to a $predicate_E(object)$ describing if this object belongs to this class. In the same way, we define $predicates_{\mathcal{R}}$, the set of binary predicates corresponding to the set of $\mathcal{R}$ in the ER schema, where each relationship class R corresponds to a $predicate_R(object1, object2)$.
Attributes classes $\mathcal{A}(X)$ help us to define the set of predicates_$\mathcal{A}$ which are divided into two sets predicates_$\mathcal{A}(\mathcal{E})$ and predicates_$\mathcal{A}(\mathcal{R})$.

— Each $predicate_A(object, value!)$ corresponds to an attribute class $A \in \mathcal{A}(\mathcal{E})$ where the first argument is the entity class to which this attribute is associated and the second one unique value for this attribute.

— Each $predicate_A(object1, object2, value!)$ corresponds to an attribute class $A \in \mathcal{A}(\mathcal{R})$ with the first two arguments are the entities involved in the corresponding relationship class.

In the Markov Logic formalism, we also have to declare that a separate weight must be learned for each formula obtained by grounding that variable to one of its values. So we define:

— Each $predicate_A$(object,value$_A$+) corresponds to an attribute class $A \in \mathcal{A}(\mathcal{E})$ with value$_{\mathcal{A}}$+ is used to learn "per constant weight".

— Each $predicate_A(object1, object2, val_A+)$ corresponds to an attribute class $A \in \mathcal{A}(\mathcal{R})$.

Finally, some prior knowledge can be added. For instance, in our case, each object belongs to a unique class.

$$predicate_{E_i}(object1) <=> !predicate_{E_j(object2)} \tag{6.1}$$

Example 6.2.1. *Let us illustrate these steps by considering the DAPER described in Figure 6.1 which defines the set of predicates and the prior knowledge depicted in Table 6.1.*

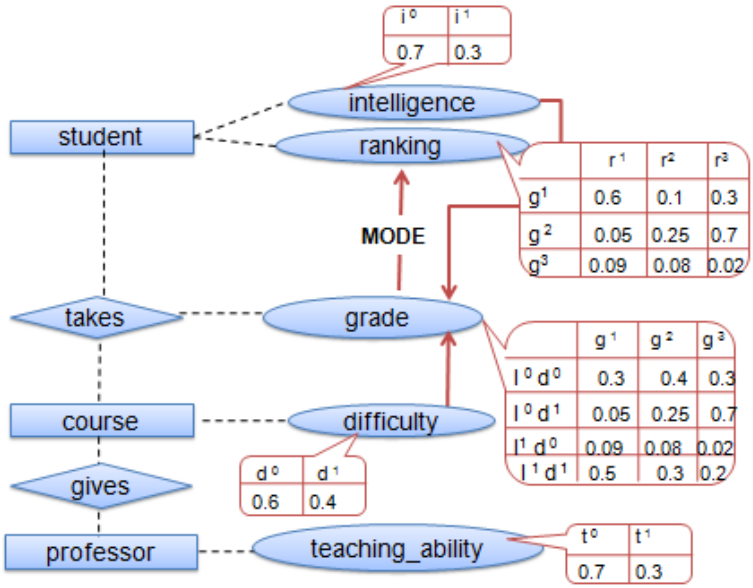


Figure 6.1 – Example of a DAPER model (inspired from (Getoor, 2001) and detailed in Chapter 3).

Predicates ER	Predicates Attributes	Prior knowledge
student(s) course(c) professor(p) takes(s,c) gives(p,c)	s_intelligence(s,ival+) s_ranking(s,rval+) c_difficulty(c,dval+) p_teaching_ability(p,taval+) t_grade(s,c,gval!) s_intelligence(s,sval+) s_ranking(s,rval+) c_difficulty(c,cval+) p_teaching_ability(p,taval+) t_grade(s,c,gval+)	student(x) ⇔ !course(x) student(x) ⇔ !professor(x) course(x) ⇔! professor(x)

Table 6.1 – Knowledge base corresponding to the DAPER described in Figure 6.1.

After defining all these concepts, we can now define the ER schema as a set of clauses $\mathcal{F}_{ER}$, one for each relationship, $f_{ER} : predicate_R \Rightarrow predicate_{E1} \wedge predicate_{E2}$ when the relationship R is defined for entities $E1 \times E2$. With perfectly structured data, these formulas are certain.

Equation 6.2 shows us the exact formulas corresponding to the ER schema defined in Figure 6.1.

$$\begin{aligned} takes(s,c) &\Rightarrow student(s) \wedge course(c) \\ gives(p,c) &\Rightarrow professor(p) \wedge course(c) \end{aligned} \tag{6.2}$$

We can finally describe the probabilistic dependencies of the DAPER with a set of formulas $\mathcal{F}_{PD}$. Each formula is defined with one head related to one predicate attribute (and a specific value) and whose body involves the values of the parents of this attribute (as defined in the DAPER dependency structure) and the logical description of the slot chain between this attribute and its parents. For instance, in the DAPER described in Figure 6.1, parents of $takes.grade$ are $takes.student.intelligence$ and $takes.course.difficulty$. The logical description of this specific dependency would involve 12 formulas (as the total number of values in the conditional probability distribution) with the pattern described in equation 6.3.

$$\begin{aligned} takes_grade(x,y,valG) \quad &\Rightarrow student(x) \wedge s_intellligence(x,valI) \\ &\wedge course(y) \wedge c_difficulty(y,valD) \end{aligned} \tag{6.3}$$

Another important information in DAPER is *aggregators*. For instance, in Figure 6.1, stud.ranking depends probabilistically on takes.grade. If a student take 8 courses, stud.ranking will depend on takes.grade of 8 takes object and this number will not be the same for all students. So, to get a summary of such dependencies, an aggregator is used. For MLN, there will be at least 8 satisfying groundings of the conjunction $takes(s,c) \bigwedge s_ranking(s,rval) \bigwedge t_grade(t,gval)$. We can notice that the MLN framework does not take into account aggregation functions to compute the probability of an object given a set of other objects.

For DAPERs to ensure the navigation between classes to achieve the relational path between descriptive attributes, slot chains are used.

By inspiring ourselves from the definition of *chain* introduced by (Dinh, 2011) to optimize their structure learning algorithm. We propose to define the set of the chains for as the set of slot chains including in the relational path between the different predicates.

Definition 6.2.1. ***Chain***
A chain $\langle p_1, ..., p_i \rangle$ is a list of ground literals $\mathcal{L}$= $\langle l_1, ..., l_k, ... \rangle$ such that $\mathcal{L}$= $\{predicates_\mathcal{E} \cup predicates_\mathcal{R} \cup predicates_\mathcal{A}\}$ and $\forall i > 1$, the link between two consecutive literals $link(l_{i-1}, l_i)$ have a shared constant.

Example 6.2.2. *Let us start from the true ground predicate takes(A,C). The connected group that can be built starting from this ground predicate contains 6 predicates as:*

$$\{takes(A,C), student(A), takes(A,D), course(D), gives(B,D), professor(B)\}$$

in which, for example, takes(A,C) and takes(A,D) are linked by the constant A.

In the case of DAPER the previous chain can be expressed by the following slot chain:

$$(takes.student.[student]^{-1}.takes.course.[gives]^{-1}.gives.professor)$$

These chains are used to define the relational path between attributes. In other words, the chain defines the format of the body of $\mathcal{F}_{PD}$.

Example 6.2.3. *Let us consider the following example F_{PD} which corresponds to one of the formulas that expresses the dependency: t_grade depends on $s_intelligence$ attainable via the chain in red.*

$$t_grade(t, valT) \Rightarrow takes(s, c) \bigwedge student(s) \bigwedge s_intelligence(s, valS)$$

6.3 DAPER joint learning using MLN framework

We propose in this work a joint DAPER learning from partially structured graph databases, where we want to learn at the same time the ER schema and the probabilistic dependencies. The Markov Logic Network formalism is an efficient solution for this task. We have seen in Section 6.2 that both parts of the model are described in the same logical way, so dedicated structure learning algorithm could be able to retrieve both information simultaneously.

6.3.1 DAPER joint learning

To identify (1) the ER schema constraints and (2) the set of probabilistic dependencies, we have to look for the set of formulas that target (1) $predicates_{\mathcal{R}}$ in function of a pair of predicate$_E$ and (2) $predicates_{\mathcal{A}}$ in function of other $predicates_{\mathcal{A}}$ and attainable via chains of literals. Figure 6.2 shows all steps of our joint learning process.

6.3.1.1 ER schema learning

The first formats of clauses we are looking for are the set of formulas $\mathcal{F}_{ER} = \{\omega_i, f_i\}$ that learn the ER schema, as described in section 6.2, where
f_i: $\neg$predicate$_R$ $\vee$ predicate$_{E1}$ $\vee$ predicate$_{E2}$

As we are dealing with partially structured data, several formulas can be extracted for the same predicate. We normalize the weight of each formula f_{ER} by computing:

$$\frac{\exp^{\omega_i}}{\exp^{max(\omega_i)}} \tag{6.4}$$

where $max(\omega_i)$ is the maximum learned weight of the identified $\mathcal{F}_{ER}$. Then, we propose to consider only the strongest f_{ER} in term of weight (in a normalized value) as the relevant relationship. We then apply an user-defined threshold $\lambda_{er} \in [0, 1]$ to identify our ER schema.

6.3.1.2 Probabilistic dependencies learning

The second format of clauses we are looking for is the set of formulas $\mathcal{F}_{PD} = \{\omega_i, f_i\}$ that correspond to the probabilistic dependencies, where f_i: $\neg$predicate$_Y$ $\vee$ K$_1$.predicate$_{X1}$ $\vee$,...,$\vee$ K$_n$.predicate$_{Xn}$.

As we have seen in the equation 6.3, a probabilistic dependency between one random variable and its parents can generate several logical formulas, one for each possible configuration of the variables. We propose to normalize the weight of each formulas f_{PD} by computing:

$$\frac{\exp^{\omega_i}}{\exp^{max(\omega_i)}} \tag{6.5}$$

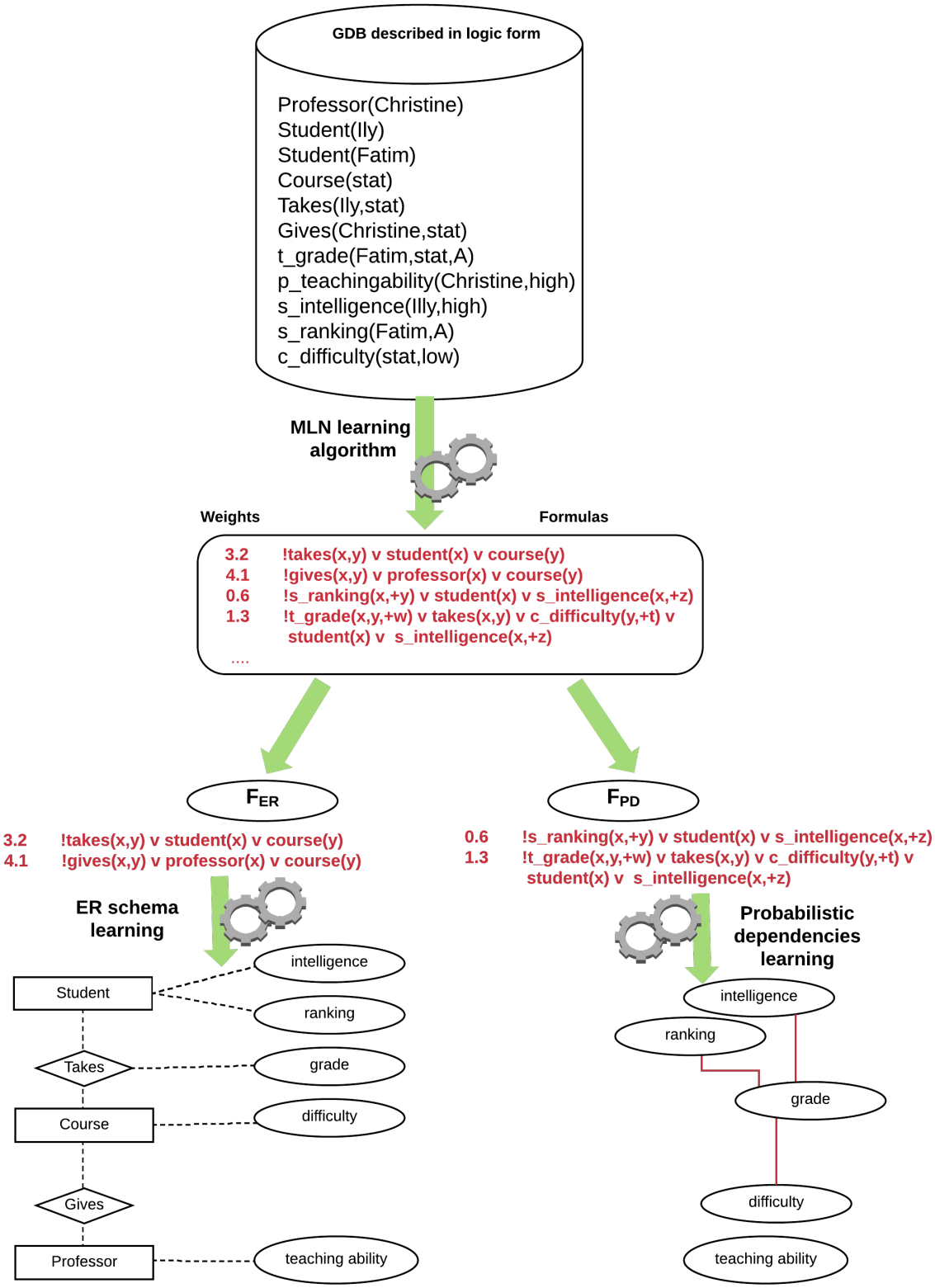


Figure 6.2 – Process of DAPER joint learning using MLN framework.

where $max(\omega_i)$ is the maximum learned weight of the identified $\mathcal{F}_{PD}$. We then propose to extract such a dependency only when the probability of the corresponding compatible f_{PD} is greater than another user-defined threshold $\lambda_{pd} \in [0, 1]$.

As the formulas are described in their clausal form, it is not always possible to identify the direction of the dependency, as defined in DAPER framework, nor the possible aggregation function. Only the slot

chains between variables can be retrieved from the formulas. We then summarized our dependency graph with an undirected graph where the nodes are the attributes; the edges are the probabilistic dependencies discovered, labeled with their associated slot chains.

6.3.2 Evaluation process

Several metrics have been proposed to evaluate DAPER structure learning algorithms. Concerning the ER schema, we define a Hamming distance (RSHD_ER) between the graphs describing the original ER schema and the learned one to evaluate the quality of this step. The RSHD_ER penalizes extra relationships and missing relationships by an increase of the distance by 1. When the relationship exists in both ER schemes, we turn to check whether the start and the end entity class and the direction of the learned relationship are those of the true one, if not the distance will be increased by 1.

The task concerning the probabilistic dependencies is more complex. We can not use directly existing metrics described for instance in (Maier et al., 2013a; Ben Ishak, 2015) to compare the original probabilistic directed model and the undirected one obtained in the section 6.3.1. Thus, we propose to use a weighted Hamming distance between the undirected counterpart of the original model and the one created from the logic formulas of the MLN (RSHD_PD).

RSHD_PD penalizes extra edges and missing edges by an increase of the distance by 1. When the edge exists in both structures, we turn to check whether the slot chain of the learned dependence are those of the true dependence. If not, RSHD_PD will be incremented by a penalization of $\alpha > 0$.

(Ben Ishak, 2015) proposed the same idea for directed models, with a weight for a given edge defined as the similarity between the slot chains associated to this edge in both models. This computation doesn't take into account the starting class of the slot chain, which leads to overestimating the error when comparing an empty slot chain to a non-empty one. We propose here to add this starting class in the similarity computation to solve this problem.

6.4 Experiments

6.4.1 Experiment methodology

The purpose of the experiments is twofold: show that DAPER joint learning using MLN framework gives correct results by studying the influence of the thresholds and compare this approach to our approach described in Chapter 5. Therefore, we have used both proposed methods to learn our DAPER model. The first one (**PRM**) consists of using the method proposed in Chapter 5 to identify the ER schema from the noisy graph database, then to learn the DAPER structure. The second one (**MLN**) consists to DAPER joint learning using an MLN learning algorithm, and then extract the ER schema and the undirected dependencies structure as described in Section 6.3.1. The output of the first experiment is a directed DAPER. To compare both methods, we have to disorientate this DAPER. Finally, we evaluate both learned models in term of ER schema accuracy, probabilistic structure reconstruction quality and running time with metrics defined at Section 6.3.2 .

6.4.1.1 Networks and datasets

We have used the generating process defined by (Ben Ishak et al., 2016) to first generate theoretical DAPER (ER models and probabilistic dependencies) and then to sample relational database instance from these probabilistic models. We have randomly generated 3 DAPERs whose characteristics are

	$\mid \mathcal{E} \mid$	$\mid \mathcal{R} \mid$	$\mid \mathcal{A} \mid$	$\mid \mathcal{V} \mid$	k_{max}:$\mid \mathcal{K} \mid$
DAPER1	5	4	16 (2-5)	2-3	1 : 9-4
DAPER2	6	5	28 (2-5)	2-4	2 : 4-8-11
DAPER3	6	5	94 (1-10)	2-6	2 : 9-36-42

Table 6.2 – characteristics of DAPER networks used for dataset generation. Values in parenthesis are min/max values per class. Values in parenthesis are min/max values per class. Values in the last column correspond to the maximum length of slot chain $kmax$ and the number of dependencies for each slot chain length (from 0 to k_{max}).

described in table 6.2. These DAPER contains binary relationship classes and attributes of type integer.

Our relational databases are transformed into graph databases to which we have added some additional relationships (exceptions)(with $\alpha = 10\%$, 30% and 50%) to perturb the underlying ER model. The signature of relationship instances has been replaced by another signature not conform with the underlying ER model. Thus, we obtain partially structured graph databases which are in turn converted into knowledge databases to use them for MLN learning.

6.4.1.2 Algorithms

For the first learning process, we use our approach described in Chapter 5 to identify the ER schema from the noisy graph database, then to learn the DAPER structure. The parameters used for this process are:
- λ_{er} : identification threshold for ER identification, which is varied as 0, 0.1, 0.3, 0.5 and 0.8. We have chosen different values of λ_{er} to study the impact of generated exception on ER schema and probabilistic dependencies learning.
- k_{max} : maximum possible slot chain length during greedy search set to 1 for DAPER1 and 2 for DAPER2 and DAPER3. This parameter controls the complexity of the algorithm by defining "how far" (in the relational schema) can be some dependent attributes. Here, we choose the same horizon than the one used for generating the theoretical DAPER.

