Statistical relational learning : Structure learning for Markov logic networks

Quang-Thang Dinh

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE

présentée par :

Quang-Thang DINH

soutenue le : **28 novembre 2011**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/ Spécialité : Informatique

Apprentissage Statistique Relationnel : Apprentissage de Structures de Réseaux de Markov Logiques

THÈSE dirigée par :

Christel Vrain	Professeur, Université d'Orléans
Matthieu Exbrayat	Maître de Conférences, Université d'Orléans

RAPPORTEURS :

Céline Rouveirol	Professeur, Université Paris 13, France
Lorenza Saitta	Professeur, Université Piémont Oriental, Italie

JURY :

Matthieu Exbrayat	Maître de Conférences, Université d'Orléans
Patrick Gallinari	Professeur, Université Pierre et Marie Curie
Philippe Leray	Professeur, Université de Nantes, Président du jury
Céline Rouveirol	Professeur, Université Paris 13
Lorenza Saitta	Professeur, Université Piémont Oriental
Christel Vrain	Professeur, Université d'Orléans

Acknowledgments

First and foremost, I would like to thank my advisor, Christel VRAIN, for her guidance, encouragement and support during the past three years. She gave me a very good chance to start my research career at LIFO laboratory, University of Orléans, France. I would like to thank equally my co-advisor, Matthieu EXBRAYAT, for having given me great support during all these years. They created a mentoring environment that support my growth as a researcher here and mastered a difficult task of giving me advise and direction, while allowing me freedom and time to persue and develop my ideas. Our weekly discussions have formed the main contributions of this dissertation. I deeply appreciate their great patience with my poor English in the first two years and with my French accent since the beginning of the thirst year.

I would like to thank the other members of the committee of this dissertation, who brought a remarkable support for my work: Céline Rouveirol and Lorenza Saitta for having accepted the heavy task of reviewing and Patrick Gallinari and Philippe Leray for their participation in the committee.

I also want to thank all the members of the Constraint and Learning (CA) team and all my colleagues at LIFO who have daily shared with me my work in Machine Learning. They also have gradually brought me into the French research community in particular and into the French social life in general.

This thesis would not have been possible without the tremendous support of my family: my parents who helped me understanding the meaning of life, my sweet wife who has always been of great support during these years of hard work and shared with me all my difficult moments, my four-year-old boy who brought me a lot of motivations and my younger sister who always believed in me. I dedicate this dissertation to them.

Finally, I would like to thank the Région Centre, France for having funded this work.

Quang-Thang DINH

The University of Orléans
November 2011

Contents

List of Figures

List of Tables

Introduction

L'apprentissage relationnel (Relational Learning) et l'apprentissage statistique (Statistical Learning) sont deux sous-problèmes traditionnels de l'apprentissage automatique (Machine Learning - ML). Les représentations logiques sont souvent utilisées dans l'apprentissage relationnel. Par exemple, la logique du premier ordre (First Order Logic - FOL) a été largement étudiée par le biais de la programmation logique inductive (Inductive Logic Programming - ILP). L'apprentissage relationnel vise à traiter la complexité des données relationnelles, qu'elles soient séquentielles, graphiques, multi-relationnelles, ou autres. L'apprentissage statistique permet quant à lui de prendre en compte la notion d'incertitude. Cependant, beaucoup de cas réels comportent des données à la fois complexes et incertaines. L'union de l'apprentissage relationnel et de l'apprentissage statistique est devenue un aspect important de l'Intelligence Artificielle.

L'apprentissage statistique relationnel (Statistical Relational Learning - SRL) [Getoor & Taskar 2007] consiste à combiner le pouvoir descriptif de l'apprentissage relationnel à la souplesse de l'apprentissage statistique. Diverses approches ont été proposées au cours des quinze dernières années, tels que PRISM (PRogramming In Statistical Modeling) [Sato & Kameya 1997], MACCENT (MAximum ENTropy modelling with Clausal Constraints) [Dehaspe 1997], les modèles relationnels probabilistes (Probabilistic Relational Models - PRMs) [Friedman *et al.* 1999], les programmes logiques bayésiens (Bayesian Logic Programs - BLPs) [Kersting & De Raedt 2007], les réseaux relationnels de dépendances (Relational Dependency Networks - RDNs) [Neville & Jensen 2004], les modèles relationnels de Markov (Relational Markov Models - RMMs) [Anderson *et al.* 2002] et les réseaux logiques de Markov (Markov Logic Networks - MLNs) [Richardson & Domingos 2006].

Les réseaux logiques de Markov (MLNs) [Richardson & Domingos 2006] constituent l'une des approches les plus récentes de l'apprentissage relationnel statistique et reposent sur la combinaison de la logique du premier ordre avec les réseaux de Markov. Un réseau de Markov (Markov Network - MN) [Pearl 1988] est un graphe, dont les nœuds représentent des variables aléatoires et dont les arêtes expriment les dépendances conditionnelles entre ces variables. Chaque clique du graphe correspond ainsi à un ensemble de variables conditionnellement dépendantes. On associe à chaque clique un poids. Un tel réseau permet ensuite d'inférer la valeur d'une ou plusieurs variables. Un réseau logique de Markov est constitué d'un ensemble de clauses logiques pondérées. La syntaxe des formules suit la syntaxe standard de la logique du premier ordre et les poids sont des nombres réels. Ces clauses sont constituées d'atomes, lesquels peuvent être vus comme des prototypes pour la construction de réseaux de Markov. En effet, si l'on dispose d'un ensemble de constantes, on peut produire, en instantiant les clauses, un ensemble d'atomes clos qui constitueront les nœuds d'un réseau de Markov. Les nœuds issus d'une même instanciation de clause seront liés, et les cliques ainsi produites seront affectées du poids de la clause dont elles dérivent. Les MLNs sont capables de représenter des lois de probabilités sur un nombre fini d'objets. De plus, beaucoup de représentations en apprentissage relationnel statistique peuvent être considérées comme des cas particuliers de logique du premier ordre associée à des modèles graphiques probabilistes et peuvent être transformés pour entrer dans le cadre formel des réseaux logiques de Markov. Pour ces raisons, nous avons retenu les réseaux

logiques de Markov comme modèle étudié dans le cadre de cette thèse.

Les deux tâches principales effectuées par un réseau logique de Markov sont l'inférence et l'apprentissage. Il existe deux types d'inférence : la recherche de l'état le plus probable d'un monde, étant donnée la valeur d'une partie des objets qui le composent (évidences) et le calcul de probabilités marginales ou conditionnelles arbitraires. L'apprentissage d'un réseau logique de Markov peut être décomposé en deux phases, consistant à apprendre respectivement la structure (i.e. les clauses en logique du premier ordre) et les paramètres (i.e. le poids des clauses) de ce réseau.

L'apprentissage automatique de la structure d'un MLN à partir d'un jeu de données constitue une tâche importante car il permet de découvrir une structure décrivant de nouvelles connaissances dans le domaine, sur laquelle on peut inférer afin de prédire des événements futurs. Cet apprentissage automatique devient incontournable quand les données sont trop grandes pour la lecture humaine et quand l'utilisateur manque de connaissance experte. Cependant, il reste un défi lié à la dimension de l'espace de recherche et au besoin, dans les systèmes actuels, d'apprendre les poids des formules afin d'évaluer celles-ci, processus qui exige beaucoup de temps de calcul.

Contributions de cette thèse

Ce mémoire de thèse porte sur le problème de l'apprentissage de la structure d'un réseau logique de Markov à partir d'un jeu de données relationnelles. Étant donné un ensemble de prédicats et les types de leurs arguments dans un domaine, et un jeu de données relationnelles contenant des atomes clos vrais ou faux, un ensemble de clauses pondérées va être découvert par un système d'apprentissage de la structure d'un réseau logique de Markov. La figure 1.1 montre un exemple des entrée et sortie d'un tel système.

Les contributions de cette thèse sont des méthodes pour l'apprentissage de la structure d'un réseau logique de Markov tant génératif que discriminant. Ces méthodes peuvent être divisées en deux classes : les méthodes reposant sur la notion de propositionnalisation et les méthodes reposant sur une notion introduite dans la thèse et appelée *Graphe des Prédicats*.

Apprentissage de la structure par propositionnalisation

La propositionnalisation est une technique populaire en programmation logique inductive. Il s'agit d'un processus permettant de produire des attributs utiles, habituellement sous la forme de tableaux attributs-valeurs, à partir des représentations relationnelles, et d'utiliser ensuite des algorithmes standards en logique propositionnelle pour apprendre des motifs dans ces tableaux. L'idée de base dans nos méthodes consiste à effectuer une propositionnalisation pour transformer les informations relationnelles liant les atomes clos du jeu de données en tableaux booléens dont chaque colonne correspond à un littéral. Ces tableaux booléens sont alors utilisés pour trouver des littéraux dépendants, à partir desquels les clauses candidates seront créées. La technique de propositionnalisation utilisée dans nos méthodes consiste en deux étapes : la création d'un ensemble de littéraux à partir du jeu de données, en partant d'un prédicat donné du domaine, puis le remplissage de ces tableaux booléens.

Nous avons d'abord développé l'algorithme HGSM (Heuristic Generative Structure for MLNs) [Dinh *et al.* 2010b] en implémentant une première technique de propositionnalisa-

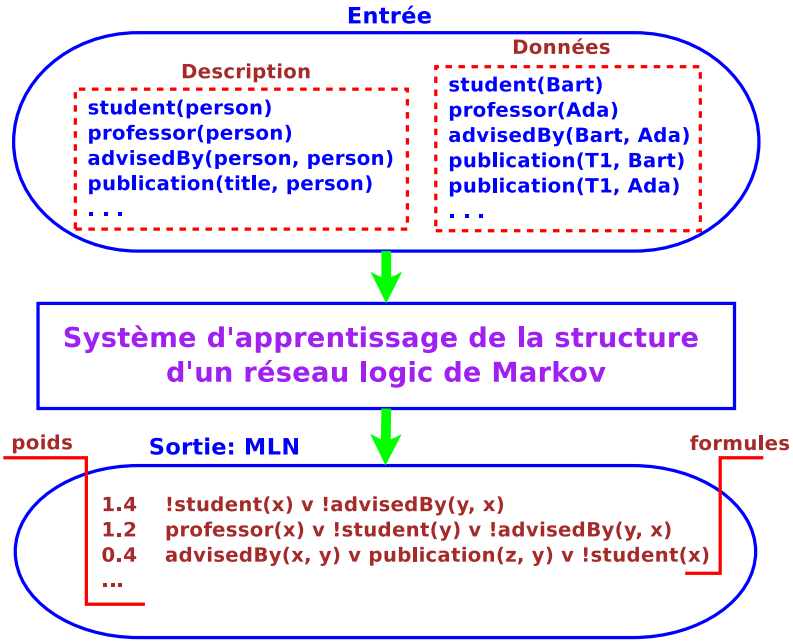


Figure 1: L'entrée et la sortie d'un système d'apprentissage de la structure d'un MLN

tion dans laquelle, pour chaque prédicat, le jeu de données est séparé en groupes distincts d'atomes connectés. Une méthode heuristique de variabilisation est alors appliquée sur ces groupes d'atomes connectés, du plus grand au plus petit, afin de construire un ensemble de littéraux avec seulement des variables. HGSM transpose ensuite l'information contenue dans le jeu de données aux tableaux booléens. Une technique de recherche des variables dépendantes dans un réseau de Markov (Grow-Shrink Markov Network) [Bromberg *et al.* 2006] est appliquée dans chaque tableau afin de trouver les littéraux (comportant seulement des variables) dépendants. Des clauses candidates sont créées à partir de cet ensemble de littéraux dépendants et sont enfin évaluées, l'une après l'autre, pour apprendre le réseau logique de Markov final.

Nous avons ensuite adapté l'approche utilisée dans HGSM à l'apprentissage discriminant de réseaux logiques de Markov. Cette nouvelle approche, intitulée HDSM (Heuristic Discriminative Structure learning for MLNs) [Dinh *et al.* 2010c], se focalise sur la capacité prédictive du réseau pour un prédicat de requête, au lieu de chercher une capacité de prédiction satisfaisante pour tous les prédicats du domaine. HDSM utilise une mesure discriminant pour choisir les clauses.

Nous avons également développé une approche appelée DMSP (Discriminative MLN Structure learning based on Propositionalization) [Dinh *et al.* 2010a] pour effectuer l'apprentissage discriminant de la structure d'un réseau logique de Markov, en mettant en œuvre une seconde technique de propositionnalisation pour la construction de l'ensemble de littéraux. Cette deuxième technique repose sur l'idée de variabiliser des groupes de chemins d'atomes connectés dans le jeu de données et permet de créer un ensemble de littéraux beaucoup plus compact, et ceci beaucoup plus rapidement. Nous montrons que cet ensemble de littéraux est le plus petit ensemble capable de décrire les relations liées

au prédicat de requête dans le jeu de données. DMSP utilise le test d'indépendance du χ^2 afin de générer plus de clauses candidates que HDSM.

Apprentissage de la structure à partir des Graphes des Prédicats

L'idée de base de cette autre famille de méthodes est de coder les informations du jeu de données dans un graphe dont on pourra extraire des clauses candidates. Nous introduisons la définition de Graphe des Prédicats (GoP), qui modélise les liens entre deux prédicats du domaine ainsi que la mesure de couverture classique en programmation logique inductive. A chaque lien entre deux prédicats est associé une formule. Un Graphe des Prédicats, dont chaque nœud correspond à un prédicat ou à sa négation et chaque arête à un lien possible entre deux prédicats, souligne les associations binaires entre les prédicats. Nous avons ensuite développé l'algorithme GSLP (Generative Structure Learning based on graph of Predicates) [Dinh *et al.* 2011a] pour l'apprentissage génératif de la structure d'un MLN. GSLP suit une stratégie descendant pour limiter la recherche des clauses candidates contenues dans le Graphe des Prédicats, ce qui est beaucoup plus rapide qu'une recherche exhaustive dans l'espace de clauses.

Lors des expérimentations menées pour évaluer GSLP, nous avons détecté plusieurs limites concernant la mesure pour choisir des clauses, la stratégie visant à réduire le nombre d'arêtes du Graphe des Prédicats et les méthodes pour construire un ensemble de clauses candidates et pour apprendre le MLN final. Ces problèmes ont été résolus dans la version modifiée de GSLP, intitulée M-GSLP (Modified-GSLP) [Dinh *et al.* 2011b].

Nous avons enfin développé le système DSLP pour la tâche d'apprentissage discriminant de la structure d'un MLN. Dans DSLP, nous avons défini une nouvelle sémantique du Graphe des Prédicats afin d'adapter celui-ci à la tâche d'apprentissage discriminant et d'accélérer le processus de recherche des clauses de Horn. Les résultats de l'expérimentation montrent que DSLP dépasse les systèmes de l'état de l'art pour l'apprentissage discriminant de la structure d'un MLN, sur les jeux de données classiques.

Organisation de la thèse

Ce mémoire est organisé comme suit :

- Le chapitre 2 rappelle tout d'abord des notions de base sur l'apprentissage statistique relationnel contenant les deux modèles graphiques probabilistes (les réseaux Bayésiens et les réseaux de Markov) et des connaissances de base de la logique du premier ordre. Il présente ensuite un bref aperçu de méthodes de Programmation Logique Inductive. Enfin, ce chapitre introduit les principaux modèles d'apprentissage statistique relationnel.

- Le chapitre 3 est consacré aux réseaux logiques de Markov. Il commence par la définition et la représentation d'un réseau logique de Markov, puis donne un aperçu des deux tâches principales des réseaux logiques de Markov: l'inférence et l'apprentissage.

- Le chapitre 4 présente nos méthodes reposant sur la propositionnalisation que nous avons proposé pour apprendre la structure d'un réseau logique de Markov.

- Le chapitre 5 présente nos méthodes reposant sur les Graphes des Prédicats pour apprendre la structure d'un réseau logique de Markov.
- Le chapitre 6 conclut cette thèse par un résumé de ses contributions et l'évocation de ses perspectives.

Les contributions du chapitre 4 ont été publiées dans [Dinh *et al.* 2010b, Dinh *et al.* 2010a, Dinh *et al.* 2010c], et celles du chapitre 5 dans [Dinh *et al.* 2011a, Dinh *et al.* 2011b].

Chapter 1

Introduction

Contents

Relational Learning and Statistical Learning are two traditional subfields of Machine Learning (ML). Relational learning is often based on logical representations. For instance, Machine Learning First-Order Logic (FOL) has been widely studied by the mean of Inductive Logic Programming (ILP). Relational Learning tends to handle complexity of relational data such as sequences, graphs, multi-relational data, etc. Statistical Learning is based on statistical representation and thus allows to handle uncertainty. However, many real-world applications require to deal with both uncertainty and complexity of data. Combining relational learning and statistical learning has become an important goal of Artificial Intelligence (AI).

Statistical Relational Learning (SRL) seeks to combine the power of both statistical learning and relational learning.A large number of statistical relational learning approaches have been proposed, including PRISM [Sato & Kameya 1997], MACCENT [Dehaspe 1997], Probabilistic Relational Models [Friedman *et al.* 1999], Relational Markov Models [Anderson *et al.* 2002], Relational Dependency Networks [Neville & Jensen 2004], Markov Logic Networks [Richardson & Domingos 2006], Bayesian Logic Programs [Kersting & De Raedt 2007], and others.

Markov Logic Networks are a kind of statistical relational model that generalizes both full FOL and Markov networks (MN). 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. A MLN can be viewed as a *template* to construct Markov networks of various sizes. Given a set of constants in a particular domain, a MLN can be grounded into a Markov network that can then be used to infer probabilities for a set of query literals given the truth values of a set of evidence literals. MLNs are able to represent probability distributions over a set of objects [Richardson & Domingos 2004, Richardson & Domingos 2006]. Moreover, many representations in SRL, that can be viewed as special cases of first-order logic and probabilistic graphical models, can be mapped into MLNs [Richardson & Domingos 2004]. For these reasons, we have chosen MLNs as the model on which we focused the research presented in this dissertation.

The two main tasks related to MLNs are inference and learning. There exists two basic types of inference: finding the most likely state of the world consistent with some evidence, and computing arbitrary marginal or conditional probabilities. Learning a MLN can be decomposed into structure learning and weight learning. Structure learning leads

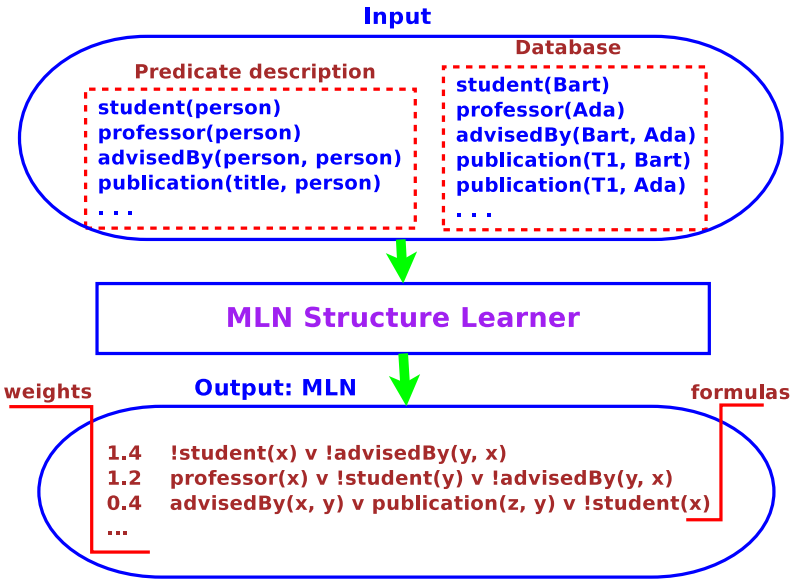


Figure 1.1: Input and output of a MLN structure learner

to finding a set of weighted first-order formulas from data. Parameter learning leads to estimating the weight associated with each formula of a given structure.

Learning MLN structure from data is an important task as it allows us to discover a structure describing novel knowledge inside the domain, from which one can perform inference in order to predict future events. It becomes more and more necessary when data is too large for human perusal and the user lacks expert knowledge about it. However, this remains challenging, due to the very large search space and to the need for current MLN learners to repeatedly learn the weights of formulas in order to evaluate these latters, a process which is computationally expensive.

1.1 Dissertation Contributions

This dissertation focuses on addressing the question of how to efficiently learn a MLN structure from relational databases. Given a description about predicate symbols and their types of arguments in a domain and a relational database containing true/false ground atoms of these predicates, a set of relevant weighted clauses are discovered by a MLN structure learner. Figure 1.1 shows an example of input and output of a MLN structure learner.

Contributions of this dissertation are approaches for both generative and discriminative MLN structure learning, which can be divided into two classes: approaches based on propositionalization and approaches based on Graphs of Predicates.

MLN structure learning based on propositionalization

Propositionalization is a popular technique in ILP. It is the process of generating a num-

ber of useful attributes, mostly in form of attribute-value tables, starting from relational representations and then using traditional propositional algorithms for learning and mining [Alphonse & Rouveirol 1999, Alphonse & Rouveirol 2000, De Raedt 2008]. The basic idea in our methods is to perform propositionalization in which information expressed by shared ground atoms (by their constants) in the dataset is transformed into boolean tables, each column of which corresponds to a variable literal. These boolean tables are then used to find dependent variable literals from which to create candidate clauses. The propositionalization technique used in our methods includes two steps: it first creates a set of variable literals from the dataset corresponding to a given predicate and then fills the boolean tables.

We first developed the Heuristic Generative Structure for MLNs (HGSM) algorithm [Dinh *et al.* 2010b], implementing a first propositionalization technique in which, corresponding to each predicate, the training dataset is separated into groups of connected ground atoms starting from a true ground atom of the current predicate. A heuristic variabilization method is applied on these groups of connected ground atoms, from the largest to the shortest, in order to build a set of variable literals. HGSM then transforms the information in the dataset into boolean tables, each of which corresponds to a variable literal of the target predicate, from which to apply the GSMN (Grow-Shrink Markov Network) algorithm [Bromberg *et al.* 2006] to find the set of variable literals that are dependent to the current variable literal. Candidate clauses are created from this set of dependent variable literals and are then considered in turn to learn the final MLN.

We then adapted this approach to form the discriminative MLN structure learning system HDSM (Heuristic Discriminative Structure learning for MLNs) [Dinh *et al.* 2010c]. Instead of searching for a MLN that predicts well the truth value of the atoms of any predicate, HDSM only learns for a single query predicate. HDSM also uses a discriminative measure to choose clauses instead of the generative one in HGSM in order to make the system more suitable for the task of discriminative learning.

Next, we developed the DMSP (Discriminative MLN Structure learning based on Propositionalization) algorithm [Dinh *et al.* 2010a] for the task of learning discriminatively a MLN structure, implementing a second propositionalization technique. The difference with the first technique lies in the way to build a set of variable literals. This second technique can create much faster a set of variable literals that is more compact than the first one, based on the idea to variabilize groups of similar paths of shared ground atoms in the database. By this way, the set of variable literals is found much faster and we also prove that it is the smallest set to describe relations related to the query predicate in the database. DMSP then uses the χ^2-test of dependence instead of Grow-Shrink in order to generate a little more candidate clauses than HDSM does.

MLN structure learning based on Graphs of Predicates

The basic idea behind this second approach is to encode information in the training database into a graph from which to search for candidate clauses. We propose the definition of a Graph of Predicates (GoP), which is inspired from links between predicates in a domain and the coverage measure in ILP. For each link between two predicates, we define a corresponding formula. A Graph of Predicates, each node of which corresponds to a predicate or its negation and each edge corresponds to a possible link between two

predicates, highlights the binary associations of predicates that share constants in terms of the number of true instantiations in the database (of the corresponding formulas). We then developed the Generative Structure Learning based on graph of Predicates (GSLP) algorithm [Dinh *et al.* 2011a] to learn generatively a MLN structure from a relational database. GSLP relies on a top-down strategy to narrow the search for candidate clauses within the Graph of Predicates, which is much faster than an exhaustive search in the space of clauses.

During the experiments to evaluate GSLP, we notice several limitations related to the measure used to choose clauses, the strategy to reduce the number of edges in the Graph of Predicates and the methods to build a set of candidate clauses and to learn the final MLN. These limitations have been overcome in the Modified-GSLP (M-GSLP) system [Dinh *et al.* 2011b].

We finally propose the DSLP (Discriminative Structure Learning based on Graph of Predicates) system for the task of discriminative MLN structure learning. In this system, we define a new semantic of Graph of Predicates in order to adapt it to the task of discriminative learning and accelerate the process of finding Horn clauses. Experiment results show that DSLP dominates the state-of-the-art discriminative MLN structure learners on the standard datasets.

1.2 Dissertation Organization

This dissertation is organized as follows:

- Chapter 2 first presents some backgrounds of Statistical Relational Learning, including two probabilistic graphical models (Bayesian networks and Markov networks) and several notions of first order logic. It then gives a brief overview of methods in Inductive Logic Programming. Finally this chapter gives an introduction of several models in statistical relational learning.

- Chapter 3 concentrates on Markov Logic Networks. It begins by the definition and representation of MLN, then brings an overview of the two main tasks for MLNs: inference and learning.

- Chapter 4 presents methods for MLN structure learning based on the propositionalization in ILP.

- Chapter 5 presents methods for MLN structure learning based on Graphs of Predicates.

- Chapter 6 concludes this dissertation with a summary of its contributions and directions for future work.

The contributions of Chapter 4 have been published in [Dinh *et al.* 2010b, Dinh *et al.* 2010a, Dinh *et al.* 2010c], and the ones of Chapter 5 in [Dinh *et al.* 2011a, Dinh *et al.* 2011b].

Chapter 2

Statistical Relational Learning

Résumé: Dans ce chapitre, nous présentons des connaissances de base en apprentissage statistique relationnel. Nous décrivons principalement deux modèles graphiques probabilistes et des notions de la logique du premier ordre. Un bref aperçu des méthodes en PLI (Programmation Logique Inductive) est donné. Enfin, nous décrivons brièvement plusieurs modèles de l'apprentissage statistique relationnel, en nous intéressant dans un premier temps aux modèles graphiques probabilistes orientés tels que les modèles probabilistes relationnels (Probabilistic Relational Models) [Friedman *et al.* 1999] et les programmes logiques bayésiens (Bayesian Logic Programs) [Kersting & De Raedt 2007] puis aux modèles fondés sur des graphes non-orientés tels que les réseaux relationnels de Markov (Relational Markov Networks) [Taskar *et al.* 2007] et les réseaux logiques de Markov (Markov Logic Networks) [Richardson & Domingos 2006]. Il faut noter que, ici, nous nous limitons aux modèles les plus populaires en apprentissage statistique relationnel. Il existe beaucoup d'autres modèles comme les réseaux relationnels de dépendance (Relational Dependency Networks) [Neville & Jensen 2004], les modèles relationnels de Markov (Relational Markov Models) [Anderson *et al.* 2002], MACCENT (Maximum Entropy Modelling with Clausal Constraints) [Dehaspe 1997], PRISM (Programming in Statistical Modelling) [Sato & Kameya 1997], etc. Nous recommandons au lecteur à la publication [Getoor & Taskar 2007] pour des lectures plus approfondies sur les différents modèles d'apprentissage statistique relationnel.

Contents

This chapter presents basic notions and several approaches of Statistical Relational Learning (SRL). It begins with a description of two popular probabilistic graphical models to handle uncertainty in Statistical Learning called Bayesian Networks (BNs) and Markov

Networks (MNs). Before depicting an introduction to Inductive Logic Programming (ILP) (or Relational Learning), we recall several definitions of First-Order Logic (FOL) as this later is commonly used to represent knowledge in ILP. Next, the principles as well as several approaches of SRL are presented. Finally, we conclude with a summary of this chapter.

2.1 Probabilistic Graphical Models

In this section, we present basic notions of Probabilistic Graphical Models (PGMs) [Pearl 1988, Bishop 2006, Jordan *et al.* 1999]. As pointed out in [Jordan *et al.* 1999], PGMs are an integration of probability theory and graph theory. They have become extremely popular tools to deal with uncertainty through the use of probability theory, and effective approaches to cope with complexity through the use of graph theory. PGMs are graphs in which nodes represent random variables and edges represent probabilistic relationships between these variables. The graph captures the way in which the joint distribution over the whole set of random variables can be decomposed into a product of factors each depending only on a subset of variables. The two most common types of probabilistic graphical models are Bayesian Networks (BNs) and Markov Networks (MNs). BNs are directed graphical models. MNs are undirected graphical models. In the following the key aspects of these two PGMs, needed to understand the SRL setting, are discussed. Further details of these two models can be found in [Bishop 2006, Jordan *et al.* 1999, Edwards 2000].

2.1.1 Bayesian Networks

A Bayesian Network (BN) is represented by a directed acyclic graph (DAG). The nodes of this DAG are the random variables in the domain and the edges correspond, intuitively, to the direct influence of one node on another. One way to view this graph is as a data structure that provides the skeleton for representing the joint distribution compactly in a factorized way.

Let $\mathcal{G}$ be a BN graph over the variables $X = X_1, \ldots, X_n$. Each random variable X_i in the network has an associated *conditional probability distribution (CPD)* or *local probabilistic model*. The CPD for X_i, given its parents in the graph (denoted by $Pa(X_i)$), is $P(X_i|Pa(X_i))$. It captures the conditional probability of the random variable, given its parents in the graph. CPDs can be described in a variety of ways in which the most common is the representation of a table which contains a row for each possible set of values for the parents of the node, describing the probability of different values for X_i. These are often referred to as table CPDs. We illustrate here by an example.

Example 1 *Suppose that there are two boolean events* A *and* B *which could cause a boolean event* C*. Also, suppose that the event* B *has a direct effect on the event* A*. Then the dependency structure among these three events can be modeled by a Bayesian network shown in Figure 2.1. Beside each event (variable or node in the Bayesian network) is the corresponding CPD table. At the root, we have the probabilities of* B *being respectively true (T) or false (F). For the other non-root nodes, the conditional probabilities of the corresponding variable given its parents are reported. These are the tables of conditional probabilities of* A *given* B *and conditional probabilities of* C *given* A and B.

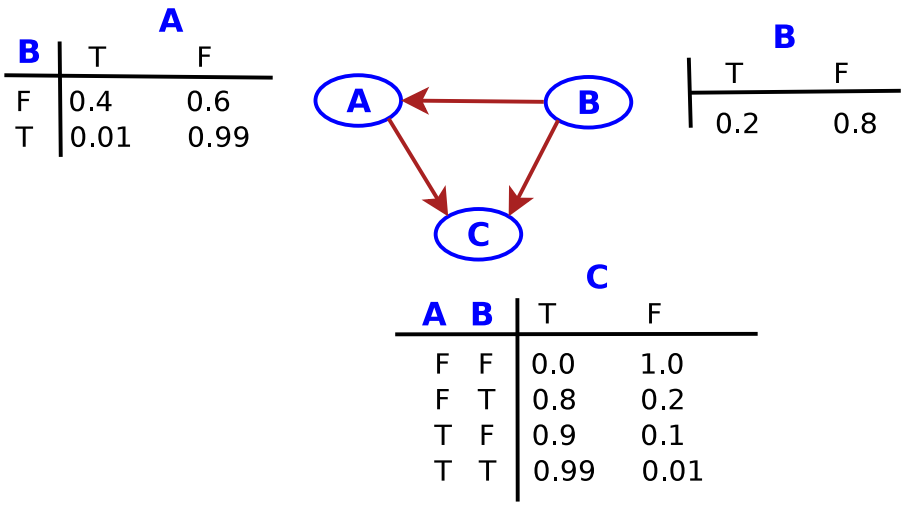


Figure 2.1: Example of a Bayesian network

Formally, a BN is a pair $(\mathcal{G}, \theta_{\mathcal{G}})$ where $\theta_{\mathcal{G}}$ is the set of CPDs associated with the nodes of $\mathcal{G}$. It defines a distribution $P_{\mathcal{B}}$ factorizing over $\mathcal{G}$:

$$P_{\mathcal{B}}(X) = \prod_{i=1}^{n} P(X_i | Pa(X_i)). \tag{2.1}$$

Example 2 *The BN in Figure 2.1 describes the following factorization:*

$$P(A,\ B,\ C) = P(B)P(A|B)P(C|A,B).$$

2.1.2 Markov Networks

A Markov Network (MN) (also known as Markov Random Field (MRF)) is a model for the joint distribution of a set of variables $\mathcal{X} = (X_1, X_2, \ldots, X_n)$ [Pearl 1988]. It is composed of an undirected graph G and a set of potential functions ϕ_k. The graph has a node for each variable, and the model has a potential function ϕ_k for each clique in the graph. A clique is defined as a subset of the nodes in the graph such that there exists a link between all pairs of nodes in the subset. A potential function is a non-negative real-valued function of the state of the corresponding clique. The joint distribution represented by a MN is given by:

$$P(X = x) = \frac{1}{Z} \prod_{k} \phi_k(x_{\{k\}}) \tag{2.2}$$

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

$$Z = \sum_{x \in \mathcal{X}} \prod_{k} \phi_k(x_{\{k\}}) \tag{2.3}$$

Example 3 *The graphical structure in Figure 2.2 over four variables* X = {A, B, C, D} *contains four maximal cliques* {A, B}, {B, C}, {C, D} *and* {D, A}. *Suppose that*

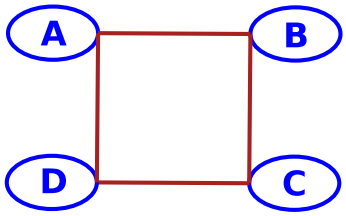


Figure 2.2: Example of a graph structure of a MLN

$\phi_1(A,B)$, $\phi_2(B,C)$ and $\phi_3(C,A)$ and $\phi_4(D,A)$ are four potential functions corresponding respectively to these maximal cliques and $x = (a,b,c,d) \in X$. We have:

$$P(A=a, B=b, C=c, D=d) = \frac{1}{Z}\phi_1(a,b)\phi_2(b,c)\phi_3(c,d)\phi_4(d,a)$$

where $Z = \sum_{(a_1,b_1,c_1,d_1)\in X} \phi_1(a_1,b_1)\phi_2(b_1,c_1)\phi_3(c_1,d_1)\phi_4(d_1,a_1)$. We can consider only functions of the maximal cliques, without loss of generality, as all other cliques, by definition, are subsets of the maximal cliques.

Markov networks are often conveniently represented as *log-linear models*, with each clique potential replaced by an exponentiated weighted sum of features of the state, $\phi_k(x_{\{k\}}) = \exp(w_k f_k(x))$, leading to:

$$P(X=x) = \frac{1}{Z}\exp\left(\sum_k w_k f_k(x)\right) \tag{2.4}$$

A feature may be any real-valued function of the state. This dissertation will focus on binary features $f_k(x) \in \{0,1\}$. In the most direct translation from the potential-function form (Equation 2.2), there is one feature corresponding to each possible state $x_{\{k\}}$ of each clique, with its weight being $\log \phi_k(x_{\{k\}})$. This representation is exponential in the size of the cliques.

In Markov networks, the notion of Markov blanket defined below is often used.

Definition 4 *In a Markov network, the Markov blanket (MB) of a node is simply a set of its neighbors in the graph.*

2.2 First-Order Logic

We present in this section basic definitions in first-order logic which are used throughout this dissertation.

A first-order knowledge base (KB) is a set of formulas in first-order logic (FOL) [Genesereth & Nilsson 1987]. Formulas are constructed using four types of symbols: *constants*, *variables*, *functions*, and *predicates*. Constant symbols represent objects in the domain of interest. Variable symbols range over the objects in the domain. Function symbols represent mappings from tuples of objects to objects. Predicate symbols represent relations among objects in the domain. A *term* is any expression representing an object in the domain. It can be a constant, a variable, or a function applied to a tuple of terms.

An *atomic formula* or *atom* is a predicate symbol applied to a tuple of terms. A *ground term* is a term containing no variables. A *ground atom* or ground predicate is an atomic formula all of whose arguments are ground terms. Formulas are inductively constructed from atomic formulas using logical connectives and quantifiers. A *positive literal* is an atomic formula; a *negative literal* is the negation of an atomic formula. A *ground literal* (resp. *variable literal*) contains no variable (resp. only variables).

Example 5 *Consider a domain of two predicate symbols* Smoke *and* Cancer *with a variable* X *ranges over two constants* {a, b}. Smoke(X) *and* Cancer(X) *are two positive literals while* Smoke(a), Smoke(b), Cancer(a) *and* Cancer(b) *are four ground atoms.*

A KB in *clausal form (conjunctive normal form (CNF))* is a conjunction of clauses, a *clause* being a disjunction of literals. A *ground clauses* contains all ground atoms. A *definite clause* or *Horn clause* is a clause with exactly one positive literal (the head, with the negative literals constituting the body). The *length* of a clause c, denoted by *len(c)*, is the number of literals in c. Two ground literals (resp. variable literals) are said to be *connected* if they share at least one ground constant (one variable). A clause (resp. a ground clause) is connected when there is an ordering of its literals $L_1, \ldots, L_p$, such that for each L_j, $j = 2, \ldots, p$, there exists a variable (a constant) occurring both in L_j and in L_i, with $i < j$. A *variabilization* of a ground clause e, denoted by *var(e)*, is obtained by assigning a new variable to each constant and replacing all its occurrences accordingly. An *interpretation* specifies which objects, functions and relations in the domain are represented by which symbols. A Herbrand base (HB) is the set of all ground atoms constructed with the predicate, constant, and available function symbols. A *possible world* or *Herbrand interpretation* is an assignment of truth value to all possible ground atoms, which is a subset of the Herbrand base. A *database* is a partial specification of a world, each atom in it being either true, false or (implicitly) unknown.

Example 6 *Consider a connected formula* $Smoke(X) \Longrightarrow Cancer(X)$ *in which* X *is a variable,* Smoke *and* Cancer *are two predicate symbols,* Smoke(X) *and* Cancer(X) *are two variable literals. With two constants* a *and* b*, by assigning variable* X *to one of these two constants in all possible manner, a possible world (Herbrand interpretation) is:*

{Smoke(a), ¬Smoke(a), Smoke(b), ¬Smoke(b), Cancer(a), ¬Cancer(a), Cancer(b), ¬Cancer(b)}.

A formula is satisfiable if and only if there exists at least one world in which it is true. The basic inference problem in first-order logic is to determine whether a knowledge base KB entails a formula F, i.e., if F is true in all worlds where KB is true (denoted by $KB \models F$).

Inference in first-order logic is only semi-decidable. Because of this, KBs are often constructed using a restricted subset of FOL where inference and learning is more tractable. Considering only Horn clauses is the most widely-used restriction. The field of inductive logic programming (ILP) [Lavrac & Dzeroski 1994] deals exactly with this problem.

2.3 Inductive Logic Programming

Inductive Logic Programming (ILP) is an area within Machine Learning that studies algorithms for learning sets of first-order clauses [Lavrac & Dzeroski 1994]. Usually, the task consists in discriminatively learning rules for a particular target predicate given background knowledge. This background knowledge may consist either of general clauses, or, more commonly, of a list of the true/false ground atoms of all predicates in the domain except the target predicate. The negative and positive examples are provided by the true and false ground atoms of the target predicate. The form that learned rules can take is frequently restricted to definite clauses [Richards & Mooney 1995]. Based on the setting of learning, methods of ILP can be classified into *learning from entailment*, *learning from interpretation* or *learning from proofs*. Based on the way clauses are searched, methods of ILP can be classified into top-down, bottom-up or hybrid approaches. Details of *learning from entailment* and *learning from interpretation* are presented in [De Raedt 2008]. In the following, we briefly describe methods of ILP according to the way clauses are searched.

- **Top-Down approaches:** The top-down ILP algorithms (e.g. [Quinlan 1990, De Raedt & Dehaspe 1997]) search the hypothesis space by considering, at each iteration, all valid refinements of a current set of candidate hypotheses. These candidates are then evaluated based on how well they cover positive examples and reject negative ones. A set of well-performing candidates is greedily selected, and the process continues with the next iteration. In addition to classification accuracy, several other heuristics for scoring, or evaluating, candidates have been used.

  For example, *FOIL* [Quinlan 1990], CLAUDIEN [De Raedt & Dehaspe 1997] and ALEPH [Srinivasan 2003] are three top-down ILP system in which ALEPH (A Learning Engine for Proposing Hypotheses) is a very popular one. ALEPH is based on an earlier ILP system called PROGOL [Muggleton 1995]; it preserves all abilities of PROGOL but incorporates additional search strategies and some extra options. It takes as input background knowledge in the form of either intensional or extensional facts, a list of modes declaring how literals can be chained together, and a designation of one literal as the "head" predicate to be learned, as well as lists of positive and negative examples of the head literal. ALEPH sequentially generates clauses covering some positive examples by picking a random example, considered as a seed. This example is used to create the bottom clause, i.e. the most specific clause (relative to a given background knowledge) that covers this example. This bottom clause is created by chaining through atoms until no more facts about the seed example can be added or a specified limit is reached. The bottom clause determines the possible search space for clauses. Aleph heuristically searches through the space of possible clauses until the "best" clause is found or time runs out. When enough clauses are learned to cover (almost) all the positive training examples, the set of learned clauses compose the learned theory.

  To sum up, top-down ILP techniques use data only to evaluate candidate hypotheses but not to suggest ways of forming new candidates.

- **Bottom-Up approaches:** Bottom-up ILP algorithms start with the most specific hypothesis and proceed to generalize it until no further generalizations are possible

without covering some negative examples [Lavrac & Dzeroski 1994]. For example, the initial hypothesis may be a set of rules where the body of each rule is simply a conjunction of the true ground atoms in the background knowledge, and the head is one of the positive examples. One way of generalizing this initial set of clauses is via the technique of least general generalization (LGG) [Plotkin 1970]. GOLEM [Muggleton & Feng 1992] is an example of a bottom-up ILP algorithm that uses LGG.

To sum up, bottom-up ILP algorithms take stronger guidance from examples, which is also used to propose clause candidates. This is in contrast with top-down algorithms, which use data only to evaluate candidate clauses.

- **Hybrid approaches:** Hybrid approaches [Zelle *et al.* 1994, Muggleton 1995] aim at exploiting the strength of both top-down and bottom-up techniques while avoiding their weaknesses. Because bottom-up techniques generalize from single examples, they are very sensitive to outliers and noise in the training data; however, because many bottom-up techniques employ LGGs, they are better-suited to handle functions. Similarly, top-down techniques can make a better use of general background knowledge to evaluate their hypotheses, but the greedy search through the hypothesis space can lead to long training times. Relational path-finding, developed by [Richards & Mooney 1992], is a hybrid approach to clausal discovery. It views the relational domain as a graph G in which constants are nodes and two constants are connected by an edge if they appear together in a true ground atom. Intuitively, relational path-finding forms definite clauses in which the head is a particular true ground atom, and the body consists of ground atoms that define a path in the relational graph G. The resulting clause is a disjunction of these ground atoms (in a corresponding path) with constants replaced by variables. This is the bottom-up part of the process. Hill-climbing search, which proceeds in a top-down fashion, is used to further improve the clauses by possibly adding unary predicates.

Often, *coverage* is used to measure how important a clause is in ILP, which measures the number of true examples in data covered by the clause.

Propositionalization is a popular technique in ILP which has been, as far as we know, first introduced in LINUS [Lavrac & Dzeroski 1994]. Propositionalization aims at converting a relational problem into an attribute-value one [Alphonse & Rouveirol 1999, Knobbe *et al.* 2001, De Raedt 2008, Kuželka & Železný 2008]. It is the process of generating a number of useful attributes or features starting from relational representations and then applying traditional attribute-value algorithms to solve the problem. Approaches for constructing automatically the new set of attributes or features can be divided into two trends [Krogel *et al.* 2003, Lesbegueries *et al.* 2009]: approaches based on logic and approaches inspired from databases.

- **Logic-based approaches:** The first trend in propositionalization approaches follows the ILP tradition which is logic-based. This trend includes the first representative LINUS [Lavrac & Dzeroski 1994] and its descendants, the latest being RSD [Lavrac *et al.* 2002] and HiFi [Kuželka & Železný 2008].

LINUS: In order to transform first-order into propositional form and vice versa, some restrictions on the hypothesis language and background knowledge are taken into

account in LINUS. The hypothesis language of LINUS is restricted to database clauses with non-recursive predicate definitions and non-recursive types where the body variables are a subset of the head variables. Background knowledge has the form of deductive database clauses, and may thus contain recursive clauses with typed variables. The background knowledge consists of two types of predicate definitions; utility functions and utility predicates. Utility functions are predicates which compute a unique output value for given input values. When occurring in an induced clause, the output arguments are bound to constants. Utility predicates are boolean functions with only input arguments.

Training examples in LINUS are ground facts which may contain structured, but non-recursive terms. The basic principle of the transformation from first-order into propositional form is that all body literals which may possibly appear in a hypothesis clause are determined, thereby taking into account variable types. Each of these body literals corresponds to a boolean attribute in the propositional formalism. For each given example, its argument values are substituted for the variables of the body literal. Since all variables in the body literals are required to occur also as head variables in a hypothesis clause, the substitution yields a ground fact. If it is a true fact, the corresponding propositional attribute value of the example is true, and false otherwise. The learning results generated by the propositional learning algorithms are then re-transformed in the obvious way.

RSD: (Relational Subgroup Discovery): This method has been originally designed as a system for relational subgroup discovery. To propositionalize data, RSD conducts the following three steps.

1. Identify all first-order literal conjunctions which form a feature, and at the same time comply to user-defined constraints (mode-language). Such features do not contain any constants and the task can be completed independently of the input data.
2. Extend the feature set by variable instantiations. Some features are copied several times with some variables substituted by constants detected by inspecting the input data. During this process, some irrelevant features are detected and eliminated.
3. Detect irrelevant features and generate propositionalized representations of the input data using the generated feature set.

To identify features, RSD accepts declarations very similar to those used by Aleph and Progol, including variable types, modes, setting a *recall* parameter etc. RSD produces the exhaustive set of features satisfying the mode and setting declarations. When an appropriate set of features has been generated, RSD can use it to produce a single relational table representing the original data. At the final step of feature construction, the system exports an attribute representation of the input data based on the truth values of respective features.

HiFi: This method is based on constructing first-order features with a hierarchical structure. Unlike RSD and LINUS, HiFi simultaneously constructs features and determines for which examples they are true (or computes their extensions). All

features produced by HiFi are the smallest in their semantic equivalence class [Kuželka & Železný 2008]. Specifically, the core algorithm accepts a learning example and a feature template. It then produces all features complying to the template and subsuming the example. These features are obtained by combinatorial composition of sub-features, which act as the primitive building blocks. The advantage of this assembly approach is that subsumption of the given example can already be checked for individual sub-features; if it is refuted for a given sub-feature, this sub-feature is not used in the subsequent feature composition.

- **Database-inspired approaches:** The second trend in propositionalization approaches are inspired from databases and appeared later beginning with two systems Polka [Knobbe *et al.* 2001] and RELAGGS (*REL*ational *AGG*regation*S* [Krogel & Wrobel 2001]. The method in RELAGGS is very similar to the one in Polka developed independently by a different research group. A difference between them concerns efficiency of the implementation.

  Those systems build attributes which summarize information stored in non-target tables by applying usual database aggregate functions such as count, min, max, etc. Besides the focus on aggregation functions, RELAGGS concentrates on the exploitation of relational database schema information, especially foreign key relationships as well as the use of optimization techniques such as indexes, which are usually applied for relational databases.

  Predicate logic and relational databases and their query languages. A relation (table) as a collection of tuples largely corresponds to ground facts of a logical predicate, and an attribute (column) of a relation to an argument of a predicate. A relational database can be represented as a graph with its relations as nodes and foreign key relationships as edges, conventionally by arrows pointing from the foreign key attribute in the dependent table to the corresponding primary key attribute in the independent table. The main idea in RELAGGS is to summarize non-target relations with respect to the target relation. In order to relate non-target relation tuples to the individuals, RELAGGS propagates the identifiers of the individuals to the non-target tables via foreign key relationships. This can be accomplished by join operations that use indexes on primary and foreign key attributes. These joins as views on the database are materialized in order to allow for fast aggregation. Aggregation functions are applied to single columns and to pairs of columns of single tables.

Most of these techniques, either based on logic or inspired from database, transform relational representations to an approximation in form of attribute-value ones. A disadvantage is that the approximation is incomplete and that some information might get lost in the propositionalization process. However, it allows to apply the whole set of traditional learning algorithms, as for instance decision trees, support vector machines and so on.

2.4 Statistical Relational Learning

Statistical Relational Learning (SRL) [Getoor & Taskar 2007] combines ideas from *Relational Learning* and *Probabilistic Graphical Models* to develop learning models and al-

gorithms capable of representing complex relationships among entities in uncertain domains. Popular SRL models include, among others, Probabilistic Relational Models [Friedman *et al.* 1999] and Bayesian Logic Programs [Kersting & De Raedt 2007], which can be seen as the extensions of Bayesian Networks and Relational Markov Networks [Taskar *et al.* 2007] and Markov Logic Networks [Richardson & Domingos 2006], which are the extensions of Markov Networks. In the following sections, we describe briefly these models.

2.4.1 Probabilistic Relational Models

Probabilistic relational models (PRMs) [Friedman *et al.* 1999] extend the standard attribute-based Bayesian network representation to incorporate a much richer relational structure. A PRM specifies a template for a probability distribution over a database. The template describes the relational schema for the domain, and the probabilistic dependencies between attributes in the domain. A PRM, together with a particular database of objects and relations, defines a probability distribution over the attributes of the objects and the relations.

A *relational schema* for a relational model describes a set of *classes*, $\mathcal{X} = X_1, \ldots, X_n$. Each class is associated with a set of *descriptive attributes* and a set of *reference slots*. There is a direct mapping between the notion of classes and the tables in a relational database: descriptive attributes correspond to standard table attributes, and reference slots correspond to foreign keys (key attributes of another table). The set of descriptive attributes of a class X is denoted by $\mathcal{A}(X)$. Attribute A of class X is denoted by $X.A$, and its domain of values is denoted by $V(X.A)$.

The set of reference slots of a class X is denoted by $\mathcal{R}(X)$. $X.\rho$ is used to denote the reference slot ρ of X. Each reference slot ρ is typed: the domain type of $Dom\,[\rho] = X$ and the range type $Range\,[\rho] = Y$, where Y is some class in $\mathcal{X}$. A slot ρ denotes a function from $Dom\,[\rho] = X$ to $Range\,[\rho] = Y$.

An instantiation $\mathcal{I}$ specifies the set of objects in each class X and the values for each attribute and each reference slots of each object. Figure 2.3 shows an instantiation of a simple movie schema. In this schema, the *Actor* class might have the descriptive attributes *Gender* with domain *male, female*. We might have a class *Role* with the reference slots *Actor* whose range is the class *Actor* and *Movie* whose range is the class *Movie*.

A *relational skeleton* σ_r, specifies the set of objects in all classes, as well as all the relationships that hold between them. In other words, it specifies the values for all of the reference slots.

A probabilistic relational model (PRM) Π for a relational schema (S) is defined as follows: for each class $X \in \mathcal{X}$ and each descriptive attribute $A \in \mathcal{A}(X)$, we have a set of *parents* $Pa(X.A)$ and a CPD that represents $P_\Pi(X.A|Pa(X.A))$.

Given a relational skeleton σ_r, a PRM Π specifies a distribution over a set of instantiations $\mathcal{I}$ consistent with σ_r.

$$P(\mathcal{I}|\sigma_r, \Pi) = \prod_{x \in \sigma_r(X)} \prod_{A \in \mathcal{A}(x)} P(x.A|Pa(x.A)) \tag{2.5}$$

where $\sigma_r(X)$ are the objects of each class as specified by the relational skeleton σ_r.

ACTOR	
name	gender
fred	male
ginger	female
bing	male

MOVIE	
name	genre
m1	drama
m2	comedy

ROLE			
role	movie	actor	role-type
r1	m1	fred	hero
r2	m1	ginger	heroine
r3	m1	bing	villain
r4	m2	bing	hero
r5	m2	ginger	love-interest

Figure 2.3: An instantiation of the relational schema for a simple movie domain

For this definition to specify coherent probability distribution over instantiations, it must be ensured that the probabilistic dependencies are acyclic, so that a random variable does not depend, directly or indirectly, on its own value. As pointed out in [Tasker *et al.* 2002], the need to avoid cycles in PRMs causes significant representational and computational difficulties.

Given a PRM and a set of objects, inference is performed by constructing the corresponding BN and applying standard inference techniques to it. Inference in PRMs is done by creating the complete ground network, which limits their scalability. PRMs require specifying a complete conditional model for each attribute of each class, which in large complex domains can be quite burdensome.

2.4.2 Bayesian Logic Programs

Bayesian Logic Programs (BLPs) [Kersting & De Raedt 2007] tightly integrate definite logic programs with Bayesian networks. A Bayesian logic program B consists of a (finite) set of Bayesian clauses. A Bayesian (definite) clause c is an expression of the form $A|A_1, \ldots, A_n$ where $n \geq 0$, $A, A_1, \ldots, A_n$ are Bayesian atoms and all Bayesian atoms are (implicitly) universally quantified. When $n = 0$, c is called a Bayesian fact and expressed as A. For each Bayesian clause c there is exactly one conditional probability distribution *CPD(c)*, and for each Bayesian predicate p of l-arity, denoted by p/l, there is exactly one combining rule *cr(p/l)*. In this context, a combining rule is a function that maps finite sets of conditional probability distributions $P(A|A_{i1}, \ldots, A_{in_1})|i = 1, \ldots, m$ onto one (combined) conditional probability distribution $P(A|B_1, \ldots, B_k)$ with $\{B_1, \ldots, B_k\} \subseteq \cup_{i=1}^{m}\{A_{i1}, \ldots, A_{in_1}\}$.

Example 7 *We reuse in Figure 2.4 an example of a BLP blood type presented in [Kersting & De Raedt 2007]. It is a genetic model of the inheritance of a single gene that determines a person* X*'s blood type* bt(X)*. Each person* X *has two copies of the chromosome containing this gene, one,* mc(Y)*, inherited from her mother* m(Y, X)*, and one,* pc(Z)*, inherited from her father* f(Z, X)*. This figure encodes blood type example for a particular family, in which* Dorothy*'s blood type is influenced by the genetic information of her parents* Ann *and* Brian*. The set of possible states of* bt(dorothy) *is* S(bt(dorothy)) =

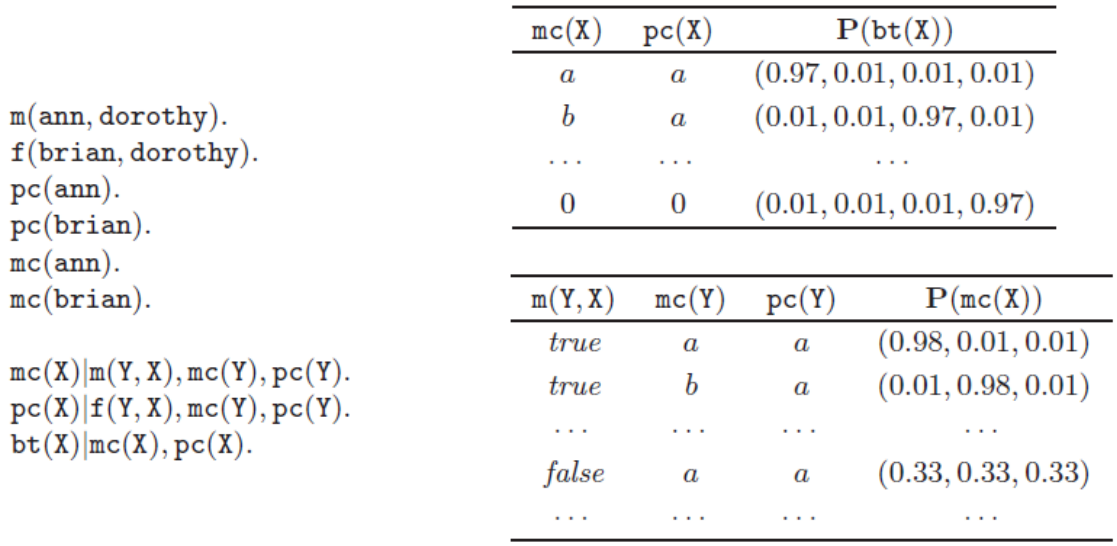


Figure 2.4: An example of a BLP

{a, b, ab, 0}*; the set of possible states of* pc(dorothy) *and* mc(dorothy) *are* S(pc(dorothy)) = S(mc(dorothy)) = {a, b, 0}. *The same holds for* ann *and* brian. *For each Bayesian predicate, the identity is the combining rule. The conditional probability distributions associated with the Bayesian clauses* bt(X) | mc(X), pc(X) *and* mc(X) | m(Y, X), mc(X), pc(Y) *are represented as tables. The other distributions are correspondingly defined. The Bayesian predicates m/2 and f/2 have as possible states* {true, false}.

Intuitively, each Bayesian logic program B represents a (possibly infinite) Bayesian network, where nodes are the atoms in the *least Herbrand* (LH) model of B, which is a set of all ground atoms $f \in HB(B)$ such that B logically entails f, i.e., $B \models f$. These declarative semantics can be formalized using the annotated dependency graph. The dependency graph *DG(B)* is the directed graph whose nodes correspond to the ground atoms in the least Herbrand model *LH(B)*. It encodes the direct influence relation over the random variables in LH(B): there is an edge from a node x to a node y if and only if there exists a clause $c \in B$ and a substitution θ, s.t. $y = head(c\theta)$, $x \in body(c_)$ and for all ground atoms $z \in c\theta : z \in LH(B)$.

Let B be a Bayesian logic program. If i) $LH(B) \neq \emptyset$, ii) *DG(B)* is acyclic, and iii) each node in *DG(B)* is influenced by a finite set of random variables, then B specifies a unique probability distribution P_B over *LH(B)*:

$$P(LH(B)) = \prod_{x \in LH(B)} P(x|Pa(x)), \tag{2.6}$$

where the parent relation Pa is according to the dependency graph.

2.4.3 Relational Markov Networks

Relational Markov networks (RMNs) [Taskar *et al.* 2007] compactly define a Markov network over a relational data set. It has been first proposed for an entire collection of related

entities in a collection of hypertext documents.

Consider hypertext as a simple example of a relational domain. A relational domain is defined by a schema, which describes entities, their attributes, and the relations between them. Formally, a *schema* specifies of a set of entity types $\mathcal{E} = \{E_1, \ldots, E_n\}$. Each type E is associated with three sets of attributes: content attributes $E.X$, label attributes $E.Y$ and reference attributes $E.R$. For simplicity, label and content attributes are restricted to take categorical values. Reference attributes include a special unique key attribute $E.K$ that identifies each entity. Other reference attributes $E.R$ refer to entities of a single type $E' = Range(E.R)$ and take values in $Domain(E'.K)$. An *instantiation* $\mathcal{I}$ of a schema $\mathcal{E}$ specifies the set of entities $\mathcal{I}(E)$ of each entity type $E \in \mathcal{E}$ and the values of all attributes for all of the entities. For example, an instantiation of the hypertext schema is a collection of web-pages, specifying their labels, the words they contain, and the links between them. Notations $\mathcal{I}.X$, $\mathcal{I}.Y$, and $\mathcal{I}.R$ are used respectively to denote the content, label, and reference attributes in the instantiation $\mathcal{I}$; $\mathcal{I}.x$, $\mathcal{I}.y$, and $\mathcal{I}.r$ are used respectively to denote the values of those attributes. $\mathcal{I}.r$ is called an *instantiation skeleton* or *instantiation graph*. It specifies the set of entities (nodes) and their reference attributes (edges). A hypertext instantiation graph specifies a set of web-pages and links between them, but not their words or labels.

A relational Markov network specifies a conditional distribution over all of the labels of all of the entities in an instantiation given the relational structure and the content attributes. Roughly speaking, it specifies the cliques and potentials between attributes of related entities at a template level, so a single model provides a coherent distribution for any collection of instances from the schema. To specify which cliques should be constructed in an instantiation, the notion of a relational clique template is used. A relational clique template specifies tuples of variables in the instantiation by using a relational query language.

A *relational clique template* $C = (\mathbf{F}, \mathbf{W}, \mathbf{S})$ consists of three components:

- $\mathbf{F} = \{F_i\}$ - a set of entity variables, where an entity variable F_i is of type $E(F_i)$.
- $\mathbf{W}(\mathbf{F}.\mathbf{R})$ - a Boolean formula using conditions of the form $F_i.R_j = F_k.R_l$.
- $\mathbf{F}.\mathbf{S} \subset \mathbf{F}.\mathbf{X} \cup \mathbf{F}.\mathbf{Y}$ - a selected subset of content and label attributes in $\mathbf{F}$.

A *relational Markov network* $M = (\mathbf{C}, \mathbf{\Phi})$ specifies a set of clique templates $\mathbf{C}$ and corresponding potentials $\mathbf{\Phi} = \{\phi_C\}_{C \in \mathbf{C}}$ to define a conditional distribution:

$$P(\mathcal{I}.\mathbf{y}|\mathcal{I}.\mathbf{x}, \mathcal{I}.\mathbf{r}) = \frac{1}{Z(\mathcal{I}.\mathbf{x}, \mathcal{I}.\mathbf{r})} \prod_{C \in \mathbf{C}} \prod_{c \in C(\mathcal{I})} \phi_C(\mathcal{I}.\mathbf{x}_c, \mathcal{I}.\mathbf{y}_c) \tag{2.7}$$

where $Z(\mathcal{I}.\mathbf{x}, \mathcal{I}.\mathbf{r})$ is the normalizing partition function:

$$Z(\mathcal{I}.\mathbf{x}, \mathcal{I}.\mathbf{r}) = \sum_{\mathcal{I}.\mathbf{y}'} \prod_{C \in \mathbf{C}} \prod_{c \in C(\mathcal{I})} \phi_C(\mathcal{I}.\mathbf{x}_c, \mathcal{I}.\mathbf{y}'_c).$$

2.4.4 Markov Logic Networks

Markov Logic Networks (MLNs) [Richardson & Domingos 2006] are a recently developed SRL model that generalizes both full first-order logic and Markov Networks. A Markov Network (MN) is a graph, where each vertex corresponds to a random variable. Each edge indicates that two variables are conditionally dependent. Each clique of this graph

is associated with a weight, which is a real number. A Markov Logic Network consists of a set of pairs (F_i,w_i), where F_i is a formula in First Order Logic, to which a weight w_i is associated. The higher w_i, the more likely a grounding of F_i to be true. Given a MLN and a set of constants $C = \{c_1, c_2, \ldots c_{|C|}\}$, a MN can be generated. The nodes (vertices) of this latter correspond to all ground predicates that can be generated by grounding any formula F_i with constants of C.

In [Saitta & Vrain 2008], the authors provide a running example to perform the comparison between Markov Logic Networks and Bayesian Logic Programs. In [Domingos & Richardson 2007] the authors show that MLNs can be a unifying framework for SRL because many representations used in SRL, such as PRM, BLP, etc, can be easily mapped into MLN. We present MLNs in more details in Chapter 3 of this dissertation.

2.5 Summary

In this chapter, some backgrounds of SRL, including two probabilistic graphical models and several notions of first order logic have been presented. Then a brief overview of methods in Inductive Logic Programming is given. Finally, we describe briefly several models of statistical relational learning, including both models based on the directed probabilistic graphical model such as Probabilistic Relational Models, Bayesian Logic Programs and models based on the undirected probabilistic graphical model such as Relational Markov Networks and Markov Logic Networks. It must be noted that, here we just present several popular models of SRL. There exists a lot of other models such as Relational Dependency Networks (RDNs) [Neville & Jensen 2004], Relational Markov Models (RMMs) [Anderson *et al.* 2002], Maximum Entropy Modelling with Clausal Constraints (MACCENT) [Dehaspe 1997], Programming in Statistical Modelling (PRISM) [Sato & Kameya 1997], etc. We refer the reader to [Getoor & Taskar 2007] for further reading about SRL models.