For the second learning process, we apply the Beam search MSL algorithm (Kok and Domingos, 2005)to learn MLN, reference algorithm already implemented in Alchemy C++ Library [1].

Beam search MSL uses as an input the knowledge base without needing a relational schema (directly imported from SQL database, or identified from a graph database). It uses L-BFGS to learn maximum weighted pseudo-log-likelihood (WPLL) weights during clause search. The parameters used for this process are:
- Maximum predicates per clause: 4 for DAPER1 and 6 for DAPER2 and 3. This parameter controls the complexity of the algorithm by defining "how far" (in the relational schema) can be some dependent attributes.
- Penalization of weighted pseudo-likelihood = 0.1
- Maximum iterations to run L-BFGS when optimizing WPLL= 10,000 (tight) and 10 maximum iteration to run L-BFGS when evaluating the candidates' clauses.

The parameters used for DAPER joint learning process are :
- λ_{er} : identification threshold for ER identification, which is varied as 0, 0.1, 0.3, 0.5 and 0.8.
- λ_{pd} : identification threshold for probabilistic dependencies identification, which is varied as 0, 0.1,

1. https://alchemy.cs.washington.edu/

0.3 and 0.6.

6.4.2 Evaluation metrics

We have compared the two approaches in term of execution speed and in term of quality of reconstruction. The quality of ER schema reconstruction is measured using RSHD_ER. The quality of graph dependency reconstruction is measured using RSHD_PD (cf. Section 6.3.2).

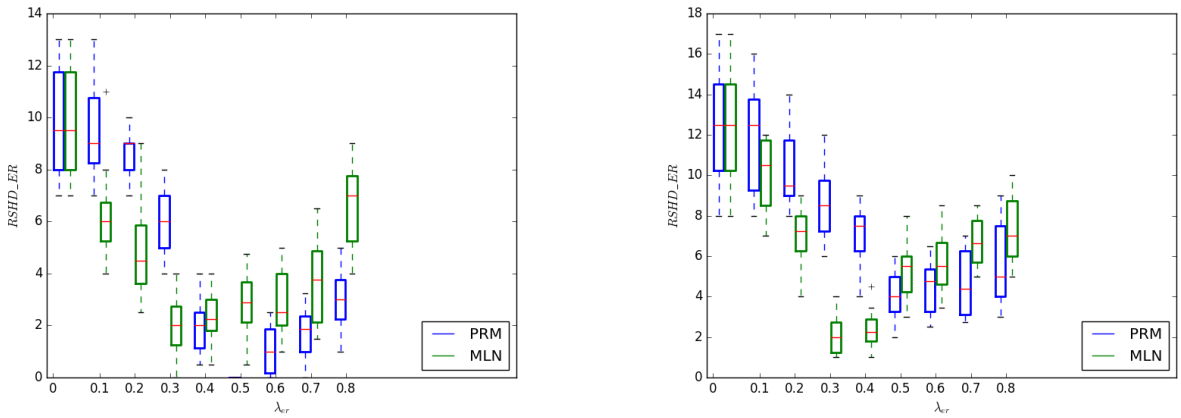
6.4.3 Results and interpretations

Table 6.3 depicts the quality ER schema reconstruction for both approaches for each DAPER and each data size when dealing with partially structured data by varying the α parameter of exceptions generating and the λ_{er} parameter of ER schema identification. For $\alpha = 10\%$ (few exceptions) the RSHD_ER values tend to zero This is explained by the fact that the relationship signatures are not enough perturbed, so the exceptions added in the database are not sufficient to be identified as new (and false) relationships.. Then, by increasing α the RSHD_ER increases because the number of exceptions in the relationship instances increases and generates more false relationships.
We can also notice that for λ_{er}=0, the RSHD_ER values for both approaches are high because the learning algorithms can identify all exceptions without excluding any one of them.
By increasing λ_{er}, RSHD_ER first decreases then increases. This is explained by the fact that firstly wrong relationships are progressively removed to the point where right formulas, but with low weights which decrease rapidly because of the normalization.
For $\lambda_{er} = 0.3$ we obtain better results with the second approach than the first one. However, for $\lambda_{er} > 0.3$ the RSHD_ER using PRM learning framework gives the best results compared to the second algorithm which uses MLN framework. This is impressive in Figure 6.3, where the RSHD_ER of MLN approach gradually decreases concerning λ_{er} until $\lambda_{er} = 0.3$.
Therefore, we notice that best results of first and second algorithms are given by applying different values of λ_{er} (λ^*_{er}=0.3 for MLN framework and λ^*_{er}=0.5 for PRM framework). This can be explained by the fact that the λ_{er} parameter for ER schema identification using PRM framework intervenes in the number of occurrence of relationship signatures and λ_{er} for MLN framework acts on the weights of identified Formulas of ER schema F_{ER}.

As the results are very close, we present in this section the most representative boxplots and figures. Table 6.4 and Figure 6.4 depict the quality reconstruction of learned DAPER using both approaches for α=10% for each data size of each DAPER.
We vary λ_{er} parameter for DAPER learning using PRM framework. We vary λ_{er} and λ_{pd} parameters when we use MLN framework. We can notice that for low values of $\lambda_{er} < 0.3$ the obtained RSHD_PD values are high for both methods because the number of perturbed relationships for one given signature is sufficient to generate false relationships and thus to generate false probabilistic dependencies. However, the MLN method can provide best results when applying the λ_{pd}.
We can notice that applying the threshold λ_{er} (for PRM framework) and applying λ_{er} and λ_{pd} (for MLN framework), the RSHD_PD gradually decreases then increases for both approaches but for different values of λ_{er}. Best results for MLN framework are generally given for $\lambda^*_{pd} = 0.3$. Until this value, λ_{pd} has a positive impact on dependencies extracting because the RSHD_PD improves and the quality of reconstruction is getting better and better.
Best results for PRM framework are given for $\lambda^*_{er} = 0.5$. This is more detailed in the Figure 6.4 which shows the RSHD_PD with respect to λ_{pd} and λ_{er} for MLN method and concerning λ_{er} for PRM method. We even obtain the best undirected learned structure using MLN comparing to the learned one using the first process. For instance, with λ_{pd}=0.3 we often obtain better RSHD_PD

values than the first method. However, by increasing λ_{pd} more and more (λ_{pd}=0.6) the RSHD_PD increases. This is explained by the fact that existing dependencies in the DAPER are no more discovered because the fact of the normalization decreases the weights of their corresponding formulas F_{PD}.
This is even worse for $\lambda_{er} = 0.8$, because existing relationships in the DAPER are no more discovered so RSHD_ER decreases. By consequence, RSHD_PD decreases because some probabilistic dependencies cannot be discovered because the corresponding attributes (and classes) are no more related in the ER model . For this reason, the corresponding reconstruction metric progressively increases.

Tables 6.5 and 6.6 depict the quality of reconstruction of learned DAPER using both approaches for respectively α=30% and α=50% for each data size of each DAPER.
From these results, we can also conclude that we can extract two optimum values of $\lambda^*_{pd} = 0.3$ and $\lambda^*_{er} = 0.3$ for MLN method and an optimum value $\lambda^*_{er} = 0.5$ for PRM method. This is further shown in Figures 6.5 and 6.6 , where RSHD_PD of MLN method drops drastically with respect to λ_{pd} values improving the learned dependency structure until λ^*_{pd}=0.3 then it increases.
We also notice that RSHD_PD values of the second approach in Table 6.4 are almost close to those of Table 6.5. This can be explained by the fact that using an MLN framework, with exceptions generation, the same relationships (predicate) can link more than one pair of entity classes, but we do not discover new attributes and classes as the case of the first approach in which we rename any exceptional relationship. Therefore, same probabilistic dependencies can be discovered due to the power of first-order-logic to manage the exceptions generated in the datasets.

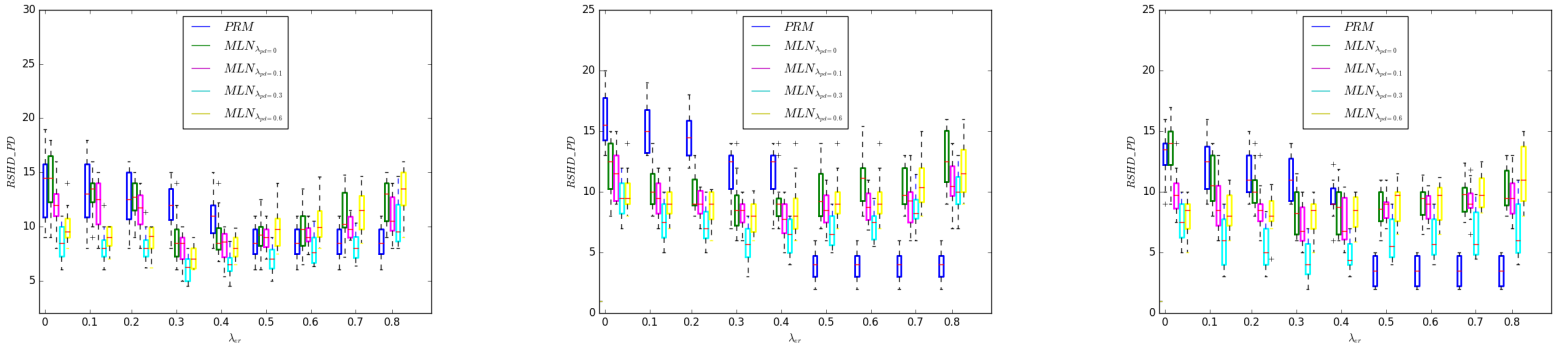


(a) α=30% (b) α=50%

Figure 6.3 – The Average values of RSHD_ER with respect to λ_{er} when executing approach 1 and 2 with (a) α=30% and (b) α=50% for a particular size (5000 instances) of DAPER1.

	Algo	λ_{er}														
		0			0.1			0.3			0.5			0.8		
		α			α			α			α			α		
		10%	30%	50%	10%	30%	50%	10%	30%	50%	10%	30%	50%	10%	30%	50%
D1	PRM	7.2±2.33	10.2±1.98	12.9±2.67	5.9±1.23	9.9±1.85	12.3±2.45	3.1±0.89	6.1±1.25	8.9±1.71	**0**±0.00	**0**±0.00	**4**±1.26	**0**±0.00	**3**±1.09	**5**±1.94
	MLN	7.2±2.33	10.2±1.98	12.9±2.67	**2**±0.00	**6.9**±2.10	**6.2**±1.88	**0**±0.00	**2.6**±0.91	**1.5**±0.80	2.4±1.23	3.6±1.32	5.7±1.39	4.3±0.89	6.9±1.42	7.6±1.70
D2	PRM	11.9±1.50	12±2.44	14.9±3.91	10.6±2.14	11±2.39	14±2.81	6.2±2.71	8.2±1.41	9±2.33	**0**±0.00	**0**±0.00	**3.5**±0.40	**0**±0.00	**5.3**±0.33	**4.7**±0.79
	MLN	11.9±5.50	12±2.44	14.9±3.91	**6.8**±0.40	**3.4**±0.79	**4.5**±0.70	**0**±0.00	**2.9**±0.74	**3.9**±0.89	4.3±0.71	6.4±1.32	8.9±1.14	7.8±2.40	9±1.79	10±1.70

Table 6.3 – Average ± standard deviation of RSHD_ER for DAPERs for a particular size (5000 instances).

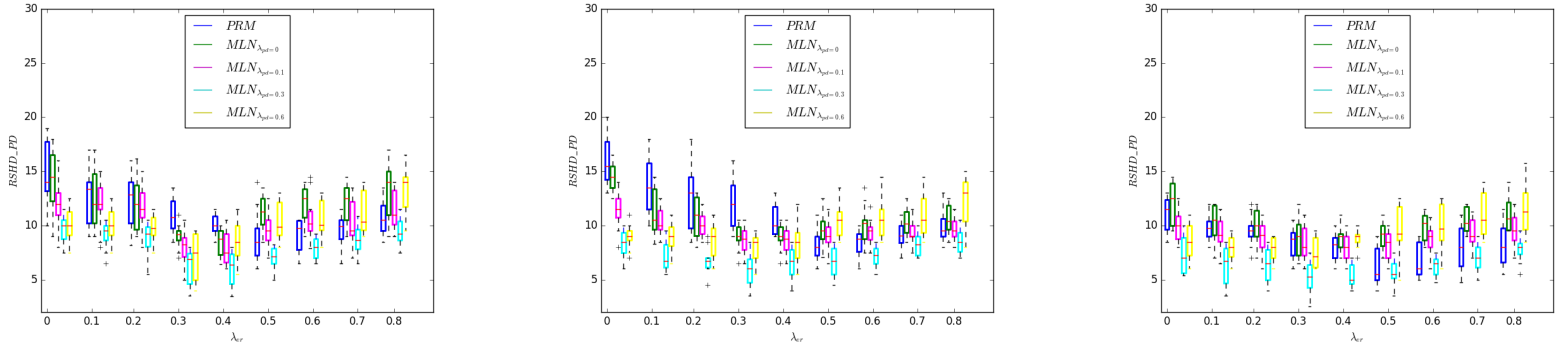


(a) Data size =1000 (b) Data size =3000 (c) Data size =5000

Figure 6.4 – The Average values of RSHD_PD with respect to λ_{pd} when executing approach 1 and 2 with (a) Data size =1000 and (b) Data size =3000 and (c) Data size =5000 for DAPER1 with a percentage α of exceptions in the relationship instances fixed to 10%.

	Algo		λ_{er}														
			0			0.1			0.3			0.5			0.8		
			Data size			Data size			Data size			Data size			Data size		
			1000	3000	5000	1000	3000	5000	1000	3000	5000	1000	3000	5000	1000	3000	5000
D1	PRM		14.57±3.21	16.33 ±2.3	13.32 ±1.91	13.80 ± 2.91	15.57 ± 2.09	12.55 ± 2.08	12.44 ± 2.34	12.01 ± 1.98	11.39 ± 1.81	8.75 ± 1.46	**4.4** ± 1.26	**4.4** ± 1.26	8.75 ± 1.46	**4.4** ± 1.26	**4.4** ± 1.26
	MLN	$\lambda_{pd} = 0$	14.58±2.70	12.43 ±2.14	13.93 ± 2.26	13.32 ±1.91	10.73± 1.94	11.45±2.20	9.6 ± 2.49	9.6 ±2.49	8.74 ± 1.90	9.46 ± 1.63	10.54± 1.75	10.37±2.14	12.70 ± 1.85	9.78± 2.39	9.15 ± 1.26
		$\lambda_{pd} = 0.1$	12.44 ±2.13	11.88 ± 2.10	10.33 ±1.91	12.43 ±2.14	9.82 ± 1.71	9.53 ±2.26	8.32 ± 1.50	8.75 ± 1.46	7.7 ± 1.82	9.05± 1.15	9.55 ± 2.01	9.89 ± 1.71	10.39 ± 1.43	8.13 ± 1.55	9 ± 1.16
		$\lambda_{pd} = 0.3$	**9.02** ±1.63	**9.82** ±1.71	**8.02** ± 1.72	**8.61** ± 1.72	**8.02**± 1.72	**6.76**± 2.05	**6.41**±1.26	**6.5** ±1.40	**4.9** ± 1.38	**7.17** ± 1.10	6.59 ± 1.40	6.71 ± 1.89	9.08 ± 1.50	9.13 ± 1.18	7.78 ±2.46
		$\lambda_{pd} = 0.6$	10.32 ±1.90	10.32 ± 1.80	8.32 ± 1.50	9.11 ± 0.95	9.45 ± 1.53	8.81 ± 1.64	7.42 ± 1.14	9.37 ± 2.57	8.32 ± 1.50	10.26 ± 2.00	12.15 ± 2.25	11.85 ± 2.36	13.69 ± 2.03	9.94 ± 2.19	9.39 ± 1.58
D2	PRM		15.25 ± 2.43	14.5 ± 2.25	14 ± 1.73	13.9± 1.45	13± 1.13	13.2± 0.69	9.2± 1.23	9.4± 0.55	8.12± 0.53	**8** ± 0.55	**4** ± 0.43	**3.2** ± 0.50	**7.6** ± 0.65	**4** ± 0.43	**3.2** ± 0.50
	MLN	$\lambda_{pd} = 0$	13 ± 2.12	16 ± 3.45	12.4 ± 1.89	11.5 ± 1.76	14 ± 1.24	11 ± 1.76	10 ± 1.63	12.5 ± 2.32	10.5 ±1.26	11± 1.23	12.78± 1.80	11± 1.45	11.76± 0.92	14.2± 0.87	11.66± 1.06
		$\lambda_{pd} = 0.1$	12 ±2.01	14 ±1.73	12 ±1.55	10 ±1.46	10.8 ±1.83	9.75 ±1.40	8.25 ±0.64	9.4 ±1.95	9 ±0.79	9.5 ±0.57	9.98 ±0.97	9.75±1.40	10.35±0.94	10.4 ±0.87	11.3±0.75
		$\lambda_{pd} = 0.3$	**9.5** ±0.75	**9** ±1.04	**7.33** ±1.82	**8** ±1.87	**7.9** ±2.05	**6.75** ±0.97	**6.4** ± 1.20	**5.3** ±1.43	**3.75**±0.85	8.17 ±1.40	7.23±0.76	6.4±0.45	9.72±1.34	10.2±1.07	9±0.75
		$\lambda_{pd} = 0.6$	10.5 ±1.30	10.67± 0.92	9.5±1.05	9.25 ±1.27	10±0.60	8.25 ±0.79	8.5± 1.33	8.2 ±0.93	6.3±1.50	10.25± 1.46	8.9±1.51	7.25 ±0.78	12±1.27	11.7±1.20	10.33±0.77
D3	PRM		13.5 ±2.40	13 ±2.13	10.75 ±1.53	12.2±1.13	12.2±1.40	9.23±1.87	10.23±1.23	8±1.45	6±1.67	7.5 ± 0.50	**6.2** ± 1.75	**3** ±1.20	**7.5** ± 0.50	**6.2** ± 1.75	**3** ±1.20
	MLN	$\lambda_{pd} = 0$	12 ±1.39	12 ±2.06	10.5±1.45	11.5 ±1.65	11±2.43	10±1.75	10.3 ±1.23	9.75 ±1.72	9 ±0.97	10.9±0.78	11±1.29	9.56±1.87	11.73±1.55	11.75 ±0.69	10±0.57
		$\lambda_{pd} = 0.1$	11 ± 1.30	10 ± 2.08	10 ±2.34	9.25 ± 1.67	9 ± 1.75	8.33 ± 1.33	7.75 ± 1.20	8 ± 1.45	8.25 ±1.55	9 ± 0.65	9.3 ± 0.88	8.98 ± 0.69	10.75 ±1.10	10.88 ± 1.32	11.25 ± 1.08
		$\lambda_{pd} = 0.3$	**8.6** ± 1.20	**7** ± 1.13	**8.4** ±0.78	**8** ± 1.45	**7.4** ± 2.13	**6.6** ±1.55	**5.4** ± 1.29	**6.9** ± 1.65	**4** ±0.57	**6.4** ± 0.99	7.46 ± 1.20	6.3±1.65	9.12 ± 0.79	10.4 ± 1.13	9.4 ± 0.63
		$\lambda_{pd} = 0.6$	9.6 ±0.76	10.84 ±1.14	9 ±1.76	8.4 ±0.77	8.4 ± 1.56	8.6 ±1.71	6.2 ±0.77	7.75 ±0.65	7.25 ± 1.03	9 ±1.20	8.9±0.88	7.6 ±0.79	10.2 ±1.00	9.25 ± 1.46	9.25±0.78

Table 6.4 – Average ± standard deviation of RSHD_PD for each algorithm for a particular sample size and network with a percentage α of exceptions in the relationship instances fixed to 10% . Bold values present the best values for a given model and a given sample size.