Among SRL models, many representations, that can be viewed as special cases of first-order logic and probabilistic graphical models, are able to be mapped into MLNs [Richardson & Domingos 2004]. For these reasons, we have chosen MLNs as the model on which we conduct our research in the framework of this dissertation. In the next chapter, we will detail MLNs and their principal tasks as well as solutions for each task.

Chapter 3

Markov Logic Networks and Alchemy

Résumé: Ce chapitre présente plus en détail les réseaux logiques de Markov. Les deux tâches principales sur les réseaux logiques de Markov sont l'apprentissage et l'inférence. L'inférence peut être réalisée de deux manières: effectuer une inférence MAP (maximum a posteriori) ou une inférence probabiliste conditionnelle. L'apprentissage peut être séparé en l'apprentissage des poids (paramètres du réseau logique de Markov) et l'apprentissage de la structure, auxquels les deux modèles (génératif et discriminant) peuvent être appliqués. Ce chapitre donne ensuite un aperçu de ces tâches. En particulier, pour l'apprentissage de la structure, il décrit les méthodes proposées à ce jour. Pour l'inférence et l'apprentissage des paramètres, il présente un aperçu des méthodes mises en œuvre dans Alchemy [Kok *et al.* 2009], un open source pour des réseaux logiques de Markov.

Contents

In this Chapter, we present Markov Logic Networks in details, related to their two main tasks: learning and inference.

3.1 Markov Logic Network

In first-order logic, a first-order knowledge base KB can be seen as a set of hard constraints on the set of possible worlds. A formula is false if there exists a world that violates it. Some systems allow a limited violation of constraints but this is more in the sense of some tolerance to noise. The basic idea in MLNs is to soften these constraints so that

when a world violates one formula in the KB, this formula becomes less probable, but not impossible. 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: the higher the weight, the greater the difference in log probability between a world that satisfies the formula and one that does not, other things being equal.

Definition 8 *[Richardson & Domingos 2006] A Markov logic network* L *is a set of pairs* (F_i, w_i)*, where* F_i *is a formula in first-order logic and* w_i *is a real number. Together with a finite set of constants* $C = \{c_1, c_2, \ldots, c_{|C|}\}$*, it defines a Markov network* $M_{L,C}$ *(Equations 2.2 and 2.4) as follows:*

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* w_i *associated with* F_i *in* L*. To be more precise, 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.*

The syntax of the formulas in an MLN is the standard syntax of first-order logic [Genesereth & Nilsson 1987]. Free (unquantified) variables are treated as universally quantified at the outermost level of the formula. In this dissertation, we focus on MLNs whose formulas are function-free clauses, and assume domain closure (i.e., the only objects in the domain are those representable using the constant symbols in C), thereby ensuring that the generated Markov networks are finite.

A MLN can be viewed as a *template* for constructing Markov networks. For different sets of constants, a MLN can construct Markov networks of different sizes, all of which sharing regularities in structure and parameters as defined by the MLN. From Definition 8 and Equation 2.4, the probability distribution over a possible world x specified by the Markov network $M_{L,C}$ is given by:

$$P(X = x|M_{L,C}) = \frac{1}{Z} \exp\left(\sum_{i \in F} \sum_{j \in G_i} w_i g_j(x)\right) \tag{3.1}$$

where Z is the normalization constant, F is the set of all first-order formulas in the MLN L, G_i and w_i are respectively the set of groundings and weight of the i-th first-order formula, and $g_j(x) = 1$ if the j-th ground formula is true and $g_j(x) = 0$ otherwise.

Example 9 *To illustrate this, we use an example from [Domingos et al. 2008a]. Let us consider a domain of three predicates* Friends(person, person)*,* Smoke(person) *and* Cancer(person)*. Here is a possible MLN composed of two formulas in this domain:*

$$1.5 \quad \forall x \; Smoke(x) \Longrightarrow Cancer(x)$$
$$1.1 \quad \forall x, y \; Friends(x, y) \Longrightarrow (Smoke(x) \Longleftrightarrow Smoke(x))$$

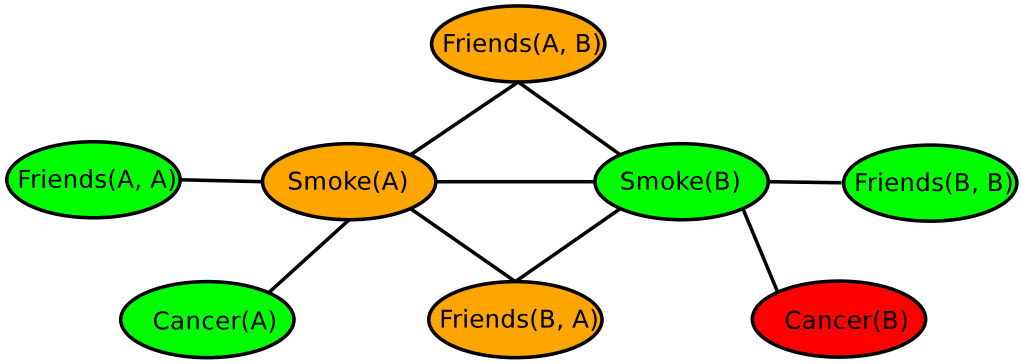


Figure 3.1: Example of a ground MN

Let us consider also a set of two constants of type person*:* Anna *and* Bob *(respectively* A *and* B *for short). By replacing variables* x *and* y *by one of these two constants in all possible ways, we have* Smoke(A)*,* Smoke(B)*,* Cancer(A)*,* Cancer(B)*,* Friends(A, A)*,* Friends(A, B)*,* Friends(B, A)*,* Friends(B, B) *as nodes of the Markov network. By replacing variable* x *in the first formula by the constant* A*,* $Smoke(A) \Longrightarrow Cancer(A)$ *is an instantiation (grounding). There exists an edge between these two nodes,* Smoke(A) *and* Cancer(A)*, and they form a clique in the instantiated MN. The corresponding feature is:*

$$f_1(Smoke(A), Cancer(A)) = \begin{cases} 1 & if \neg Smoke(A) \vee Cancer(A) \\ 0 & otherwise \end{cases}$$

Similarly, considering all the instantiations of the two formulas, we obtain the corresponding ground MN shown in Figure 3.1.

The two main tasks concerning MLNs are learning and inference. Inference is conducted either to find the most probable state of the world y given some evidence x, where x is a set of literals, or to calculate the probability of query q given evidence x based on equation 3.1. Learning can be divided into structure learning and parameter (weight) learning; both generative and discriminative models can be applied to them. Structure learning consists in 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.

3.1.1 Weight Learning

Given a set of formulas and a relational database, weight 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. In the following, we present the mathematical formulas of these measure as well as some methods for solving this problem.

3.1.1.1 Generative Weight Learning

Generative approaches try to induce a global organization of the world and thus, in the case of a statistical approach, optimize the joint probability distribution of all the variables.

Concerning MLNs, given a database, which is a vector of n possible ground atoms in a domain or $x = (x_1, \ldots, x_l, \ldots, x_n)$, where x_l is the truth value of the l-th ground atom ($x_l = 1$ if the atom is true, and $x_l = 0$ otherwise), weights can be generatively learned by maximizing the *likelihood* of a relational database (Equation 3.1):

$$P(X = x|M_{L,C}) = \frac{1}{Z} \exp\left(\sum_{i \in F} \sum_{j \in G_i} w_i g_j(x)\right) = \frac{1}{Z} \exp\left(\sum_{i \in F} w_i n_i(x)\right)$$

where $n_i(x)$ is the number of true instantiations of F_i in x. It must be noted that database is a set of entirely ground atoms in the domain, which is always difficult to collect. In fact, we often have only a set of gathered incomplete training examples. A closed-world assumption is therefore used so that all ground atoms not expressed in the set of training examples are considered to be false in database[1] in order to have such a database. The derivative of the log-likelihood with respect to its weight is:

$$\frac{\delta}{\delta w_i} \log P_w(X = x) = n_i(x) - \sum_{x'} P_w(X = x') n_i(x') \tag{3.2}$$

where the sum is over all possible databases x', and $P_w(X = x')$ is $P(X = x')$ computed using the current weight vector $w = (w_1, \ldots, w_i, \ldots)$. In other words, the i-th 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.

In the worst case, counting the number of true groundings ($n_i(x)$) of a formula in a database is intractable [Richardson & Domingos 2006, Domingos & Lowd 2009]. Computing the expected number of true groundings is also intractable, thus requiring inference over the model. The pseudo-log-likelihood of data is widely used instead of the likelihood [Besag 1975]. The pseudo-log-likelihood (PLL) of x given weights w is:

$$\log P_w^*(X = x) = \sum_{l=1}^{n} \log P_w(X_l = x_l | MB_x(X_l)) \tag{3.3}$$

where $MB_x(X_l)$ is the state of the Markov blanket (MB) of the ground atom X_l. The gradient of the pseudo-log-likelihood is:

$$\begin{aligned} \frac{\delta}{\delta w_i} \log P_w^*(X = x) = \sum_{l=1}^{n} [n_i(x) &- P_w(X_l = 0 | MB_x(X_l)) n_i(x_{[X_l=0]}) \\ &- P_w(X_l = 1 | MB_x(X_l)) n_i(x_{[X_l=1]})] \end{aligned} \tag{3.4}$$

where $n_i(x_{[X_l=0]})$ is the number of true groundings of the i-th formula when we forced $X_l = 0$ and leaved the remaining data unchanged, and similarly for $n_i(x_{[X_l=1]})$. Computing this expression does not require inference over the model, and is therefore much faster.

Both likelihood and pseudo-likelihood are convex functions therefore MLN weights might be learned using methods based on convex optimization such as iterative scaling [Della Pietra *et al.* 1997] or a quasi-Newton optimization method like the *L-BFGS* algorithm [Liu *et al.* 1989]. In fact, it has been shown that L-BFGS combined with the pseudo-likelihood yields efficient learning of MLN weights even in domain with millions of ground

[1] This assumption can be removed by using the expectation maximization algorithm to learn from the resulting incomplete data [Dempster *et al.* 1977].

atoms [Richardson & Domingos 2006]. However, the pseudo-likelihood parameters may lead to poor results when long chains of reference are required [Domingos & Lowd 2009].

3.1.1.2 Discriminative Weight Learning

In discriminative learning, a weight vector w is discriminatively learned by maximizing the conditional log-likelihood (CLL) or the max-margin of a set of query atoms Y given a set of evidence atoms X. We first present the CLL and its optimization methods then the ones related to the max-margin.

The *CLL* of Y given X is defined by:

$$P(Y = y|X = x) = \frac{1}{Z_x} exp\left(\sum_i w_i n_i(x, y)\right) \tag{3.5}$$

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 i-th formula in data.

The negative CLL is a convex function in case the truth values of all predicates are known in training data. It may no longer be convex when there is missing data [Poon & Domingos 2008, Domingos & Lowd 2009]. All methods that we present in this section are based on convex optimization hence complete data is required by using the closed-world assumption. Concerning missing data, we recommend the reader to [Poon & Domingos 2008] for further reading.

[Singla & Domingos 2005] proposed methods for both two classes of optimization algorithms; first-order and second-order. First-order methods pick a search direction based on the gradient function. Second-order methods derive a search direction by approximating the function as a quadratic surface. While this approximation may not hold globally, it can be good enough in local regions to greatly speed up convergence. Since most of the optimization literature assumes that the objective function is to be minimized, they minimize the negative conditional log likelihood. In an MLN, the derivative of the negative CLL with respect to a weight is the difference of the expected number of true groundings of the corresponding clause and their actual number according to data:

$$\begin{aligned} \frac{\delta}{\delta w_i}(-\log P_w(Y = y|X = x)) &= -n_i(x, y) + \sum_{y'} P_w(Y = y'|X = x) n_i(x, y') \\ &= E_{w,y}[n_i(x, y)] - n_i(x, y) \end{aligned} \tag{3.6}$$

where y is the state of the non-evidence atoms in data, x is the state of the evidence, and the expectation $E_{w,y}$ is over the non-evidence atoms Y.

Computing the expected counts $E_{w,y}[n_i(x, y)]$ is intractable. However, these can be approximated by the counts $n_i(x, y^*_w)$ in the MAP state $y^*_w(x)$, which is the most probable state of y given x (the MAP inference is presented in detail in Subsection 3.1.3.1). Thus, computing the gradient needs only MAP inference to find $y^*_w(x)$ which is much faster than full conditional inference for computing $E_{w,y}[n_i(x, y)]$. This approach was successfully used in [Collins 2002] for a special case of MNs where the query nodes form a linear chain. In this case the MAP state can be found using the *Viterbi* algorithm [Rabiner 1990] and the voted perceptron algorithm in [Collins 2002] follows this approach. To generalize this method to arbitrary MLNs, it is necessary to replace the Viterbi algorithm with a general-purpose

algorithm for MAP inference in MLNs. From Equation 3.5, we can see that since $y_w^*(x)$ is the state that maximizes the sum of the weights of the satisfied ground clauses, it can be found using a MAX-SAT solver. The authors in [Singla & Domingos 2005] generalized the voted-perceptron algorithm to arbitrary MLNs by replacing the Viterbi algorithm with the MaxWalkSAT solver [Kautz *et al.* 1996]. Given an MLN and set of evidence atoms, the KB to be passed to MaxWalkSAT is formed by constructing all groundings of clauses in the MLN involving query atoms, replacing the evidence atoms in those groundings by their truth values, and simplifying.

However, unlike the Viterbi algorithm, MaxWalkSAT is not guaranteed to reach the global MAP state. This can potentially lead to errors in the weight estimates produced. The quality of the estimates can be improved by running a Gibbs sampler starting at the state returned by MaxWalkSAT, and averaging counts over the samples. If the $P_w(x|y)$ distribution has more than one mode, doing multiple runs of MaxWalkSAT followed by Gibbs sampling can be helpful. This approach is followed in the algorithm in [Singla & Domingos 2005] which is essentially gradient descent.

In [Lowd & Domingos 2007] was proposed another approach for discriminative weight learning of MLNs. In this work the conjugated gradient [Shewchuk 1994] is used. Gradient descent can be sped up by performing a line search to find the optimum along the chosen descent direction instead of taking a small step of constant size at each iteration. This can be inefficient on ill-conditioned problems, since line searches along successive directions tend to partly undo the effect of each other: each line search makes the gradient along its direction zero, but the next line search will generally make it non-zero again. This can be solved by imposing at each step the condition that the gradient along previous directions remain zero. The directions chosen in this way are called conjugate, and the method conjugated gradient. In [Lowd & Domingos 2007] the authors used the Polak-Ribiere method (see in [Nocedal & Wright 1999] for details) for choosing conjugate gradients since it has generally been found to be the best-performing one.

Conjugated gradient methods are among the most efficient ones, on a par with quasi-Newton ones. Unfortunately, applying them to MLNs is difficult because line searches require computing the objective function, and therefore the partition function Z, which is intractable [Lowd & Domingos 2007]. Fortunately, the Hessian can be used instead of a line search to choose a step size. This method is known as scaled conjugate gradient (SCG), and was proposed in [Moller 1993] for training neural networks. In [Lowd & Domingos 2007], a step size was chosen by using the Hessian similar to a diagonal Newton method. Conjugate gradient methods are often more effective with a pre-conditioner, a linear transformation that attempts to reduce the condition number of the problem [Sha & Pereira 2003]. Good pre-conditioners approximate the inverse Hessian. In [Lowd & Domingos 2007], the authors used the inverse diagonal Hessian as pre-conditioner and called the SCG algorithm *Preconditioned SCG (PSCG)*. PSCG was shown to outperform the voted perceptron algorithm of [Singla & Domingos 2005] on two real-world domains both for CLL and AUC measures. For the same learning time, PSCG learned much more accurate models.

Instead of finding a weight vector w that optimizes the conditional log-likelihood $P(y|x)$ of the query atoms y given the evidence x, [Huynh & Mooney 2008] propose an alternative approach to learn a weight vector w that maximizes the ratio:

$$\frac{p(y|x,w)}{p(\hat{y}|x,w)}$$

between the probability of the correct truth assignment y and the closest competing incorrect truth assignment $\hat{y} = \arg\max_{\bar{y} \in Y \setminus y} P(\bar{y}|x)$. Applying equation 3.5 and taking the log, this problem is translated to maximizing the margin:

$$\begin{aligned} \gamma(x, y; w) &= w^T n(x, y) - w^T n(x, \hat{(y)}) \\ &= w^T n(x, y) - \max_{\bar{y} \in Y \setminus y} w^T n(x, \bar{y}) \end{aligned} \tag{3.7}$$

To train the proposed model, they design a new approximation algorithm for loss-augmented inference in MLNs based on Linear Programming (LP). Concerning this method, we recommend the paper of [Huynh & Mooney 2008] for further reading.

3.1.2 Structure Learning

The structure of a MLN is a set of formulas or clauses to which are attached weights. In principle, this structure can be learned or revised using any inductive logic programming (ILP) technique [Richardson & Domingos 2006]. However, the results obtained by this method were not "good" enough due to the lack of ability to deal with uncertainty in ILP [Richardson & Domingos 2006]. Therefore, finding new methods for MLN structure learning has been necessary. In the following, we present methods for MLN structure learning up to date in both generative and discriminative models.

3.1.2.1 Generative Structure Learning

The first generative algorithm is presented in [Richardson & Domingos 2006], where the authors used the CLAUDIEN system [De Raedt & Dehaspe 1997] in a first step to learn the clauses of MLNs and then learned the weights in a second step with a fixed structure. CLAUDIEN, unlike most other ILP systems which learn only *Horn clauses*, is able to learn arbitrary first-order clauses, making it well suited to MLNs. The PLL measure function is used to evaluate the quality of the learned MLN. They are also able to direct CLAUDIEN to search for refinements of the MLN structure by constructing a particular language bias. However, the results obtained by this method were not better than learning the weights for a hand-coded KB. This is due to the fact that CLAUDIEN searches clauses using typical ILP coverage evaluation measures [Richardson & Domingos 2006] which are not effective in the case of MLN, which represents a probability distribution. A method, which can integrate both searching clauses and suitable evaluation measures for MLNs, is hence necessary.

The PLL measure used in [Richardson & Domingos 2006] is potentially more adopted than the ones coming from ILP, but the authors in [Kok & Domingos 2005] shown that the log-likelihood measure gives undue weight to the largest arity predicates, resulting in poor modeling of the reminder. They defined, instead, the weighted pseudo-log-likelihood (WPLL):

$$\log P_w^*(X = x) = \sum_{r \in R} c_r \sum_{k=1}^{g_r} \log P_w(X_{r,k} = x_{r,k} | MB_x(X_{r,k})) \tag{3.8}$$

where R is the set of predicates, g_r is the number of groundings of first-order predicate r, $x_{r,k}$ is the truth value (0 or 1) of the k-th grounding of r. The choice of predicate

weights c_r depends on the user's goals. In this dissertation we set $c_r = 1/g_r$, as in [Kok & Domingos 2005], which has the effect of balancing the impact of clauses whatever their length.

Using this WPLL measure, several methods have been proposed for generative MLN structure learning. Those are MSL [Kok & Domingos 2005], BUSL [Mihalkova & Mooney 2007], ILS [Biba *et al.* 2008b], LHL [Kok & Domingos 2009], LSM [Kok & Domingos 2010] and MBN [Khosravi *et al.* 2010]. Note that structure learning is a difficult problem due to the exponential search space, hence there exist just a few methods up to date, and these methods are designed directly for MLNs without using any system in ILP. All these methods compose of two components: searching clauses and selecting clauses. We next describe them focusing on techniques using in these two components.

- **MSL** (Markov Logic Structure Learner) [Kok & Domingos 2005] involves the L-BFGS algorithm for learning weights, uses the WPLL measure for choosing clauses and searches clauses in the space of ground atoms in dataset using two strategies: *beam search* and *shortest first search*. New clauses are constructed by applying operations on a given clause to add, remove or replace literals from it. The first approach adds clauses to the MLN one at a time, using beam search to find the best clause to add: starting with the unit clauses it applies addition and deletion operators to each clause, keeps the b best ones, applies the operators to those, and repeats until no new clause improves the WPLL measure. The chosen clause is the one with the highest WPLL found in any iteration of the search. The second approach adds k clauses at a time to the MLN. In contrast to beam search, which adds the best clause of any length found, this approach adds all "good" clauses of length l before considering any clause of length $l + 1$.

- **BUSL** (Bottom-up Structure Learner) [Mihalkova & Mooney 2007] is a bottom-up method that use data to guide the search of clauses. In more details, it uses a propositional Markov Network learning method to construct structure networks that guide the construction of candidate clauses. The structure of BUSL is composed of three main phases: Propositionalization, Building clauses and Putting clauses into the MLN.

  In the propositionalization phase, BUSL creates a boolean table *MP* for each predicate P in the domain. The boolean table contains a column corresponding to a *TNode* and a row corresponding to a ground atom of P. A TNode is a data structure that contains conjunctions of one or more literals of variables. Intuitively, TNodes are constructed by looking for groups of constant sharing true ground atoms in form of relational path-finding [Richards & Mooney 1992] and variablizing them. Thus, TNodes could also be viewed as portions of clauses that have true groundings in data. Each entry *MP[r][c]* is a boolean value that indicates whether data contains a true grounding of the TNode corresponding to column c with at least one of the constants of the ground atom corresponding to row r.

  When building clauses, BUSL applies the Grow Shrink Markov Network (GSMN) algorithm [Bromberg *et al.* 2006] on *MP* to find every clique of the network, from which it builds candidate clauses. Candidates are evaluated using the WPLL measure

Candidate clauses are then considered in turn to put into the MLN in order of decreasing WPLL. Candidates that do not increase the overall WPLL of the learned structure are discarded. Weights are learned by the mean of the the L-BFGS algorithm.

- **ILS** (Iterated Local Search) [Biba *et al.* 2008b] is based on the iterated local search (ILS) [Hoos & Stutzle 2004] meta-heuristic that explores the space of structures through a biased sampling of the set of local optima. An intuitive key idea in ILS is the use of two types of search steps alternatively (one to reach local optima as efficiently as possible, and the other to effectively escape local optima) to perform a walk in the space of local optima w.r.t. the given evaluation function. The algorithm focuses the search not on the full space of solutions but on a smaller subspace defined by the solutions that are locally optimal according to the optimization engine. In more details, it starts by randomly choosing a unit clause CL_C in the search space. Then it performs a greedy local search to efficiently reach a local optimum CL_S. At this point, a perturbation method is applied leading to the neighbor CL'_C of CL_S and then a greedy local search is applied to CL'_C to reach another local optimum CL'_S. The algorithm has to decide whether the search must continue from the previous local optimum CL_C or from the last found local optimum CL'_S. ILS uses the L-BFGS to learn the weights and the WPLL measure as the evaluation function.

- **LHL** (Learning via Hypergraph Lifting) performs through two steps; building a lifted-graph then creating candidate clauses over this lifted-graph.

  In LHL, a training database is viewed as a hyper-graph with constants as nodes and true ground atoms as hyper-edges. Each hyper-edge is labeled with a predicate symbol. Nodes (constants) are linked by a hyper-edge if they appear as arguments in the atom. A path of hyper-edges can be variabilized to form a first-order clause. To avoid tracing the exponential number of paths in the hyper-graph, LHL first jointly clusters the nodes into higher-level concepts stored into a lifted-graph, whose each node is a set of constants and each edge corresponds to a predicate. The lifted-graph has fewer nodes and hyper-edges and therefore fewer paths than the hyper-graph.

  LHL next uses relational path-finding [Richards & Mooney 1992] over the lifted-graph to find paths then variabilizes them in order to create clauses. Each clause is evaluated using the WPLL measure and the L-BFGS algorithm to learn weights. Evaluated clauses are iterated from shortest to longest. For each clause, LHL compares its measure (i.e. WPLL) against those of its sub-clauses (considered separately) that have already been retained. If the clause is better than all of its sub-clauses, it is retained; otherwise, it is discarded. Finally, the retained clauses are added to the final MLN. LHL provides an option to choose either to add one clause at a time or all clauses at a time.

- **LSM** (Learning using Structural Motifs) [Kok & Domingos 2010] is a further development of LHL which is based on the lifted-graph, *truncated hitting time* [Sarkar *et al.* 2008] and *random walks* [Lovasz 1996] to identify densely connected objects in database, group them with their associated relations into a motif, and then constrain the search for clauses to occur within motifs.

A structural motif is a set of literals, which defines a set of clauses that can be created by forming disjunctions over the negation/non-negation of one or more of the literals. Thus, it defines a subspace within the space of all clauses. LSM discovers subspaces where literals are densely connected and groups them into a motif. To do so, LSM also views a database as a hyper-graph, then groups nodes that are densely connected by many paths and the hyper-edges connecting them into a motif. Then it compresses the motif by clustering nodes into high-level concepts, reducing the search space of clauses in the motif. Next it quickly estimates whether the motif appears often enough in data to be retained. Finally, LSM runs relational path-finding on each motif to find candidate rules and retains the good ones in an MLN.

LSM differs from LHL in the following ways: first, LHL finds a single clustering of nodes while in LSM a node can belong to several clusters. Second, LSM uses longer paths instead of length-2 paths in LHL, and thus more information, to find various clusterings of nodes. Third, spurious edges present in the initial ground hyper-graph of LHL are retained in the lifted one, but these edges are ignored by LSM.

LSM creates candidate clauses from each path in a similar way as LHL with some modifications. At the beginning, LSM counts the true groundings of all possible unit and binary clauses to find those that are always true or always false in data. It then removes every candidate clause that contains unit/binary sub-clauses that are always true as they are always satisfied. If a candidate clause c contains unit/binary sub-clauses that are always false, and if the clause c' formed by removing the unit/binary sub-clauses is also a candidate clause, then c is removed because it is a duplicate of c'. LSM also detects whether a binary predicate R is symmetric by evaluating whether $R(x, y)$, $R(y, x)$ is always true. LSM then removes clauses that are identical modulo the order of variables in symmetric binary predicates. At the end LSM adds all clauses to the MLN, finds their optimal weights, and removes those whose weights are less than a given threshold θ_{wt}. LHL also use L-BFGS to learn weights and WPLL to evaluate clauses. Before evaluating the WPLLs of candidate clauses against the whole data, it evaluates them against the ground hyper-graphs that give rise to the motifs where the candidate clauses are found. Since such ground hypergraphs contain fewer atoms, it is faster to evaluate against them to quickly prune bad candidates.

LHL adds clauses into the final MLN in the same way as LHL.

- **MBN** (Moralized Bayes Net) [Khosravi *et al.* 2010] first learns a parametrized Bayes net (BN) [Poole 2003] then applies a standard moralization technique [Domingos *et al.* 2008a] to the BN to produce a MLN.

  A parametrized Bayes net structure [Poole 2003] consists of a directed acyclic graph (DAG) whose nodes are parametrized random variables, a population for each first-order variable and an assignment of a range to each functor. A population is a set of individuals, corresponding to a domain or type in logic. A parametrized random variable is of the form $f(t_1, \ldots, t_k)$ where f is a functor (either a function symbol or a predicate symbol) and each t_i is a first-order variable or a constant. Each functor has a set of values (constants) called the range of the functor.

  MBN first learns the structure of a parametrized BN. The algorithm upgrades a

single table BN learner, which can be chosen by the user, by decomposing the learning problem for the entire database into learning problems for smaller tables. The basic idea is to apply the BN learner repeatedly to tables and join tables from database, and combine the resulting graphs into a single graphical model for the entire database. As the computational cost of the merge step is negligible, the run-time of this learn-and-join algorithm is essentially determined by the cost of applying the BN learner to each (join) table separately. Thus MBN leverages the scalability of single-table BN learning to achieve scalability for MLN structure learning.

Bayes net DAGs can be converted into MLN structures through the standard moralization method [Domingos *et al.* 2008a] connecting all spouses that share a common child, and making all edges in the resulting graph undirected. In the moralized BN, a child forms a clique with its parents. For each assignment of values to a child and its parents, a formula is added to the MLN.

3.1.2.2 Discriminative Structure Learning

In many learning problems, there is a specific target predicate that must be inferred given evidence data; discriminative learning is then preferred. Concerning discriminative structure learning, to the best of our knowledge, there exists only two systems; the one developed by [Huynh & Mooney 2008] and the Iterated Local Search - Discriminative Structure Learning (ILS-DSL) developed by [Biba *et al.* 2008a].

- **The method of** [Huynh & Mooney 2008] is the first approach to construct MLNs that discriminatively learns both structure and parameters to optimize predictive accuracy for a query predicate given evidence specified by a set of defined background predicates. It uses a variant of an existing ILP system (ALEPH [Srinivasan 2007]) to construct a large number of potential clauses and then effectively learns their parameters by altering existing discriminative MLN weight-learning methods in Alchemy [Kok *et al.* 2009] to use exact inference and L_1-regularization [Lee *et al.* 2007].

- **ILS-DSL (Iterated Local Search - Discriminative Structure Learning)** [Biba *et al.* 2008a] learns discriminatively first-order clauses and their weights. This algorithm shares the same structure with the ILS approach for generative MLN structure learning. It also learns the parameters by maximum pseudo-likelihood using the L-BFGS algorithm but scores the candidate structures using the CLL measure.

3.1.3 Inference

This subsection presents two basic types of inference: finding the most likely state of the world consistent with some evidence, and computing arbitrary conditional probabilities.

It must be noticed that, in this dissertation, we concentrates on the development of algorithms for MLN structure learning in both generative and discriminative models. Our algorithms are implemented over the Alchemy package [Kok *et al.* 2009] which involve several built-in tools for weight learning and inference. Inference methods that we present in this section do not form an exhaustive because there has been a lot of work conducted on this problem in the past two years. We limit to the methods implemented in Alchemy (detail in Section 3.2). Some of them are used in our approaches.

3.1.3.1 MAP Inference

A basic inference task consists in finding the most probable state of the world y given some evidence x, where x is a set of literals. This is known as the MAP (Maximum A Posteriori) inference in the Markov network literature. In the context of Markov logic, this is formally defined as follows [Domingos & Lowd 2009]:

$$\begin{aligned} \arg\max_y P(y|x) &= \arg\max_y \frac{1}{Z_x} \exp\left(\sum_i w_i n_i(x,y)\right) \\ &= \arg\max_y \sum_i w_i n_i(x,y) \end{aligned} \tag{3.9}$$

where *n(x, y)* is the number of true instantiations of F_i in $x \cup y$. The first equality is due to Equation 3.1, which defines the probability of a possible world. The normalization constant is written as Z_x to reflect the fact that we are only normalizing over possible worlds consistent with x. In the second equality, Z_x is removed since, being constant, it does not affect the arg max operation. The exponentiation can also be removed because it is a monotonic function. Therefore, the MAP problem in Markov logic reduces to finding the truth assignment that maximizes the sum of weights of satisfied clauses.

This problem can be solved using any weighted satisfiability solver. It is NP-hard in general, but there exist both exact and approximate effective solvers. The most commonly used approximate solver is MaxWalkSAT [Richardson & Domingos 2006], a weighted variant of the WalkSAT local-search satisfiability solver, which can solve hard problems with hundreds of thousands of variables in minutes [Kautz *et al.* 1996].

3.1.3.2 Marginal and Conditional Probabilities

Another inference task consists in computing the probability that a formula F_1 holds, given a MLN and a set of constants, and possibly other formulas F_2 as evidence. If C is a finite set of constants including the constants that appear in F_1 or F_2, and L is an MLN, then:

$$\begin{aligned} P(F_1|F_2, M_{L,C}) &= \frac{P(F_1 \wedge F_2, M_{L,C})}{P(F_2|M_{L,C})} \\ &= \frac{\sum_{x \in \mathcal{X}_{F_1} \cap \mathcal{X}_{F_2}} P(X = x|M_{L,C})}{\sum_{x \in \mathcal{X}_{F_2}} P(X = x|M_{L,C})} \end{aligned} \tag{3.10}$$

where $\mathcal{X}_{F_i}$ is the set of worlds where F_i holds, $M_{L,C}$ is the MN defined by L and C, and $P(X = x|M_{L,C})$ is given by Equation 3.1.