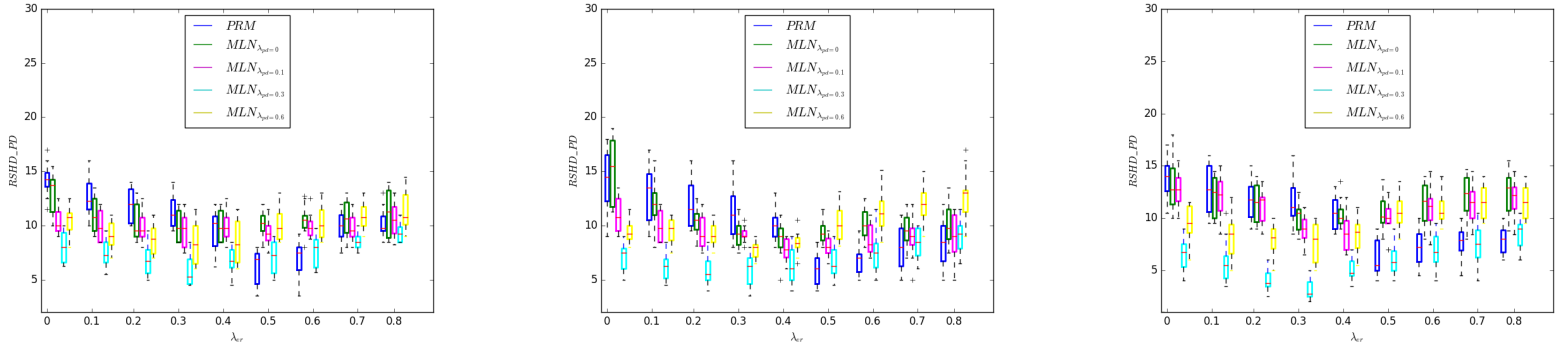


(a) Data size =1000 (b) Data size =3000 (c) Data size =5000

Figure 6.5 – The Average values of RSHD_PD with respect to λ_{pd} when executing approach 1 and 2 with (a) Data size =1000 and (b) Data size =3000 and (c) Data size =5000 for DAPER1 with a percentage α of exceptions in the relationship instances fixed to 30%.

	Algo		λ_{er}														
			0			0.1			0.3			0.5			0.8		
			Data size			Data size			Data size			Data size			Data size		
			1000	3000	5000	1000	3000	5000	1000	3000	5000	1000	3000	5000	1000	3000	5000
D1	PRM		15.46 ±3.08	16.33 ± 2.34	11.21 ± 1.50	13.43 ± 2.67	14.22 ± 2.64	10 ± 1.31	11.21 ± 1.54	12.48 ± 2.41	8.67 ± 1.31	9.6 ± 2.49	8.81 ± 1.64	6.73 ±1.74	9.9 ±1.17	9.13 ±1.07	8.74 ±2.02
	MLN	$\lambda_{pd} = 0$	14.58 ± 2.70	14.63 ± 1.40	12.31 ±1.89	13.11 ± 2.76	11.45 ± 2.20	10.51 ± 1.55	9.13 ± 1.07	9.16 ±1.13	10.2 ± 2.14	11.5 ± 1.63	9.9 ± 1.45	9.37 ±1.24	13.69 ±2.54	10.76 ± 1.26	11.28 ± 1.68
		$\lambda_{pd} = 0.1$	12.44 ± 2.13	11.75 ± 1.62	10.21 ± 1.47	12.4 ± 2.06	10.54 ± 1.24	9.48 ± 1.43	8.28 ± 1.42	8.83 ± 1.19	8.81 ±1.64	9.94 ±1.68	9.33± 1.25	8.4 ± 1.32	11.7 ±1.72	9.76 ± 1.05	9.83 ± 1.47
		$\lambda_{pd} = 0.3$	**9.9** ± 1.17	**8.44** ± 1.21	**7.72** ±1.71	**9.31** ±1.15	**7.16** ± 1.09	**6.78** ± 1.57	**6.58** ± 1.49	**6.38** ± 1.48	**5.68** ± 1.38	**7.1** ±0.97	**6.97**±1.30	**5.9** ± 1.14	**9.58** ±1.12	**8.73** ± 1.10	**7.61** ±0.97
		$\lambda_{pd} = 0.6$	10.22 ± 1.47	9.13 ± 1.07	9.02±1.63	10.22 ± 1.47	9.17 ± 1.22	8.15 ± 1.07	7.78 ± 1.81	8.2 ± 1.16	7.7 ± 1.33	10.72±1.78	10.65±1.51	10.14 ±1.98	12.86± 1.76	12.67 ± 2.17	11.92 ± 2.20
D2	PRM		14.4 ± 1.49	14.58 ± 2.70	14.07 ± 2.04	12.82 ± 1.86	13.5 ± 2.49	13.29 ± 2.36	11.58± 1.68	11.81 ± 2.47	11.91 ± 2.12	**6.58**±1.49	**6.4** ±1.52	6.73 ± 1.74	10.42± 1.38	**8.63**±1.70	**7.16** ±1.09
	MLN	$\lambda_{pd} = 0$	13.26 ± 1.79	15.66 ± 2.95	13.72 ± 2.57	11.25 ± 1.66	12.44 ± 2.13	12.31 ± 1.89	10.2 ± 1.44	9.49 ± 1.25	10.21 ± 1.47	10.33± 1.16	9.62 ± 1.09	10.57 ±1.43	10.5 ±1.46	10.41 ± 1.81	11.42 ±1.93
		$\lambda_{pd} = 0.1$	10.56 ± 1.12	11.25 ± 1.66	12.98 ± 1.70	10.2 ± 1.44	10.2 ± 1.40	12.3 ± 1.84	10.19 ± 1.43	9.25 ± 0.68	9.17 ± 1.22	9.553 ± 1.20	8.15 ±0.92	9.07 ± 1.96	10.4 ±1.46	8.82 ± 1.83	8.9± 1.41
		$\lambda_{pd} = 0.3$	**8.24** ± 1.37	**7.21** ±1.18	**5.77** ± 1.19	**7.5** ± 1.34	**6.47** ± 1.31	**4.31** ± 1.13	**6.47** ± 1.31	**6.3** ±1.29	**3.57** ± 1.08	7.47± 1.47	6.9±1.43	**5.32**±1.08	**9.18** ±0.89	9.3 ±1.46	7.7 ± 1.87
		$\lambda_{pd} = 0.6$	10.128 ± 1.09	9.6 ± 1.09	8.44 ± 1.21	9.17 ± 1.22	9.69 ± 1.15	8.12 ± 1.54	8.74 ± 1.90	7.83 ±0.78	8.22 ± 1.67	10.32± 1.54	10.61±1.74	10.73±1.57	11.67± 1.78	13.18 ±2.46	11.73±1.62
D3	PRM		15.6 ±2.21	14.3 ± 2.40	13.89 ±1.87	13.5 ± 1.62	12 ± 1.86	12.6 ±1.06	12.21 ± 0.92	10.39 ±1.72	10 ± 1.12	**6.4** ± 0.89	**4.29** ±0.62	**4.12**± 1.40	9.8 ±1.13	9.12 ± 1.24	8.45 ± 1.70
	MLN	$\lambda_{pd} = 0$	15.9 ±2.40	15.23 ±2.31	13.34 ± 1.79	14.12 ± 1.91	12.71 ±1.45	12 ±1.03	12.13 ± 1.09	10.8 ± 1.56	10 ± 1.21	11.59 ±1.03	11 ± 1.55	10.78 ±2.34	12.3 ± 1.89	12.5 ± 1.70	12 ±1.20
		$\lambda_{pd} = 0.1$	13.2 ± 2.34	12.22 ± 1.45	11.4 ± 1.07	12.23 ± 2.01	10.9 ± 1.79	10.21 ±1.54	10.4 ± 1.14	9 ± 0.78	9.3 ± 1.04	10.89 ± 0.68	9.56 ± 0.98	10± 0.73	12 ± 1.12	11.79 ± 1.05	10.87 ±0.76
		$\lambda_{pd} = 0.3$	**8.6** ± 0.67	**7.7** ± 1.15	**6.43** ± 0.73	**7.8** ± 1.04	**6.63**± 1.09	**6** ± 0.65	**6.8** ± 1.12	**5.8** ± 0.55	**4.2** ± 1.03	7.21 ± 0.72	6.9 ± 0.97	6.4 ± 0.76	**7.14** ± 1.02	**7.14** ± 1.12	**8.34** ± 0.78
		$\lambda_{pd} = 0.6$	10 ±1.40	12.25 ±2.13	11 ± 1.72	9.3 ± 1.14	9.12 ±1.50	9.45 ±0.65	8.73 ± 1.25	8.2 ±1.42	9 ±0.79	9.13 ± 1.45	9.7 ±1.55	9.89 ±1.74	12.63 ± 2.15	12.3 ±1.23	11.45 ±1.43

Table 6.5 – Average ± standard deviation of RSHD_PD for each algorithm for a particular sample size and network with a percentage α of exceptions in the relationship instances fixed to 30% . Bold values present the best values for a given model and a given sample size.

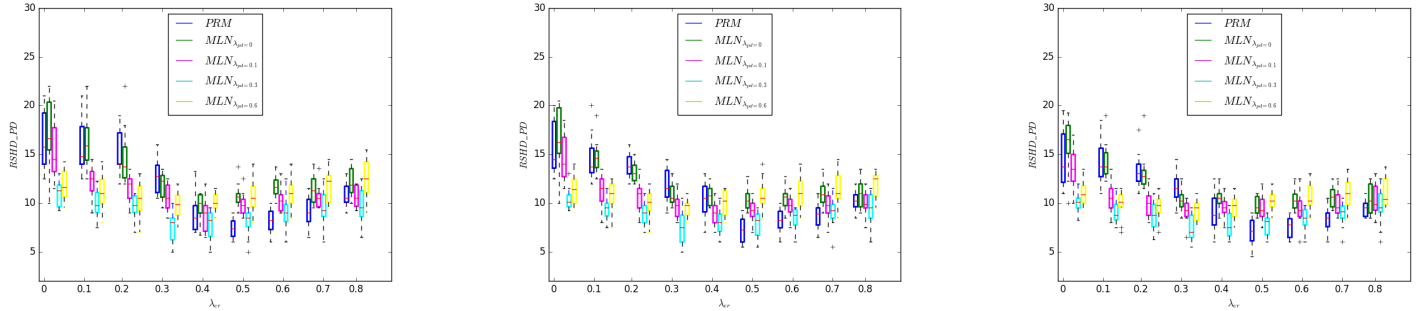


(a) Data size =1000 (b) Data size =3000 (c) Data size =5000

Figure 6.6 – The Average values of RSHD_PD with respect to λ_{pd} when executing approach 1 and 2 with (a) Data size =1000 and (b) Data size =3000 and (c) Data size =5000 for DAPER2 with a percentage α of exceptions in the relationship instances fixed to 30%.

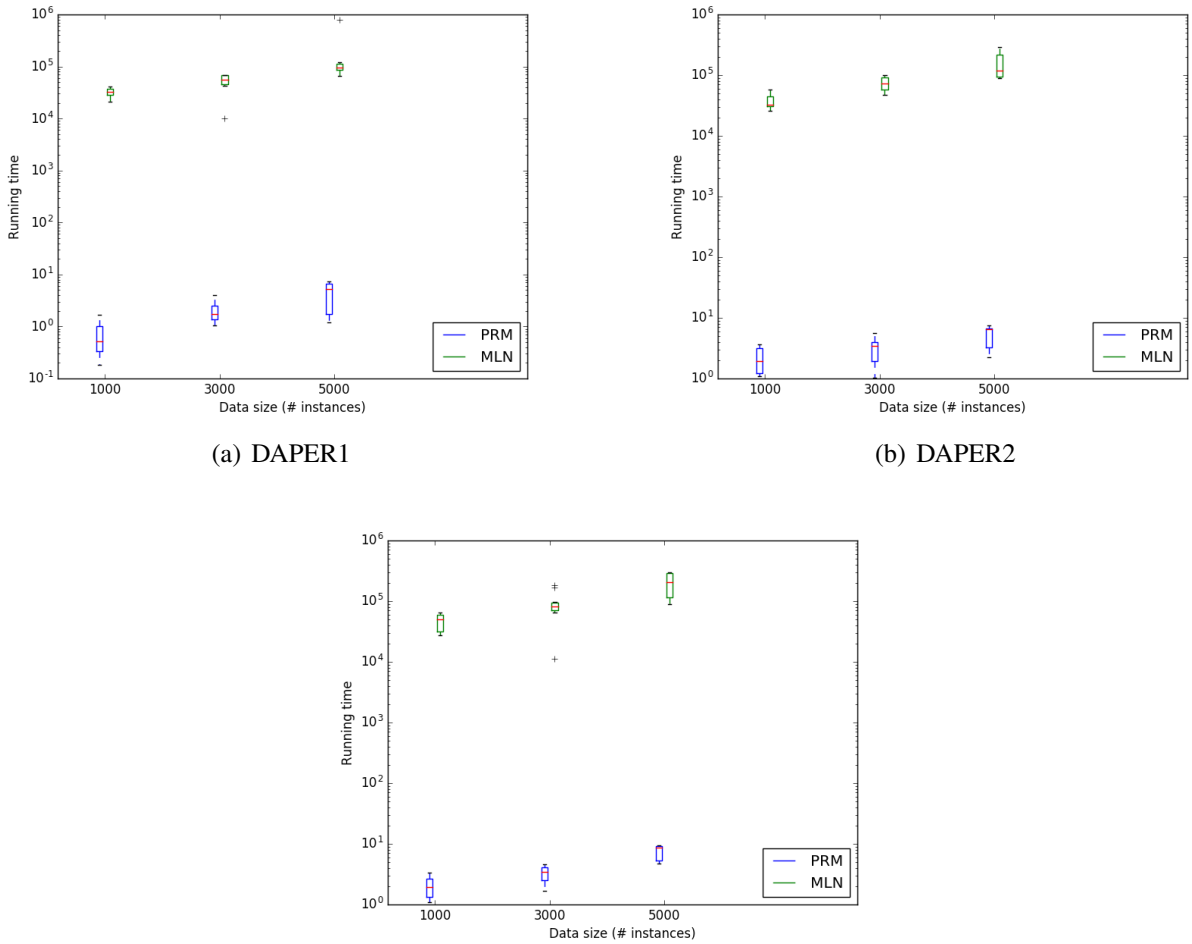
	Algo		λ_{er}														
			0			0.1			0.3			0.5			0.8		
			Data size			Data size			Data size			Data size			Data size		
			1000	3000	5000	1000	3000	5000	1000	3000	5000	1000	3000	5000	1000	3000	5000
D1	PRM		16.3 ±2.12	15.79 ± 2.73	14.27 ± 1.19	14.71 ± 2.16	12.54 ±1.50	11.39 ±2.74	12 ± 1.73	11.63± 1.07	10.76 ± 1.34	9.65 ±1.72	9.2 ±1.15	**7.12** ±1.32	11.98 ± 1.14	10.88 ±1.72	9.61±0.89
	MLN	$\lambda_{pd} = 0$	16±2.24	15.45 ± 1.90	13.17 ±1.19	14.27 ± 1.72	12.24 ± 0.73	11.5 ± 1.10	11.3 ± 1.55	10.98 ± 1.77	10.38 ± 1.45	12.4±1.22	11.37 ± 1.84	10.82 ± 0.56	13.6± 1.21	11.98 ±1.62	11.75±1.53
		$\lambda_{pd} = 0.1$	13.9 ±2.20	10.75 ± 2.12	10.14 ± 2.44	12. ±1.40	11.2 ± 1.09	9.51 ± 0.75	9.46 ± 1.49	9 ± 1.74	8.74 ± 1.56	10.5±1.78	9.73±1.55	9.57± 1.63	11.28±1.54	11.76±1.45	10.73 ±0.76
		$\lambda_{pd} = 0.3$	**10.2**±1.14	**9.96** ± 2.00	**9.2** ± 1.50	**8.5** ± 1.64	**8.63** ± 1.55	**7.94** ± 1.18	**7.39** ± 1.89	**6.76** ± 1.75	**5.7**± 1.76	**9.19** ±1.23	**8.98** ±1.73	7.46±1.80	**9.79**±0.92	**9.68**±1.37	**8.75**±1.55
		$\lambda_{pd} = 0.6$	11.45 ± 1.10	10.79 ± 1.45	11 ± 1.89	10.65 ± 1.66	10 ± 1.20	9.76 ± 1.42	10 ± 0.74	8.5 ± 1.19	8.73 ± 1.47	11.74± 1.04	11± 1.55	10±1.00	12.39± 1.15	12± 1.88	11.13±1.77
D2	PRM		16.96 ±2.93	16.19 ± 2.73	15.17 ± 3.11	16.31 ± 2.82	15.03 ±2.53	14.59 ±2.25	12.94 ± 1.79	12.13± 1.67	11.76 ± 1.56	**7.56** ±0.92	**7.48** ±1.35	**7.39** ±1.32	10.98 ± 1.24	10.35 ±1.02	9.51±0.79
	MLN	$\lambda_{pd} = 0$	18±3.24	17.29 ± 2.91	16.37 ±2.49	16.87 ± 2.97	15 ± 1.83	14.56 ± 2.08	11.96 ± 1.34	11 ± 1.18	10.48 ± 1.19	10.94±1.32	10.17 ± 0.86	9.82 ± 0.86	11.6± 1.51	10.98 ±1.60	10.72±1.53
		$\lambda_{pd} = 0.1$	15.9 ±2.80	14.95 ± 2.32	13.94 ± 2.14	12.28 ±1.48	11.51 ± 1.55	10.71 ± 1.65	10.69 ± 1.39	10.08 ± 1.40	9.34 ± 1.06	10±1.18	9.43±1.06	8.97± 1.13	10.8±1.04	10±1.45	10.13 ±1.16
		$\lambda_{pd} = 0.3$	**11**±1.24	**10.56** ± 1.13	**10.15** ± 1.00	**10** ± 1.40	**9.53** ± 1.10	**9.24** ± 1.18	**8.32** ± 1.15	**8.26** ± 1.31	**7.7**± 1.40	8.3 ±1.26	8 ±1.23	7.6±1.20	**9.49**±0.92	**8.8**±1.57	**8.5**±1.50
		$\lambda_{pd} = 0.6$	12 ± 1.60	11.59 ± 1.66	11.11 ± 1.36	11.46 ± 1.86	10.86 ± 1.55	10 ± 1.31	10 ± 1.38	9.66 ± 1.01	9.33 ± 1.09	11.24± 1.64	11.12± 1.60	10.35±1.02	12.28± 1.85	11.47± 1.18	11.13±1.47
D3	PRM		15.9 ±2.13	16.59 ± 2.20	16.17 ± 2.81	15.2 ± 2.71	15.77 ±2.23	15.19 ±2.65	13.24 ± 1.12	13± 0.77	12.56 ± 1.74	**8.36** ±1.72	**8** ±1.04	**7.59** ±1.40	11.28 ± 1.10	10 ±1.54	9±1.49
	MLN	$\lambda_{pd} = 0$	18.79±3.77	17.88 ± 3.71	17.17 ±2.86	17 ± 2.50	15.4 ± 1.07	15.16 ± 2.00	11.56 ± 1.24	11.2 ± 1.77	10.78 ± 1.76	11±1.44	10.97 ± 1.18	10.3 ± 1.14	11.9± 1.92	11.43 ±1.66	11.8±1.03
		$\lambda_{pd} = 0.1$	16.2 ±2.70	15 ± 2.40	13.94 ± 2.14	13 ±1.77	12 ± 1.04	11.3 ± 1.43	10 ± 1.12	10.78 ± 1.47	10.71 ± 1.54	10.56±1.23	10.16±1.77	9.77± 1.87	11±1.57	11.3±1.77	11 ±1.24
		$\lambda_{pd} = 0.3$	**11.4**±1.70	**11** ± 1.20	**10.75** ± 1.50	**10.86** ± 1.55	**9.73** ± 1.48	**9.84** ± 0.88	**8.72** ± 1.60	**8.88** ± 1.40	**8**± 1.50	9.66 ±1.74	9 ±1.23	9.2±1.20	**10.49**±0.92	**9.8**±1.57	**9.75**±1.77
		$\lambda_{pd} = 0.6$	13 ± 1.00	12.19 ± 1.72	11.76 ± 1.77	12.3 ± 1.44	11 ± 1.46	10.87 ± 1.41	10 .67 ± 1.45	9.86 ± 1.53	9.86 ± 1.59	11.77± 1.24	12± 1.70	10.85±1.54	12.74± 0.85	12± 1.15	11.76±1.52

Table 6.6 – Average ± standard deviation of RSHD_PD for each algorithm for a particular sample size and network with a percentage α of exceptions in the relationship instances fixed to 50% . Bold values present the best values for a given model and a given sample size.



(a) Data size =1000 (b) Data size =3000 (c) Data size =5000

Figure 6.7 – The Average values of RSHD_PD with respect to λ_{pd} when executing approach 1 and 2 with (a) Data size =1000 and (b) Data size =3000 and (c) Data size =5000 for DAPER2 with a percentage α of exceptions in the relationship instances fixed to 50%.



(a) DAPER1

(b) DAPER2

(c) DAPER3

Figure 6.8 – Boxplot of running time (in seconds and log scale) for each sample size of each DAPER.

However, for the first approach, if we compare its RSHD_PD values presented in Table 6.5 with the ones of the same approach of Table 6.4, we notice that RSHD_PD values increase for all datasets sizes. This is explained by the fact that we have increased the percentage of exceptions. So, the number of perturbed relationships for one given signature is sufficient to generate more and more false relationships so the RSHD_PD increases.

This is more impressive in Table 6.6, where we have more increased the percentage of exceptions to α=50%.

Figure 6.7 shows the RSHD_PD with respect to λ_{er} parameter for DAPER learning using PRM framework and with respect to λ_{pd} and λ_{er} parameters when we use MLN framework for a percentage of exceptions α=50%. This figure proves our conclusions deduced from the Figures 6.5 and 6.6.

Figure 6.8 represents running time for each DAPER for each approach, in log scale time. We can notice that increasing the size of our sample do not have a bad impact when we use the first process. However, with the second one the running time is multiplied by a factor greater than 10.

Despite, the second approach gives best results in term of quality reconstruction when dealing with partially structured datasets, especially when applying specified values of the threshold λ_{PD}. This is thanks to the power of first-order-logic to handle exceptions. However, it gives the worst results,

		DAPER1		DAPER2	
α		10%	30%	30%	50%
Data size	1000	MLN	MLN	MLN	MLN
	3000	PRM	MLN	OK	PRM
	5000	PRM	MLN	PRM	MLN

Table 6.7 – A comparative study of obtained optimum values of both methods using the statistical student test with a critical threshold= 5%.

comparing with the first approach, in term of running time.

Table 6.7 gives a statistical comparison between optimum values obtained by learning algorithms using PRM framework and learning using MLN framework with a critical threshold= 5% and for a degree of liberty: $d.d.l = n_a - n_b - 2$, where n_a and n_b are the size of each observation values.
We can conclude that with few data, MLN gives best results in term of quality reconstruction measures. However, by increasing the data size and the percentage of exception generating α, the gap between both approaches is decreasing. MLN are very useful for partially structured contexts thanks to their ability to manage exception. By using MLN, we can ask for the probability of an object even it is an exception for the dataset. However, it is not possible to infer such object using DAPER model. In the other side, learning PRM algorithm are faster than MLN one.

To explain the critical results of MLN method in term of running time, we make the following discussion. In fact, the running time of such algorithm involves two things.

1. Access time to the database. An MLN algorithm needs to know if a formula is relevant or not by computing how many groundings are true for this formula in the database.
2. The algorithm complexity.

The converted graph databases of MLN algorithm are stored in text files. However, our first contribution data are stored in graph databases. That's why we obtained interesting results in term of running time for the first approach. Now, if we compare running time of MLN algorithm with PRM algorithm which uses a relational database to learn its DAPER as shown in Chapter 5 (cf. Section 5.3.5, Figure 5.7) we can notice that MLN algorithm always gives bad results in term of running time. Therefore, we can conclude that converting the used database of MLN from text file to graph database cannot resolve the problem of the high running time. This is explained by the complexity of MLN structure learning algorithm itself which must be revised and improved.

6.5 Conclusion

In this chapter, we were interested in learning Directed Acyclic Probabilistic Entity Relationship (DAPER) models, from partially structured databases. We proposed here joint learning from partially structured graph databases, where we simultaneously learn the ER schema and the probabilistic dependencies. The Markov Logic Network formalism appeared as an efficient solution for this task. We show with the experimental study that MLN structure learning can effectively identify both parts of the DAPER model in one single task, with a comparative precision, but with very high time complexity. MLN semantics also restrict ourself by only identify undirected dependency structures instead of the DAPER directed one. Reversely, the logical formulas used by MLN to describe both the ER schema and the probabilistic dependencies can manage exceptions, i.e., data which are not coherent with an underlying structured model, which is not possible with the DAPER framework.

7

Conclusion

Contents

7.1 Summary 108
7.2 Future works 109

7.1 Summary

In this thesis, we explored the potential for using Probabilistic Relational models (PRMs) with partially structured databases especially with graph databases. We focused on Directed Acyclic Probabilistic Entity Relationship (DAPER) models and Markov Logic Networks (MLNs). These models are in fact quite related because both of them are probabilistic models that can be learned from relational data. Also, each one of them has its advantages that can be combined to benefit from the flexibility of statistical learning with the descriptive power of First-Order-Logic (FOL) to develop learning models and algorithms capable of representing complex relationships among entities in uncertain and partially structured domains. However, the intersection of DAPER and MLN is not much explored in the machine learning community.

The first part of this dissertation was dedicated to a survey on Probabilistic Graphical Models (PGMs), PRMs and Graph database. Chapter 2 deals with PGMs, namely Bayesian networks and Markov networks which are used to learn from flat data. The chapter gives their definition, utility and learning approaches allowing their construction from observational data. Chapter 3 applies to PRMs namely, DAPER models and MLNs which are extensions of respectively Bayesian networks and Markov networks to the relational context. The chapter provides DAPERs and MLNs definition and existing approaches to learn them from relational data. Also, we have discussed the complexity of learning these models from relational data in term of time-consuming because of the expensive cost of the "join" operation in relational databases. Also, we have shown the lack of techniques to compare an MLN and a DAPER learning algorithms in term of quality reconstruction. Chapter 4 provides a quick overview on NoSQL databases and focuses mainly on the graph database. In this chapter, we have shown that in contrast to relational databases, where join-intensive query performance deteriorates as the dataset gets bigger, the graph database depends less on a rigid schema and its performance tends to remain relatively constant, even as the dataset grows. All these observations were the subject of the second part of this thesis, which is dedicated to our contributions.

Our contributions were developed in Chapters 5 and 6 in which they were approved experimentally. Chapter 5 provides a method to learn DAPERs from partially structured graph databases. First, we presented how to extract the underlying ER model. Then, we described a method to compute sufficient statistics based on graph traversal techniques. The main goals of this contribution are to learn DAPERs with less structured data and to speed up the learning process by querying graph databases. The second part of the chapter concerns the evaluation of our approach. We have presented three experimental protocols. The first experimental protocol deals with a limited size of generated datasets. Then, to evaluate the scalability of our approach we have performed our second experimental protocol with larger datasets. In this experimental protocol, we have increased the number of instances so we have fully justified our choice of adopting graph databases for DAPER learning. Also, we have chosen a simple value and maximum value of k_{max} as we have fixed the same value for some generated models to illustrate the impact of this parameter on both the running time and the quality of reconstruction. The third experimental protocol deals with learning from noisy datasets. We have generated random exceptions in relationship instances of some datasets in order to study the impact of these exceptions on the learned structures. In these experiments, we have used naive synthetic datasets that are generated by applying the naive sampling algorithm. To evaluate the performance of the proposed approach when it will be used in real-world situations, we had generated more realistic datasets by applying k-partite sampling algorithm as described in our fourth experimental protocol. Experimental results showed that our approach presents good results either in term of running time or in term of accuracy. However, by using more realistic k-partite graph databases the running time increases as well as the number of stored instances.

Our second contribution introduced in Chapter 6, addressed the missing theoretical work for the overlapping of DAPERs and FOL to deal with partially structured data. We have proposed to learn DAPERs from partially structured graph databases using MLN frameworks. We have described how to express DAPER in Markov logic. Then, a method to learn at the same time the ER schema and the probabilistic dependencies using an MLN learning algorithm is detailed. Our contribution consists of two essential components. First, we have described how to identify the ER schema and the set of probabilistic dependencies expressed via FOL framework from the learned MLN. Second, we have proposed an evaluation distance measure which will be used to compare the extracted DAPER components from the learned MLN and the true DAPER model. We have redefined the RSHD measure to the relational and logical context. The second part of the chapter consists of an experimental study, where we have shown that MLN structure learning can effectively learn both parts (ER schema and probabilistic dependencies) of DAPER model in one single task. We have performed an experimental protocol in which we have compared this approach to our first contribution. The experiments have shown that DAPER joint learning using MLN framework gives correct results by studying the influence of the thresholds λ_{er} and λ_{pd} but with a remarkably higher computation time and with undirected dependencies.

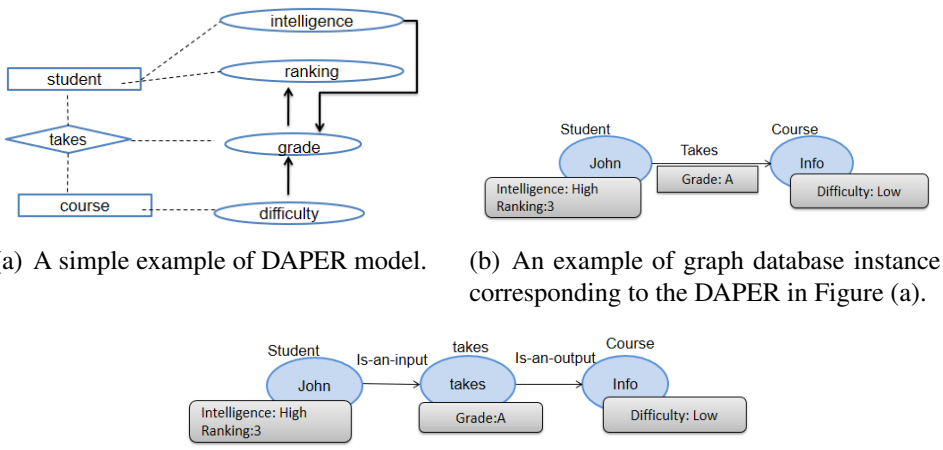
7.2 Future works

During the research presented in this thesis, many potential directions for future works have opened up. In the following, we list some of them.

Improvement of DAPER structure learning algorithms from graph databases by giving a new definition to the DAPER model As relationships in the graph database are not indexed, we propose to change the manner how we define the DAPER model.

In fact, the DAPER model is defined based on the ER schema. We propose to define the relationshipclass of the ER schema as a node object in the graph database instead of a direct edge and the relationships between nodes will be represented by "is-an-input" or "is-an-output" labeled links in the graph database as shown in Figure 7.1.

Improvement of DAPER structure learning algorithms from graph databases with new algo-



(a) A simple example of DAPER model.

(b) An example of graph database instance corresponding to the DAPER in Figure (a).

(c) The new proposed database structure corresponding to the DAPER in Figure (a).

Figure 7.1 – A new definition for DAPER model in context of graph databases.

rithms for DAPER structure learning Our first proposed method for learning DAPER structure from graph database are based on the standard relational greedy search (RGS) algorithm for learning DAPERs. (Ben Ishak, 2015) has shown through experiments that RGS algorithm is less accurate compared to newer algorithms, such as RMMHC. DAPER learning from graph databases with these newer algorithms might improve the quality and the running time of the learned DAPER. Therefore, we are currently performing other rigorous experiments with more realistic datasets with these new algorithms.

Improvement of our approach to deal with DAPERs with structural uncertainty Our contributions deal with regular DAPER models which provide a model for domains where attribute values are uncertain. In these models, all relations between attributes are known, uncertainty exists in the descriptive attributes only. We propose to extend our approaches to deal with the cases where both attributes and link structure are uncertain modeling more complex structural uncertainty namely: RU (reference uncertainty) and EU (existence uncertainty).

Application to recommender systems We could explore the potential for using DAPER in recommendation systems. We are proposing the overlapping of PRM-PrefReco (Probabilistic Relational Model for Preference-based Recommenders) (Chulyadyo, 2016) and graph databases to address the recommendation task.

Improvement of DAPER joint learning using other logic frameworks As our objective is obtaining an efficient probabilistic framework dealing with partially structured graph databases, we are now interested by improving this work into several directions. Since MLN structure learning algorithms seem to suffer from complexity issues, we are also interested by other probabilistic and relational frameworks derived from Logic such as ProbLog models (Raedt et al., 2007). If we can confirm that DAPER structure learning is less complex than its MLN/ProbLog counterparts, the last perspective would be to improve MLN/ProbLog structure learning by first learning a more restrictive by less complex DAPER model.

Applying our proposed contributions in diverse domains using real datasets Though we have evaluated our approaches with only synthetic graph databases, it is a generic solution and can be applied in the domain of real states too . Some of the applicable domains are community detection in social networks, hotel/restaurant recommendation systems, etc.

Incremental DAPER structure learning from graph data streams Nowadays a growing number of applications generating massive streams of data, which need intelligent data processing and on-line analysis. These systems have to update their existing knowledge in the light of novel data. Therefore, incremental data mining has been an active area of research. As a long-term perspective, we want to propose new algorithm to deal with the continuous arrival of a large amount of data in the form of graph data streams. Also, an incremental learning algorithm (Yasin, 2013) could be used to study the interest of our method under incremental contexts.

IV

Appendices

A

PILGRIM relational

A.1 Introduction

In order to evaluate our contributions, we implemented all the proposed algorithms into the C++ PILGRIM project which is under development. The PILGRIM software tool allows to deal with several probabilistic graphical models. It presents the development effort of several participants who have made or are making their academic researchs within the Data User KnowledgE (DUKe) research group of the LS2N lab [1].

As our contributions are based on using graph databases. So we will also introduce Neo4j [2] graph database management system. We have chosen to work with Neo4j because it is the most popular, used and developed system. Also there is a whole community that work under this system who provide many documentations, tutorials and forums to help people who want to use it. Neo4j is available on commercial and community editions.

Section A.2 [3] provides an overview of the PILGRIM project and lists the used environment and softwares. Then, Section A.3 focuses especially in the PILGRIM Relational framework.

A.2 Pilgrim project

PILGRIM (**P**robab**IL**istic **GR**aphIcal **M**odels) is actually under development to provide an efficient tool to deal with several probabilistic graphical models (e.g., BNs, PRMs, DAPER), or in PILGRIM platform all Bayesian Networks learned from relational datasets (e.g., PostgreSQL) and not from flat formats (e.g., PRMs, DAPER) are considered as Relational Bayesian Network. So DAPER or PRM models are implemented as RBN objects in PILGRIM. In this section we give a brief representation of the PILGRIM project. PILGRIM is a software tool for modeling, learning and reasoning upon probabilistic networks. It has support to:

1. Probabilistic networks modeling: We are particularly interested with directed acyclic graphs. Actually, PILGRIM allows to model:

1. https://ls2n.fr/
2. https://neo4j.com/product/
3. pilgrim.univ-nantes.fr

— Bayesian Network (BN)

— Relational Bayesian Network (RBN)

— Relational Bayesian Network with spatial attributes (RBN-SA)

— Relational Bayesian Network with Reference Uncertainty (RBN-RU)

2. Probabilistic networks learning: Besides models representation, Pilgrim addresses the main learning tasks related to those models, namely:

   — Parameter learning

   — structure learning

3. Probabilistic networks benchmarking: The validation of any learning approach goes through an evaluation process where benchmarks availability is essential. Thus, PILGRIM allows:

   — Model generation

   — Sampling

4. Probabilistic networks evaluation metrics: There are several metrics allowing to evaluate probabilistic graphical models structure learning algorithms. PILGRIM implements a set of metrics to evaluate the quality of reconstruction of the graph structure, namely:

   — Distance-based measures

   — Performance measures

PILGRIM is developed around four main sub-projects, namely PILGRIM general, PILGRIM structure learning, PILGRIM relational and PILGRIM application.