Computing Equation 3.10 directly has been shown intractable for large domains [Richardson & Domingos 2006]. Although, it can be approximated using an MCMC (Markov Chain Monte Carlo) algorithm [Gilks & Spiegelhalter 1999] that rejects all moves to states where F_2 does not hold, and counts the number of samples in which F_1 holds. However, this is still likely to be too slow for arbitrary formulas [Richardson & Domingos 2006].

There are several methods for inference in MLN that restrict the problem by considering the typical case where the evidence F_2 is a conjunction of ground atoms. The authors in

[Richardson & Domingos 2006] proposed an algorithm that works in two phases. The first phase returns the minimal set M of the ground MN required to compute $P(F1|F2, L, C)$. The second phase performs inference on this network, with the nodes in F_2 being set to their values in F_2. A possible method is Gibbs sampling, but any other inference method may be used. The basic Gibbs step consists of sampling one ground atom given its *Markov blanket*. In MN, a Markov blanket of a node is simply the set of its neighbors in the graph. The probability of a ground atom X_l when its Markov blanket B_l is in state b_l is given by:

$$P(X_l = x_l) = \frac{\exp(\sum_{f_i \in F_l} w_i f_i(X_l = x_l, B_l = b_l))}{\exp(\sum_{f_i \in F_l} w_i f_i(X_l = 0, B_l = b_l)) + \exp(\sum_{f_i \in F_l} w_i f_i(X_l = 1, B_l = b_l))} \tag{3.11}$$

where F_l is the set of ground formulas that X_l appears in, and $f_i(X_l = x_l, B_l = b_l)$ is the value (0 or 1) of the feature corresponding to the *i-th* ground formula when $X_l = x_l$ and $B_l = b_l$. The estimated probability of a conjunction of ground literals is simply the fraction of samples in which the ground literals are true, once the Markov chain has converged.

One of the problems that arise in real-world applications, is that an inference method must be able to handle probabilistic and deterministic dependencies that might hold in the domain. MCMC methods are suitable to handle probabilistic dependencies but give poor results when deterministic or near deterministic dependencies characterize a certain domain. On the other side, logical methods, like satisfiability testing cannot be applied to probabilistic dependencies. One approach to deal with both kinds of dependencies is that of [Poon & Domingos 2006], called *MC-SAT*, where the authors use the SampleSAT [Wei *et al.* 2004] in a MCMC algorithm to uniformly sample from the set of satisfying solutions.

Experimental results in [Poon & Domingos 2006] show that MC-SAT greatly outperforms Gibbs sampling. They then developed *Lazy-MC-SAT* [Poon *et al.* 2008], a lazy version of MC-SAT. Lazy-MC-SAT was shown to greatly reduce memory requirements for the inference task.

It is also possible to carry out lifted first-order probabilistic inference (akin to resolution) in Markov logic [Braz *et al.* 2005]. These methods speed up inference by reasoning at the first-order level about groups of indistinguishable objects rather than propositionalizing the entire domain. Several lifted inference methods based on belief propagation [Bishop 2006] can be found in [Singla & Domingos 2008, Kersting *et al.* 2009].

3.2 Alchemy

Alchemy [Kok *et al.* 2009] is an open source software package for MLNs written in *C++*. It includes implementations for all the major existing algorithms for structure learning, generative weight learning, discriminative weight learning, and inference. Full documentation on Alchemy is available at `http://alchemy.cs.washington.edu`. The syntax and options of algorithms are available in the user's guide; information about the internal workings of Alchemy is in the developer's guide; and a tutorial (including MLN and database files) is available that covers much of the functionalities of Alchemy through examples. Our proposed algorithms are implemented using the API functions of Alchemy.

```
//predicate declarations
Friends(person, person)
Smokes(person)
Cancer(person)
// If you smoke, you get cancer
1.5 Smokes(x) => Cancer(x)
// Friends have similar smoking habits
1.1 Friends(x, y) => (Smokes(x) <=> Smokes(y))
```

Table 3.1: example.mln: An .mln input file in Alchemy

3.2.1 Input files

In Alchemy, predicates and functions are declared and first-order formulas are specified in *.mln* files. The first appearance of a predicate or function in a .mln file is taken to be its *declaration*. A variable in a formula must begin with a lowercase character, and a constant with an uppercase one in a formula. A formula can be preceded by a weight or terminated by a period, but not both. A period means that a formula is "hard" (i.e., worlds that violate it should have zero probability). Types and constants are also declared in .mln files. Comments can be included with // and /* */ as in C++ or Java. Blank lines are ignored. Table 3.1 contains a .mln input file example, named *example.mln*, for the MLN presented in Example 9. The MLN begins by declaring the predicates Friends, Smokes, and Cancer, each taking one or two arguments of type person followed by two formulas and their weights.

Ground atoms are defined in .db (database) files. Ground atoms preceded by "!" (e.g., !Friends(Anna,Bob)) are false, by "?" are unknown, and by neither are true.

Alchemy supports also several others notation and declarations. We refer the reader to the on-line manual at url `http://alchemy.cs.washington.edu/` for more information on Alchemy syntax and other features.

3.2.2 Inference

The *infer* executable is used to perform inference. For example, the command:

```
infer -i smoking.mln -e example.db -r inference.out -q Smokes -ms
```

performs inference for the query predicate *Smokes* given the MLN in *smoking.mln* and data in *example.db*, in which:

-i specifies the input .mln file,

-e specifies the evidence .db file,

-r specifies the output file, which contains the inference results,

-q specifies the query predicates.

-ms specifies to use the MC-SAT algorithm

The output is stored in the file *inference.out* containing the list of query ground atoms and its probabilities like this:

Smokes(Chris) 0.238926

Smokes(Daniel) 0.141286

The command *infer* without any parameters is used to view all arguments for inference.

3.2.3 Weight Learning

Weight learning can be achieved by using the *learnwts* command. This latter supports both generative and discriminative weight learning. For example, the command:

learnwts -d -i example.mln -o learnWeightOut.mln -t example.db -ne Smokes

learns discriminatively weights for the MLN in *example.mln*, for the query predicate *Smokes* given database in *example.db*. The parameters:

-d specifies discriminative learning; for generative learning, use -g instead,

-i and -o specify the input and output .mln files

-t specifies the .db file

-ne specifies the non-evidence predicates, those that will be unknown and inferred at inference time.

After weight learning, the output .mln file contains the weights of the original formulas as well as those of their derived clauses because each formula is converted to conjunctive normal form (CNF), and a weight is learned for each of its clauses.

One can view all the options by using the command *learnwts* without any parameters.

3.2.4 Structure Learning

Structure learning can be done by using the command *learnstruct*. For example, the command:

learnstruct -i example.mln -o example-out.mln -t example.db

learns a structure given the initial MLN in *example.mln* and the database in *example.db*. The learned MLN is stored in *example-out.mln*. The parameters -i, -o, -t specify respectively the input .mln file, output .mln file and database file.

One can use the command *learnstruct* without any parameters to view all its options.

3.3 Summary

Inference and learning are two principle tasks for MLNs. Inference can be achieved by performing a MAP inference or a conditional probabilistic inference. Learning can be separated into parameter (weight) learning and structure learning and both generative model and discriminative model can be applied to them. We bring in this chapter an overview of these tasks for MLNs. In particular, for structure learning, we draw a portrait of proposed methods up to date and for the remaining, we presented an overview of methods implemented in Alchemy.

In the next two chapters, we present the methods we proposed for MLN structure learning.

Chapter 4

Learning MLN Structure Based on Propositionalization

Résumé: Nous présentons dans ce chapitre trois méthodes pour apprendre la structure d'un réseau logique de Markov: l'algorithme HGSM pour l'apprentissage génératif et HDSM et DMSP pour l'apprentissage discriminant. L'idée de base dans ces méthodes est d'effectuer une propositionnalisation dans laquelle des informations relationnelles exprimées par les atomes du jeu de données sont transformées en des tableaux booléens, dont chaque colonne correspond à un littéral avec seulement des variables. Un algorithme d'apprentissage automatique est appliqué sur ces tableaux booléens afin de trouver des relations entre les littéraux, à partir desquelles des clauses candidates sont ensuite créées pour apprendre le réseau logique de Markov final.

Notre première technique de propositionnalisation est mise en œuvre dans l'algorithme génératif HGSM. Pour chaque prédicat à apprendre, le jeu de données est divisé en groupes disjoints d'atomes connectés à partir d'un atome vrai de ce prédicat. Une méthode heuristique de variabilisation est appliquée sur ces groupes d'atomes connectés, du plus grand au plus petit, afin de construire un ensemble de littéraux avec seulement des variables. Les informations relationnelles entre atomes dans le jeu de données sont utilisées pour remplir les tableaux booléens. L'algorithme GSMN (Grow-Shrink Markov Network) est ensuite appliqué pour chaque tableau booléen afin de trouver une couverture de Markov (Markov blanket) du littéral correspondant. Les clauses candidates sont ensuite créées à partir de cette couverture de Markov. L'intérêt des clauses est évalué à l'aide de la mesure de pseudo log vraisemblance pondérée (WPLL).

L'approche appliquée dans HGSM est adaptée à l'apprentissage discriminant dans le système HDSM. Au lieu de considérer tous les prédicats du domaine, HDSM n'apprend que pour un seul prédicat de requête. HDSM utilise une mesure discriminante, la log vraisemblance conditionnelle (CLL), pour choisir les clauses au lieu de la WPLL qui constitue une mesure générative, Les résultats des expérimentations montrent que HDSM donne des résultats meilleurs que les systèmes de l'état de l'art pour l'apprentissage discriminant de la structure d'un réseau logique de Markov, sur les jeux de données classiquement utilisés.

Une deuxième technique de propositionnalisation est implémentée dans l'algorithme DMSP pour l'apprentissage discriminant de la structure d'un réseau logique de Markov. La différence avec la première technique réside dans la façon de construire un ensemble de littéraux avec seulement des variables. Cette deuxième technique permet de créer un ensemble de littéraux beaucoup plus rapidement et de manière plus compacte que la première. Elle repose sur l'idée que beaucoup de chemins entre atomes clos peuvent être décrits par un seul chemin entre littéraux avec seulement des variables. L'algorithme, par conséquent, variabilise d'abord un chemin de littéraux, qui est ensuite utilisé comme filtre pour ignorer beaucoup d'autres chemins. De cette manière, l'ensemble des littéraux est

engendré beaucoup plus rapidement et nous avons également prouvé qu'il est le plus petit ensemble, capable de décrire toutes les relations liées au prédicat de requête dans le jeu de données. DMSP utilise le test d'indépendance du χ^2 au lieu de l'approche Grow-Shrink afin de générer un peu plus de clauses candidates.

Nous terminons ce chapitre en discutant en détail les différences entre notre méthode et BUSL, un algorithme d'apprentissage génératif de structure d'un réseau logique de Markov, qui utilise aussi une technique de propositionnalisation pour créer un ensemble de clauses candidates.

Contents

4.1 Introduction

In this chapter, we develop several algorithms for learning the structure of MLNs from relational databases which store data in form of ground atoms. Our algorithms combine ideas from Propositionalization in ILP and the Markov blanket in Markov networks.

We recall that the Markov blanket of a node in a Markov network is simply the set of its neighbors in the graph [Domingos *et al.* 2008a]. A MLN can be viewed as a template to construct Markov networks. Given different sets of constants, it will produce different MNs of ground atoms with varying sizes, but all of them contain regularities in structure and parameters, given by the MLN. For example, given a clause in a MLN, all its instantiations have the same weight, and a set of ground atoms occurring in an instantiation form a clique in the graph which is similar to the clique formed by the variable literals of this clause. This means that if two ground atoms occur in an instantiation of a clause, then they are dependent and further in the template graph (corresponding to the MLN), if two variable literals L_i and L_j occur in a clause, then they are dependent and by construction, L_i is in the Markov blanket of L_j (i.e. L_i is a neighbor of L_j), and vice versa. In other words,

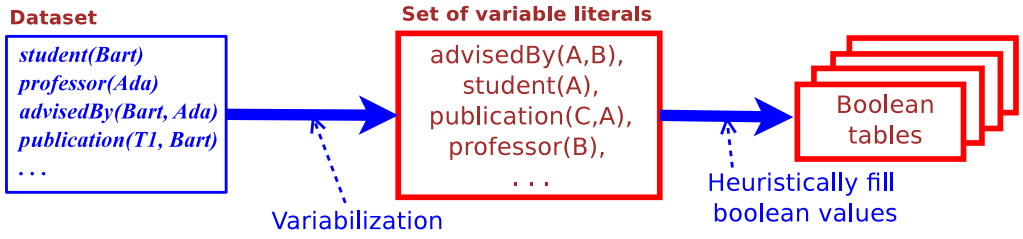


Figure 4.1: Propositionalization

any two variable literals participating together in a clause are dependent to each other and they must be connected by an edge in the graph of variable literals participating in the clauses of this MLN.

In our methods, candidate clauses are generated from sets of *dependent variable literals*. However, the problem of finding such a set of variable literals is a difficult task since the dataset is only a set of ground atoms without any predefined template. In addition, most of efficient statistical independence testing methods such as the Pearson's conditional independence χ^2-test [Agresti 2002] and G-test of independence [McDonald 2009] require data to be arranged into a contingency table [Agresti 2002], which is very different from a set of ground atoms. We propose in this chapter two propositional methods in order to transform information in the training database into an approximative representation in form of boolean tables, from which statistical methods for testing of independence can be applied.

Basically, a boolean table in our propositionalization methods holds information about relational ground atoms: each column corresponds to a variable literal, each row corresponds to a ground atom of the considered predicate. The boolean values on each row r express information on other ground atoms connected to the corresponding ground atom of r. These propositionalization methods can be briefly described as in Figure 4.1. Starting from data, a set of variable literals is created by a heuristic variabilization technique. This set of variable literals is then used to transform information about relational ground atoms in database into an approximative representation of boolean tables.

In this chapter, we present the first propositionalization technique developed in the HGSM [Dinh *et al.* 2010b] and HDSM [Dinh *et al.* 2010c] algorithms in Section 4.2. The second propositionalization technique developed in the DMSP algorithm [Dinh *et al.* 2010a] is exposed in Section 4.3. Related works are discussed in Section 4.4. Finally, Section 4.5 is the summary of this chapter.

4.2 The HGSM and HDSM Algorithms

We have as inputs a training database DB composed of true/false ground atoms and initial knowledge in form of a MLN (which can be emply). We are looking for a set of weighted clauses for all predicates in the domain for the task of generative MLN structure learning, or a set of weighted clauses that correctly discriminates between true and false ground atoms of a given query predicate QP for the task of discriminative MLN struc-

ture learning. We present in this section an algorithm for Heuristic Generative Structure learning for MLNs, called *HGSM* [Dinh *et al.* 2010b] and an algorithm for Heuristic Discriminative Structure learning for MLNs, called *HDSM* [Dinh *et al.* 2010c]. As these two methods exploit a similar method of propositionalization, in the following, we first introduce some definitions supported to facilitate our presentation then describe in details how propositionalization is implemented before explaining step by step each algorithm.

4.2.1 Definitions

Definition 10 *Let* g *and* s *be two ground literals. A link of* g *and* s*, denoted by* link(g,s)*, is a list composed of the name of the predicates of* g *and* s *followed by the positions of the shared constants. It is written by* link(g, s) = {G S g_0 s_0 | g_1 s_1 | ... } *where* G *and* S *are respectively two predicate symbols of* g *and* s *and two constants respectively at positions* $g_i \in [0, arity(g) - 1]$ *and* $s_i \in [0, arity(s) - 1]$ *are the same.*
If there is no shared constant between g *and* s *then* link(g,s) *is empty.*

Definition 11 *A* g-chain *of ground literals starting from a ground literal* g_1 *is a list of ground literals* $\langle g_1, ..., g_k, ...\rangle$ *such that* $\forall i > 1$*,* $link(g_{i-1}, g_i)$ *is not empty and every shared constant is not shared by* $g_{j-1}, g_j, 1 < j < i$*. It is denoted by:* $g\text{-}chain(g_1) = \langle g_1, ..., g_k, ...\rangle$*. The length of a g-chain is the number of ground atoms in it.*

Definition 12 *A group of ground atoms* $SGA \in DB$ *is connected to a ground atom* $a \in DB$ *if* $a \in SGA$ *and* $\forall b \in SGA$*,* $b \neq a$ *there exists at least a g-chain of ground atoms in* SGA *starting from* a *and containing* b.
If SGA is connected to the ground atom a *then SGA is called a connected group starting from* a.
The width *of a connected group being the number of ground atoms in it.*

Similarly, we define a *link* of two variable literals as a list composed of the name of the predicates followed by the positions of the shared variables, a *v-chain* as a *chain* of variable literals.

Example 13 *Let* G = {P(a, b), Q(b, a), R(b, c), S(b), S(c)} *be a set of ground atoms.* P(a, b) *and* Q(b, a) *are connected by the two shared constants* a *and* b*. The constant* a *occurs respectively at position* 0 *of the ground atom* P(a, b) *and at position* 1 *of the ground atom* Q(b, a)*. Similarly, the constant* b *occurs respectively at position* 1 *of the ground atom* P(a, b) *and at position* 0 *of the ground atom* Q(b, a)*. We have:*

$$\text{link}(\text{P(a, b)}, \text{Q(b, a)}) = \{\text{P Q 0 1} \mid \text{1 0}\}$$

A possible g-chain gc_1 *starting from the ground atom* P(a, b) *is:*

$$gc_1 \equiv \text{g-chain}(\text{P(a, b)}) = \langle \text{P(a, b)}, \text{R(b, c)}, \text{S(c)}\rangle.$$

Note that there might be a lot of g-chains starting from a ground atom, we use here the symbol "≡" to denote that gc_1 *is exactly the g-chain* ⟨P(a, b), R(b, c), S(c)⟩.

On the other hand, ⟨P(a, b), R(b, c), S(b)⟩ *is not a g-chain because the shared constant* b *between the two ground atoms* R(b, c) *and* S(b) *is already used for the two ground atoms* P(a, b) *and* R(b, c).

It is easy to verify that G *is a connected group starting from* P(a, b).

The definition of *g-chain* ensures that its variabilization produces a connected clause. A *v-chain* can also form a connected clause. The notion of g-chain is related to the relational path-finding [Richards & Mooney 1992] and relational cliché [Silverstein & Pazzani 1991], which have been used in the Inductive Logic Programming [Lavrac & Dzeroski 1994].

Definition 14 *A* template clause *is a disjunction of positive variable literals.*

Definition 15 *A template graph is a graph, each node of which corresponds to a variable literal.*

4.2.2 Propositionalization Method

In our approach, propositionalization is applied on for every predicate in the domain in case of generative learning and on only the query predicate in case of discriminative learning. For the sake of simplicity, we use "the target predicate" for both cases (it means the query predicate for discriminative learning and the considering predicate in each iteration for generative learning). As it is briefly described in Figure 4.1, propositionalization is performed through two consecutive steps. At the first step, a set of variable literals is created starting from the dataset. This set of variable literals is then used in the second step to form boolean tables and fill values to them. The process of generating variable literals and then building boolean tables produces an approximative representation of relational ground atoms in the database. The size and quality of the boolean tables affect the performance of the algorithms. The size of boolean tables depends on the number of generated variable literals. We want to find a set of variable literals as small as possible but large enough to encode relational information of the database.

Section 4.2.2.1 explains the first step in this propositionalization method.

4.2.2.1 Creating a Set of Variable Literals

Let us consider a target predicate QP. The task is to build a set of variable literals linked to QP. Finding this set is difficult since the database is only a set of ground atoms without any given template of literal. Our idea is to split data into distinct groups, each of them is a connected group starting from some true ground atom of QP in the database. To save time, while searching for such a connected group, a *bread-first-search* strategy is applied so that the algorithm stops whenever no ground atom can be added into the group. Each connected group is then variabilized to generate variable literals. Because there might be a lot of connected groups hence a lot of generated variable literals, a heuritic variabilization technique is used in order to generate a number of variable literals as small as possible. In more details, this heuritic variabilization technique is iterated for each connected group, from the largest to the smallest. For each connected group (or for each iteration), it tries to describe relational information between ground atoms by reusing as much former generated variable literals as possible, thereby it can, more or less, avoid generating new variable literals.

Algorithm 1 outlines the steps to create a set of variable literals. It takes a training database DB and a target predicate QP as inputs and then returns a set SL of variable literals. For each true ground atom tga of QP in DB (*line 2*), the connected group of ground atoms in DB starting from tga is built by a function called *MakeGroup* (*line*

5). Let us emphasize that, here, we have a single and unique connected group starting from *tga*, which is far different from the above definition of a *g-chain* starting from *tga*. Intuitively, the connected group starting from *tga* contains all ground atoms, each of which is in some g-chain starting from *tga*. With the support of the bread-first-search strategy, this connected group is found much faster than finding every g-chain, which needs involve a *deep-first-search* procedure. From these connected groups, the set *SL* is then built so that for each connected group, there exists a variabilization of it with all variable literals belonging to *SL*. It must also be noted that the function *MakeGroup* will stop whenever all the ground atoms connected to the ground atom *tga* (via one or more intermediate another ground atoms) are already in the group, therefore it does not take so much time even when the dataset is large ($O(n^2)$ where n is the number of true ground atoms in the database).

Algorithm 1: Generate a set of variable literals(DB, QP)

Input : *DB:* a training database;
QP: a target predicate;
Output: *SL:* a set of variable literals;

```
// Initialization
index = −1; mI the number of true ground atoms of QP;
// Separate set of connected ground atoms
for each true ground atom tga of QP do
    if tga is not considered then
        index = index + 1;
        Groups[index] ⟵ MakeGroup(tga);
    end
end
// Variabilizing
Sort Groups by decreasing length;
SL ⟵ Variabilize(Groups[0]);
SL ⟵ CaptureLiteral(Groups[i]), 1 ≤ i ≤ mI ;
Return(SL);
```

Regarding this variabilization problem, we use a *simple variabilization strategy* to variabilize each connected group ensuring that different constants in the connected group are replaced by different variables. However, instead of variablizing separately each group as the traditional simple variabilization strategy does, variable literals are reused as much as possible and extended throughout the connected groups, from the largest to the smallest, therefore reducing the number of new variable literals. A heuristic technique is used to check whether a new variable literal needs to be generated for a link between ground atoms in a connected group or not.

We illustrate this variabilization technique by an example below before explaining it in detail.

Example 16 *In order to illustrate our approach, let us consider a training database consisting of 13 ground atoms as follows:*

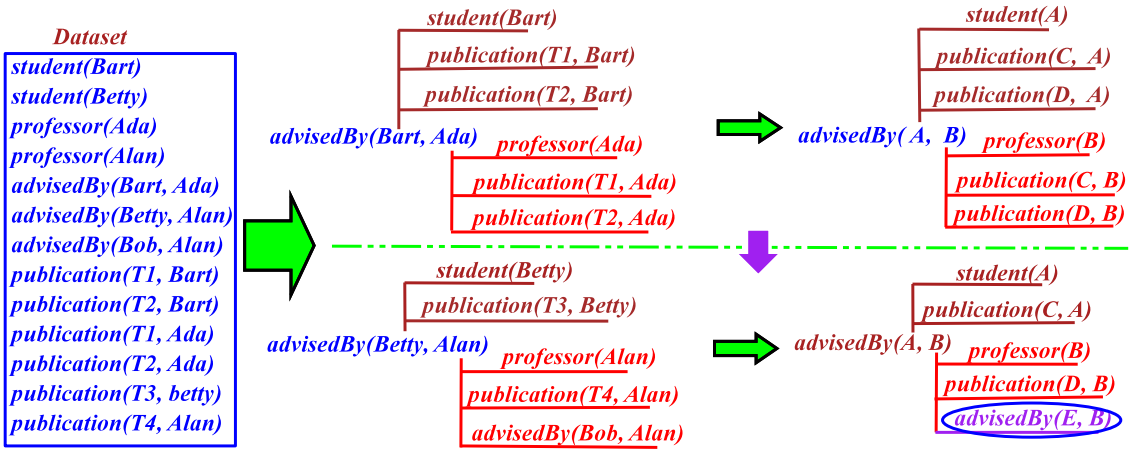


Figure 4.2: Example of chains in the variabilization process of HGSM

student(Bart), student(Betty), professor(Ada), professor(Alan), advisedBy(Bart, Ada), advisedBy(Betty, Alan), advisedBy(Bob, Alan), publication(T1, Bart), publication(T1, Ada), publication(T2, Ada), publication(T2, Bart), publication(T3, Betty), publication(T4, Alan).

Let QP={advisedBy}. Figure 4.2 illustrates the different steps in the variable process presented in Algorithm 1.

Let us start from the true ground atom advisedBy(Bart, Ada). *The widest connected group that can be built starting from this ground atom contains 7 atoms as:*

{advisedBy(Bart, Ada), stu(Bart), publication(T1, Bart), publication(T2, Bart), professor(Ada), publication(T1, Ada), publication(T2, Ada)},

in which, for example advisedBy(Bart, Ada) *and* publication(T1, Bart) *are linked by the constant* Bart.

The algorithm then variabilizes this connected group by assigning each constant a variable, so that two different constants are assigned different variables, to get the following set of literals:

SL={advisedBy(A, B), student(A), publication(C,A), publication(D, A), professor(B), publication(C, B), publication(D, B)}.

Let us consider now the true ground atom advisedBy(Betty, Alan), *then we get a smaller connected group of six ground atoms:*

{advisedBy(Betty, Alan), student(Betty), professor(Alan), advisedBy(Bob, Alan), publication(T3, Betty), publication(T4, Alan)}.

The above set SL *of variable literals is not sufficient to capture the relation among the two ground atoms* {advisedBy(Betty, Alan), advisedBy(Bob, Alan)}, *and one more variable literal* advisedBy(E, B) *is added to the set* SL.

Let us describe this heuristic variabilization technique here in more details. Given a set, *Groups*, of connected groups in decreasing order of the width, the variabilization technique is performed as follows:

1. *Variabilizing the first (also the widest) connected group:* a simple variabilization method is used to assign each constant a variable so that two different constants are assigned different variables.

2. *Variabilizing smaller connected groups:*

   (a) The starting ground atom is variabilized by the same variable literal for every connected group. For example both ground atoms *advisedBy(Bart, Ada)* and *advisedBy(Betty, Alan)* in Figure 4.2 are assigned to the same variable literal *advisedBy(A, B)*. The two variables *A* and *B* are then reused respectively for the two constants *Betty* and *Alan* for the connected group starting from *advisedBy(Betty, Alan)*.

   (b) A scanning process is performed to try to reuse variable literals as well as to capture additional variable literals. During the scanning, the algorithm follows precisely two principles:

   - **Check for the existing links:** This happens when there are several similar links which have occurred in some previous considered connected group. In this case, the algorithm tries to assign constants to the *first variables it can find*. For instances, consider the link between *advisedby(Betty, Alan)* and *publication(T3, Betty)*, it now has to find variable for the constant *T3* because *Betty* and *Alan* are already assigned to *A* and *B*. In this case, it reuses the variable *C* for the constant *T3* as *C* is formerly used for the constant *T1* in the link(advisedBy(Bart, Ada), publication(T1, Bart)), which is equal to the link(advisedby(Betty, Alan), publication(T3, Betty)). Similarly, it reuses variable *D* for the constant *T4* in the link between *advisedby(Betty, Alan)* and *publication(T4, Alan)*. Here, the variable *C* is found firstly for this link, but it is already used for the constant *T3*, the algorithm has to turn to other variable, hence *D* is chosen. When the algorithm *can not find any variabe*, it *creates a new one*. For example, we assume that the dataset in our example contains one more true ground atom *publication(T5, Alan)* as illustrated in Figure 4.3. The relation between the two ground atoms *advisedBy(Betty, Alan)* and *publication(T5, Alan)* is similar to the relation between the two ground atoms *advisedBy(Betty, Alan)* and *publication(T4, Alan)*, both of them sharing the constant *Alan* at the same position. The set *SL* is sufficient to variabilize the relations among the three ground atoms *advBy(Betty, Alan)*, *publication(T3, Betty)*, *publication(T4, Alan)* (respectively by *advisedBy(A, B)*, *publication(C, A)* and *publication(D, B)*) but it is not sufficient to variabilize relations among the four ground atoms *advisedBy(Betty, Alan)*, *publication(T3, Betty)*, *publication(T4, Alan)* and *publication(T5, Alan)*. In this case, a new variable literal *publication(F, B)* is used to variabilize the ground atom *publication(T5, Alan)*.

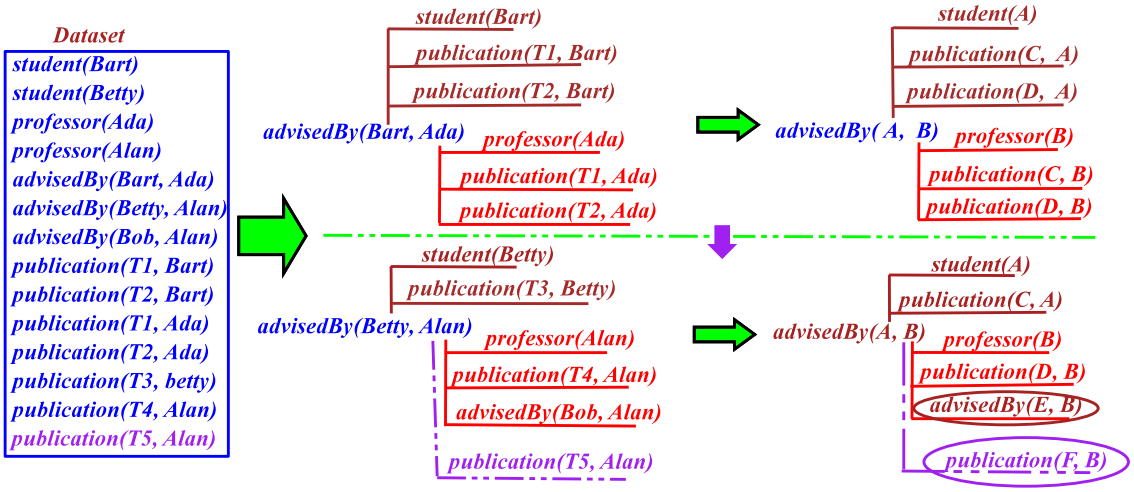


Figure 4.3: Example of adding new variable in HGSM

- **Check for a new link:** This happens when there is a new link between two ground atoms in the considering connected group, or in other words, when a link between two ground atoms is never considered before to variabilize. For example, the relation between the two ground atoms *advisedBy(Betty, Alan)* and *advisedBy(Bob, Alan)* is a new one. Because the ground atom *advisedBy(Betty, Alan)* is already variabilized by the variable literal *advisedBy(A, B)* and the two variables C and D are already used for other relations in the set SL, a new variable E is introduced to form a new variable literal *advisedBy(E, B)* in order to variabilize the true ground atom *advisedBy(Bob, Alan)*. This guarantees the link between the two ground atoms *advisedBy(Betty, Alan)* and *advisedBy(Bob, Alan)* by the shared variable B between the two variable literals *advisedBy(A, B)* and *advisedBy(E, B)*.

Example 17 *Repeating this scanning process until the last true ground atom of the given predicate* advisedBy*, Algorithm 1 produces the set of eight variable literals as follows:*

SL={advisedBy(A, B), student(A), publication(C, A), publication(D, A), professor(B), publication(C, B), publication(D, B), advisedBy(E, B)}.

One can realize that more than one variable literal of the target predicate advisedBy *are generated. They are* advisedBy(A, B) *and* advisedBy(E, B)*. The second step of propositionalization will build a boolean table corresponding to each one of them.*

4.2.2.2 Building a Boolean Table

The next step in our approach of propositionalization transforms information from the relational database into boolean tables corresponding to every target predicate QP. We remind that there might be more than one generated variable literals of the target predicate, and thus create a boolean table corresponding to each such variable literal. This

boolean table, called BT, needs to catch information related to the target predicate as much as possible. It is organized as follows: each column corresponds to a variable literal; each row corresponds to a true/false ground atom of the query predicate. Let us assume that data concerning a given ground atom q_r of QP is stored in row r. Let us also assume that column c corresponds to a given variable literal vl_c. *BT[r][c]* = true means that starting from q_r we can reach a literal that is variabilized as vl_c. In other words, there exists at least a v-chain $vc \in SL$ containing the variable literal vl_c, a g-chain gc_r starting from the ground atom q_r, and a variabilization of gc_r such that $vc \subseteq var(gc_r)$.