1. **PILGRIM general project**: was the first developed project it allows modeling standard probabilistic models. Also, it provides several structure learning algorithms for BNs (e.g., GS, MMHC). It also implements several score functions (e.g., BDeu, BIC, AIC, MDL), statistical tests (e.g., mutual information) and metrics to evaluate the quality of the reconstructed networks (e.g., SHD).
2. **PILGRIM relational project**: provides several functionalities to deal with probabilistic networks for relational data representation.
3. **PILGRIM application project**: contains some domain applications (e.g., RBN application to recommender systems) where probabilistic networks implemented by either PILGRIM general or PILGRIM relational have been used.

The next section is dedicated to the PILGRIM relational project and we specify libraries and datasets used by PILGRIM. Then, we will focus on the main module on which we have contributed.

A.3 PILGRIM Relational

PILGRIM relational is built on the Relation Bayesian Network (RBN) specification, thus it requires the data to be modeled through a relational schema (ER schema). The probabilistic structure defines the dependencies among the probabilistic attributes of that schema. Reference slots encode One-to-Many cardinality type, therefore it is possible that some children nodes have multiple parents for the same probabilistic dependency. As multiple parents have to be aggregated, PILGRIM relational defines several aggregation functions and Multi-set operators (e.g., AVERAGE, MODE, INTERSECTION). To do so, The PILGRIM Relational project uses several existing C++ libraries to manage graph structures, to manipulate BNs objects, to communicate with a RDBMS or to run unit tests.

A.3.1 Existing libraries

The Boost Graph Library (BGL): is a C++ open source library that provides a generic open interface for traversing graphs. Its implementation follows a generic programming principle and its source can be found as part of the Boost distribution [4]. BGL is characterized by its easy of use and integration in any program: no need to be built to be used, wealth of documentation and multiple code examples. It consists of a set of core algorithm patterns, namely, Breadth First Search, Depth First Search and Uniform Cost Search, and a set set of graph algorithms (e.g., Dijkstra Shortest Paths, Connected Components, Topological Sort).
Database Template Library (dtl): dtl is a C++ open source library. The specificity of this library is that it can run on multiple platforms and C++ compilers. In addition, Several DBMS are supported (Oracle, PostgreSQL, MySql, etc) and Database access is ensured through ODBC drivers. dtl allows to perform several manipulations on databases such as reading and writing records. It also allows to perform more other requests such as creating schemas, tables, constraints, etc. dtl is well documented and a variety of examples are given and commented. Moreover, instructions for using the library are provided and precision on how to use it with each DBMS is given [5].
Googletest: Released under the BSD 3-clause license [6], Google Test presents a library for writing C++ unit tests. It works on a variety of platforms and can be easily integrated to any c++ program. The library allows several test types and several options for running the tests [7].
ProBT Library: when we instantiate RBNs, ground Bayesian networks are BN objects defined using the ProBT library.

A.3.2 Additional libraries

Cpprest SDK also known as Casablanca. This Microsoft open source project is evolving in CodePlex [8] and takes advantage of the new set of capabilities introduced in C++. Microsoft developed the C++ REST SDK on top of the Parallel Patterns Library (PPL), and leverages PPL's task-based programming model. It enables you to stay in C++ when consuming REST services or developing other code closely related to the cloud. Such as, making calls to a synchronous API to make an HTTP GET call, retrieving and sending JSON documents via JSON parser and writer, etc.

A.3.3 Data accessibility

Some functionalities (e.g., parameter learning, structure learning) involve datasets as input. For standard probabilistic models, flat data representation (e.g., a text file) is used. For relational probabilistic models, relational data representation is needed. For the second case, the PostgreSQL Relational database management system has been used. Accordingly, to our contribution; learning RBN from graph database so we added Neo4j as graph database management system to deal with.

PostgreSQL: is an open-source object-relational database management system. Initially created at the University of California at Berkeley, PostgreSQL is now considered among the most advanced open-source database. It supports a large part of the SQL standard and provides the possibility to be used, modified, and distributed by anyone free of charge for any purpose, be it private, commercial, or academic [9].
Neo4j: Neo4j is an open source graph database developed by Neo Technology, implemented in Java

4. http://sourceforge.net/projects/boost/files/
5. http://dtemplatelib.sourceforge.net/
6. http://opensource.org/licenses/BSD-3-Clause
7. https://code.google.com/p/googletest/
8. https://casablanca.codeplex.com/
9. http://www.postgresql.org/docs/9.4/interactive/index.html

and accessible from software written in other languages using the Cypher query language [10] through a transactional HTTP endpoint. It is highly scalable, and schema free graph database that stores data as graphs. It allows developers to achieve excellent performance in queries over large, complex graph datasets and at the same time, it is very simple and intuitive to use. it can be used as is or can be embedded inside applications as well [11].

A.3.4 PRM specification in PILGRIM

A PRM is defined through the *RBN* class. This is the most important class in this module. All other modules rely on this class for PRM specification. An *RBN* object is composed of a *RelationalSchema* object, a dependency structure described by a set of *IRBNVariable* objects, and the associated CPDs described by *RBNDistribution objects*.

A.3.5 Relational schema in PILGRIM

All meta-model(ER schema or relational schema)are specified in PILGRIM through the class *RelationalSchema*. A relational schema is stored in the form of a graph such that classes are represented by vertices, and the reference slots (or foreign keys) by edges. Each vertex of this graph is an object of Class. The class *RelationalSchema* allows us to manage the set of relations (modify, consult, add, move,...). Another also makes it possible to
We can also browse the graph according to a starting relation and a sequence of references, via set of methods such as *getClassRefForSlotChain()* and *isGuaranteedAcyclicSlotChain()*.

Class: represents a class of a relational schema. It consists of a set of attributes and an identifier (a primary key).
Attribute class: represents an attribute of a class. Each attribute is identified by a name, and can take a value from a finite set. This set of values is modeled by Domain class.
Domain: is an abstract class wich represent a finit set ofpossible values for an attribute. It has the following implementations: *MultinomialDomain*, *ProBTDomain*, *ContainingNullDomain*, and *CompositeDomain*.

Figure A.1 shows how classes RelationalSchema, Class, Attribute, and Domain are related to each other.

A.3.6 PRM learning module

This package includes algorithms for RBN parameter learning as well as structure learning as well as needed score functions and statistical tests. It offers statistical (MLE) as well as Bayesian (MAP, EAP, and Laplace) approaches to parameter learning. For structure learning, the current version of PILGRIM-Relational provide the implementation of RGS, RMMHC, RMMPC, and $\overline{RMMPC}$ algorithms which are implemented in the classes *RGS*, *AlgoRMMHC*, *AlgoRMMPC* and *AlgoRMMPCBar* respectively.

A.3.7 PRM benchmark generation module

To evaluate implemented learning approaches, PILGRIM-Relational offers the relational database benchmarks that allow to check probabilistic dependencies among the attributes. This package is

10. http://neo4j.com/docs/stable/cypher-query-lang.html
11. http://neo4j.com/

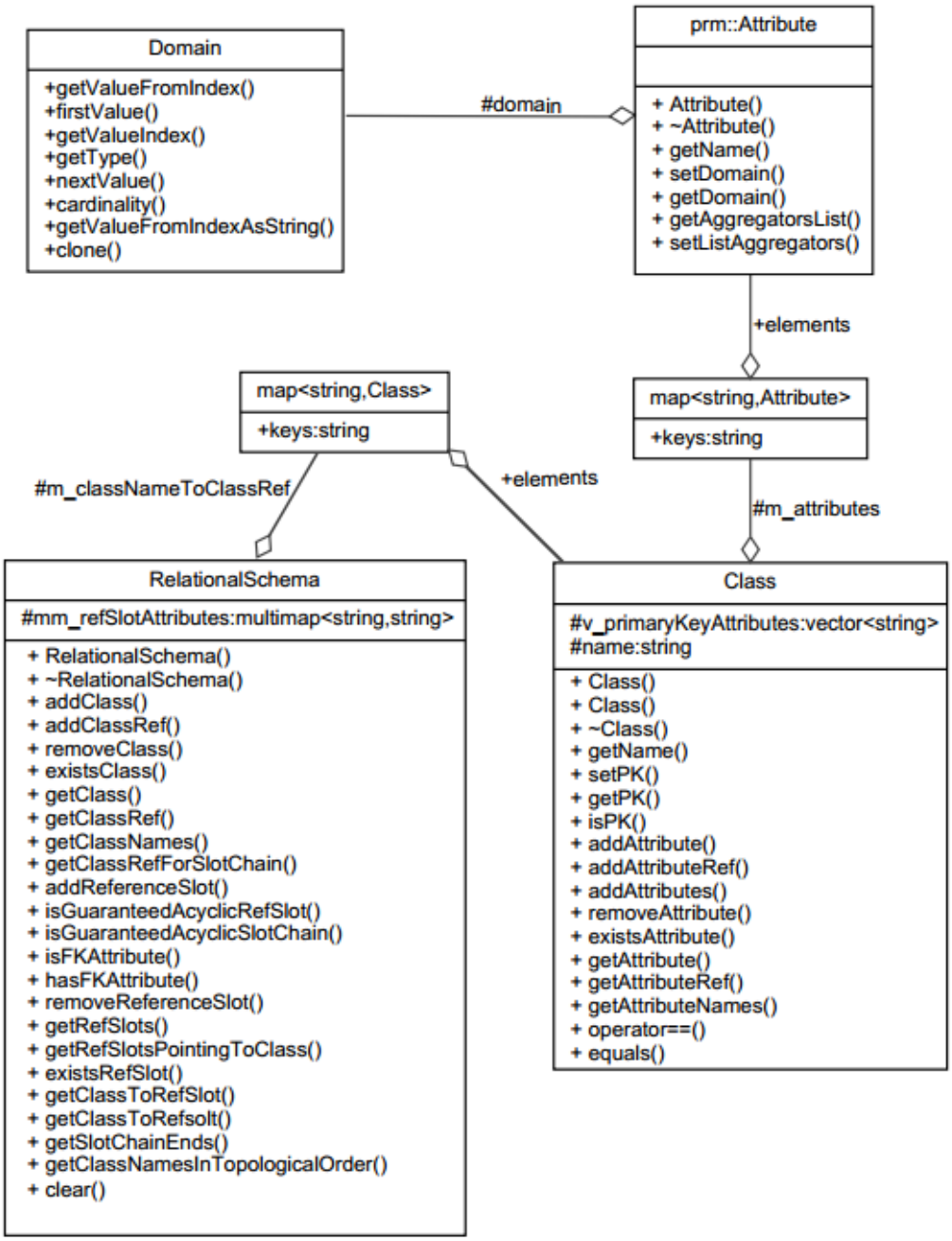


Figure A.1 – Class diagram for the *RelationalSchema*, *Class*, *Attribute* and *Domain* classes.

initially implemented by (Ben Ishak, 2015) and extended by (Chulyadyo, 2016).

A.3.8 PRM extension module

Three PRM extensions are also implemented in PILGRIM-Relational. Namely, PRM with Reference Uncertainty (PRM-RU),PRM with Clustering Uncertainty (PRM-CU) developed by (Coutant, 2015) and PRM with Spatial Attributes (PRM-SA) developed by (Chulyadyo, 2016). These extensions inherit RBN class, and add data structures and functionalities specific to them

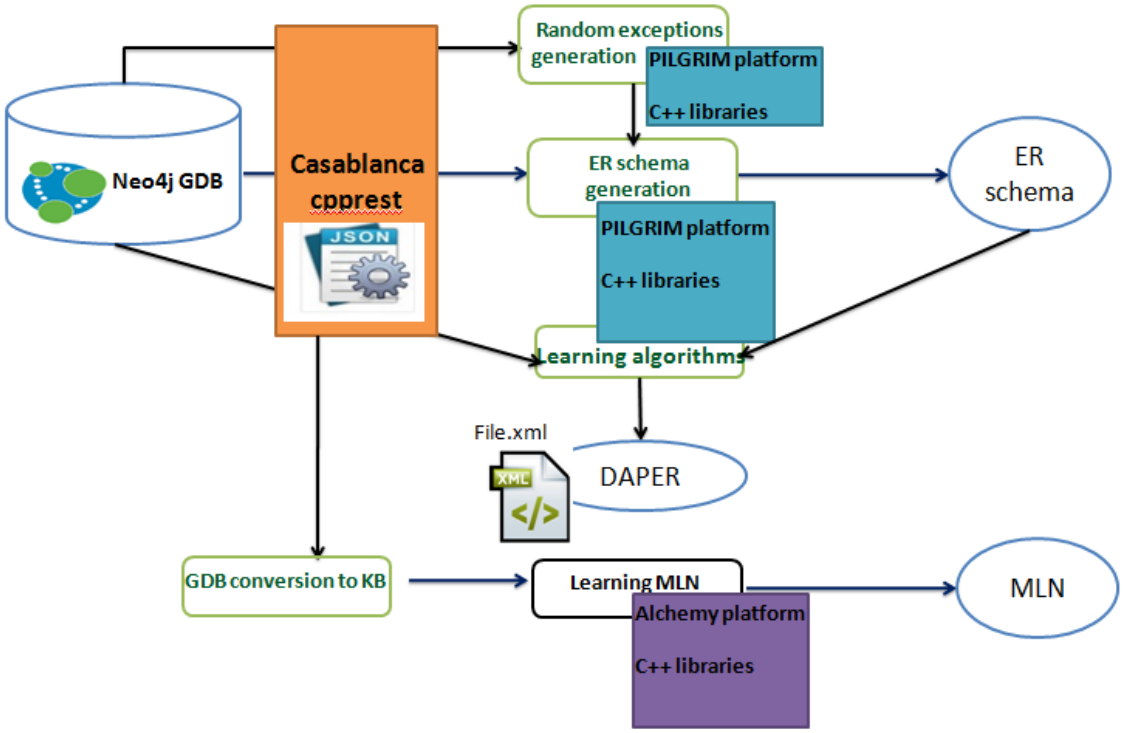


Figure A.2 – Implementation environment

A.3.9 Learning from graph database

As this thesis contributes to the implementation of counting methods used for learning DAPER from graph databases, we will explain its implementation in detail in this section. An overview of the implementation environment of our contributions and used software tools is shown in Figure A.2.

A.3.9.1 ER schema identification

The implementation of *RelationalSchema* class in PILGRIM-Relational does not have support for graph database. Therefore, a new implementation was needed to generate an ER schema. In fact in the case of graph database, we generate in the reality an ER schema but we return as a structure a *RelationalSchema* object respecting the specificities of PILGRIM-Relational.

So to identify the ER schema from graph database we have defined a *Neo4jGenerator* class. In this class we implemented our algorithms of ER schema identification described in Chapter 5, Section 5.2.2.

Neo4jGenerator class also contain *generateExceptions ()* method which add randomly some exception to a given graph database. This method is used for our contribution and experiments: learning DAPER from a noisy graph database. An other method was implemented under this class is *ConvertToKnowledgeDB ()*. This method allow user to convert a graph database to a knowledge database used for MLN learning.

To ensure the communication between PILGRIM-Relational and Neo4j graph database, we have implemented the class *Neo4jRequest*. In this class, we find all methods which access to the graph database to return information we need about data stored inside. *GetCountNode* returns the number of instance of a particular node type, *WhoHasProperty ()* returns the type of object who has a particular property,*GetNodesOfProperty ()* returns all nodes which have a given property, *GetRela-*

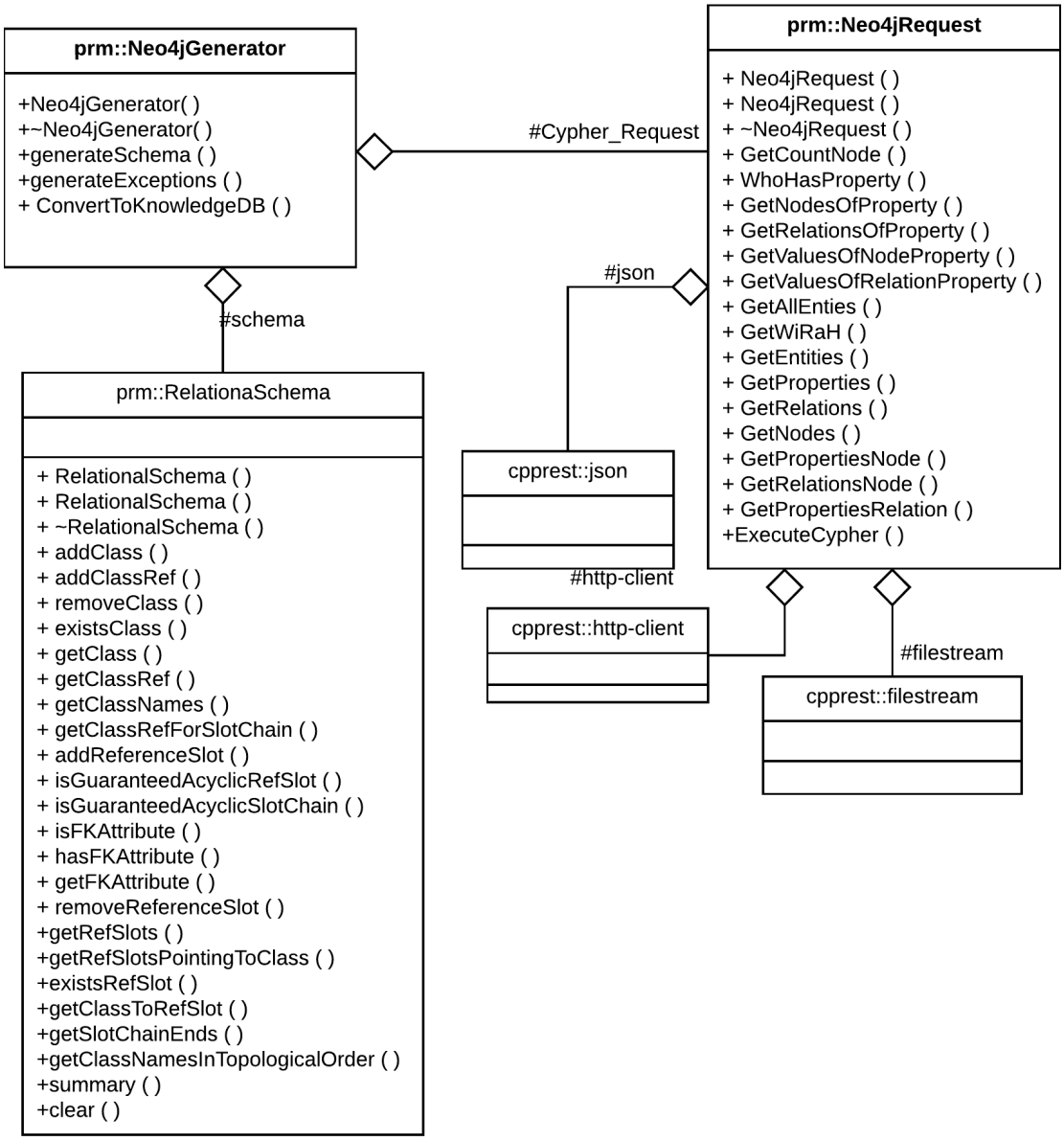


Figure A.3 – Class diagram for *Neo4jGenerator* and *Neo4jRequest*.