Let us consider a connected clause $C = A_1 \vee A_2 \vee \cdots \vee A_k$, where the number of variable literals in C is k. Since the clause is connected, from any variable literal $A_i, 1 \leq i \leq k$ we can reach some other variable literal $A_j, 1 \leq j \leq k, j \neq i$ with at most k links. For instance, considering the clause $P(x) \vee !Q(x, y) \vee R(y, z)$, *R(y,z)* can be reached from *P(x)* through two links: *link(P(x), Q(x,y))* by the argument x and *link(Q(x,y), R(y,z))* by the argument y. This implies that to find information related to the target variable literal L_{QP} of the target predicate QP, we only need to consider the subset of variable literals appearing in the set SVC of v-chains starting from L_{QP}, which is much smaller than the complete set SL, especially when the database is large, therefore the size of boolean table is reduced. We continue Example 17 to illustrate this remark.

Example 18 *Les us consider now the variable literal* advisedBy(E, B) *with* k = 2. *Every v-chain of* 2 *variable literals starting from* advisedBy(E, B) *contains the variable literal such that one of its variables is* E *or* B. *Therefore, the set of variable literals in the set of v-chains starting from* advisedBy(E, B) *with* k = 2 *is:*

SVC={ advisedBy(E, B), advisedBy(A, B), professor(B), publication(C, B), publication(D, B)}.

In this case, the set SVC *has only* 5 *elements while* SL *has* 8 *elements.*

Each column of BT corresponds to a variable literal in the set SVC. Each row of BT corresponds to a ground atom of QP. Values on each row of BT are next, heuristically filled. We remains that the value at row r (corresponding to the ground atom q_r) column c (corresponding to the variable literal $vl_c \in SVC$) is true if there exist at least a v-chain $vc \in SL$ containing the variable literal vl_c, a g-chain gc_r starting from the ground atom q_r, and a variabilization of gc_r such that $vc \subseteq var(gc_r)$. Checking this property by considering all g-chains starting from q_r is too expensive because it has to involve an exhaustive search in the database. We overcome this obstacle by inversely, heuristically finding v-chains in SL, each v-chain is then used to guide the search in database. Given a v-chain, the search in the database can be performed much faster because it already knows the order of predicates and positions of shared constants between two consecutive predicates in that v-chain; the search space is only a part of the whole database. In addition, the set SL was built as small as possible, we can expect that there are not so many v-chains in SL starting from a target variable literal L_QP. Algorithm 2 describes steps for building a boolean table starting from a target variable literal L_{QP}.

Example 19 *Let us continue Example 16 by starting from the variable literal* advisedBy(A, B). *From this variable literal, the algorithm can reach any element in the set:*

Algorithm 2: Build a Boolean Table (DB, SL, L_{QP})

Input : *DB:* a training database;
SL: a set of variable literals;
L_{QP}: a target variable literal
Output: *BT:* a boolean table

```
// Initialize
Initialization BT = ∅;
// Find a set of v-chains starting from L_QP
SVC ⟵ Find every v-chain(L_QP) in SL;
// Fill values of the boolean table BT
for each true/false ground atom qga of QP do
    fillchar(OneRowOfTable, 0);
    for each g-chain gc = g-chain(qga) do
        if ∃ a v-chain vc ∈ SVC s.t. vc ⊆ var(gc) then
            OneRowOfTable[L] = 1 for each variable literal L in vc;
        end
    end
    BT ⟵ Append OneRowOfTable;
end
Return(BT);
```

{student(A), publication(C, A), publication(D, A), professor(B), publication(C, B), publication(D, B), advisedBy(E, B)}.

In this case, it is the whole set SL*, but as mentioned above, this set could be much smaller when the database is large. In the boolean table* BT*, each column corresponds to an element in this set and each row corresponds to a true/false ground atom of the target predicate* advisedBy*. Table 4.1 shows several rows of this table in our example starting from the target variable literal* advisedBy(A, B)*. Let us consider, for example, the row corresponding to the false ground atom* advisedBy(Betty, Bob)*. There exists a g-chain starting from this ground atom:* {advisedBy(Betty, Bob), student(Betty)} *that satisfies the v-chain* {advisedBy(A, B), student(A)}*, and a g-chain* {advisedBy(Betty, Bob), publication(T3, Betty)} *that satisfies the two v-chains* {advisedBy(A, B), publication(C, A)} *and* {advisedBy(A, B), publication(D, A)}*. The values at columns corresponding to variable literals* student(A), publication(C, A), publication(D, A) *are thus set to* true*. The others are set to* false*.*

4.2.3 Structure of HGSM

We describe in this section our algorithm HGSM for generative MLN Structure learning [Dinh *et al.* 2010b]. The idea is to apply a process of propositionalization for every predicate in the domain. It means that, as described in Section 4.3.4, for every target predicate, a set SL of variable literals is first generated then a boolean table is created for each corresponding target variable literal. This boolean table is next used to find dependent relations between variable literals from which to form template clauses. Candidate clauses are created from template clauses. This process is repeated by every target predi-

Ground atoms	advisedBy (A, B)	student (A)	publication (C, A)	publication (D, A)	professor (B)	publication (C, B)	publication (D, B)	advisedBy (E, B)
advisedBy(Bart, Ada)	1	1	1	1	1	1	1	0
advisedBy(Betty, Alan)	1	1	1	1	1	1	1	1
advisedBy(Bob, Alan)	1	0	0	0	1	1	1	1
advisedBy(Betty, Bob)	0	1	1	1	0	0	0	0
advisedBy(Alan, Bart)	0	0	1	1	0	1	1	0
...	...	...	...	...	...	...	...	...

Table 4.1: Example of several rows in a boolean table of HGSM

cate thus giving a set of candidate clauses for all predicates in the domain. These clauses are used to learn the final MLN.

Algorithm 3 sketches out the global structure of HGSM. The algorithm tries to find existing clauses considering predicates in the domain in turn (*line 4*). For a given predicate QP, it correspondingly builds a set SL of variable literals (*line 5*), then it forms template clauses from several subsets of SL (*line 6 - 11*), each of them containing at least an occurrence of the target predicate QP. To build the set SL of variable literals, HGSM constructs the largest possible set of connected ground atoms in the database starting from any true ground atom of QP, then heuristically variabilizes them. For each literal $L_{QP} \in SL$ (*line 6*), HGSM generates a set of template clauses from which it extracts a set of relevant candidate clauses to add into the final MLN. A template clause is built from the variable literal L_{QP} and its *neighbors* (i.e. its dependent variable literals in the Markov Blanket). Once every predicate has been considered, we get a set STC of template clauses. From each template clause, HGSM generates all possible candidate clauses by flipping the sign of literals and then keeps the best one according to a given measure (i.e. WPLL). Having considered every template clause, we get a set composed of the "most interesting clauses", a subset of which will form the final MLN (*line 13*).

Algorithm 3: HGSM(*DB*, *MLN*, *maxLength*)

Input : *DB:* a training database;
MLN: an initial (empty) Markov Logic Network;
maxLength: a positive integer;
Output: *MLN:* A final learned Markov Logic Network;

```
   // Initialization
1  A set of template clauses STC = ∅;
2  A set of possible variable literal SL = ∅;
3  A boolean table BT is empty;
   // Creating template clauses
4  for each predicate QP do
5  |  SL ⟵ Generate heuristically a set of possible variable literals(DB, QP);
6  |  for each variable literal L_QP ∈ SL do
7  |  |  BT ⟵ Build a boolean table(DB, SL, L_QP);
8  |  |  MB(L_QP) ⟵ Find the Markov blanket (BT, L_QP);
9  |  |  TC ⟵ Create template clauses(L_QP, MB(L_QP), maxLength);
10 |  |  STC = STC ∪ TC;
11 |  end
12 end
   // Learn the final Markov Logic Network
13 Create the final MLN (DB, MLN, STC, modeClause);
14 Return(MLN);
```

We must emphasize that our approach is, at a first glance, somewhat similar to the principle underlying BUSL [Mihalkova & Mooney 2007]. In fact, HGSM differs from BUSL deeply in all steps of propositionalization and learning the final MLN. We discuss these differences in Section 4.4.

We next detail how a set of template clauses is composed in Subsection 4.2.3.1. In Subsection 4.2.3.2 we present how the set STC of template clauses can be used to learn the final MLN.

4.2.3.1 Building a Set of Template Clauses

A set of template clauses is built using the set SL of variable literals and the set of boolean tables. Let us remind that, in a MLN, if two variable literals occur in a same clause then they are conditionally dependent to each other and this dependency relation is described by an edge in the template graph that corresponds to this MLN. Clauses can then be searched within this template graph in order to reduce the search-space. As a consequence, we aim at finding all possible incident edges or, in other words, we are looking for the Markov blanket of each target variable literal L_{QP}. This problem can be managed using two broad classes of algorithms for learning the structure of undirected graphical models: *score-based* and *independence-based* [Bromberg *et al.* 2006, Spirtes *et al.* 2001]. Score-based approaches conduct a search in the space of structures (of size exponential in the number of variables in the domain) in an attempt to discover a model structure that maximizes a score. They are more robust for smaller datasets because of the smaller space of structures. Independence-based algorithms are based on the fact that a graphical model implies that a set of conditional independences exist in the distribution of the domain, and hence in the dataset provided as input to the algorithm. They work by conducting a set of conditional independence tests on data, successively restricting the number of possible structures consistent with the results of those tests to a single one (if possible), and inferring that structure as the only possible one. It has the advantage of requiring no search and of performing well with large datasets [Spirtes *et al.* 2001]. For these reasons, we propose, as in [Mihalkova & Mooney 2007], to use the independence-based Grow-Shrink Markov Network algorithm (GSMN) [Bromberg *et al.* 2006].

Originally, GSMN is used to learn the structure of a Markov network. Given as input a dataset and a set of variables V, GSMN returns the sets of nodes (variables) B^X adjacent to each variable $X \in V$, which completely determine the structure of the Markov network. To find the set B^X for a variable $X \in V$, it carries out successively three phases: *a grow phase*, *a shrink phase* and *a collaboration phase*. In the grow phase, it attempts to add each variable Y to the current set of hypothesized neighbors of X, contained in a set S. At the end of the grow phase, some of the variables in S, called false positive, might not be true neighbors of X in the underlying Markov Network, thus it uses the shrink phase to remove each false positive Y' by testing for independence with X conditioned on $S \setminus \{Y'\}$. If Y' is found independent of X, it cannot be a true neighbor and GSMN removes it from S. After the neighbors of each X has been produced in B^X, GSMN executes a collaboration phase. During this phase, the algorithm adds X to B^Y for every node Y that is in B^X, because in undirected graphs, X is adjacent to Y if and only if Y is adjacent to X. To determine whether two variables are conditionally independent or not, GSMN uses the Pearson's conditional independence chi-square (χ^2) test (see in [Agresti 2002] for details of its computation).

In HGSM, we can consider the boolean table BT and the set of variable literals corresponding to columns of BT (which is always a subset of SL connected to the target variable literal) respectively as inputs for the GSMN algorithm. However, instead of find-

Algorithm 4: Create template clauses (L_{QP}, $MB(L_{QP})$, $maxLength$)

Input : L_{QP}: a target variable literal;
$MB(L_{QP})$: the Markov blanket of L_{QP};
$maxLength$: a positive integer;
Output: TC: a set of template clauses

```
// Initialize
```
Initialization $TC = \emptyset$;
```
// Generate template clauses
```
for $j = 2$ **to** $maxLength$ **do**
  for *each subset* $S \subseteq MB(L_{QP})$ *of* $(j-1)$*-elements* **do**
    $c = CreateTempClause(L_{QP}, S)$;
    $TC \longleftarrow c$ if c is a connected clause;
  end
end
Return(TC);

ing the Markov blankets for every variable of the input set, we apply GSMN to find only the Markov blanket $MB(L_{QP})$ of the target variable literal L_{QP} because the boolean table BT is designed to contain mostly information related to ground atoms of the target predicate. Every other target variable literals (and further every other target predicate) will be considered in turn thus at each step it is sufficient to find only and uniquely the Markov blanket of L_{QP}.

Example 20 *By applying the* GSMN *algorithm on the boolean table given in Example 19 we get the Markov blanket of the target variable literal* advisedBy(A, B)*:*

$$\text{MB(advisedBy(A, B))} = \{\text{student(A), professor(B), advisedBy(E, B)}\}.$$

Having got the Markov blanket $MB(L_{QP})$, our algorithm composes a set TC of template clauses. We recall that a template clause is simply a disjunction of positive literals. A set of template clauses is created from the target variable literal L_{QP} and its neighbors, i.e. the variable literals forming its Markov blanket $MB(L_{QP})$. Each template clause is built from L_{QP} and a subset $S \subseteq 2^{MB(L_{QP})}$ such that $S \cup L_{QP}$ is a connected clause. Algorithm 4 illustrates the steps for this task.

Example 21 *We finish this subsection by illustrating the set of template clauses created from the target variable literal* advisedBy(A, B) *and its neighbors in Example 20. Among template clauses, we found:*

$$advisedBy(A, B) \vee student(A),$$
$$advisedBy(A, B) \vee professor(B),$$
$$advisedBy(A, B) \vee advisedBy(E, B),$$
$$advisedBy(A, B) \vee student(A) \vee professor(B),$$
$$\ldots$$

4.2.3.2 Learning the Final MLN

Clauses are created from each template clause by flipping the sign of its variable literals. HGSM can be restricted to consider only Horn clauses. Clauses created from the same template clause are evaluated in turn. To evaluate a clause c, we learn the weights, then compute the measure (i.e. WPLL) of the *temporary MLN* consisting of the unit MLN (a clause is simply a variable literal) plus c. We define the *gain* of c as the difference between the measure of this temporary MLN and the one of the unit MLN and the *weight* of c is its weight in the *temporary MLN*. If this gain is positive and the weight is greater than a given threshold *minWeight*, c becomes an *available* candidate clause. Because every available candidate clause created from a template clause composes similar cliques in the network, for each connected template clause, HGSM keeps only the clause with the maximum measure as a candidate clause.

Candidate clauses are sorted by decreasing gain and our algorithm adds them in turn in this order to the current MLN. The weights of the resulting MLN are learned and this latter is scored using the chosen measure (i.e. WPLL). Each clause that improves the measure is kept in the current MLN, otherwise it is discarded.

Finally, as adding a clause into the MLN might drop down the weight of clauses added before, once all clauses have been considered, HGSM tries to prune some clauses of the MLN. This pruning is based on their weight: a clause with the weight less than the *minWeight* is discarded from the MLN if its removal also increases the measure.

This process is given in Algorithm 5. In [Richardson & Domingos 2006, Kok & Domingos 2005], the authors showed that it is useful to start learning generatively a MLN structure by adding all *unit clauses* (clauses of a single variable literal). We therefore provide them as the first step in process of learning the final MLN (*line 2-4*). For each template clause $Tc \in STC$ (*line 5*), HGSM generates a set *Clauses* of clauses by flipping the sign of its variable literals. Depending on the goal (or on time constraints), HGSM can restrict the search to *Horn* clauses or consider all *arbitrary* clauses (2^n possible clauses). For each clause $c_i \in Clauses$ (*line 7*), HGSM first learns weights for a MLN composed of all unit clauses *plus* c_i (*line 8*), and then computes the *WPLL* of this MLN (*line 9*). Because every clause created from a template clause composes the same clique in the network, HGSM chooses only one clause in *Clauses* that leads to the best *WPLL* and adds it into the set *CanClauses* of candidate clauses (*line 11*). As proposed in [Richardson & Domingos 2006, Kok & Domingos 2005], we eliminate several clauses from *CanClauses* by keeping only clauses that improve the *WPLL* measures and have a weight greater than a given threshold; this allows time saving during the next steps (*line 13*). A set of best candidate clauses is stored into *BestClauses*, which is used to learn the final MLN.

4.2.4 Evaluating HGSM

HGSM has been implemented on top of the Alchemy package [Kok *et al.* 2009]. We used the APIs implementation of L-BFGS to learn weights. We performed experiments to answer the following questions:

1. Does HGSM outperform BUSL?

Algorithm 5: Create the final MLN (*DB*, *MLN*, *STC*, *modeClause*)

Input : *DB*: a training database;
MLN: an initial Markov Logic Network;
STC: a set of template clauses;
modeClause: an integer for choosing Horn or arbitrary clauses;
Output: *MLN*: the final Markov Logic Network;

```
// Initialization
```
A set of candidate clause $CanClauses = \emptyset$;
```
// Add unit clauses into the final MLN
```
Add Unit Clauses Into the MLN;
LearnWeightsWPLL(*MLN*, *DB*);
BestScore = *measureWPLL(MLN, DB)*;
```
// Evaluate candidate clauses
```
for *each template clause* $Tc \in STC$ **do**
    Clauses = *CreateClauses(Tc, modeClause)*;
    for *each clause* $c_i \in Clauses$ **do**
        LearnWeightsWPLL(c_i, *MLN*, *DB*);
        $c_i.gain$ = measureWPLL(c_i, *MLN*, *DB*);
    end
    $CanClauses = CanClauses \cup c$ where $c.gain = \max_{c_i \in Clauses}\{c_i.gain\}$;
end
```
// Add candidate clauses into the final MLN
```
Choose *BestClauses* from *CanClauses*;
for *each clause* $c \in BestClauses$ **do**
    LearnWeightsWPLL(*c, MLN, DB*);
    NewScore = measureWPLL(*c, MLN, DB*);
    if *NewScore* > *BestScore* **then**
        Add clause *c* into MLN;
        BestScore = *NewScore*;
    end
end
Prune Clauses out of the MLN (*MLN*, *DB*);
Return(*MLN*);

Datasets	IMDB	UW-CSE	CORA
Types	4	9	5
Constants	316	1323	3079
Predicates	10	15	10
True atoms	1540	2673	70367

Table 4.2: Details of the IMDB, UW-CSE and CORA datasets

2. Does HGSM perform better than the state-of-the-art generative systems for MLN structure learning?

3. How does HGSM perform considering only Horn clauses?

4.2.4.1 Systems, Datasets and Methodology

To answer question 1, we directly compare HGSM to BUSL. To answer question 2, we compare HGSM to the state-of-the-art generative system ILS (Iterated Local Search) [Biba *et al.* 2008b]. Finally, for question 3, we introduce *HGSM-H*, a restricted version of HGSM that limits ti Horn clauses.

We used three publicly-available datasets: IMDB, UW-CSE and CORA[1], which have been used to evaluate most of the former algorithms, such as MSL [Kok & Domingos 2005], BUSL [Mihalkova & Mooney 2007], ILS [Biba *et al.* 2008b] and others. Each dataset is splitted into mega-examples, where each megaexample contains a connected group of facts. Mega-examples are independent of each other.

- **IMDB:** This dataset, created by Mihalkova and Mooney (2007) from the IMDB.com database, describes a movie domain. It contains predicates describing movies, actors, directors, and their relationships (e.g, *Actor(person)*, *WorkedUnder(person, movie)*, etc.). The genres of each director is based on the genres of the movies he or she directed. The Gender predicate is only used to state the gender of actors.

- **UW-CSE:** This dataset, prepared by Richardson and Domingos (2006), describes an academic department. It lists facts about people in an academic department (i.e. Student, Professor) and their relationships (i.e. AdvisedBy, Publication) and is divided into five independent folds based on five areas of computer science.

- **CORA:** This dataset is a collection of citations to computer science papers, created by Andrew McCallum, and later processed by Singla and Domingos (2006) into five independent folds for the task of predicting which *citations* refer to the same *paper*, given the words in their *author*, *title* and *venue* fields. Predicates include: *SameCitation(cit1, cit2)*, *TitleHasWord(title, word)*, etc.

Details of these datasets are reported in Table 4.2.

We performed experiments through a *5-fold cross-validation*. For each system on each test fold, we performed inference over each test ground atom to compute its probability of being true, using all other ground atoms as evidence. The log of this probability is the

[1]These datasets are publicly-available at the URL `http://alchemy.cs.washington.edu`

		Datasets		
Measure	Systems	IMDB	UW-CSE	CORA
CLL	BUSL	-0.240±0.017	-0.358±0.019	-0.341±0.014
	ILS	-0.210±0.021	-0.180±0.015	-0.131±0.011
	HGSM-H	-0.201±0.021	-0.138±0.017	-0.108±0.013
	HGSM	**-0.183±0.028**	**-0.103±0.020**	**-0.087±0.016**
		Datasets		
Measure	Systems	IMDB	UW-CSE	CORA
AUC-PR	BUSL	0.470±0.07	0.291±0.02	0.435±0.01
	ILS	0.400±0.06	0.257±0.02	0.501±0.02
	HGSM-H	0.531±0.06	0.292±0.02	0.701±0.01
	HGSM	**0.692±0.07**	**0.311±0.01**	**0.762± 0.02**

Table 4.3: CLL, AUC-PR measures

conditional log-likelihood (CLL) of the test ground atom. To evaluate the performance of each system, we measured the average CLL of the test atoms and the area under the precision-recall curve (AUC-PR). The advantage of CLL is that it directly measures the quality of the probability estimates produced. The advantage of AUC is that it is insensitive to a large number of true negatives (i.e., ground atoms that are false and predicted to be false). The precision-recall curve for a predicate is computed by varying the threshold CLL above which a ground atom is predicted to be true. We used the package provided in [Davis & Goadrich 2006] to compute the AUC-PR measure.

Concerning parameter setting, we used the same settings for ILS and BUSL as in [Biba *et al.* 2008b] and [Mihalkova & Mooney 2007] respectively. In order to limit the search space for all systems, we set the maximum number of variable literals per clause to 5, as in [Biba *et al.* 2008b].

Having learned the MLN, we performed inference for every predicate on the test folds for all datasets, using the recent build-in inference algorithm *Lazy-MC-SAT* [Poon & Domingos 2006] in Alchemy [Kok *et al.* 2009]. Lazy-MC-SAT produces probabilities for every grounding of the query predicate on the test fold.

For each testing dataset, we ran each system on a Dual-core AMD 2.4 GHz CPU - 4GB RAM machine.

4.2.4.2 Results

Table 4.3 reports the CLL and AUC-PR measures for all approaches on all datasets. These are the average of CLLs and AUC-PRs over all test folds. It must be noted that, while we used the same parameter settings, our results do slightly differ from the ones in [Biba *et al.* 2008b]. This comes from the fact that we conducted inference using *Lazy-MC-SAT* instead of *MC-SAT*. Table 4.4 gives the average number of candidate clauses *(NOC)*, the finally number of learned clauses *(FC)* and the training time (in minutes) *(TT)* over all train folds.

First, let us compare HGSM respectively to BUSL and ILS. HGSM outperforms BUSL

Datasets →	IMDB			UW-CSE			CORA		
Systems	NOC	FC	TT	NOC	FC	TT	NOC	FC	TT
BUSL	337	17.60	24.12	362.00	**37.00**	360.85	188.60	17.75	3412.08
ILS	983	15.40	19.38	**3104.20**	23.40	191.84	**1755.00**	18.25	1730.08
HGSM-H	346	13.20	19.41	570.20	23.20	281.43	114.60	17.00	1963.74
HGSM	551	15.40	23.87	1520.00	23.60	**521.40**	143.40	**20.25**	**3849.52**

Table 4.4: Number of clauses and runtimes (minutes)

and ILS in terms of both CLL and AUC-PR. Concerning the CLL measure, HGSM-A increases it approximately *23%* for IMDB, *71%* for UW-CSE, *74%* for CORA compared to BUSL and respectively *12%*, *42%* and *33%* compared to ILS. Concerning the AUC-PR measure, HGSM-A increases it approximately *47%* for IMDB, *6%* for UW-CSE, *75%* for CORA compared to BUSL and respectively *73%*, *21%* and *52%* compared to ILS. We would like to emphasize that HGSM dominates them not only on average values, but also for each test fold of each dataset. However, HGSM is slower than BUSL while ILS appears to be the fastest system. We can answer to question 1 that HGSM performs better than BUSL in the sense of CLL and AUC-PR. Based on these results, we believe in the domination of our method compared to the state-of-the-art generative structure learning algorithms for MLNs, especially for the task of classification. This answers question 2. Since CLL determines the quality of the probability predictions output by the algorithm, HGSM outperforms ILS and BUSL in the sense of the ability to predict correctly the query predicates given evidence. Since AUC-PR is insensitive to the large number of true negatives (i.e., ground atoms are false and predicted to be false), HGSM enhances the ability to predict the few positives in data.

Last, let us compare all systems together. HGSM is the best system in terms of CLL and AUC-PR, and ILS is the best one in terms of runtime. However, in theory, all of these algorithms involve the L-BFGS algorithm to set weights for clauses, hence the times all depend on the performance of this weight learning algorithm. In practice, as revealed in [Shavlik & Natarajan 2009], the presence of a challenging clause like $AdvisedBy(s,p) \wedge AdvisedBy(s,q) \rightarrow SamePerson(p,q)$ for the *UW-CSE* dataset will have a great impact on optimization as well as on inference. Runtime therefore depends mostly on the number of candidate clauses and on the occurrence of literals together in each clause. From practice we also verify that the time used to find candidate clauses is much less than the time used to learn and to infer. From Table 4.4 we can see that, although BUSL and HGSM evaluate fewer candidates than ILS, they are both slower than ILS. This is due to the fact that BUSL and HGSM change the MLN completely at each step, calculating the WPLL measure becoming thus very expensive. In ILS this does not happen because at each step L-BFGS is initialized with the current weights (and zero weight for a new clause) and converges in a few iterations [Biba *et al.* 2008a]. Regarding HGSM and BUSL, for the CORA dataset, despite HGSM evaluates fewer candidate clauses than BUSL, it gives better CLL and AUC-PR values but unfortunately runs slower. This implies that the set of candidate clauses created by HGSM is better than the one created by BUSL, furthermore our method to create candidate clauses is better than the one in BUSL. This issue also urges us to apply a method like it is done in ILS to accelerate HGSM. It is very interesting that HGSM considering only Horn clauses (HGSM-H) takes much less time than it does with arbitrary clauses while it also outperforms both ILS and BUSL in terms of both CLL and AUC-PR measures. HGSM-H gets only a little loss in the sense of CLL and AUC-PR compared to HGSM with arbitrary clauses. From the logic point of view, a Horn-clause MLN might integrate easier in further processing than a MLN based on arbitrary-clauses. These results give us belief in restricting our system to solve with *Horn clauses* in a more acceptable runtime. This answers question 3.

4.2.5 Structure of HDSM

The HGSM algorithm presented in the previous section is designed for a generative learning purpose which attempts to learn a set of clauses, that can then be used to predict the truth value of all predicates given an arbitrary set of evidence. This kind of learning is useful when it is not known ahead of time how the model will be used, so that the learned model needs to capture as many aspects of a domain as possible. However, in many learning problems, there is a specific target predicate to learn, the values of its ground atoms being unknown at test-time, assuming that values of the remaining predicates in the domain will be given. In this case, discriminative learning is more appropriate. More detailed studies of the relative advantages of these two frameworks of learning are available in [Liang & Jordan 2008].

Most traditional ILP methods focus on discriminative learning [Dzeroski 2007]; however, they do not address the issue of uncertainty. Several discriminative methods have been developed for parameter learning in MLNs [Singla & Domingos 2005, Lowd & Domingos 2007]; however, they do not address structure learning. To the best of our knowledge, there only exists two systems for discriminative MLN structure learning that integrate both a clause searching step and a weight learning step. The first one uses ALEPH [Srinivasan 2007] system to learn a large set of potential clauses, then learns the weights and prunes useless clauses [Huynh & Mooney 2008]. The second method, called ILS-DSL (Iterated Local Search - Discriminative Structure Learning), chooses the structure by maximizing CLL and sets the parameters using L-BFGS to maximize PLL [Biba *et al.* 2008a]. These systems were described in more details in Chapter 3 of this dissertation.

Due to the lack of algorithms for discriminative MLN structure learning, this research direction needs to be further investigated. In this section, we present an adaptation of the techniques implemented in HGSM for the task of discriminative MLN structure learning. We call this version HDSM [Dinh *et al.* 2010c], which stands for Heuristic Discriminative Structure learning for MLNs.

4.2.5.1 Structure of HDSM

We apply basically the structure of HGSM to the task of learning MLN structure for a specific query predicate in the domain. However, the generative measure (i.e. WPLL) used in HGSM to choose clauses is not really effective, compared to a discriminative measure such as CLL, when we want to find clauses that maximize the score of only a specific predicate assuming that the remainders are known. We therefore use a discriminative measure (i.e. CLL) to choose clauses in this discriminative version. Besides, in [Biba *et al.* 2008a], the authors have shown that using the generative weight learning algorithm L-BFGS for discriminative MLN structure learning can lead to acceptable results in terms of runtime and of the CLL and AUC-PR measures, compared to a discriminative weight learning method such as P-SCG [Lowd & Domingos 2007]. We decided to keep using L-BFGS to learn the weights in order to take these advantages.

We have as input a training database DB composed of true/false ground atoms and a query predicate QP (several query predicates can be given). A set of clauses (i.e. an initial MLN) defining some background knowledge may also be given. We aim at learning

a MLN that correctly discriminates between true and false groundings of *QP*. We describe in this subsection the structure of *HDSM*.

Algorithm 6: HDSM(*DB*, *MLN*, *QP*, *maxLength*)

Input : *DB:* a training database;
MLN: an initial (empty) Markov Logic Network;
QP: a query predicate;
maxLength: a positive Integer;
Output: *MLN:* a final learned Markov Logic Network

```
// Initialization
```
1 A set of template clauses $STC = \emptyset$;
2 A set of possible variable literal $SL = \emptyset$;

```
// Creating template clauses
```
3 $SL \longleftarrow$ Generate heuristically a set of possible variable literals(DB, QP);
4 **for** *each variable literal* $L_{QP} \in SL$ **do**
5 $\quad BT \longleftarrow$ Build a boolean table(DB, SL, L_{QP});
6 $\quad MB(L_{QP}) \longleftarrow$ Find the Markov blanket (BT, L_{QP});
7 $\quad TC \longleftarrow$ Create template clauses(L_{QP}, $MB(L_{QP})$, *maxLength*);
8 $\quad STC = STC \cup TC$;
9 **end**

```
// Learn the final Markov Logic Network
```
10 Learn the final MLN (DB, MLN, STC, *modeClause*);
11 Return(*MLN*);

We sketch the global structure of HDSM in Algorithm 6. HDSM tries to find existing clauses containing query predicate *QP* by building a set *SL* of variable literals, from which to generate a set of template clauses, each of them containing at least one occurrence of *QP*. To build the set *SL* of variable literals, HDSM constructs the largest possible set of connected ground atoms corresponding to every true ground atom of *QP*, then heuristically variabilizes them (*line 2*). For each literal $L_{QP} \in SL$, HDSM generates a set of template clauses from which it extracts a set of relevant candidate clauses (*line 3-8*). A template clause is built from the variable literal L_{QP} and its *neighbors*. As in HGSM, we have to transform the information in the database into a boolean table (*line 4*) corresponding to each $L_{QP} \in SL$ in order to be able to apply GSMN [Bromberg *et al.* 2006] to find the Markov blanket of L_{QP} (*line 5*). Once every variable literals of the query predicate have been considered we get a set of template clauses *STC*. Candidate clauses are created from this set and provided to learn the final MLN (*line 9*).

We emphasize that HDSM does perform in the same manner as HGSM but for only a query predicate. To be more suitable to discriminative learning, it uses the discriminative measure (i.e. CLL) instead of the generative one (i.e. WPLL) to choose clauses. Computing this discriminative measure, however, takes more time as it requires in addition an inference step to compute the empirical expectation.

4.2.6 Evaluating HDSM

HDSM is implemented over the Alchemy package [Kok *et al.* 2009]. We performed experiments to answer the following questions:

1. How does HDSM carry out with Horn clauses instead of arbitrary clauses?
2. Is HDSM better than the state-of-the-art algorithms for discriminative MLN structure learning?
3. Is HDSM better than HGSM for the task of discriminative learning?
4. Between two propositionalization methods respectively implemented in BUSL and HDSM, which one is the more suitable for the task of discriminative MLN structure learning?

While the first three questions are intended to evaluate how HDSM performs compared to both state-of-the-art generative and discriminative systems, the last question is focused on comparing to the BUSL algorithm. In the previous section, we have mentioned that HGSM (and now also HDSM) relies on similar principles to the one underlying BUSL but deeply differs in all three steps; performing a propositionalization, building candidate clauses and learning the final MLN, and we shown in the experiments that HGSM produces better results (for both CLL and AUC-PR) than BUSL does on the three standard datasets we used. We, however, were not able to show which step affects mostly the performance of HGSM compared to BUSL because of the setting for all predicates and the implemented codes of these two generative systems. In this experiment, we want to take the advantage of discriminative learning for only a single query predicate in order to figure out which method of propositionnalization (in BUSL or in HDSM (HGSM)) fits better for the task of MLN structure learning.

4.2.6.1 Systems, Datasets and Methodology

To answer question 1, we ran HDSM twice to perform respectively with Horn clauses (HDSM-H) and with arbitrary clauses (HDSM-A). To answer question 2, we compared HDSM to the state-of-the-art discriminative system ISL-DSL [Biba *et al.* 2008a]. To answer question 3 we compared directly HDSM to HGSM [Dinh *et al.* 2010b]. To answer question 4, we implemented HDSM using L-BFGS to set weights and the WPLL measure to choose clauses (HDSM-W). Moreover, HDSM-W creates template clauses from cliques and considers, as in BUSL, all possible clauses from a template clause. We configured BUSL to run only for single predicates. In this case, BUSL and HDSM-W are different only in the step of building boolean tables, hence we can compare the interest of the boolean tables created by them.

We also used the three datasets IMDB, UW-CSE and CORA and performed a *5-fold cross-validation*, as described in Subsection 4.2.4.1. For IMDB, we predicted the probability of pairs of person occurring in the relation *WorkedUnder*. For UW-CSE we have chosen the discriminative task of predicting who is the *advisor* of who. For CORA, we learned four discriminative MLNs, respectively according to the four predicates: *sameBib, sameTitle, sameAuthor, sameVenue*. These predicates have often been used in previous

studies of MLN discriminative learning [Singla & Domingos 2005, Biba *et al.* 2008a]. Both CLL and AUC-PR are also used as evaluation measures as described in Subsection 4.2.4.1.

Parameters for ILS-DSL and ILS were respectively set as in [Biba *et al.* 2008a], [Biba *et al.* 2008a]. We set the maximum number of literals per clause to *5* for all the systems as in [Biba *et al.* 2008a]. We performed inference on the learned MLN using the *Lazy-MC-SAT* algorithm. We ran our tests on a Dual-core AMD 2.4 GHz CPU - 4GB RAM machine.

4.2.6.2 Results

Table 4.5 presents the average CLL and AUC-PR measures for the learning predicates over test folds for all the algorithms estimating on the three datasets. It must be noted that, while we used the same parameter setting, our results do slightly differ from the ones in [Biba *et al.* 2008a]. This comes from the fact that we performed inference using Lazy-MC-SAT instead of MC-SAT, and that in the training process, ILS-DSL used only one of the training folds to compute the CLL measure[Biba *et al.* 2008a]. Table 4.6 gives the average runtime over all train folds for IMDB and UW-CSE, and over four learning predicates for CORA.

First, we consider HDSM respectively with Horn clauses (HDSM-H) and with arbitrary clauses (HDSM-A). We can see that HDSM-A performs only better than HDSM-H for some predicates in terms of both CLL and AUC-PR. HDSM-A is even worse than HDSM-H for the predicate *advisedBy* (UW-CSE). This sounds a bit strange as HDSM-A and HDSM-H share the same set of template clauses and HDSM-A considers more clauses than HDSM-H does. We can explain this issue as follows: in our method, for each template clause the algorithm keeps at most one candidate clause (the one with the highest score), thus the one kept by HDSM-A has a score greater than or equal to the score of the one kept by HDSM-H. The set of candidate clauses of HDSM-A is hence different from the one of HDSM-H. The order of template clauses and the order of candidate clauses affect the final learned MLN. When candidate clauses are considered in turn, the set of candidate clauses of HDSM-A is not always better than the one of HDSM-H because the algorithm has to learn the weights again for each considered candidate clause, and it may affect the weights of clauses added before into the MLN. It is interesting that HDSM performs much faster with Horn clauses than with arbitrary clauses while it gets a little loss in the CLL and AUC-PR measures. This is the answer to question 1.

Second, we compare HDSM to ILS-DSL. Both versions of HDSM perform better than ILS-DSL in terms of both CLL and AUC-PR. Since CLL determines the quality of the probability predictions output by the algorithms, in our experiments, our algorithm outperforms this state-of-the-art discriminative algorithm in the sense of its ability to predict correctly the query predicates given evidence. Since AUC-PR is useful to predict the few positives in data, we can conclude that HDSM enhances the ability of predicting the few positives in data. Question 2 is answered.

Third, we compare HDSM to HGSM. HDSM produces really better values in both CLL and AUC-PR for all predicates for all datasets. We believe that our method, designed for classification, behaves better in such tasks than generative structure learning algorithms for MLNs.

Concerning runtimes, ILS-DSL is the fastest system, then are HDSM-H, BUSL and

Algorithms →		HGSM		BUSL		HDSM-W	
Datasets	Predicates	CLL	AUC-PR	CLL	AUC-PR	CLL	AUC-PR
IMDB	WorkedUnder	-0.035±0.006	0.315	-0.225±0.011	0.129	-0.035±0.007	0.315
UW-CSE	AdvisedBy	-0.029±0.005	0.229	-0.044±0.006	0.204	-0.029±0.008	0.215
CORA	SameBib	-0.150±0.005	0.390	-0.325±0.009	0.229	-0.154±0.011	0.394
	SameTitle	-0.121±0.009	0.415	-0.284±0.009	0.418	-0.127±0.006	0.411
	SameAuthor	-0.169±0.007	0.415	-0.356±0.008	0.347	-0.176±0.007	0.410
	SameVenue	-0.121±0.006	0.357	-0.383±0.010	0.276	-0.121±0.007	0.327
Algorithms →		ILS-DSL		HDSM-H		HDSM-A	
Datasets	Predicates	CLL	AUC-PR	CLL	AUC-PR	CLL	AUC-PR
IMDB	WorkedUnder	-0.029±0.007	0.311	**-0.028±0.006**	0.323	**-0.028±0.008**	**0.325**
UW-CSE	AdvisedBy	-0.028±0.006	0.194	**-0.023±0.004**	**0.231**	-0.025±0.010	0.230
CORA	SameBib	-0.141±0.009	0.461	**-0.140±0.008**	**0.480**	**-0.140±0.011**	**0.480**
	SameTitle	-0.134±0.010	0.427	-0.110±0.007	**0.502**	**-0.108±0.010**	0.498
	SameAuthor	-0.188±0.008	0.560	-0.155±0.008	0.581	**-0.146±0.009**	**0.594**
	SameVenue	-0.132±0.009	0.297	**-0.115±0.009**	0.338	**-0.115±0.011**	**0.342**

Table 4.5: CLL, AUC-PR measures

Algorithms →	HGSM	ILS-DSL	HDSM-H	HDSM-A	HDSM-W	BUSL
IMDB	0.42	0.49	0.56	1.09	0.42	0.38
UW-CSE	8.62	4.30	7.00	9.30	8.56	8.05
CORA	39.46	30.41	33.71	50.15	38.24	34.52

Table 4.6: Runtimes(hour)

HDSM-A. The runtime (of each system) includes the time to find candidate clauses, the time to learn weights for each candidate clause and the time to choose clauses for the final MLN. In practice we verify that the time spent to find candidate clauses is much less important than the time spent to learn weights and the time to compute CLL when performing inference. To set weights for clauses, all systems involve the L-BFGS algorithm, and thus the runtime depends on the performance of this weight learning algorithm. BUSL and HGSM change the MLN completely at each step, thus calculating WPLL (required to learn weights by L-BFGS) becomes very expensive. In ILS-DSL, this does not happen because at each step L-BFGS is initialized with the current weights (and zero weight for a new clause) and it converges in a few iterations [Biba *et al.* 2008b], [Biba *et al.* 2008a]. ILS-DSL also uses some tactics to ignore a candidate clause whenever it consumes time more than a given threshold for inferring [Biba *et al.* 2008a]. We plan to accelerate our systems to a more reasonable time as it has been done in ILS-DSL, especially by finding an efficient solution to filter candidate clauses. It is very interesting that HDSM-H takes much less time than HDSM does while it gets only a little loss in the sense of CLL and AUC-PR. We also would like to notice that the MLN produced by HDSM-H might be an advantage, from the logical point of view, since a set of Horn clauses can be more easily interpreted.

4.3 The DMSP Algorithm

We have developed in the previous section the HGSM algorithm for generative MLN structure learning and the HDSM algorithm for discriminative MLN structure learning. Both are implemented on the same structure including three steps: performing a propositionalization to transform information in the relational database into boolean tables, creating template clauses from every query variable literal and its Markov blanket using the GSMN algorithm (Grow-Shrink Markov Network algorithm) [Bromberg *et al.* 2006], and finally learning the final MLN by considering candidate clauses in turn. Our experiments show that they give better results than the state-of-the-art approaches for MLN structure learning while considering less candidate clauses. Especially, the proposed propositionalization method is more suitable for the task of MLN structure learning than the one implemented in BUSL.

Beside these satisfying results, however, HGSM and HDSM spend too much time to produce the final MLN, even if they consider less candidate clauses than ILS or BUSL. It must be noted that time-consumption and performance are affected during all the three steps in our methods:

- **Performing propositionalization:** This step includes two tasks: first forming a

set of variable literals starting from the query predicate and then encoding information in the dataset into boolean tables, each boolean table corresponding to a target variable literal, each column of which corresponding to a variable literal. The task of forming a set of variable literals is based on the idea of separating the dataset in distinct connected groups starting from true ground atoms of the query predicate in the dataset. A heuristic variabilization technique is used in order to generate a set of variable literals such that we can always describe links between ground atoms in a connected group by using only elements in this generated set of variable literals. It means that for every g-chain gc = *g-chain(qga)* starting from a true ground atom *qga* of the query predicate, there always exists a v-chain, vc, of variable literals and a variable replacement such that $var(gc) \subseteq vc$. During the variabilization process, two criteria are checked repeatedly to decide whether a new variable literal needs to be generated. This process, nevertheless, is iterated many times when the dataset is composed of a lot of distinct connected groups. When there are a lot of new variable literals that need to be created, this checking operation, more or less, leads to an exhaustive search in the dataset. In addition, a large number of variable literals increases time-consumption in the two next steps.

- **Creating template clauses:** Having got a boolean table corresponding to each target variable literal, the GSMN algorithm is applied in order to find the Markov blanket of this target variable literal. We recall that a variable Y is in the MB of a target variable X if X and Y are conditionally dependent to each other. Because a real-world dataset is always incomplete and our propositionalization method is a kind of reduction which causes information loss, both of these factors affect results of the GSMN. The set of template clauses hence could miss some good ones containing good candidate clauses.

- **Learning the final MLN:** Candidate clauses are considered in turn to determine if they are added into the final MLN. A candidate is added if it causes an improvement of the measure. It is obvious that, in this method, we have to apply the weight learning process as many times as the number of candidate clauses and the larger a MLN is (in number of clauses) the more time is required for weight learning. Adding a clause also influences the weights of all preceding clauses in the MLN. Therefore the order of clauses added into the MLN is very important to produce the final MLN. Concerning the step of learning the final MLN, we use a built-in state-of-the-art weight learning algorithm in Alchemy (i.e. the L-BFGS). This algorithm also takes times when there is a long inference in the set of clauses and when variable literals have many grounding atoms. This explains why our methods are slower than the others while considering less candidate clauses.

Improving the performance of a weight learning algorithm or designing a new one is a quite different task that is out of the scope of this dissertation. In this Subsection, we will concentrate on a new method for discriminative MLN structure learning. We develop a new propositionalization technique, modify the way to create candidate clauses and change the order of candidate clauses to learn the final MLN while keeping the same structure of HDSM. We call this new method *DMSP* [Dinh *et al.* 2010a], which stands for Discriminative MLN Structure learning based on Propositionalization. As the structure of

DMSP is similar to the one of HDSM, we next introduce several definitions in Subsection 4.3.1 to facilitate our representation then focus on presenting the specific aspects of DMSP.

4.3.1 Definitions

Definition 22 *A link of a g-chain* $gc = \langle g_1, ..., g_k, ... \rangle$ *is an ordered list of links* $link(g_i, g_{i+1})$, $i \geq 1$, *denoted by:*

$$g - link(gc) = \langle link(g_1, g_2)/.../link(g_i, g_{i+1})/... \rangle.$$

Definition 23 *A g-link* $gc = \langle g_1, ..., g_k \rangle$ *is said to be a prefix of a g-link* $gs = \langle s_1, ..., s_n \rangle$, $k \leq n$ *if* $link(g_i, g_{i+1})$ *is equal to* $link(s_i, s_{i+1})$, $\forall i, 0 \leq i < k$.

We also define a *v-link* as a *link* of a *v-chain*. We can see that if there exists a variabilization vc of a g-chain $gc = \langle g_1, ..., g_k \rangle$ such that $var(gc) = vc = \langle v_1, ..., v_k \rangle$ then *g-link(gc)* is equal to *v-link(vc)*.

Example 24 *The g-link of the g-chain* $gc_1 = \langle$P(a, b), R(b, c), S(c)$\rangle$ *is:* g-link(gc_1) = $\langle$\{P R 1 0 \} / \{R S 0 0\}$\rangle$.

Obviously, we can see that the g-link of a g-chain:

$$gc_2 \equiv \text{g-chain(P(a, b))} = \langle \text{P(a, b), R(b, c)} \rangle$$

is g-link(gc_2) = $\langle$\{P R 1 0 \}$\rangle$, *which is a prefix of the g-link of* gc_1.

4.3.2 Propositionalization Method

The two main tasks in our first propositionalization method are the task of generating a set of variable literals corresponding to the query predicate and the task of building a boolean table corresponding to each query variable literal. The number of generated variable literals affects the size of the boolean tables, therefore the performance of the algorithm. In this section, we focus on a new technique to generate the set of variable literals while using the same definition and strategies to create boolean tables as HDSM.

The problem is to generate a set *SL* of variable literals given a dataset *DB* and a query predicate *QP* such that for each g-chain, *g-chain(e)* in *DB*, of a true ground atom e of *QP* in *DB*, there exists a variablilization such that $var(g\text{-}chain(e)) \subseteq SL$. Our first propositionalization method separates *DB* into distinct connected groups of ground atoms from which to variabilize heuristically. The idea underlying in this second method is to variabilize by detecting regularities of ground atoms, based on the observation that relational data usually contains regularities. As a consequence, it is reasonable to expect that many g-chains (starting from several ground atoms) are similar, and could thus be variabilized by a single v-chain *vc* (i.e. v-link(vc) similarly to every g-link of some g-chain). In this case, only a set of variable literals appearing in *vc* has to be stored into the set *SL*. Moreover, if a g-link of a *g-chain(e)* is a prefix of another one which has already been processed, there exists at least a v-chain *vc*, with its variable literals in *SL*, such that $g\text{-}link(g\text{-}chain(e))$ is a prefix of *vc*. This *g-chain(e)* is thus no longer to be considered for variabilizing. The task is now to variabilize such sets of similar g-chains to get a set of v-chains from which to achieve a set *SL* of variable literals.

Let us recall that a g-chain is also a connected clause, hence each *g-chain(e)* starting from e could be variabilized to produce a candidate clause. Unfortunately, there are a lot of candidate clauses like that, and learning the final MLN would be very complex. The set of v-chains achieved by variabilizing sets of similar g-chains could also be used as the set of candidate clauses, but this set remains very large and a v-chain may not be a good candidate, mainly if its variable literals are not statistically dependent. In order to generate less candidate clauses, we aim at using the variabilization process to create a minimum set SL of variable literals. During the process of variabilization when forming v-chains, we try to reuse as many variables and variable literals as possible that have been previously introduced in order to reduce the number of variable literals, and thus to reduce the search space and time-consumption for the next steps. It is noted that this method variabilizes g-chains that are completely different from the first one that variabilizes distinct connected groups of ground atoms.

Algorithm 7: Generating a set of variable literals (*DB*, *QP*)

Input : *DB:* a training database;
QP: a query predicates;
Output: *SL:* a set of variable literals;

```
// Initialization
```
1 A maximum number of used variables $maxVar = 0$;
2 Every constant is not variabilized $mapVar[c_i] = 0, 1 \leq i \leq mc$, where mc is the number of constants in *DB*;
3 A set of g-links $SOGL = \emptyset$;
```
// variabilization
```
4 **for** *each true ground atom tga of QP* **do**
5 **for** *every* g-chain(tga) **do**
6 **if** *CheckLinkOf(g-chain(tga), SOGL)* **then**
7 $SOGL \longleftarrow$ *g-link*(*g-chain*(tga)) ;
8 $SL \longleftarrow variabilize(\text{g-chain}(tga), maxVar, mapVar)$;
9 **end**
10 **end**
11 **end**
12 Return(*SL*);

Algorithm 7 sketches our idea to build the set SL of variable literals given the learning dataset DB and the query predicate QP. The algorithm considers each true ground atom *tga* of the query predicate QP (*line 4*) and finds every *g-chain(tga)* (*line 5*). Function *CheckLinkOf(g-chain(tga))* (*line 6*) performs two operations. It first creates a g-link *gl* = *g-link(g-chain(tga))* then checks whether *gl* is already in the set $SOGL$ containing all already built *g-links* (which is really the set of v-links after variabilizing). Variabilization will occur only if *gl* does not appear in the set $SOGL$ (*line 8*). By using the set $SOGL$ of g-links instead of g-chains we can reduce the needed memory bacause there are a lot of g-chains sharing a similar g-link. By checking whether *gl* is in $SOGL$, we can remove a lot of g-chains (with g-link was already kept) and thus accelerate the process of finding g-chains. Regarding the variabilization problem, we use the *simple variabilization strategy* to

variabilize each *g-chain(tga)* ensuring that different constants in this *g-chain* are replaced by different variables. In more details, to variabilize a *g-chain(tga)*, the algorithm uses the same variable literal for the starting true ground atom *tga*, and for the remaining ones a new variable is only assigned to the constants that have not previously been assigned a variable. Algorithm 8 describes gradually this step.

Algorithm 8: Variabilizing(*g-chain(tga)*, *maxVar*, *mapVar*)

Input : $g-chain(tga) = \langle g_1(t_1^1, ..., t_{m_1}^1), ..., g_k(t_1^k, ..., t_{m_k}^k)\rangle$ is a g-chain starting from the true ground atom *tga*, where $t_{m_j}^i$ is the constant at position m_j of the ground atom g_i;
maxVar: the maximum number of variables used;
mapVar: A list that maps constants to variables where *mapVar[c] = -v* implies that a constant *c* is replaced by a variable *-v*;

Output: *SL:* a set of variable literals;

```
   // variabilize the first ground atom
1  for (i = 1; i ≤ m_1; i++) do
2  |  if (∃j, 1 ≤ j < i such that t_i^1 == t_j^1) then
3  |  |  mapVar[t_i^1] = mapVar[t_j^1];
4  |  else
5  |  |  maxVar++ ;
6  |  |  mapVar[t_i^1] = −maxVar;
7  |  end
8  end
   // variabilize the remainders
9  for (i = 2; i ≤ k; i++) do
10 |  for (j = 1; j ≤ m_i; j++) do
11 |  |  if (mapVar[t_j^i] == 0) then
12 |  |  |  maxVar++;
13 |  |  |  mapVar[t_j^i] = −maxVar;
14 |  |  end
15 |  end
16 end
   // Replace constants by variables
17 θ =< t_1^1/mapVar[t_1^1], ..., t_j^i/mapVar[t_j^i], ..., t_{m_k}^k/mapVar[t_{m_k}^k] >;
18 v − chain = g − chain_k(tga)θ;
19 Return(v-chain, maxVar, mapVar);
```

We detail how to variabilize a g-chain in particular and illustrate step by step the process of generating literals through Example 25 below:

Example 25 *Let* DB *be a database composed of 15 ground atoms as follows:*

advisedBy(Bart, Ada), student(Bart), professor(Ada), publication(T1,Bart), publication(T2, Bart), publication(T1, Ada), publication(T2, Ada), advisedBy(Betty, Alan), advisedBy(Bob, Alan), student(Betty), professor(Alan), publication(T3,Betty), publication(T4, Bob), publication(T4, Alan), publication(T5, Alan).

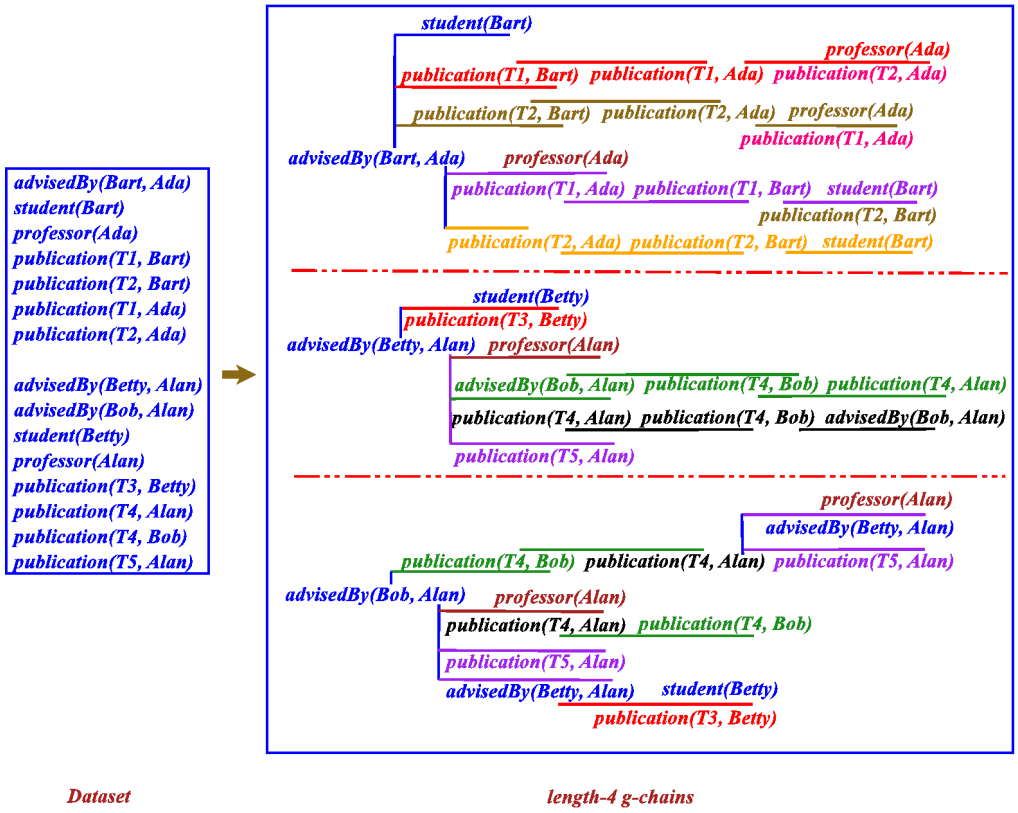


Figure 4.4: Example of g-chains in DMSP

Let QP={advisedBy}, maxLength = 4. *Figure 4.4 shows all possible* g-chains *of true ground atoms* advisedBy(Bart, Ada), advisedBy(Betty, Alan) *and* advisedBy(Bob, Alan). *Figure 4.5 exhibits all g-links of the corresponding* g-chains *shown in Figure 4.4 and Figure 4.6 gives the variable literals according to the process of variabilization. Corresponding to every* g-chain, *function* CheckLinkOf *(*line 6 in Algorithm 7*) creates a g-link. At the beginning, the* g-link {advisedBy student 1 1} *corresponding to the* g-chain {advisedBy(Bart, Ada) student(Bart)} *is created. It is the first g-chain, therefore the g-link is added into a set* SOGL *of g-links and the* g-chain {advisedBy(Bart, Ada) student(Bart)} *is variabilized to get the set of variable literals* SL = {advisedBy(A, B), student(A)}, *where* A *and* B *are variables respectively for the two constants* Bart *and* Ada.

The algorithm next takes into account the g-chain*:*

{advisedBy(Bart, Ada), publication(T1, Bart), publication(T1, Ada), professor(Ada)}

and creates a g-link:

gl=⟨{advisedBy publication 1 2} / {publication publication 1 1} / {publication professor 2 1}⟩.

Because gl *is not in the set* SOGL, gl *is added into* SOGL *and the* g-chain *is variabilized to get the set of variable literals:*

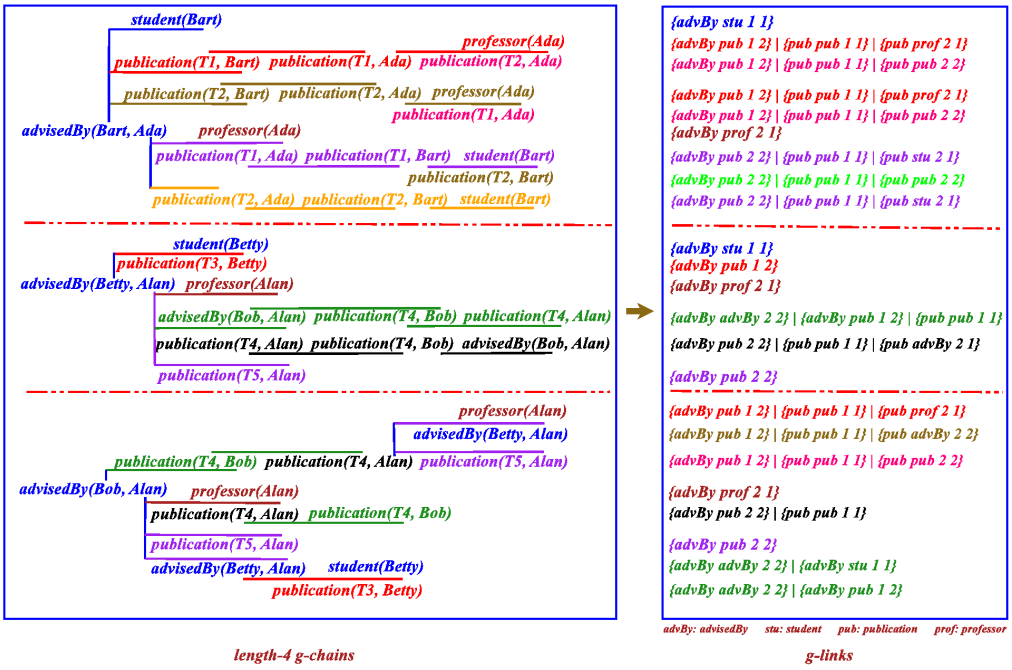


Figure 4.5: Example of g-links in DMSP

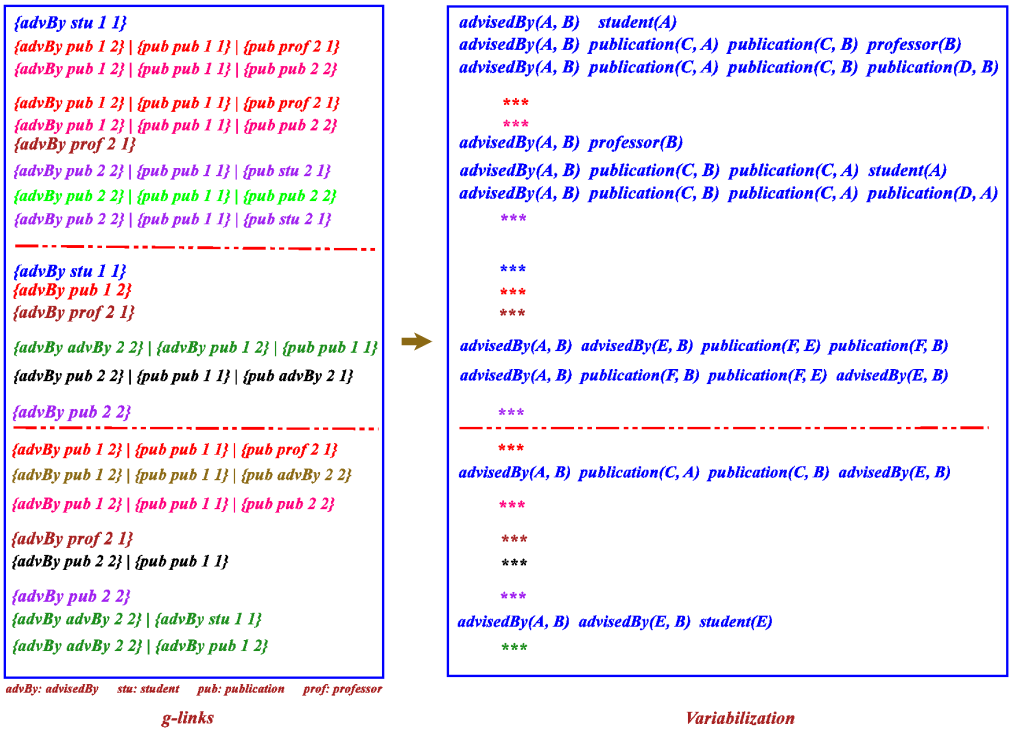


Figure 4.6: Example of the variabilization in DMSP

SL={advisedBy(A, B), student(A), publication(C, A), publication(C, B), professor(B)}.

Considering then the g-chain*:*

{advisedBy(Bart, Ada) publication(T2, Bart) publication(T2, Ada) professor(Ada)},

the algorithm also creates the g-link:

gl1=⟨{advisedBy publication 1 2} / {publication publication 1 1} / {publication professor 2 1}⟩,

but gl1 *is already present in the set of g-links (*gl1 *and* gl *are identical), therefore variabilizing for g-chain is not useful. The three stars sign (***) displayed in Figure 4.6 means that there is no new variabilization for the corresponding g-chain. As we can see from Figure 4.6, this situation occurs quite frequently in this example database. It must be noticed that, in the case of the g-chain:*

{advisedBy(Betty, Alan) publication(T3, Betty)},

the corresponding g-link:

⟨{advisedBy publication 1 2}⟩

is included as a prefix of a g-link and thus the algorithm also does not variabilize this g-chain.

Let us consider now the g-chain:

{advisedBy(Betty, Alan) advisedBy(Bob, Alan) publication(T4,Bob) publication(T4, Alan)}.

The algorithm creates the g-link:

gl2=⟨{advisedBy advisedByBy 2 2} / {advisedBy publication 1 2} / {publication publication 1 1 }⟩.

This g-link is then variabilized because gl2 *has not yet occurred in the set of g-links. At the beginning of the variabilization step, the variable literal* advisedBy(A, B) *is reused to map the starting ground atom* advisedBy(Betty, Alan) *(as we mentioned before, the algorithm uses the same variable literal for all starting true ground atoms of the query predicate), hence two constants* Betty *and* Alan *are respectively mapped to two variables* A *and* B*. The two constants* Bob *and* T4 *are new constants to be considered, thus they are respectively assigned to two new variables* E *and* F*. After this process, three new variable literals were created. They are* advisedBy(E, B), publication(F, E) *and* publication(F, B).

Having repeated this process until the last true ground atom of the query predicate advisedBy*, algorithm 7 produces a set of 11 variable literals as follows:*

SL={advisedBy(A, B), student(A), publication(C, A), publication(C, B), professor(B), publication(D, B), publication(D, A), advisedBy(E, B), publication(F, E), publication(F, B), student(E)}.

We end this subsection by introducing the following lemma and an example to compare to our first method.

Lemma 26 *The set SL of variable literals created by Algorithm 7 is the minimum set such that for each ground atom* e *of the query predicate* QP*, for each* g-chain(e)*, there always exists at least a variabilization:* $var(g\text{-}chain(e)) \subseteq SL$.