*tionsOfProperty ()*returns all relationships which have a given property, *GetValuesOfNodeProperty ()*, *GetValuesOfRelationProperty ()* return the values taken by a given property associated to a particular node and relationship respectively. *GetAllEnties ()* to get all nodes in the database, *GetWiRaH ()* to get what is related and how. In other worlds, return each relationship and the peer of nodes that it connects, *GetProperties ()* to get all properties, *GetRelations ()* to get all relationships in the graph database, *GetNodes ()* to get all nodes of a particular type, *GetPropertiesNode ()* returns the list of properties of a given node, *GetRelationsNode ()* returns the list of relationships of a given node, *GetPropertiesRelation ()* returns the list of properties of a given relationship, *ExecuteCypher ()* to execute a cypher query.

The class diagram of *Neo4jGenerator* and *Neo4jRequest* is presented in Figure A.3.

A.3.9.2 Instance

Instance is an abstract class that defines an interface for a complete or a partial instantiation of a relational schema identified from a relational database. In our case we have developed a *Neo4jInstance*

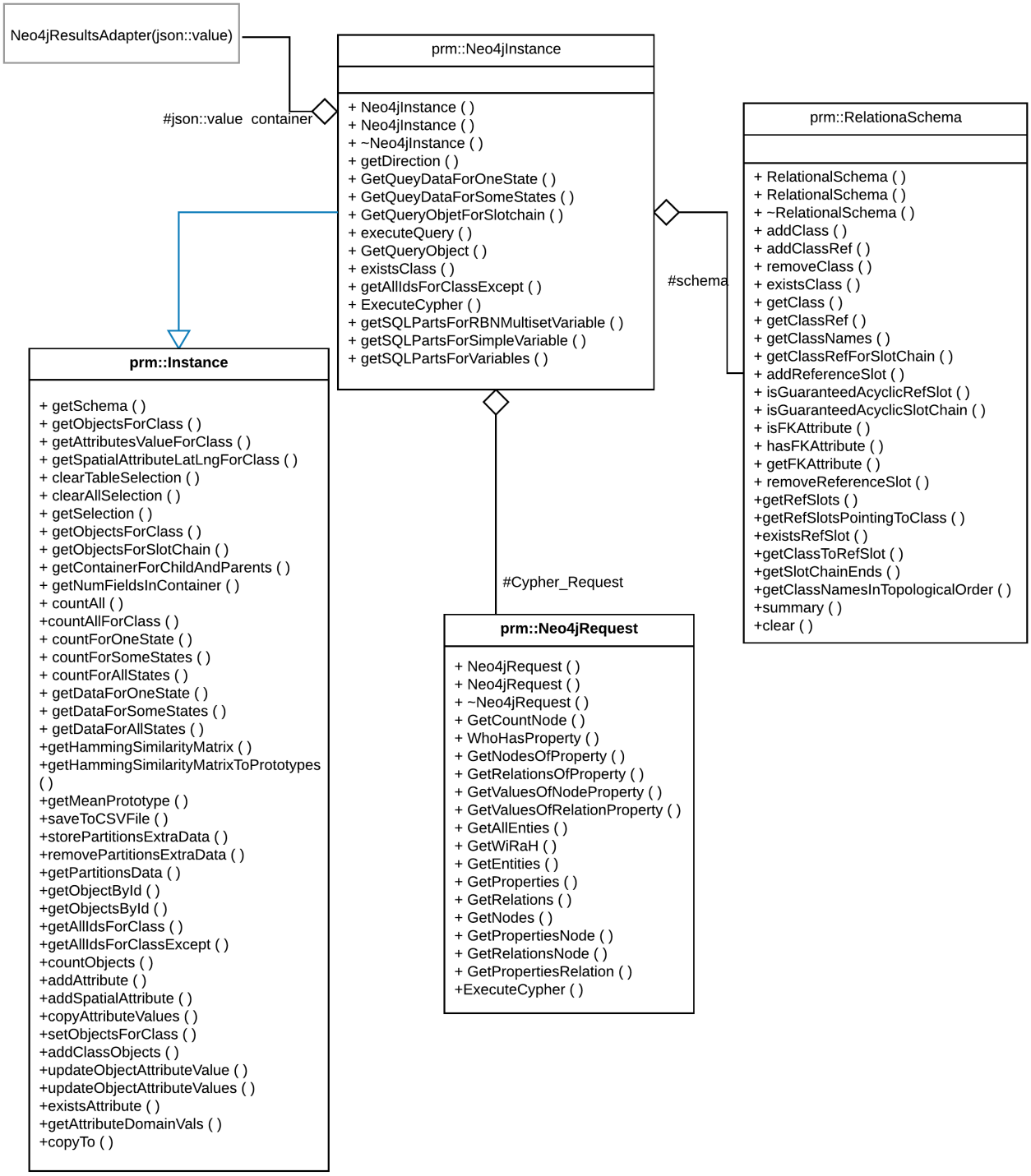


Figure A.4 – Class diagram for *Neo4jInstance*

which inherits the class *Instance* and in which we have defined all methods which will be used specifically with Neo4j graph database. Before, PILGRIM-Relational has two implementations of the class *Instance*: (1) *MockInstance*, which is an in-memory storage of a relational schema instantiation in the form of a graph, and is basically implemented for testing purposes, and (2) *DBInstance*, which is used to communicate with a relational schema instantiation stored as a relational database, and is specialized for PostgreSQL databases only and based on SQL queries. Currently, by adding our new class PILGRIM-Relational has three implementations of the class *Instance*.

Neo4jInstance: we have implemented in this class the main methods family which allows to count

the occurrences of different configurations of variable values. The methods of this family include *CountAll ()* to count all individuals in a sequence of particular variables without considering precise values, *countForOneState ()* to consider only a precise set of values for a sequence of variables, *countForSomeStates ()* that extends the previous method to a list of value configurations, and finally *CountForAllStates ()* which performs a set of counts for all possible value configurations in the order obtained by an iterator of type RBNValues on the considered view. These methods are used to compute sufficient statistics and other statistical indicators when learning models.

The second family of methods is used to retrieve values from the dataset. The set of methods is organized as follow:
getAllData () to get a full graph, *getDataForOneState ()* to retrieve the pattern of a particular configuration, *getDataForSomeStates ()* for several configurations, and finally *GetDataForAllStates ()* for all configurations.

A third family of methods is used to manage the visibility of the data set represented by the instance. Indeed, in order to manage learning data sets And test data sets without physically dividing the information into several Databases, we have allowed to filter the information of each relationship scheme at the level of the instance classes. The *clearTableSelection ()*, *clearAllSelection ()*, *getSelection ()* methods are used to manage relational filters, either by a list of primary key IDs or by executing a request or by both. Queries are managed by the list of methods: *GetQueyDataForOneStates ()*, *GetQueyDataForSomeStates()*, *GetDirection()*,...
Also we have implemented the methods : *getSQLPartsForRBNMultisetVariable ()*, *getSQLPartsForSimpleVariable ()* and *getSQLPartsForVariables ()* but by adapting these methods to Cypher graph database language. In other worlds, we left the same methods' signatures because we respect the *Instance* class from which we inherit our *Neo4jInstance* class, but inside the methods we return our cypher queries which we need to manage our graph dataset.
An other class of methods is also defined in PILGRIM-Relational but we didn't use in our case because it is dedicated for clustering algorithms such as *getHammingSimilarityMatrix()*.

The simplified class diagram of the different classes associated with the class *Neo4jInstance* is given in Figure A.4.

A.4 Conclusion

In this chapter, we introduced PILGRIM, a software for working with probabilistic graphical models. We presented a brief overview of different sub-projects of PILGRIM. We focused on project where this thesis has contributed the most: PILGRIM-Relational and we have listed our contributions in the development of this software.

B

Neo4j graph database

B.1 Introduction

As previously mentioned, relational databases have their appropriate use cases. For highly structured, predetermined schemas, an RDBMS is the perfect tool. But as we've seen, relational databases aren't always enough. Applications that require connected data insights can't rely on the relational model. We already know now that relational databases are not very performant for handling PRMs learning specially for highly connected and unstructured data. So that, our contribution was to use as alternative graph database. In this section we introduce the graph database management system that we have used during our implementations. the so called Neo4j. In this chapter, we introduce Neo4j graph database.

B.2 Neo4j graph database

Neo4j [1] is a NOSQL graph database. It is a fully transactional database (ACID) that stores data structured as graphs. It offers high query performance on complex data, while remaining intuitive and simple for the developer. Neo4j is developed by the Swedish-American company Neo technology. It has been in commercial development for 10 years and in production for over 7 years. Most importantly it has the largest and most vibrant, helpful and contributing community surrounding it.

The Neo4j database is built to be extremely efficient for handling node links. This performance is due to the fact that Neo4j pre-calculates joins at the time of data writing, compared to relational databases that calculate joins to read using indexes and key logic. Neo4j is the only graph database that combines native graph storage, scalable architecture optimized for speed, and ACID compliance to ensure predictability of relationship-based queries. This graph database uses Cypher query language as a declarative, pattern-matching language for connected data. This language offer a query plan visualization. This can be really useful, and allows possibilities to rephrase queries in order to allow an optimization to occur. Also, Neo4j provides results based on real-time data providing real time insights of whats happening with the data. This makes Neo4j a suitable technology for large,

1. https://neo4j.com/

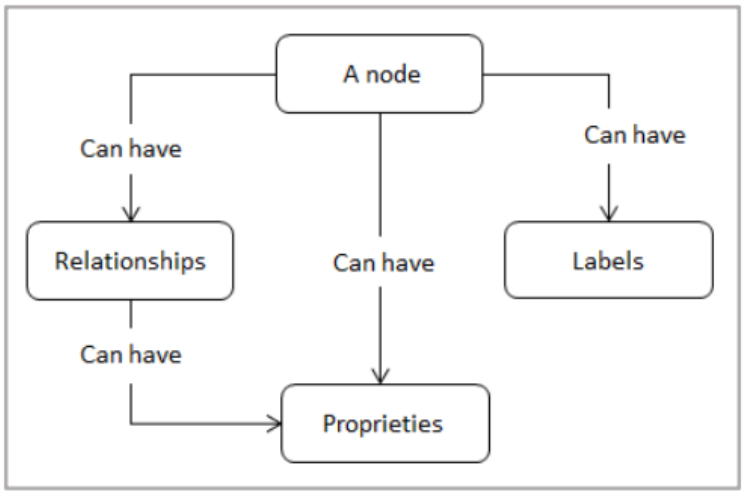


Figure B.1 – Neo4j graph database model.

connected data sets.

In Neo4j, both nodes and relationships can contain properties. Nodes and relationships can also be labeled with zero or more labels. Figure B.1 represents the Neo4j graph database model.
Some particular features make Neo4j very popular among users and developers:

— has an intuitive, rich graph-oriented model for data representation.

— has a native storage manager optimized for storing graph structures with maximum performance and scalability. Each piece of data in Neo4j has an explicit connection to every related entity, meaning database queries can ignore anything that's not connected rather than crawl the entire dataset. The result is unparalleled speed and scale.

— provides full database characteristics including ACID transaction.

— is scalable. Neo4j can handle graphs with many billions of nodes/relationships/properties on a single machine, but can also be scaled out across multiple machines for high availability.

— has a powerful traversal framework and query languages for traversing the graph.

— can be deployed as a standalone server or an embedded database.

— Written on top of the JVM with a core Java API.

— Materialize relationships at creation time, resulting in no penalties for complex runtime queries.

Figure B.2 resumes the Neo4j graph database features.

B.2.1 Connecting to Neo4J graph database

You can download and install Neo4j as a server on all operating systems. Then you can connect to Neo4j via its browser which an interactive, web-based database interface, via REST API or via language drivers.

The Neo4j browser [2] the easiest way to connect to the Neo4j Browser is through the localhost:7474 port to execute your graph queries (written in Cypher) in a workbench-like fashion. Results are presented as either intuitive graph visualizations or as easy-to-read, exportable tables. Figure B.3 is a snapshot from the Neo4j browser. Once you have data in Neo4j, the "database icon" in the upper-left

2. https://neo4j.com/developer/guide-neo4j-browser/

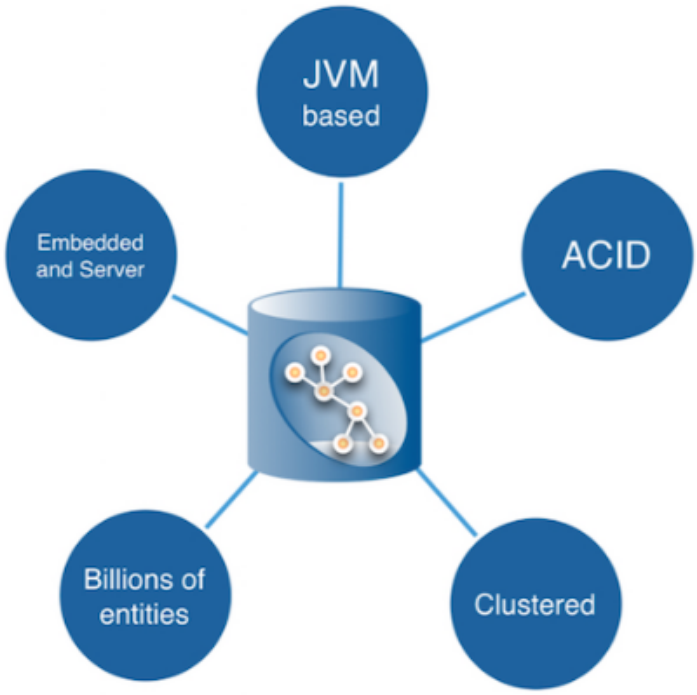


Figure B.2 – Neo4j graph database features.

corner will allow you to view the data graphically.

The simplest way of getting started with Neo4j is to use Neo4j's database browser. However, for programmers and applications developers they still want application to connect to Neo4j through other means, most often through a driver for their programming languages they use. They can do so with the REST API or via language drivers.

The Neo4j REST API [3] the Neo4j server provides access to the graph database via an HTTP-based REST API which allows you to execute a series of Cypher statements within the scope of a transaction. The REST API allows to:

— POST one or more Cypher statements with parameters per request to the server

3. https://neo4j.com/docs/developer-manual/current/http-api/

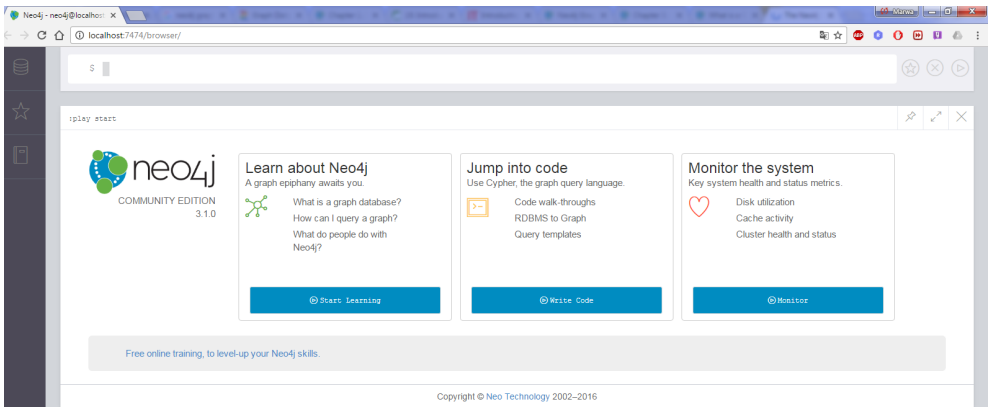


Figure B.3 – The Neo4j browser.

— Keep transactions open over multiple requests
— Choose different result formats
— Rollback an open transaction
— Include query statistics

The HTTP API supports also authentication and authorization so that requests to the HTTP API must be authorized using the username and password of a valid user.

```
:POST http://localhost:7474/db/data/transaction/commit
 {"statements":[
   {"statement":"CREATE (p:Person {name:{name}}) RETURN p",
"parameters":{"name":"Daniel"}}
  ]}
->
{"results":[{"columns":["p"],"data":[{"row":[{"name":"Daniel"}]}]}],
,"errors":[]}
```

Figure B.4 – A HTTP request example which executes Cypher query to create a Person.

An example of a HTTP request that executes Cypher to create a Person with the plain JSON response shown in Figure B.4.

Language drivers [4]: are connector libraries which provide application access to a Neo4j database. They have been designed to strike a balance between an idiomatic API for each language, and a uniform surface across all supported languages. Thanks to the Neo4j community, there are Neo4j drivers for almost every popular programming language. Namely, Java, .NET, JavaScript, Python, Ruby, PHP, R, Perl, C#, ...

A "Hello world" example [5] in Figure B.5 shows the minimal configuration necessary to interact with Neo4j through a driver designed for C# language.

B.2.2 Cypher query language

Declarative grammar with clauses (like SQL). Expressive language for creating, querying and updating graph databases. Very complicated database queries can easily be expressed through Cypher. Cypher focuses on the clarity of expressing what to retrieve from a graph, not on how to retrieve it. It borrows its structure from SQL. So that, queries are built up using various clauses such as CREATE, MATCH, MERGE,RETURN, ORDER BY, etc.