Proof Assume that the set SL of variable literals created by Algorithm 7 is not the minimum set. This means that there is a variable literal $vl \in SL$ such that: for each true ground atom e, for each $g\text{-}chain_k(e)$, there always exists at least a variabilization $var(g\text{-}chain(e)) \subseteq SL \backslash vl$. Following the process of variabilization in Algorithm 7, there exists at least some *g-chain(e)* such that it is variabilized and $vl \in var(g\text{-}chain(e))$. The positions of variable literals appearing in *var(g-chain(e))* are fixed. Besides, different variables in *var(g-chain(e))* map to different constants in *g-chain(e)*, therefore vl can not be replaced by the other element in the set SL, so that we cannot remove the variable literal vl from the set SL. Hence, the set SL is the minimum set. □

In many situations the method in HDSM creates much more variable literals than this one. We illustrate this remark by the following example.

Example 27 *Let us consider a database of four ground atoms as follows:*

advisedBy(Bob, Alan), advisedBy(Betty, Alan), publication(T1, Betty), publication(T2, Betty), publication(T3, Betty).

Let advisedBy *be the query predicate. Figure 4.7 shows the two methods of variabilization implemented in DMSP and HDSM respectively. While the method in HDSM creates* 5 *variable literals:*

SL_{HDSM} ={advisedBy(A, B), advisedBy(C, B), publication(D, C), publication(E, C), publication(F, C)},

the method in DMSP only produces 3 *variable literals:*

SL_{DMSP} ={advisedBy(A, B), advisedBy(C, B), publication(D, C)}.

The different sets of generated variable literals lead to different boolean tables hence to different results of the next steps of our approaches. For example, a boolean table corresponding to the variable literal *advisedBy(A, B)* in HDSM has 5 columns and the one in DMSP has 3 columns (in this case, they both use all variable literals). As the set of variable literals created by this method is smaller than the one in HDSM, the boolean tables are created faster also.

4.3.3 Structure of DMSP

We have as input a database DB defining positive or negative examples, a query predicate QP and an initial MLN (that can be empty). DMSP aims at learning a MLN that correctly discriminates between true and false groundings of the query predicate QP. As mentioned above, DMSP implements this new propositionalization method, modifies the way to create candidate clauses and changes the order of candidate clauses to learn the final MLN while using a structure similar to the one in HDSM (see in Subsection 4.2.5.1). In the following, we concentrate on explaining the way to generate candidate clauses and to learn the final MLN in DMSP.

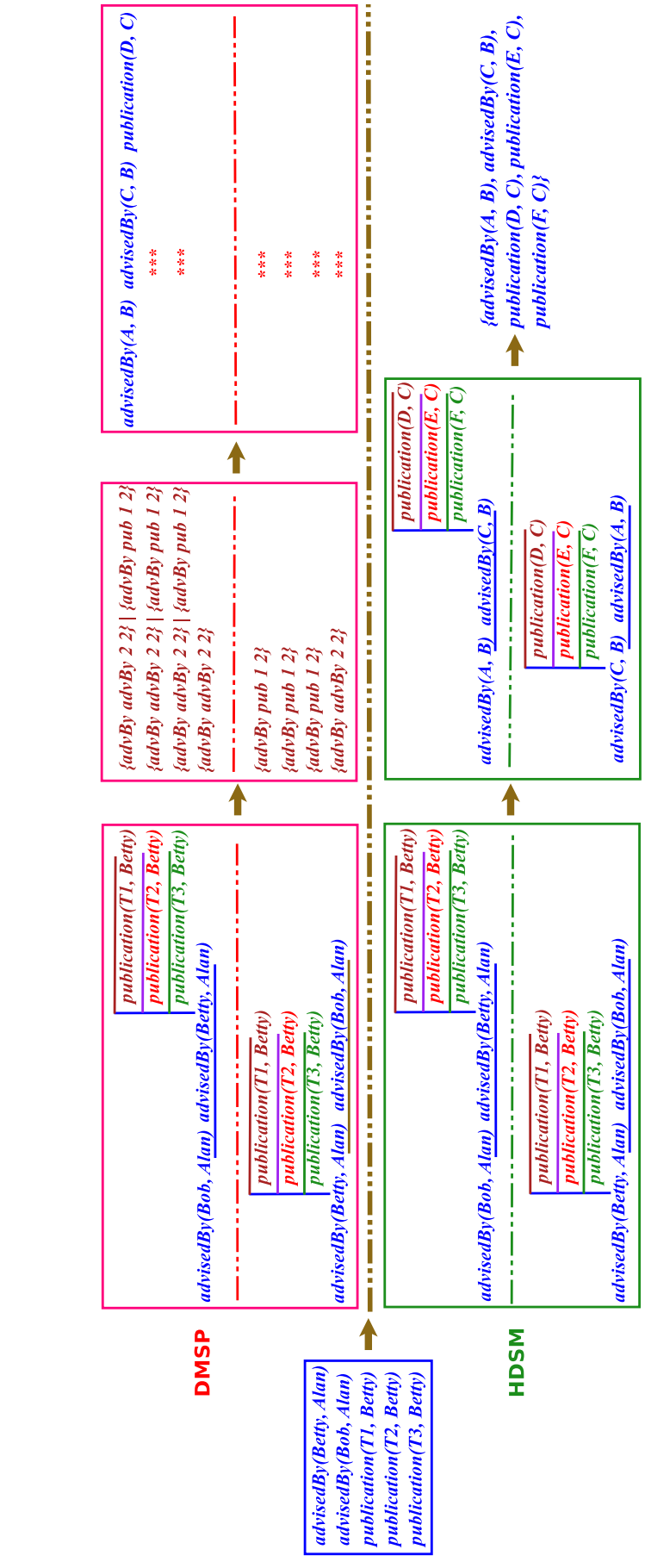


Figure 4.7: Example of the variabilization processes in DMSP and HDSM

4.3.3.1 Creating Candidate Clauses

Candidate clauses are created from template clauses in both these two methods by flipping the signs of variable literals. In HDSM, template clauses are created from every query variable literal L_{QP} and its corresponding Markov blanket $MB(L_{QP})$. The Markov blanket of each corresponding query variable literal is found by applying the GSMN algorithm on the corresponding boolean table (created in the step of performing propositionalization). If a variable literal Y is in $MB(L_{QP})$, the GSMN guarantees that the query variable literals L_{QP} is also in *MB(Y)*. Generating candidate clauses this way therefore might miss some good clauses because a real-world is often incomplete and our propositionalization method is a kind of reduction that affects the quality of boolean tables, and then the result of GSMN. We proposed in DMSP to use only the χ^2-test in order to find only a set of dependent variable literals for each query variable literal L_{QP}, from which more template clauses are generated.

4.3.3.2 Learning the Final MLN

DMSP also considers candidate clauses in turn to determine if they are added into the final MLN. However, instead of using the decreasing order of gain as in HDSM, DMSP uses two criteria: first an increasing number of literals per clause and then a decreasing *CLL*. DMSP also uses in addition an operation to replace clauses.

In details, candidate clauses are sorted by increasing number of literals. Candidate clauses having the same number of literals are sorted by decreasing CLL. DMSP then considers candidate clauses in turn. For each candidate clause c, it learns the weights for a MLN composed of the initial MLN plus the clauses kept at the previous iterations and c. If the CLL of the MLN improves, c is kept. If c is not accepted and if there exists a clause pc in the current structure such that there exists a variable renaming $\theta, pc\theta \subseteq c$, DMSP checks if replacing pc by c improves the CLL. If it does, pc is replaced by c. Finally, as adding a clause into a MLN might drop down the weight of clauses added before, once all the clauses have been considered, DMSP tries to prune some clauses of the MLN, as was done in [Kok & Domingos 2005].

4.3.4 Evaluating DMSP

4.3.4.1 Systems, Datasets and Methodology

DMSP is implemented over the Alchemy package [Kok *et al.* 2009]. We conducted experiments to compare DMSP to the state-of-the-art discriminative systems *HDSM* [Dinh *et al.* 2010a] and the state-of-the-art generative systems *ILS* [Biba *et al.* 2008b], and to *BUSL* [Kok & Domingos 2005] to find out whether boolean tables respectively created by BUSL or DMSP are better.

In order to compare to BUSL, we implemented a version of DMSP, called DMSP-W, that does only differ from BUSL in the step of building boolean tables. This means that DMSP-W also involves the L-BFGS algorithm to set weights, using the WPLL measure to choose clauses and the GSMN algorithm to find neighbors for each query variable literal, creating template clauses from every clique and considering all possible clauses of a template clause. This allows to assess directly the quality of the boolean tables created by BUSL and our method respectively.

We also used the three datasets IMDB, UW-CSE and CORA and the same settings that were used to evaluate HDSM (Section 4.2.6.1).

4.3.4.2 Results

Table 4.7 presents the average CLL and AUC-PR measures for the learning predicates over the different test folds, obtained for all the considered algorithms on the three datasets. Table 4.8 reports the average runtimes over train folds for the datasets IMDB and UW-CSE, over four learning predicates for the dataset CORA.

First, comparing DMSP to HDSM, we can notice that DMSP performs better both in terms of CLL and AUC-PR for all predicates and for all datasets. Since CLL measures the quality of the probability predictions output by the algorithm, our algorithm outperforms this state-of-the-art discriminative algorithm from the point of view of predicting correctly the query predicates given evidences. Since AUC-PR is useful to predict the few positives in data, we can conclude that DMSP enhances the ability of predicting the few positives in data. The better results of DMSP also shows that these techniques applied in DMSP are more suitable for the task of discriminative MLN structure learning than the ones implemented in HDSM, especially the new propositionalization method affects a big improvement of runtime.

Second, let us consider now DMSP and ILS. DMSP gets better values in both CLL and AUC-PR measures for all predicates and all datasets. This is the interest of DMSP compared to the state-of-the-art generative structure learning for MLNs.

Third, we compare DMSP-W to BUSL. DMSP-W highly improves CLL values and always gets better AUC-PR values. Because the main difference between DMSP-W and BUSL lies in the boolean tables, we can conclude that the boolean tables created by DMSP seem more suitable for the task of discriminative MLN structure learning than the ones created by BUSL.

Let us finally consider the algorithms all together. For all three datasets, DMSP obtains the best CLL and AUC-PR values. It must be noted that this result holds not only on average but also on every test fold of all datasets. The differences between DMSP and HDSM cause the much better results, not only in CLL and AUC-PR measure, but also in runtimes. Indeed, DMSP improves runtime approximately 60% on IMDB, 38% on UW-CSE and 38% on CORA. However, as a counterpart, DMSP still runs really slower than ILS. The runtime (of each system) includes the time to find candidate clauses, the time to learn weights for each candidate clause and the time to choose clauses for the final MLN. In practice we notice that the time spent to find candidate clauses is much less important than the time to learn weights and the time spent by inference to compute the measure (i.e. CLL). To set weights for clauses, all the systems use the L-BFGS algorithm, and thus the runtime depends on the performance of this weight learning algorithm. BUSL and DMSP change completely the MLN at each step, thus learning weights with L-BFGS is very expensive. This problem is not encountered in ILS because at each step L-BFGS is initialized with the current weights (and with a weight equal to 0 for a new clause) and it converges within a few iterations [Biba *et al.* 2008b].

CLL		Algorithms				
Datasets	Predicates	DMSP	DMSP-W	HDSM	ILS	BUSL
IMDB	WorkedUnder	**-0.022±0.007**	-0.032±0.009	-0.028±0.008	-0.036±0.006	-0.225±0.011
UW-CSE	AdvisedBy	**-0.016±0.006**	-0.027±0.008	-0.025±0.010	-0.031±0.005	-0.044±0.006
CORA	SameBib	**-0.136±0.006**	-0.151±0.009	-0.140±0.011	-0.173±0.005	-0.325±0.009
	SameTitle	**-0.085±0.009**	-0.121±0.007	-0.108±0.010	-0.144±0.009	-0.284±0.009
	SameAuthor	**-0.132±0.008**	-0.170±0.006	-0.146±0.009	-0.234±0.007	-0.356±0.008
	SameVenue	**-0.109±0.007**	-0.121±0.007	-0.115±0.011	-0.145±0.006	-0.383±0.010

AUC-PR		Algorithms				
Datasets	Predicates	DMSP	DMSP-W	HDSM	ILS	BUSL
IMDB	WorkedUnder	**0.382**	0.315	0.325	0.312	0.129
UW-CSE	AdvisedBy	**0.264**	0.216	0.230	0.187	0.204
CORA	SameBib	**0.540**	0.400	0.480	0.346	0.229
	SameTitle	**0.624**	0.421	0.498	0.415	0.418
	SameAuthor	**0.619**	0.410	0.594	0.369	0.347
	SameVenue	**0.475**	0.328	0.342	0.427	0.276

Table 4.7: CLL, AUC-PR measures

Datasets	DMSP	DMSP-W	HDSM	ILS	BUSL
IMDB	0.40	0.40	1.09	0.34	0.38
UW-CSE	5.76	6.74	9.30	2.28	8.05
CORA	31.05	33.37	50.15	28.83	34.52

Table 4.8: Runtimes(hours)

4.4 Related Works

Our methods presented in this chapter (HGSM, HDSM and DMSP) share a similar framework in which the algorithms start from the training database to generate candidate clauses. This transformation is data driven in order to limit the search space to generate candidate clauses. Concerning Inductive Logic Programming, these methods are related to bottom ILP algorithms [Lavrac & Dzeroski 1994], such as GOLEM [Muggleton & Feng 1990] and LOGAN-H [Arias *et al.* 2007], which also use training data to propose candidate clauses. This is in contrast with top-down ILP algorithms, which use training data to evaluate candidate clauses only. Our methods are also related to hybrid top-down/bottom-up ILP algorithms, such as CHILLIN [Zelle *et al.* 1994] and the method in PROGOL [Muggleton 1995], which aim at exploiting the strengths of both top-down and bottom-up techniques while avoiding their weaknesses (see in Section 2.3).

Concerning SRL, our methods are related to the growing amount of research on learning statistical relational models [Getoor & Taskar 2007]. Regarding particularly MLN structure learning, however, in contrast to the top-down strategy followed by most existing learners, our methods perform in a more bottom-up way. As far as we know, there are only the BUSL algorithm [Mihalkova & Mooney 2007] and more recently the LHL algorithm [Kok & Domingos 2009] that are also implemented in a bottom-up strategy. LHL lifts the training database into a lifted-graph, each node corresponding to a set of constants and each edge corresponding to a predicate, from which to find paths in the graph in order to generate candidate clauses. The outline of our method, at a first glance, is similar to BUSL. Nevertheless, they differ deeply in all three steps: the way propositionalization is performed, the way to build the set of candidate clauses and the way to put clauses into the final MLN:

- **Propositionalization:** The boolean tables respectively constructed by BUSL, HGSM and DMSP are different in the meaning of columns, hence in the meaning of values of entries. Each column in the boolean table *MP* of *BUSL* corresponds to a *TNode* which can be either a single variable literal or a conjunction of several variable literals, while each column in the boolean table *BT* of HGSM and DMSP only corresponds to a single variable literal. For instance, starting from the ground atom *student(a)*, knowing *advisedBy(b, a)* and then *publication(t, b)*, BUSL would produce three *TNodes t1 = {student(A)}*, *t2 = {advisedBy(B, A)}* and *t3 = {advisedBy(C, A), publication(D, C)}*, while HGSM and DMSP would produce three separated variable literals *l1 = {student(A)}*, *l2 = advisedBy(B, A)* and *l3 = publication(T, B)*. The number of *TNodes* in BUSL can be very large, depending on the number of atoms allowed per *TNode*, the size of the database and the links existing between ground atoms. On the contrary, HGSM and DMSP produce just a set of

variable literals, that is sufficient to reflect all possible links between ground atoms. For the *r-th* ground atom of the target predicate, *MP[r][t] = true* if and only if the conjunction of the set of variable literals in t is true, while *BT[r][l] = true* if there exists at least a linked-path of ground atoms starting from the *r-th* true ground atom and containing a true ground atom of l. These differences influence the performance when applying the GSMN algorithm or the χ^2-test.

- **Building candidate clauses:** BUSL uses the GSMN algorithm to determine edges amongst *TNodes* and composes candidate clauses from cliques of *TNodes*. HGSM uses just the Markov blanket of the considered variable literal and DMSP uses just the χ^2-test in order to get a little more clauses. Moreover, candidate clauses in BUSL must contain all the literals appearing in a *TNode*, meaning that, concerning our example, both *advisedBy(C, A)* and *publication(D, C)* of TNode *t3* occur together in the clause. This might not be flexible enough as it might occur that a relevant clause contains only one of these two literals.

- **Adding clauses into the final *MLN*:** For each clique, BUSL creates all possible candidate clauses, then removes duplicated clauses and finally considers them one-by-one to put into the MLN. HGSM just keeps at most one clause per template clause in the set of candidate clauses. We can also notice a difference in the order clauses are taken into account. BUSL and HGSM use a decreasing order to sort clauses while DMSP uses two criteria: first an increasing number of literals per clause and then a decreasing measure (i.e. CLL). The different orders lead to different structures.

4.5 Summary

We present in this chapter three methods for MLN structure learning: the HGSM algorithm for generative MLN structure learning, HDSM and DMSP algorithms for discriminative MLN structure learning. The basic idea of these methods is to perform a propositionalization in which information in the training dataset is transformed into boolean tables, each column of which corresponds to a variable literal. A machine learning algorithm is applied on these boolean tables in order to find relational variable literals. Candidate clauses are then created from such sets of relational variable literals. The final MLN is learned from the set of candidate clauses. Because these methods use the training database to guide and limit the search for candidate clauses, they perform in a bottom-up manner.

Our first technique of propositionalization is implemented in the HGSM algorithm: the training dataset is separated into groups of connected ground atoms starting from a true ground atom of the target predicate. A heuristic variabilization method is applied on these groups of connected ground atoms, from the largest to the shortest one, in order to build a set of variable literals. HGSM then transforms the information in the dataset into boolean tables, each one corresponding to a variable literal of the target predicate, from which GSMN (Grow-Shrink Markov Network) is applied to find a set of dependent variable literals of this variable literal. Template clauses are created from every MB and the corresponding target variable literal. Candidate clauses, generated from template clauses, are considered in turn to add into the final MLN. L-BFGS algorithm is used to

set the weights and the WPLL measure to choose clauses.

The approach applied in HGSM is then adapted to discriminative MLN structure learning system HDSM. Instead of considering all predicates in the domain, HDSM only learns for a single query predicate. HDSM uses the discriminative CLL measure to choose clauses instead of the generative WPLL measure, in order to make the system more adapted to the task of discriminative learning. The experimental results show that HDSM gives better results than the state-of-the art discriminative systems for MLN structure learning.

Our second technique of propositionalization is implemented in the DMSP algorithm for the task of learning discriminatively the MLN structure. The difference with the first technique is the way to build a set of variable literals. This second technique can create a set of variable literals much faster and more compactly than the first one based on the idea that a lot of *g-chains* can be described by only a single *v-chain*. The algorithm, therefore, first variabilizes a g-chain to generate a corresponding v-chain, which is then used as a filter to ignore a lot of other g-chains. By this way, the set of variable literals is found much faster and we also prove that it is the smallest set to describe relations related to the query predicate in the database. DMSP relies only on the χ^2-test of dependence instead of the GSMN algorithm in order to generate a little more candidate clauses.

The different steps and components implemented in HGSM, HDSM and DMSP are resumed in Table 4.9.

We end this chapter by giving the related works in ILP as well as in SRL. In particular, we discuss in details the differences between our method and BUSL, a generative MLN learning algorithm using also a propositionalization technique to find a set of candidate clauses to learn the final MLN.

	HGSM	HDSM	DMSP
Optimization algorithm	Generative: WPLL, L-BFGS	Discriminative: CLL, L-BFGS	
Propositionnalization	First method		Second method
Creating template clauses	Grow-Shrink Markov Network (GSMN) Markov blanket		χ^2-test Dependent literals
Learning the final MLN	Candidate clauses: flip the sign of literals for each template clause - Learn weights for each candidate clause - Choose the clause with the best score		
	Sort candidate clauses in descending order of gain, then test and add them into the final MLN		Sort candidate clauses by decreasing number of literals then by decreasing gain
			Replace sub-clauses
	Prunning: a clause is removed from the MLN if its weight is less than *minWeight*		

Table 4.9: A synthetic view of the different steps and components in HGSM, HDSM and DMSP

Chapter 5

Learning MLN Structure Based on Graph of Predicates

Résumé: Dans ce chapitre nous introduisons la notion de Graphe des Prédicats pour synthétiser les informations contenues dans le jeu de données en vue de l'apprentissage de la structure d'un réseau logique de Markov. Les clauses candidates sont ensuite extraites de ce graphe. Nous proposons tout d'abord la définition d'un Graphe des Prédicats, dont chaque nœud correspond à un prédicat ou à sa négation, et chaque arête correspond à un lien possible entre deux prédicats. Selon le but de l'apprentissage (génératif ou discriminant), des étiquettes différentes sont associées aux arêtes. Dans ce Graphe des Prédicats, une arête e est considérée comme une clause binaire bc et chaque arête incidente à e est considérée comme une bonne direction pour prolonger cette clause. Par conséquent, chaque chemin initié par une arête dans le graphe des prédicats est étendu progressivement pour engendrer des clauses candidates plus longues et potentiellement meilleures. En d'autres termes, nos méthodes utilisent une stratégie descendante pour limiter la recherche des clauses candidates dans le graphe des prédicats, ce qui est beaucoup plus rapide qu'une recherche exhaustive dans l'espace de clauses.

Nous avons mis en oeuvre les Graphes de Prédicats dans le cadre d'un premier système appelé GSLP (Generative Structure Learning based on graph of Predicate) pour l'apprentissage génératif de la structure d'un réseau logique de Markov. Lors des expérimentations menées pour évaluer GSLP, nous avons détecté plusieurs limites, qui ont été corrigées dans le système M-GSLP (Modified-GSLP). Nous avons enfin développé le système DSLP pour l'apprentissage discriminant de la structure d'un réseau logique de Markov. Les résultats d'expérimentation montrent que nos systèmes produisent des résultats meilleurs que les algorithmes de l'état-de-l'art, sur les jeux de données classiques.

Contents

5.1 Introduction

In Chapter 3 we described several recent Markov Logic Network structure learners: MSL (MLNs Structure Learning) [Kok & Domingos 2005], BUSL (Bottom-Up Structure Learning) [Mihalkova & Mooney 2007], ILS (Iterated Local Search) [Biba *et al.* 2008b], LHL (Learning via Hypergraph Lifting) [Kok & Domingos 2009], LSM (Learning using Structural Motifs) [Kok & Domingos 2010] and MBN (Moralized Bayes Net) [Khosravi *et al.* 2010] for generative MLN structure learning and the method of Huynh and Mooney [Huynh & Mooney 2008] and ILS-DSL (Iterated Local Search for Discriminative Structure Learning) [Biba *et al.* 2008a] for discriminative MLN structure learning. In Chapter 4 we presented our bottom-up, generate-and-test methods based on propositionalization in Inductive Logic Programming [De Raedt *et al.* 2008], which are HGSM (Heuristic Generative Structure learning for MLNs) [Dinh *et al.* 2010b] for generative learning and HDSM (Heuristic Discriminative Structure learning for MLNs) [Dinh *et al.* 2010c] and DMSP (Discriminative MLN Structure learning based on Propositionalization) [Dinh *et al.* 2010a] for discriminative learning. In order to find candidate clauses, most of these methods, more or less, reach the limitation of searching space. Indeed, MSL, ILS, ILS-DSL and the method of [Huynh & Mooney 2008] explore in an intensive way the space of clauses, generating a lot of useless candidate clauses. Using a smaller space, methods such as BUSL, LHL, HGSM and DMSP search for paths of ground atoms in the database. Despite this smaller space, searching is still computationally expensive when the database is large and there exists a lot of shared constants between ground atoms.

In this chapter, we present a technique to represent compactly relational information between shared ground atoms in the database, from which to perform an efficient search of candidate clauses. Our idea comes from two observations; the basic *concept of coverage* in Inductive Logic Programming (ILP) [De Raedt *et al.* 2008] and *associations between predicates* in the domain:

- **Coverage in ILP [De Raedt *et al.* 2008]:** The concept of coverage is used to measure how important a clause is in ILP, by which the more useful a connected clause $A_1 \vee ... \vee A_n$ is, the larger the number of true instantiations in the database it covers. We consider this concept from two points of view as follows:
  - Relevant candidate clauses are mostly the ones that are frequent enough in terms of true instantiations. Besides, if a connected formula is frequent, then its connected sub-formulas are also at least as frequent as it (similar remarks

serve as the basis of many well known strategies for frequent pattern mining). Inversely, we can find first a frequent enough binary clause from which to expand to longer clauses in terms of increasing true instantiations.

– It seems useless to consider connected clauses that do not correspond to any instantiation in the database.

- **Associations between predicates:** associations between predicates constitute a smaller model space than clauses and can thus be searched more efficiently [Kersting & De Raedt 2007].

From these observations, we propose to construct a data structure that can store information of both associations between predicates and frequencies of binary formulas, from which to search for longer formulas. We thus propose first the definition of a *Graph of Predicates* (GoP), which highlights the binary associations of predicates that share constants in terms of frequency. We then propose an top-down, generate-and-test algorithm to generate gradually candidate clauses starting from binary ones (i.e. formed of two connected atoms). This means that 3-atom clauses can then be formed on the basis of the frequent binary ones, 4-atom clauses on the basis of frequent 3-atom ones, and so on. Having got the graph of predicates, we now transform the problem into finding such a set of binary connected clauses $A_i \vee A_j$, from which the set of candidate clauses is gradually extended.

Based on these ideas, a method for generative MLN structure learning is presented in Subsection 5.2. During the experiment process, we realized that some points need to be investigated further, therefore we have applied several modifications to this generative method, that are described in Subsection 5.3. This idea is also developed for the task of learning discriminatively a MLN structure. We depict this discriminative version in Subsection 5.4.

5.2 The GSLP Algorithm

The first approach presented in this chapter called *GSLP* [Dinh *et al.* 2011a], which stands for Generative Structure Learning based on graph of Predicates, an algorithm to learn generatively the MLN structure from a training dataset *DB* and a background MLN (which can be empty). The algorithm first constructs a graph of predicates from which to generate and choose a set of candidate clauses.

5.2.1 Graph of Predicates in GSLP

Let us consider, in a domain D, a set $\mathcal{P}$ of m predicates $\{p_1, ..., p_m\}$ and a database DB consisting of true/false ground atoms of these m predicates.

Definition 28 *A template atom of a predicate p is an expression $p(type_1, \ldots, type_n)$, where argument $type_i$, $1 \leq i \leq n$ indicates the type of the i-th argument.*

Definition 29 *A link between two template atoms $p_i(a_{i_1}, ..., a_{i_q})$ and $p_j(a_{j_1}, ..., a_{j_k})$ is an ordered list of pairs of positions u and v such that the types of arguments at position u*

in p_i and v in p_j are identical. It is denoted by $link(p_i(a_{i_1}, ..., a_{i_q}),\ p_j(a_{j_1}, ..., a_{j_k})) = \langle u\ v|...\rangle$, *where* $a_{i_u} = a_{j_v}$, $1 \leq u \leq q, 1 \leq v \leq k$.

The link between two predicates p_i and p_j, denoted by $link(p_i, p_j)$, is the set of all possible links between their template atoms $p_i(a_{i_1}, ..., a_{i_q})$ and $p_j(a_{j_1}, ..., a_{j_k})$.

When two template atoms do not share any type, there exists no link between them.

We emphasize that this definition is related to but more general than the Definition 10 (Subsection 4.2.1) of link between two ground atoms. A link between two ground atoms depends on the shared constants between them, and might not exist if they do not any shared constant. A link between two template atoms depends only the native types of arguments in the domain.

Definition 30 *A formula corresponding to a* $link(p_i(a_{i_1}, ..., a_{i_q}),\ p_j(a_{j_1}, ..., a_{j_k})) = \langle si_1\ sj_1|\ ...|si_c\ sj_c|\rangle$ *is a disjunction of literals* $p_i(V_{i_1}, ..., V_{i_q}) \vee p_j(V_{j_1}, ..., V_{j_k}))$ *where* V_t, $t = i_1, ..., i_q, j_1, ..., j_k$, *are distinct variables except* $V_{i_{si_d}} = V_{j_{sj_d}}$, $1 \leq d \leq c$.

The definitions of link and formula corresponding to a link are naturally extended to negation. For example, $link(p_i(a_{i_1}, ..., a_{i_q}),\ !p_j(a_{j_1}, \ldots, a_{j_k}))$ is identical to the link between $p_i(a_{i_1}, ..., a_{i_q})$ and $p_j(a_{j_1}, ..., a_{j_k})$: $link(p_i, !p_j) \equiv link(p_i, p_j)$.

Example 31 *We consider a domain consisting of two predicates* AdvisedBy *and* Professor *respectively with two template atoms* AdvisedBy(person, person) *and* Professor(person). *The argument (type)* person *appears at position* 0 *of* Professor(person) *and at positions* 0 *and* 1 *of* AdvisedBy(person, person). *Several possible links exist between them, as for instance* $\langle 0\ 0\rangle$ *and* $\langle 0\ 1\rangle$, *and a possible formula corresponding to the latter one is* Professor(A) $\vee$ AdvisedBy(B, A).

We also have link(AdvisedBy, AdvisedBy)= $\{\langle 0\ 0\rangle, \langle 0\ 1\rangle, \langle 1\ 0\rangle, \langle 1\ 1\rangle, \langle 0\ 0 \mid 1\ 0\rangle, \langle 0\ 0 \mid 1\ 1\rangle, \langle 0\ 1 \mid 1\ 0\rangle, \langle 0\ 1 \mid 1\ 1\rangle\}$.

It must be noticed that several links are not considered here. For example, the link $\langle 0\ 0 \mid 1\ 1\rangle$ *leads to a formula composed of two similar literals. The link* $\langle 1\ 0 \mid 0\ 1\rangle$ *is similar to the link* $\langle 0\ 1 \mid 1\ 0\rangle$ *because they are corresponded by similar formulas up to a variable renaming. The link* $\langle 1\ 1 \mid 0\ 0\rangle$ *is also similar to the link* $\langle 0\ 0 \mid 1\ 1\rangle$. *The same remark can be done for* $link(Professor, Professor) = \langle 0\ 0\rangle$, $link(Professor, !Professor) = \langle 0\ 0\rangle$, *etc.*

Definition 32 *Let DB be a database, $\mathcal{P}$ the set of* m *predicates $\{p_1, ..., p_m\}$ occurring in DB, and D the domain (set of constants). The undirected graph of predicates $\{p_1, ..., p_m\}$ (GoP) is a pair* G=(V, E) *composed of a set* V *of nodes (or vertices) together with a set* E *of edges, where:*

i. A node $v_i \in V$ corresponds to a predicate p_i or its negation, $|V| = 2 \times |\mathcal{P}|$

ii. If there exists a link $link(p_i, p_j)$ between two template atoms of predicates p_i and p_j, then there exists an edge between the corresponding nodes v_i and v_j, that is associated with two labels: link-label, *which is the link itself and* num-label, *which is the number of true instantiations in the DB of the binary formula corresponding to this link.*

iii. Each node $v_i \in V$ is associated with a weight, which is defined by: $v_i.weight = k * \frac{\sum_{p=1}^{q} t_{ip}}{q}$, where q *is the number of edges incident to v_i, t_{ip} is the num-label of the* p*-th edge and* k *is a real adjustment coefficient.*

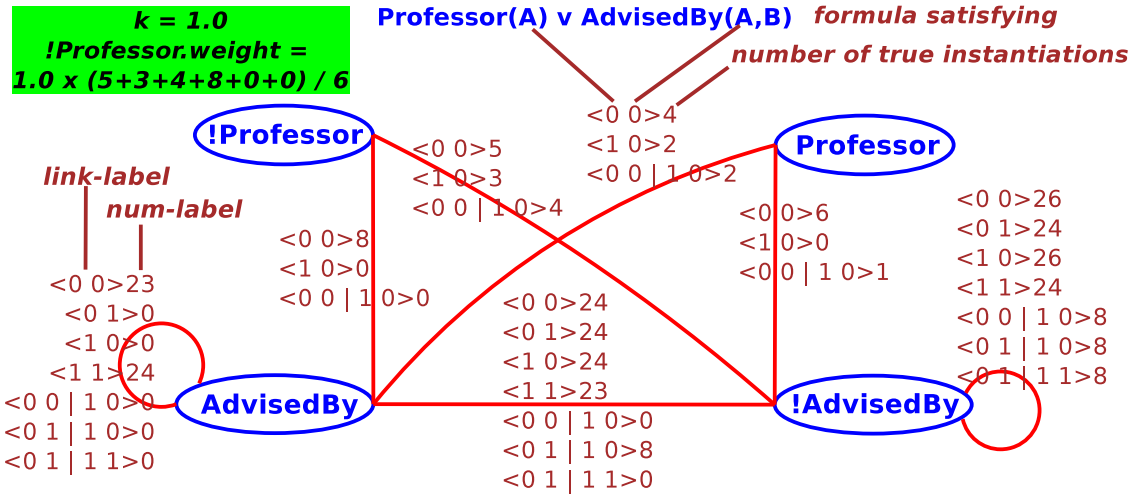


Figure 5.1: Example of graph of predicates in GSLP

Example 33 *Figure 5.1 illustrates a GoP for the domain in Example 31. A possible link between predicates* Professor *and* AdvisedBy *is* link(Professor, AdvisedBy) = $\langle 0\ 1 \rangle$. *If we suppose that we have 2 true instantiations (in the database) of the formula corresponding to this link, then we find an edge between* Professor *and* AdvisedBy *with* link-label = $\langle 0\ 1 \rangle$ *and* num-label = 2. *Inversely, we have* link(AdvisedBy, Professor)= $\langle 1\ 0 \rangle$. *When* $k = 1.0$, *the weight of node* AdvisedBy *is* $AdvisedBy.weight = 1.0 \times (23 + 24)/7 \simeq 6.714$, *if 23 and 24 are respectively the number of true instantiations of the formulas corresponding to the links* $\langle 0\ 0 \rangle$ *and* $\langle 1\ 1 \rangle$ *(and if the others do not have any instantiation in the database).*

5.2.2 Structure of GSLP

Given as input a training dataset DB consisting of ground atoms of predicates in the domain, a background MLN (which is also an empty MLN) and an integer number *maxLength*, describing the maximum length of clauses, we present here the *GSLP* algorithm to learn generatively the Markov Logic Network structure. Algorithm 9 sketches the main steps of GSLP.