In the Table B.1, we give each cypher clause and its corresponding description.

The basic notion of Cypher is that it allows to ask the database to find data that matches a specific pattern.

Example B.2.1. *The Cypher pattern in Figure B.7 describes a path which forms a triangle that connects a node we call jim to the two nodes we call ian and emil, and which also connects the ian node to the emil node.*

An other example of Cypher query in Figure B.8 uses MATCH and RETURN clauses to find the mutual friends of a user named jim from the social graph pictured on Figure B.6.

Figure B.9 shows a comparaison between Cypher and SQL clauses.

4. https://neo4j.com/docs/developer-manual/current/drivers/
5. https://neo4j.com/docs/developer-manual/current/drivers/

```
public class HelloWorldExample : IDisposable
{
    private readonly IDriver _driver;

    public HelloWorldExample(string uri, string user, string password)
    {
        _driver = GraphDatabase.Driver(uri, AuthTokens.Basic(user, password));
    }

    public void PrintGreeting(string message)
    {
        using (var session = _driver.Session())
        {
            var greeting = session.WriteTransaction(tx =>
            {
                var result = tx.Run("CREATE (a:Greeting) " +
                                    "SET a.message = $message " +
                                    "RETURN a.message + ', from node ' + id(a)",
                    new {message});
                return result.Single()[0].As<string>();
            });
            Console.WriteLine(greeting);
        }
    }

    public void Dispose()
    {
        _driver?.Dispose();
    }

    public static void Main()
    {
        using (var greeter = new HelloWorldExample("bolt://localhost:7687", "neo4j", "password"))
        {
            greeter.PrintGreeting("hello, world");
        }
    }
}
```

Figure B.5 – A "Hello world" example shows the minimal configuration necessary to interact with Neo4j through a driver designed for C language.

B.2.3 From relational database to graph database

There are three main paradigms to deploying a graph database relative to relational database. Figure B.10 (Hunger et al., 2016) shows each of the paradigms for deploying both a relational and graph database.

First paradigm consists to abandon the relational database altogether and migrate all data into a graph database. Second one consists to continue using the relational database for any use case that relies on non-graph, tabular data and use graph database to store data for any use cases that involve a lot of JOINs. The third paradigm consists to duplicate all data into both a relational database and a graph database.

In our experiments we have implemented a script based on first paradigm of relational database converting to graph database. The easiest way to import data from a relational database is to create a CSV dump of individual entity-tables and join-tables. Then we take the CSV files and use Cypher's LOAD CSV power tool to:

- ingest the data, accessing columns by header name or offset.
- convert values from strings to different formats and structures (toFloat, split, ...)
- skip rows to be ignored.
- MATCH existing nodes based on attribute lookups.
- CREATE or MERGE nodes and relationships with labels and attributes from the row data.
- SET new labels and properties or REMOVE outdated ones.

Figure B.11 is an example of importing a CSV fle into Neo4j using the LOAD CSV Cypher command.

B.3 Conclusion

We have presented the Neo4j graph database system which we have used in our contributions. We

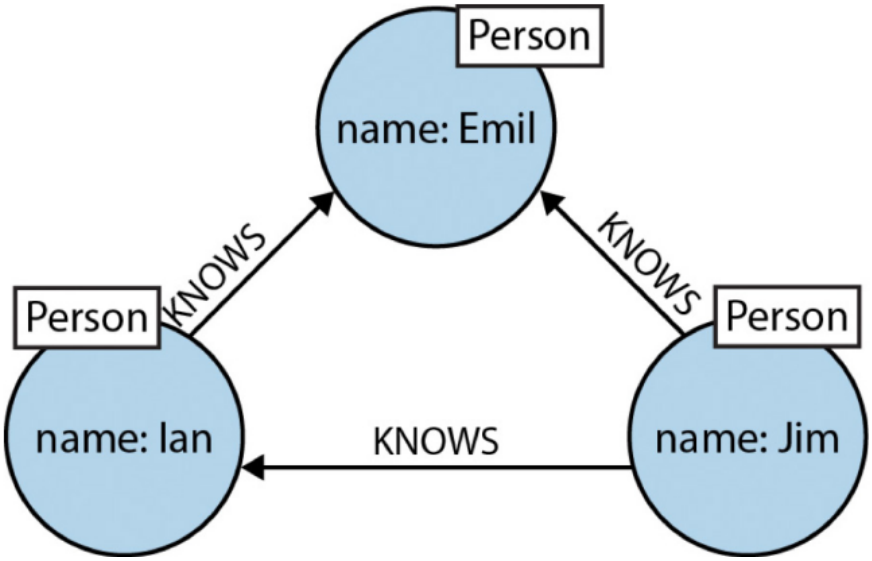


Figure B.6 – A social graph describing the relationship between three friends.

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)
```

Figure B.7 – A Cypher pattern example.

have also introduced the Cypher query language used for Neo4j graph database querying. Finally, we have gave a brief description on how we can convert a relational database to graph database.

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b:Person)-
      [:KNOWS]->(c:Person), (a)-[:KNOWS]->(c)
RETURN b, c
```

Figure B.8 – A Cypher query example contains MATCH and RETURN clauses.

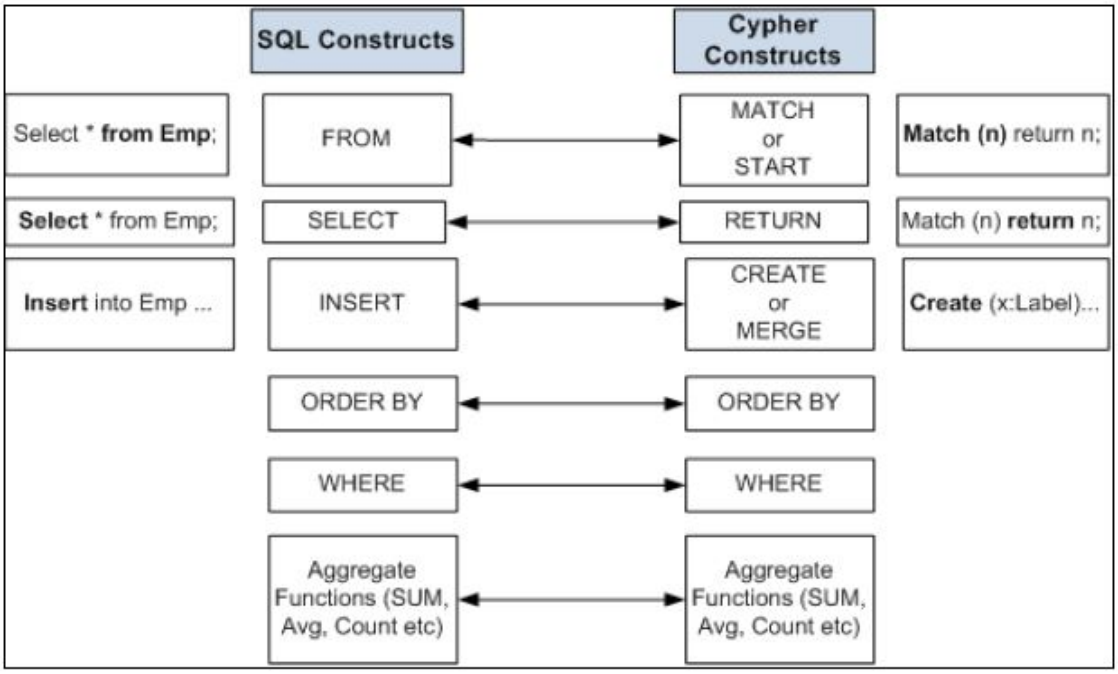


Figure B.9 – Cypher versus SQL language.

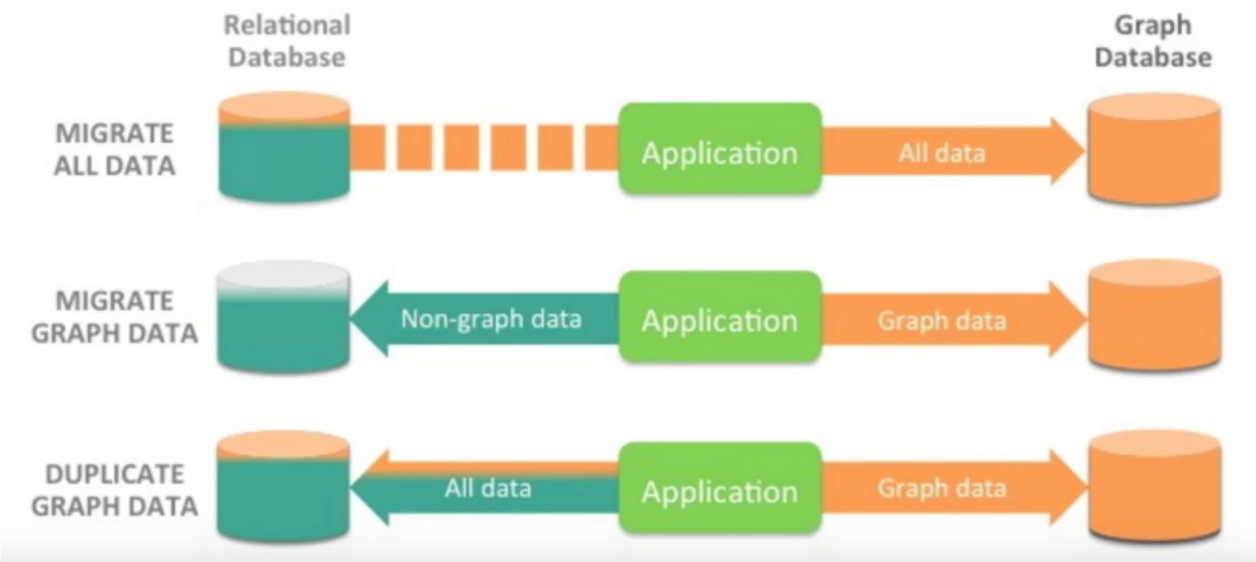


Figure B.10 – The three most common paradigms for deploying relational and graph databases (Hunger et al., 2016).

Clause	Description
MATCH	Specify the patterns to search for in the database
OPTIONAL MATCH	Specify the patterns to search for in the database while using nulls for missing parts of the pattern
START	Find starting points through legacy indexes
RETURN	Defines what to include in the query result set
WITH	Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next
UNWIND	Expands a list into a sequence of rows
WHERE	Adds constraints to the patterns in a MATCH or OPTIONAL MATCH clause or filters the results of a WITH clause
ORDER BY	A sub-clause following RETURN or WITH, specifying that the output should be sorted in particular way
SKIP	Defines from which row to start including the rows in the output
LIMIT	Constrains the number of rows in the output
USING INDEX	Index hints are used to specify which index, if any, the planner should use as a starting point
USING SCAN	Scan hints are used to force the planner to do a label scan (followed by a filtering operation) instead of using an index
USING JOIN	Join hints are used to enforce a join operation at specified points
CREATE	Create nodes and relationships
DELETE	Delete graph elements: nodes, relationships or paths. Any node to be deleted must also have all associated relationships explicitly deleted
DETACH DELETE	Delete a node or set of nodes. All associated relationships will automatically be deleted
SET	Update labels on nodes and properties on nodes and relationships
REMOVE	Remove properties and labels from nodes and relationships
FOREACH	Update data within a list, whether components of a path, or the result of aggregation
MERGE	Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created
ON CREATE	Used in conjunction with MERGE, this write sub-clause specifies the actions to take if the pattern needs to be created
ON MATCH	Used in conjunction with MERGE, this write sub-clause specifies the actions to take if the pattern already exists
CALL	Invoke a procedure deployed in the database and return any results
CREATE UNIQUE	A mixture of MATCH and CREATE,âmatching what it can, and creating what is missing
UNION	Combines the result of multiple queries into a single result set. Duplicates are removed
UNION ALL	Combines the result of multiple queries into a single result set. Duplicates are retained
LOAD CSV	Use when importing data from CSV files
USING PERIODIC COMMIT	This query hint may be used to prevent an out-of-memory error from occurring when importing large amounts of data using LOAD CSV
CREATE | DROP CONSTRAINT	Create or drop an index on all nodes with a particular label and property
CREATE | DROP INDEX	Create or drop a constraint pertaining to either a node label or relationship type, and a property

Table B.1 – A table contains information about the clauses in the Cypher query language.

```
LOAD CSV FROM 'file:///data/persons.csv' WITH HEADERS AS line
FIELDTERMINATOR ";"
MERGE (person:Person {email: line.email}) ON CREATE SET p.name = line.name
MATCH (dep:Department {name:line.dept})
CREATE (person)-[:EMPLOYEE]->(dept)
```

Figure B.11 – An example of importing a CSV file into Neo4j using the LOAD CSV Cypher command.

Bibliography

Acid, S. and de Campos, L. M. (2003). Searching for Bayesian network structures in the space of restricted acyclic partially directed graphs. *J. Artif. Intell. Res. (JAIR)*, 18:445–490. 23

Ahmadi, B., Kersting, K., and Hadiji, F. (2010). Lifted belief propagation: Pairwise marginals and beyond. In P. Myllymaeki, T. Roos, T. J., editor, *Proceedings of the 5th European Workshop on Probabilistic Graphical Models (PGM–10)*, Helsinki, Finland. 46

Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39. 61

Bartlett, M. and Cussens, J. (2017). Integer linear programming for the bayesian network structure learning problem. *Artif. Intell.*, 244:258–271. 22

Ben Ishak, M. (2015). *Probabilistic relational models: learning and evaluation*. PhD thesis, Université de Nantes, Ecole Polytechnique ; Université de Tunis, Institut Supérieur de Gestion de Tunis. 40, 81, 97, 110, 117

Ben Ishak, M., Leray, P., and Ben Amor, N. (2016). Probabilistic relational model benchmark generation. *Intelligent Data Analysis*, 20(3):615–635. 78, 79, 97

Besag, J. (1975). Statistical analysis of non-lattice data. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 24(3):179–195. 43

Biba, M., Ferilli, S., and Esposito, F. (2008a). Discriminative structure learning of Markov Logic Networks. In *ILP*, ILP '08, pages 59–76. 46

Biba, M., Ferilli, S., and Esposito, F. (2008b). Structure learning of Markov Logic Networks through iterated local search. In *18th European Conference on Artificial Intelligence ECAI2008*, pages 361–365. 44

Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer Verlag. 4

Borgelt, C. and Kruse, R. (2002). *Graphical models - methods for data analysis and mining*. Wiley. 22

Brewer, E. A. (2000). Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 7, New York, NY, USA. ACM. v, vii, 54, 55

Buntine, W. (1991). Theory refinement in Bayesian networks. In *Proceedings of the 7th Conference on Uncertainty in Artificial Intelligence*, pages 52–60. Morgan Kaufmann Publishers. 22

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39:12–27. 53

Chickering, D. M. (2003). Optimal structure identification with greedy search. *J. Mach. Learn. Res.*, 3:507–554. 20, 22

Chickering, D. M., Geiger, D., and Heckerman, D. (1995). Learning Bayesian networks: Search methods and experimental results. In *Preliminary papers of the 5th International Workshop on Artificial Intelligence and Statistics*, pages 112–128. 22

Choudhury, S., Holder, L. B., Jr, G., Mackey, P., Agarwal, K., and Feo, J. (2014). Query optimization for dynamic graphs. *CoRR*, abs/1407.3745:1–13. 63

Chow, C. J. and Liu, C. N. (1968). Approximating discrete probability distributions with dependence trees. *IEEE Trans. on Information Theory*, 14(3):462–467. 22, 26

Chulyadyo, R. (2016). *A new horizon for the recommendation: Integration of spatial dimensions to aid decision making*. PhD thesis, Université de Nantes. 110, 117

Chulyadyo, R. and Leray, P. (2018). Using probabilistic relational models to generate synthetic spatial or non-spatial databases. In *Proceedings of IEEE 12th International Conference on Research Challenges in Information Science (IEEE RCIS'2018)*, Nantes, France. 78, 80

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387. 32

Collins, M. (2002). Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 1–8. Association for Computational Linguistics. 26

Cooper, G. (1992). The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42(2-3):393–405. 22

Cooper, G. and Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–347. 21

Coutant, A. (2015). *Probabilistic Relational Models and Reference Uncertainty*. PhD thesis, Université de Nantes. 117

Cruz, I. F., Mendelzon, A. O., and Wood, P. T. (1987). A graphical query language supporting recursion. In *SIGMOD Conference*, pages 323–330. ACM Press. 56, 57

Dan, P. (2008). BASE: An ACID alternative. *Queue*, 6:48–55. 53, 54

Darwiche, A. (2013). A differential approach to inference in Bayesian networks. *CoRR*, abs/1301.3847. 28

Das, M., Wu, Y., Khot, T., Kersting, K., and Natarajan, S. (2015). Graph-based approximate counting for relational probabilistic models. In *Working Notes of the 5th International Workshop on Statistical Relational AI (StarAI@UAI)*, pages 1–9. 47, 62

Dasgupta, S. (1999). Learning polytrees. In *UAI*, pages 134–141. Morgan Kaufmann. 26

Dash, D. and Druzdzel, M. (1999). A hybrid anytime algorithm for the construction of causal models from sparse data. In Laskey, K. B. and Prade, H., editors, *UAI*, pages 142–149. Morgan Kaufmann. 23

Date, C. (2003). *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8 edition. 32

Date, C. J. (2008). *The Relational Database Dictionary, Extended Edition*. Apress, Berkely, CA, USA, 1 edition. 32

de Salvo Braz, R., Amir, E., and Roth, D. (2005). Lifted First Order Probabilistic Inference. In *IJCAI*, pages 1319–1325. 48

de Salvo Braz, R., Amir, E., and Roth, D. (2006). MPE and partial inversion in lifted probabilistic variable elimination. In *Proceedings of the 21sth National Conference on Artificial Intelligence (AAAI)*, pages 1–8. 48

de Salvo Braz, R., Natarajan, S., Bui, H., Shavlik, J., and Russell, S. (2009). Anytime lifted belief propagation. pages 10–13. 46

De Virgilio, R., Maccioni, A., and Torlone, R. (2014). *Model-Driven Design of Graph Databases*, pages 172–185. Springer International Publishing. 63, 70

Diestel, R. (2005). *Graph Theory*. Springer. 12

Dinh, Q. T. (2011). *Statistical relational learning : Structure learning for Markov logic networks. (Apprentissage statistique relationnel : apprentissage de structures de réseaux de Markov logiques).* PhD thesis, University of Orléans, France. 32, 35, 44, 94

Domingos, P. and Lowd, D. (2009). *Markov Logic: An Interface Layer for AI.* Morgan & Claypool, San Rafael, CA. 46

Domingos, P. and Richardson, M. (2004). Markov logic: A unifying framework for statistical relational learning. In *Proceedings of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields*, pages 49–54. 92

Domingos, P. and Webb, A. (2012). A tractable first-order probabilistic logic. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, pages 1902–1909. AAAI Press. 32, 35

Domke, J., Karapurkar, A., and Aloimonos, J. (2008). Who killed the directed model? In *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, pages 1 – 8, Los Alamitos, CA, USA. IEEE Computer Society. 28

Durrett, R. (2009). *Probability: Theory and Examples*. Duxbury. 12

Edlich, S. (2011). *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken.* Hanser. 53

El Abri, M., Leray, P., and Essoussi, N. (2017). Daper learning from (partially structured) graph database. In *14th ACS/IEEE International Conferenceon Computer Systems and Applications AICCSA, IEEE*, Hammamet, Tunisia. 68

El Abri, M., Leray, P., and Essoussi, N. (2018). Daper joint learning from partially structured graph databases. In *3rd International Conference on Digital Economy ICDEc, Springer*, Brest, France. 92

EMA (2013). How graph databases solve problems in network & data center management: a close look at two deployments. Technical report, An Entreprise Management Associates (EMA). 61

Fan, W., Wang, X., and Wu, Y. (2013). Incremental graph pattern matching. *ACM Trans. Database Syst.*, 38(3):18. 63

Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. (1999a). Learning probabilistic relational models. *In IJCAI*, pages 1300–1309. ix, 5, 32, 36, 37, 39, 40, 46, 73

Friedman, N., Nachman, I., and Peffer, D. (1999b). Learning Bayesian network structure from massive datasets: The "sparse candidate" algorithm. In *UAI*, pages 206–215. Morgan Kaufmann. 23

Getoor, L. (2001). *Learning Statistical Models from Relational Data*. PhD thesis, Stanford. viii, 47, 74, 93

Getoor, L., Friedman, N., Koller, D., Pfeffer, A., , and Taskar, B. (2007). *Probabilistic relational models*, chapter Introduction to Statistical Relational Learning, pages 129ï¿½–174. Cambridge, MA: MIT Press. 39

Graves, M. Bergeman, E. and Lawrence, C. (2002). Graph database systems. *Engineering in Medicine and Biology Magazine, IEEE*, 14:737 – 745. 63

Han, J., Haihong, E., Le, G., and Du, J. (2011). Survey on NoSQL database. In *Pervasive Computing and Applications (ICPCA)*, pages 363–366. 54

Heckerman, D., Geiger, D., and Chickering, D. (1995). Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, pages 197–243. 21

Heckerman, D. and Meek, M. (2004). Probabilistic entity-relationship models, prms, and plate models. *ICML*, pages 55–60. 5, 32, 36, 37, 68

Huang, Y. and Luo, T. (2014). Nosql database: A scalable, availability, high performance storage for big data. In Zu, Q., Vargas-Vera, M., and Hu, B., editors, *Pervasive Computing and the Networked World*, volume 8351 of *Lecture Notes in Computer Science*, pages 172–183. Springer International Publishing. 53

Hunger, M. (2012). Neo4j: The world's leading graph database. 63

Hunger, M., Boyd, R., and Lyon, W. (2016). *The Defnitive Guide to Graph Databases for the RDBMS Developer*. neo4j.com. viii, 127, 129

Huynh, T. N. and Mooney, R. J. (2008). Discriminative structure and parameter learning for Markov Logic Networks. In *Proceedings of the 25th International Conference on Machine Learning (ICML)*, pages 1–8. 44

Jensen, F. and Nielsen, T. (2007). *Bayesian Networks and Decision Graphs*. Springer, New York, NY, 2nd edition. 28

Jouili, S. and Vansteenberghe, V. (2013). An empirical comparison of graph databases. In *SocialCom*, pages 708–715. IEEE Computer Society. 63

Kaelin, F. and Precup, D. (2010). A study of approximate inference in probabilistic relational models. *JMLR.org*, 13:315–330. 48

Kersting, K., Ahmadi, B., and Natarajan, S. (2009). Counting Belief Propagation. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 277–284, Arlington, Virginia, United States. AUAI Press. 46

Khachana, R. T., James, A. E., and Iqbal, R. (2011). Relaxation of acid properties in autra, the adaptive user-defined transaction relaxing approach. *Future Generation Comp. Syst.*, 27(1):58–66. 35

Khosravi, H., Schulte, O., Man, T., Xu, X., and Bina, B. (2010). Structure learning for Markov Logic Networks with many descriptive attributes. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 50–57. AAAI Press. 44

Kim, J. H. and Pearl, J. (1983). A computational model for combined causal and diagnostic reasoning in inference systems. In *Proceedings of the Eighth International Joint Conferences on Artificial Intelligence (IJCAI-1983)*, pages 190–193, Karlsruhe, Germany. 27

Kinderman, R. and Snell, S. (1980). *Markov random fields and their applications*. American mathematical society. 26

Kok, S. and Domingos, P. (2005). Learning structure of Markov Logic Networks. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 441–448, New York, NY, USA. ACM. ix, 40, 44, 45, 98

Kok, S. and Domingos, P. (2009). Learning Markov Logic Networks structure via hypergraph lifting. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 505–512, New York, NY, USA. ACM. 44

Kok, S. and Domingos, P. (2010). Learning Markov Logic Networks using structural motifs. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 551–558. 44

Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press. vii, 4, 12, 28

Koller, D. and Pfeffer, A. (1998). Probabilistic frame-based systems. *AAAI/IAAI*, pages 580–587. 5, 32

Koller, D. and Pfeffer, A. (2000). Semantics and inference for recursive probability models. In *Proc. 17th National Conference on Artificial Intelligence (AAAI-00)*, pages 538–544. 48

Lauritzen, S. and Spiegelhalter, D. (1988). Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems. *Journal of the Royal Statistical Society, Series B*, 50(2):157–224. 28

Leonid, A. and Toby, W. (2012). Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm. 61

Liu, L. and Özsu, M. T. (2009). ACID transaction. In *Encyclopedia of Database Systems*, pages 21–26. Springer US. 35

Liu, Y., Shah, N., and Koutra, D. (2015). An empirical comparison of the summarization power of graph clustering methods. *CoRR*, abs/1511.06820:1–8. 63

Lowd, D. and Davis, J. (2014). Improving Markov network structure learning using decision trees. *Journal of Machine Learning Research*, 15(1):501–532. 27

Lowd, D. and Domingos, P. (2007). Efficient weight learning for Markov Logic Networks. In *Proceedings of the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases*, PKDD 2007, pages 200–211, Berlin, Heidelberg. Springer-Verlag. 44

Maier, M., Marazopoulou, K., Arbour, D., and Jensen, D. (2013a). A sound and complete algorithm for learning causal models from relational data. *CoRR*, abs/1309.6843. 80, 97

Maier, M., Marazopoulou, K., and Jensen, D. (2013b). Reasoning about independence in probabilistic models of relational data. *CoRR*, abs/1302.4381. 39

Maier, M., Taylor, B., Huseyin, O., and Jensen, D. (2010). Learning causal models of relational domains. In *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence*, pages 531–538, Atlanta, GA. AAAI Press. 39

Margaritis, D. (2003). *Learning Bayesian Network Model Structure From Data*. PhD thesis. 19, 20

McCallum, A. (2003). Efficiently inducing features of conditional random fields. In *Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI03)*. 27

Mihalkova, L. and Mooney, R. (2007). Bottom-up learning of Markov logic network structure. In *ICML*, pages 625–632. 27, 44

Milch, B., Zettlemoyer, L., Kersting, K., Haimes, M., and Pack Kaelbling, L. (2008). Lifted Probabilistic Inference with Counting Formulas. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1062–1068. 46

Moore, A. and Lee, M. (1998). Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets. *Journal of Artificial Intelligence Research (JAIR)*, 8:67–91. 46

Muggleton, S. and Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19(0):629 – 679. 71

Naïm, P., Wuillemin, P.-H., Leray, P., Pourret, O., and Becker, A. (2011). *Réseaux bayésiens*. Editions Eyrolles. 18

Neville, J. and Jensen, D. (2005). Relational dependency networks. In Getoor, L. and Taskar, B., editors, *Introduction to Statistical Relational Learning*. 5, 32, 36

Oliver, S. and Zhensong, Q. (2015). Factorbase: Sql for learning a multi-relational graphical model. *CoRR*, abs/1508.02428:1–8. 47, 48, 62

Park, Y., Hallac, H., P. Boyd, S., and Leskovec, J. (2017). Learning the network structure of heterogeneous data via pairwise exponential markov random fields. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, pages 1302–1310. 27

Partner, J., Vukotic, A., and Watt, N. (2014). *Neo4j in Action*. Manning Publications Company. 6, 47, 52, 62, 63

Pearl, J. (1982). Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proceedings AAAI Natinal Conference on AI*, pages 133–136, Pittsburgh, PA. 27

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 4, 12, 15, 23

Pfeffer, A. and Koller, D. (2000). Semantics and inference for recursive probability models. In Kautz, H. A. and Porter, B. W., editors, *AAAI/IAAI*, pages 538–544. AAAI Press / The MIT Press. 38

Pietra, S., Della Pietra, V., and Lafferty, J. (1997). Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):380–393. 27

Poole, D. (2003). First-Order Probabilistic Inference. In *IJCAI*, pages 985–991. 46

Poon, H. and Domingos, P. (2006). Sound and efficient inference with probabilistic and deterministic dependencies. In *Proceedings of the 21st National Conference on Artificial Intelligence*, AAAI'06, pages 458–463. AAAI Press. 48

Raedt, L. and Kersting, K. (2008). Probabilistic inductive logic programming. In *Probabilistic Inductive Logic Programming*, pages 1–27. Springer. 32, 35

Raedt, L., Kimmig, A., and Toivonen, H. (2007). Problog: A probabilistic prolog and its application in link discovery. In Veloso, M. M., editor, *IJCAI*, pages 2462–2467. 110

Renyi, A. (2007). *Probability theory*. Dover Publications, Mineola, N.Y. 12

Reutter, J. (2013). *Graph patterns : structure, query answering and applications in schema mappings and formal language theory.* PhD thesis, University of Edinburgh, UK. British Library, EThOS. 58

Richardson, M. and Domingos, P. (2006). Markov Logic Networks. *Machine Learning*, 62(1-2):107–136. 5, 32, 40, 43, 44

Ritter, D. (2012). From network mining to large scale business networks. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 989–996, New York, NY, USA. ACM. 61

Royer, L., Reimann, M., Andreopoulos, B., and Schroeder, M. (2008). Unraveling protein networks with power graph analysis. *PLoS Computational Biology*, 4(7):1–17. 61

Sequeda, J., Arenas, M., and Miranker, D. (2012). On directly mapping relational databases to rdf and owl. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 649–658, New York, NY, USA. ACM. 63, 70

Shin, M. and Hiroki, M. (2014). Performance evaluations of graph database using cuda and openmp compatible libraries. *SIGARCH Computer Architecture News*, 42(4):75–80. 63

Singh, M. and Valtorta, M. (1993). An algorithm for the construction of Bayesian network structures from data. In *UAI*, pages 259–265. Morgan Kaufmann. 23

Singh, M. and Valtorta, M. (1995). Construction of bayesian network structures from data: A brief survey and an efficient algorithm. *International Journal of Approximate Reasoning*, 12(2):111 – 131. 23

Singla, P. and Domingos, P. (2008). Lifted first-order belief propagation. In *AAAI*, pages 1094–1099. AAAI Press. 46, 48

Smullyan, R. (1968). *First-Order Logic*. Springer-Verlag, Berlin. 32

Spirtes, P., Glymour, C., and Scheines, R. (1990). Causality from probability. *J. Tiles, G. McKee, and G. Dean (eds.): Evolving Knowledge in the Natural and Behavioral Sciences*, Pittman, London,:181ï¿½199. 20

Spirtes, P., Glymour, C., and Scheines, R. (2000). *Causation, Prediction, and Search*. MIT press, 2nd edition. 20, 39

Srebro, N. (2003). Maximum likelihood bounded tree-width Markov networks. *Artif. Intell.*, 143(1):123–138. 27

Sun, J. and Reddy, C. K. (2013). Big data analytics for healthcare. *KDD*, page 1525. 5, 52

Taskar, B., Abbeel, P., Wong, M.-F., and Koller, D. (2007). Relational markov networks. In Getoor, L. and Taskar, B., editors, *Introduction to Statistical Relational Learning*. MIT Press. 5, 32, 38

Tsamardinos, I., Brown, L. E., and Aliferis, C. F. (2006). The max-min hill-climbing bayesian network structure learning algorithm. *Mach. Learn.*, 65(1):31–78. 23, 40

Van den Broeck, G., Taghipour, N., Meert, W., Davis, J., and Raedt, L. (2011). Lifted probabilistic inference by first-order knowledge compilation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2178–2185. 46

Verma, T. and Pearl, J. (1990). Equivalence and synthesis of causal models. In Bonissone, P. P., Henrion, M., Kanal, L. N., and Lemmer, J. F., editors, *UAI*, pages 255–270. Elsevier. 18, 20

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., and Wilkins, D. (2010). A comparison of a graph database and a relational database: A data provenance perspective. *The 48th Annual Southeast Regional Conference*, pages 421–426. 47, 62, 63

Wainwright, M. J., Ravikumar, P., and Lafferty, J. D. (2006). High-dimensional graphical model selection using l1-regularized logistic regression. In *NIPS*, pages 1465–1472. MIT Press. 27

Webber, J. and Robinson, I. (2012). The top 5 use cases of graph databases. Technical report, Neo4j, the 1 Platform for Connected Data. 61

Wei, W., Erenrich, J., and Selman, B. (2004). Towards efficient sampling: Exploiting random walk strategies. In McGuinness, D. L. and Ferguson, G., editors, *AAAI*, pages 670–676. AAAI Press / The MIT Press. 48

Wood, P. T. (2012). Query languages for graph databases. *SIGMOD Rec.*, 41(1):50–60. 57

Xiang, Y. (2010). Modeling and reasoning with Bayesian networks. *Artif. Intell.*, 174(2):147–151. 28

Yang, S., Yan, X., Zong, B., and Khan, A. (2012). Towards effective partition management for large graphs. In Candan, K. S., Chen, Y., Snodgrass, R. T., Gravano, L., and Fuxman, A., editors, *SIGMOD Conference*, pages 517–528. ACM. 63

Yasin, A. (2013). *Incremental Bayesian network structure learning from data streams*. PhD thesis, Université de Nantes. 110

Yedidia, J., Freeman, W., and Weiss, Y. (2000). Generalized belief propagation. In *Advances in Neural Information Processing Systems (NIPS)*, volume 13, pages 689–695. MIT Press. 28

Zhang, N. and Poole, D. (1996). Exploiting causal independence in Bayesian network inference. *JAIR*, 5:301–328. 37, 48

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Titre : Apprentissage des modèles probabilistes relationnels à partir des bases de données graphe

Mots clés : Modèle Relationnel Probabiliste, Modèle Entité-Association Probabiliste, Apprentissage, Bases de données graphe, Données partiellement structurées, Réseau logique de Markov.

Historiquement, les Modèles Graphiques Probabilistes (PGMs) sont une solution d'apprentissage à partir des données incertaines et plates, appelées aussi données propositionnelles ou représentations attribut-valeur. Au début des années 2000, un grand intérêt a été adressé au traitement des données relationnelles présentant un grand nombre d'objets participant à des différentes relations. Les Modèles Probabilistes Relationnels (PRMs) présentent une extension des PGMs pour le contexte relationnel. Avec l'évolution rapide issue de l'internet, des innovations technologiques et des applications web, les données sont devenues de plus en plus variées et complexes. D'où l'essor du Big Data. Plusieurs types de bases de données ont été créés pour s'adapter aux nouvelles caractéristiques des données, dont les plus utilisés sont les bases de données graphe. Toutefois, tous les travaux d'apprentissage des PRMs sont consacrés à apprendre à partir des données bien structurées et stockées dans des bases de données relationnelles. Les bases de données graphe sont non structurées et n'obéissent pas à un schéma bien défini. Les arcs entre les nœuds peuvent avoir des différentes signatures. En effet, les relations qui ne correspondent pas à un modèle ER peuvent exister dans l'instance de base de données. Ces relations sont considérées comme des exceptions. Dans ce travail de thèse, nous nous intéressons à ce type de bases de données. Nous étudions aussi deux types de PRMs à savoir, Direct Acyclic Probabilistic Entity Relationship (DAPER) et chaines de markov logiques (MLNs). Nous proposons deux contributions majeures. Premièrement, Une approche d'apprentissage des DAPERs à partir des bases de données graphe partiellement structurées. Une deuxième approche consiste à exploiter la logique de premier ordre pour apprendre les DAPERs en utilisant les MLNs pour prendre en considération les exceptions qui peuvent parvenir lors de l'apprentissage. Nous menons une étude expérimentale permettant de comparer nos méthodes proposées avec les approches déjà existantes.

Title : Probabilistic relational models learning from graph databases

Keywords : Probabilistic Relational Model, Directed Acyclic Directed Entity Relationship Model, Machine Learning, Graph Database, Partially structured data, Markov Logic Network .

Historically, Probabilistic Graphical Models (PGMs) are a solution for learning from uncertain and flat data, also called propositional data or attribute-value representations. In the early 2000s, great interest was addressed to the processing of relational data which includes a large number of objects participating in different relations. Probabilistic Relational Models (PRMs) present an extension of PGMs to the relational context. With the rise of the internet, numerous technological innovations and web applications are driving the dramatic increase of various and complex data. Consequently, Big Data has emerged. Several types of data stores have been created to manage this new data, including the graph databases. Recently there has been an increasing interest in graph databases to model objects and interactions. However, all PRMs structure learning use well-structured data that are stored in relational databases. Graph databases are unstructured and schema-free data stores. Edges between nodes can have various signatures. Since, relationships that do not correspond to an ER model could be depicted in the database instance. These relationships are considered as exceptions. In this thesis, we are interested by this type of data stores. Also, we study two kinds of PRMs namely, Direct Acyclic Probabilistic Entity Relationship (DAPER) and Markov Logic Networks (MLNs). We propose two significant contributions. First, an approach to learn DAPERs from partially structured graph databases. A second approach consists to benefit from first-order logic to learn DAPERs using MLN framework to take into account the exceptions that are dropped during DAPER learning. We are conducting experimental studies to compare our proposed methods with existing approaches.