In a MLN, a weight associated with a formula reflects how strong a constraint is: the higher the weight, the greater the difference in log probability between a world that satisfies the formula and one that does not, other things being equal [Richardson & Domingos 2004, Richardson & Domingos 2006]. The weights of unit clauses, composed of a single atom, roughly speaking, capture the marginal distribution of predicates, allowing longer clauses to focus on modeling predicate dependencies [Richardson & Domingos 2006, Domingos *et al.* 2008a]. For this reason, adding all unit clauses into the MLN is usually useful (*line 3*). This is also the first step of our algorithm and we call this first MLN the *unit MLN*.

Next, GSLP creates a graph of predicates with all possible edges. This graph, however, still contains a lot of edges. GSLP therefore eliminates some "less important" edges in order to reduce the search space for clauses. Clauses are then generated by finding paths in the

Algorithm 9: GSLP(*DB*, *MLN*, *maxLength*)

Input : *DB:* a training database;
MLN: an initial (empty) Markov Logic Network;
maxLength: a positive integer
Output: *MLN:* a final learned Markov Logic Network

```
// Initialize
```
A set of candidate clauses $CC = \emptyset$;
A set of paths $SP = \emptyset$;

```
// Learn unit clauses
```
Add all unit clauses into the MLN and learn weights(*DB*, *MLN*);

```
// Create graph of predicates
```
Create graph *G = (V, E)* with all possible edges;
Reduce edges of graph *G = (V, E)* ;

```
// Generate candidate clauses and learn the MLN
```
for *length=2* **to** *maxLength* **do**
  CC ⟵ CreateCandidateClauses(*DB*, *G*, *CC*, *SP*, *length*); `// Alg. 10`
  AddCandidateClausesIntoMLNAndPrune(*CC*, *MLN*);
end
Return(*MLN*);

reduced graph afterward valiabilizing them. A criterion of evaluation is used to describe how important a clause is. Based on this criterion of evaluation, GSLP decides which clause will be used to learn the final MLN as well as which edge is eliminated.

In the following, we describe how a clause is evaluated in Subsection 5.2.2.1, how edges of the graph are eliminated in Subsection 5.2.2.2, how a set of candidate clauses is found in Subsection 5.2.2.3 and how the final MLN is learned in Subsection 5.2.2.4.

5.2.2.1 Evaluating clauses

Before describing the algorithm, it is important to explain how a clause c is evaluated. Because in a MLN, the importance of a clause is reflected by its weight, choosing clauses according to the number of true instantiations they cover might not be always efficient. In addition, counting this number also takes much time when the database is large and literals have many groundings. Instead, we choose clauses based on their weight ($c.weight$) and gain ($c.gain$). These two values are computed according to a temporary MLN composed of the unit MLN plus c. The weights of this temporary MLN are then learned in order to compute the performance measure (i.e. WPLL). The weight of c is its weight in this temporary MLN. The gain of c is defined as: $c.gain = newMea - unitMea$, where $newMea$ and $unitMea$ are respectively the measures of performance of the temporary MLN and of the unit MLN. This reflects essentially the improvement of the measure implied by adding c into the unit MLN. Although it requires more times than counting the number of true instantiations, it is more suitable as it reflects how the score is improved when adding it into the unit MLN. The current weight-learners in Alchemy [Kok *et al.* 2009] can also learn weights for such a temporary MLN within an acceptable time. We consider c as a

candidate clause when $c.gain$ is greater than a given threshold $minGain$ and $|c.weight|$ is greater than a given threshold $minWeight$ where $|\ |$ denotes the absolute value. Because a negative-weighted clause is acceptable in a MLN, we use the absolute value of the weight of a clause in order to be able to choose both positive and negative weighted clauses. This differs from the evaluation for a clause implemented in HGSM.

5.2.2.2 Reducing the Number of Edges in the Graph of Predicates

GSLP first creates the graph of predicates with all possible edges (*line 4*). For each predicate, the algorithm creates two nodes (corresponding to the predicate itself and its negation), hence the graph has only $2 \times |\mathcal{P}|$ nodes, where $|\mathcal{P}|$ is the number of predicates in the domain. The number of edges incident to a node depends on the number of arguments (of the corresponding predicate) and the relation amongst types of arguments of predicates in the domain, but is usually not too large. The search space of paths in this graph is then much smaller than the whole search space of paths of ground atoms, especially when the database consists of a large number of predicates and there exists a lot of shared constants amongst ground atoms in the database.

Every edge e of GoP is created corresponding to a possible link between two nodes. We can see that e has a binary clause corresponding to its link. However, this graph with all possible links between predicates might contain some edges whose num-labels are not frequent in the database (possibly equal to zero). A binary clause $A_i \vee A_j$ corresponding to such a link will not have a vital role in the formula $A_1 \vee ... \vee A_n$ because its appearance does not raise much higher the number of true instantiations. A set of links $e_{i,j} = link(v_i, v_j)$, such that $e_{i,j}.num\text{-}label$ is less than the minimum of $v_i.weight$ and $v_j.weight$, is therefore eliminated (*line 5*). This criteria guarantees that the algorithm only considers frequent edges compared to the weights of their endpoints (the weight of nodes in the graph). This process of reduction therefore leads to a more efficient search because the number of edges of the GoP is remarkably reduced. The algorithm henceforth works completely on this reduced GoP and we keep calling it the *GoP* instead of the *reduced GoP* for the sake of simplicity.

5.2.2.3 Building a Set of Candidate Clauses

A clause is then generated from a path in the GoP. We would like to emphasize that, although the number of edges in the graph is remarkably reduced, the search for all paths (within a given length) in this GoP is still ineffective because it might also reach paths, the gain of which is not good (i.e. is not greater than a given threshold *minGain*) w.r.t. the database. In order to focus on the search for good paths, our algorithm lengthens successively every binary clause to generate longer ones such that every clause that get extended is good enough (i.e. its gain is greater than *minGain* and its absolute weight is greater than a given threshold *minWeight*) (*line 6-9*). By this way, the search space for longer clauses is reduced depending on the number of shorter candidate clauses discovered in the previous step. At each step corresponding to the *length* of clauses (*line 6*), the algorithm generates a set CC of same *length*-atoms candidate clauses (*line 7*), which is provided to add into the final MLN and to generate clauses in the next step (*line 8*). We next present in more details the way to generate clauses and discuss some advantages of

using such a set CC. Adding clauses from CC into the final MLN is depicted in Subsection 5.2.2.4.

Let us describe now the process of generating candidate clauses which is performed by function *CreateCandidateClauses(DB, G, CC, SP, length)* in Algorithm 9. Algorithm 10 contains a pseudo-code of this process. GSLP extends gradually clauses, each step corresponding to a given *length*; it involves this function in order to generate a set of *length*-atom candidate clauses by adding one more atom to each clause in the given set CC of *(length-1)*-atom clauses.

Initially, a set SCC of candidate clauses and a set SPG of the corresponding paths are empty. In case *length = 2*, meaning that the algorithm has to find the set SCC of *2*-atom good clauses, it considers all edges in the GoP, for each of them, its corresponding formula is evaluated (*line 3-10*). If this clause is good (regarding to its gain and its weight), it is stored into the set SCC as a candidate clause and the corresponding path (consisting only of this edge) is stored into the set SPG. For longer clauses, we extend from every path $p_1, ..., p_k$ (corresponding to some candidate clause) to get a longer path $p_1, ..., p_k, p_{k+1}$ by finding an edge connecting to at least one node in the set $\{p_1, ..., p_k\}$ (*line 11-25*). This process is repeated if we want to extend more than one edge from a given path. The longer path is then also variabilized to create a longer clause c_1. This clause c_1 is then evaluated to decide whether it becomes a candidate clause or not. It is considered as a candidate clause only if its gain is greater than *minGain* and its absolute weight is greater than *minWeight*. If the clause is considered as a candidate one then it and the corresponding path $p_1, \dots p_{k+1}$ are respectively stored into the set SCC and SPG for the next *length + 1* generation.

It must be noted that a longer clause c_1 can also be extended directly from a shorter clause c. However, there exists more than one path in the GoP corresponding to c; from each of them we can reach a lot of longer paths, each of them corresponding to a longer candidate clause. Therefore, extending from such corresponding paths of c is time-consuming and not every longer path corresponds to a good candidate clause. To overcome this issue, we keep the path corresponding to such a clause c, stored into a set SP of paths. By this way, the algorithm only needs to consider every path in SP to create longer clauses. Further, in Algorithm 10, we use cooperatively two sets holding information for longer discovered clauses; a set SCC of candidate clauses and a set SPG of paths in the GoP, which have a role as "catches" to filter clauses. Because a longer path can be extended from more than one shorter path and several paths lead to a same clause through the vriabilization process, this technique guarantees that corresponding to each shorter candidate clause, every path is considered exactly once for extending and a longer clause is evaluated exactly once also. Besides, we can also accelerate greedily the "speed" of that system by setting an option to extend the top best candidate clauses instead of the whole set of clauses in CC (i.e. the ones with the best gains), the search space therefore is narrowed remarkably.

Let us consider now the process of variabilization. Most approaches often variabilize a path of ground atoms by simply replacing constants by variables. We emphasize that, here, we produce clauses from paths in the graph of predicates, i.e. a list of edges containing only information of predicates and positions of shared arguments. There exists a lot of clauses corresponding to this path, thus building all possible clauses is not reasonable: the more clauses generated, the more time spent to evaluate them and to learn the final MLN.

Algorithm 10: CreateCandidateClauses(*DB*, *G*, *CC*, *SP*, *length*)

Input : *DB:* a training database;
G(V, E): a graph of predicates;
CC: a set of *(length-1)*-atom clauses;
SP: a set of paths corresponding to clauses in CC;
length: a positive integer;

Output: *CC:* a set of new *length*-atom candidate clauses;
SP: a set of paths corresponding to clauses in CC;

```
// Initialize
A set of candidate clauses SCC = ∅;
A set of paths SPG = ∅;
// Create and evaluate new candidate clauses
if length==2 then
    foreach edge e_ij ∈ E do
        Evaluate(c_ij), c_ij is the binary clause corresponding to e_ij;
        if (c_ij.gain > minGain) and (|c_ij.weight| > minWeight) then
            SCC ⟵ c_ij;
            SPG ⟵ the path of only e_ij;
        end
    end
else
    foreach path pc ∈ SP do          // corresponding to a clause c ∈ CC
        SPC ⟵ SearchForPathInGraph(pc, G);
        foreach path p ∈ SPC, p ∉ SPG do
            tc ⟵ Variabilize(p);
            if tc ∉ SCC then
                Evaluate(tc);
                if (tc.gain > 0) and (|tc.weight| > minWeight) then
                    SCC ⟵ tc ;
                    SPG ⟵ p;
                end
            end
        end
    end
end
// Return values
CC ⟵ SCC;
SP ⟵ SPG;
Return(CC, SP);
```

Algorithm 11: Variabilize(G, p)

Input : G = *(V, E):* a graph of predicate;
$p = \langle e_1.$*link-label*, ..., $e_n.$*link-label*$\rangle, e_i = (v_{i_q}, v_{i_k}) \in E, 1 \leq i \leq n$;
Output: *c:* a clause is variabilized from the path p;

```
Initilization: check[i] = false, 1 ≤ i ≤ n ;          // e_i is not variabilized
counter[v] ⟵ number of times that v appears in p, ∀v ∈ V;
c ⟵ variabilize some e_i ∈ p s.t. counter[v_q] = max_{1≤j≤n} counter[v_j], v_q ∈ e_i;
while ∃e_i ∈ p, !check[i] do
    if ∃ an endpoint v_{i_q} ∈ e_i s.t. the corresponding predicate of v_{i_q} is already
    variabilized then
        c ⟵ variabilize e_i starting from v_{i_q};
        check[i] = true ;                          // e_i has been variabilized
    end
end
Return(c);
```

e_1	advisedBy	$\langle 0\ 0 \vert 1\ 0\rangle$	!advisedBy
e_2	professor	$\langle 0\ 0\rangle$	!advisedBy
e_3	publication	$\langle 1\ 0\rangle$	student
e_4	student	$\langle 0\ 0\rangle$	!advisedBy

Table 5.1: A path of four edges

We variabilize heuristically a path to generate only *one connected clause*. First, in order to reduce the number of distinct variables in the clause, we handle the predicates in the order of descending frequency in the path. For each predicate, we perform variabilization for shared variables first, and then for unshared ones. An argument of a predicate is variabilized by a new variable if its position does not appear in the link of any edge in the path. This process is outlined in Algorithm 11.

Example 34 *We variabilize a path p composed of four links* $\{e_1, e_2, e_3, e_4\}$ *as in Table 5.1. Following Algorithm 11, we have:* counter[advisedBy] = 2, counter[!advisedBy] = 3, counter[professor] = 1 *and* counter[student] = 2. *The algorithm decides to start from* !advisedBy *as it is the most frequent predicate in p (3 times). The node* !advisedBy *is an endpoint of edge* e_1*, hence the algorithm variabilizes this edge to generate two variable literals* !advisedBy(A, B) *and* advisedBy(A, A). *During the first iteration, the algorithm determines* e_2 *and* e_4*, that both have* !advisedBy *as an endpoint, to next variabilize to generate the variable literals* professor(A) *and* student(A). *Finally, it variabilizes* e_3 *starting from* student *(which was already variabilized in the previous iteration) to generate a variable literal* publication(C, A). *At this time, the clause* advisedBy(A,A) $\vee$!advisedBy(A, B) $\vee$ professor(A) $\vee$ student(A) $\vee$ publication(C, A) *is returned as a variabilized clause from the path* $p = \{e_1, e_2, e_3, e_4\}$.

From a path $p_1, ..., p_k$ corresponding to some clause c_k, we find a path p_1, ..., p_k, p_{k+1}, then variabilize it to create a clause c_{k+1}. If c_k is a sub-clause of c_{k+1}, c_k is ignored.

5.2.2.4 Learning the Final MLN

At each step, having a set of similar-length candidate clauses, our algorithm puts them into the MLN in turn. This process is depicted in Algorithm 12. Before considering clauses in turn, the set CC of similar-length candidate clauses is sorted in an descending order of the gain of clauses. For each candidate clause c (*line 2*), the algorithm has to check whether there exists some clause c_i in the current MLN which is a sub-clause of c (the length of c_i is always less than the one of c) (*line 3*). If it is a sub-clause, the algorithm involves a replacing operation to replace c_i by c if this replacement causes an increase of the measure (*line 4*). Otherwise it involves an adding-operation to add c into the MLN if this addition causes an increase of the measure (*line 6*). The algorithm learns weights for the candidate MLN consisting of either the current learned MLN and c for adding or the current learned MLN (*without* c_i) and c for replacing. The measure is then computed for the candidate MLN. If there is an improvement of this measure, the candidate MLN becomes the current learned MLN. As adding clauses into the MLN can influence the weight of formerly added clauses, once all candidate clauses have been considered, we try to prune some clauses from the MLN. A clause with a weight smaller than *minWeight* will be removed from the MLN if this results in an increase of the measure. We can choose to prune a single clause or several each time. Although this pruning-step takes times, the more clauses to be eliminated, the faster the algorithm selects longer clauses, as the total number of clauses in the MLn after this step is reduced.

Algorithm 12: AddCandidateClausesIntoMLNAndPrune(CC, MLN)

Input : *CC:* a set of candidate clauses;
MLN: current Markov Logic Network;
Output: *MLN:* current learned Markov Logic Network at this step;

```
1  Sort CC in an decreasing order of the gain of clauses;
2  foreach clause c ∈ CC do
3  |  if ∃c_i ∈ MLN such that c_i is a sub-clause of c then
4  |  |  ReplaceClause(c_i, c, MLN) ;          // Try to replace c_i by c
5  |  else
6  |  |  AddClause(c, MLN) ;                  // Try to add c into the MLN
7  |  end
8  end
9  Prune(MLN);
10 Return(MLN);
```

In comparison with the method to learn the final MLN in HGSM (Subsection 4.2.3.2) this method differs in the order clauses are learned (each time considering clauses of the same length in turn, from the smallest to the greatest, instead of whole clauses of all length) and in performing the replacement operation in addition.

5.2.3 Experiments

We conducted experiments to evaluate the performance of GSLP compared to the recent approaches for generative MLN structure learning.

CLL			
Systems	IMDB	UW-CSE	CORA
GSLP	**-0.160 ± 0.03**	**-0.053± 0.06**	**-0.059 ± 0.05**
HGSM	-0.183 ± 0.03	-0.103± 0.04	-0.087 ± 0.05
LSM	-0.191 ± 0.02	-0.068± 0.07	-0.065 ± 0.02

AUC-PR			
Systems	IMDB	UW-CSE	CORA
GSLP	**0.789 ± 0.06**	**0.502± 0.07**	**0.886 ± 0.07**
HGSM	0.692 ± 0.07	0.311± 0.02	0.762 ± 0.06
LSM	0.714 ± 0.06	0.426± 0.07	0.803 ± 0.08

Table 5.2: CLL, AUC-PR measures

5.2.3.1 Systems, Datasets and Methodology

GSLP is implemented on top of the Alchemy package [Kok *et al.* 2009]. We compared it to LSM (Learning using Structural Motifs) [Kok & Domingos 2010] and HGSM (Heuristic Generative Structure learning for MLNs) [Dinh *et al.* 2010b], the two state-of-the-art methods for generative MLN structure learning.

We also used the three datasets IMDB, UW-CSE, CORA, and the same methodology to evaluate HGSM as the one described in Subsection 4.2.4.1.

Parameters for LSM and HGSM were respectively set as they were provided in [Kok & Domingos 2010] and in [Dinh *et al.* 2010b]. We set the maximum number of literals per clause to 5 for all systems ($maxLength = 5$). In order to limit the search space in GSLP, we set the coefficient of adjustment $k = 1$ for all tests and each time add one predicate to a path. We limit the number of similar predicates in each clause to 3 in order to use the weight learning algorithm in a reasonable time. We used $minGain = 0$ for all tests and $minWeight = 0.01$ for the two datasets IMDB and UW-CSE and $minWeight = 0.001$ for CORA as they were set in previous studies. For each test set, we ran each system on a Dual-core AMD 2.4 GHz CPU - 4GB RAM machine.

5.2.3.2 Results

We report the CLL, AUC-PR measures in Table 5.2 and runtimes in Table 5.3 for all approaches on all datasets. These are the average of CLLs, AUC-PRs and runtimes over all test folds. It must be noticed that, while we used the same parameter setting, our results do slightly differ once again from the ones in [Kok & Domingos 2010]. This comes from the difference in the way weights are learned for the final MLN. We only learned once and then performed inference on this MLN for every predicate. LSM relearned the weights for each predicate to obtain the best possible results, and thereby performed inference in a discriminative MLN for each predicate [Kok & Domingos 2010]. In particular, for the CORA database, we learned for clauses with at most *5* predicates instead of *4* in [Kok & Domingos 2010].

Comparing GSLP to HGSM and LSM, GSLP outperforms them in terms of both CLL and AUC-PR. The improvement is not only on average values over test folds but also on

	IMDB	UW-CSE	CORA
GSLP	1.59	4.62	7.82
HGSM	3.06	8.68	64.15
LSM	**1.23**	**2.88**	**6.05**

Table 5.3: Runtimes (hours)

average values over predicates in each test fold. Since the CLL determines the quality of the probability predictions output by the algorithms, GSLP improves the ability to predict correctly the query predicates given evidence. Since AUC-PR is insensitive to the large number of true negatives, GSLP enhances the ability to predict a few positives in the data. GSLP also runs much faster than HGSM, especially for the Cora dataset. Although, it performs a bit slower than LSM. LSM is based on a synthetic view of each dataset which is called a lifted hyper-graph. These lifted hyper-graphs are all computed only once during the preprocessing step and then directly used during the cross-validation phase, which is thus much faster. GSLP is much faster than HGSM as it considers only *good* edges (edges with a num-label greater than the threshold) and creates only one clause for each path. It therefore explores a much smaller search space than HGSM does.

We report in Table 5.4 the average CLL, AUC-PR values for all predicates in the UW-CSE dataset for the two algorithms GSLP and LSM. GSLP does not perform better than LSM for every predicate. There are several predicates for that LSM gives better results than GSLP. LSM performs impressively with predicates like *SameCourse*, *SamePerson*, *SameProject*, the ground atoms of which are true if and only if their two arguments are alike.

During the process of evaluation, beside these satisfying results, we have identified some points which might be investigated further:

- Tuning the adjustment coefficient k has a clear impact on the resulting MLN. The smaller k, the more edges considered, the larger the number of clauses in the MLN, and finally, the better its WPLL. Nevertheless, while WPLL remains computable, CLL, which requires inference, becomes intractable on current MLN softwares when the number of clauses becomes too large. The balance between performance (in terms of WPLL), tractability and time consumption will be an interesting point to study.

- Generative learning produces a globally satisfying structure, but it appears that the resulting prediction is much better for some predicates than for some others. Understanding this variability and reducing it constitutes an exciting challenge.

- Among candidate clauses, there frequently exist some that only differ on their variabilization. For instance, $P(X,Y) \vee Q(X,Y)$ and $P(X,Y) \vee Q(Y,X)$ compose evenly from two predicates P and Q having different positions of variables X and Y. An early detection and avoidance of such similar clauses should not affect the accuracy of the MLN, and would improve runtimes.

	GSLP		LSM	
Predicates	CLL	AUC-PR	CLL	AUC-PR
AdvisedBy	**-0.015**	**0.228**	-0.020	0.010
CourseLevel	**-0.311**	**0.801**	-0.321	0.581
HasPosition	**-0.057**	**0.821**	**-0.057**	0.568
InPhase	**-0.092**	**0.449**	-0.160	0.170
Professor	**-0.069**	0.965	-0.084	**1.000**
ProjectMember	**-0.001**	**0.001**	**-0.001**	0.0005
Publication	**-0.078**	**0.234**	-0.130	0.037
SameCourse	-0.009	0.921	**-0.002**	**1.000**
SamePerson	-0.010	0.922	**-0.002**	**1.000**
SameProject	-0.005	0.952	**-0.001**	**1.000**
Student	**-0.066**	**0.987**	-0.141	0.961
Ta	**-0.008**	**0.025**	**-0.008**	0.002
TempAdvisedBy	**-0.008**	**0.019**	**-0.008**	0.006
YearsInProgram	**-0.004**	**0.187**	-0.008	0.051
TaughtBy	**-0.059**	**0.014**	-0.078	0.004
Average	**-0.053**	**0.502**	-0.068	0.426

Table 5.4: Average measures for predicates in Uw-cse dataset

5.3 The Modified-GSLP Algorithm

As mentioned in the previous section, besides the satisfying experiment results, we identified also some limitations of GSLP. In this section, we first explain these limitations in more details and then propose modifications to overcome these restrictions. We name this new version of our algorithm M-GSLP [Dinh *et al.* 2011b], which stands for Modified-GSLP.

We overcame some limitations related to the *measure* used to score and choose clauses, the way to *reduce the number of edges of GoP*, to *build a set of candidate clauses* and to *learn the final MLN* as follows:

- **Choosing clauses:** GSLP evaluates a clause c by its weight and gain. If $c.gain$ is greater than a given threshold $minGain$ and the absolute value of $c.weight$ is greater than a given threshold $minWeight$ then c is considered as a candidate clause. The set of some clauses with the same length is then sorted in the decreasing order of gain. Clauses will be considered in turn for addition into the final MLN. According to this order, it can happen that chosen clauses are the small weighted ones because the algorithm takes into account the gain rather than the weight. In other words, it causes an imbalance between the gain and weight while we do not know which one is the most important; the weight reflects how important a clause is in a MLN, the gain reflects its importance in terms of the measure (of a MLN given database). To balance these two coefficients, we propose to use the product of them: $c.mf = |c.weight| \times c.gain$ where $|\ |$ denotes the absolute value, in order to estimate the interest of c. Besides, in the process of choosing clauses, GSLP considers every good clause at each iteration step, and this can lead to an explosion of the number of candidate clauses at some step. This issue also relates to the way to *build a set of*

candidate clauses. We present modifications to handle this problem in Subsection 5.3.2.1.

- **Reducing the number of edges in the graph:** GSLP considers only frequent enough edges in terms of the number of true instantiations in the database of its corresponding formula. In other words, it removes any edge, the num-label of which is less than the minimum of the weights of its two endpoints. We remind that the weight of a node v_i in the GoP in GSLP is the product of the average value of num-labels (of its incident edges) and an adjustment coefficient k: $v_i.weight = k * \frac{\sum_{p=1}^{q} t_{ip}}{q}$ where q is the number of incident edges to v_i, t_{ip} is the num-label of the p-th edge (see Definition 32). In some situations, this method does not treat fairly all edges. For example, when among the edges connecting two nodes a and b, if there is an edge having a num-label *much greater* than the other ones, the algorithm tends to consider only this single edge and to remove all the others. Despite the fact that we have used the coefficient k to control the number of chosen edges, choosing value for this coefficient itself raises difficulty. In other words, it is difficult to find some value of k which is homogeneous for all pairs of nodes in the graph. We define another semantic of the num-label of edges in order to overpass this difficulty, as presented in Subsection 5.3.1.

- **Building a set of candidate clauses:** GSLP extends gradually clauses starting from binary ones. If a clause c has a weight greater than $minWeight$ and a gain greater than $minGain$, it is considered as a candidate clause and, simultaneously, it is regarded as an input to generate longer clauses. However, by this method, there are a lot of clauses generated from c and not all of them (despite their acceptable weight and gain) are better than c. This means that a clause c_1 (with acceptable weight and gain) can be extended from c but adding it into the final MLN does not lead to a better global measure than c. In addition, the higher the number of such clauses c_1, the more time needed to evaluate. Although we used an option to choose several best candidate clauses, how to choose an effective number is also a hard issue. We propose to consider only one such clause c_1, which is better than c, as the input for the next extension and a clause is regarded as a candidate clause only when it reaches the limited length or we can not extend it to any better, longer clause. We present this modification in Subsection 5.3.2.1.

- **Learning the final MLN:** GSLP considers a set of similar-length clauses in turn to learn the final MLN at each step in relation to the length of clauses. This operation, however, requires to learn weights many times and the MLN learned at each iteration contains a set of clauses which are hard for inference to compute the CLL, or lead to poor weight learning results (see Subsection 5.2.3.2). We change the approach by first finding a set of clauses of any length, from which to learn the final MLN. Two criteria, *sub-clause* and *clauses of same predicates*, are used in order to eliminate several less important clauses hence accelerate the speed of the algorithm as well as to be able to overcome the limitation of current built-in tools in Alchemy [Kok *et al.* 2009]. This modification is described in Subsection 5.3.2.2.

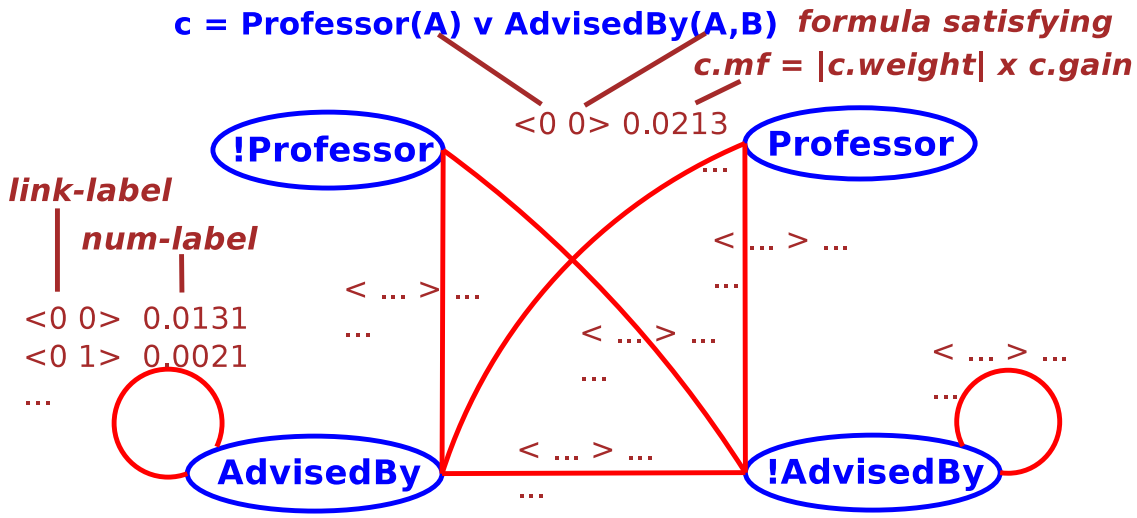


Figure 5.2: Example of graph of predicates in M-GSLP

5.3.1 Graph of Predicates in M-GSLP

We define a new semantic of the label *num-label* and eliminate the *weight* label for nodes in the GoP implemented in GSLP as follows.

Definition 35 *Let DB be a database, $\mathcal{P}$ be a set of* m *predicates* $\{p_1, ..., p_m\}$ *occurring in DB, and D the domain (set of constants). The undirected graph of predicates* $\{p_1, ..., p_m\}$ *(GoP) is a pair* G=(V, E) *composed of a set* V *of nodes (or vertices) together with a set* E *of edges, where:*

i. A node $v_i \in V$ corresponds to a predicate p_i or its negation, $|V| = 2 \times |\mathcal{P}|$

ii. If there exists a link between two template atoms of predicates p_i (or $!p_i$) and p_j (or $!p_j$), then there exists an edge between the corresponding nodes v_i and v_j, that is associated with two labels: link-label *which is the link itself (e.g. <0 0>) and* ***num-label*** **which is** ***the measure of the binary formula corresponding to this link.***

Figure 5.2 illustrates a graph of predicates for the domain in Example 31. It differs from the graph in Figure 5.1 in the meaning of the num-label of an edge and of the nodes without weights. The num-label of an edge is now the measure of its formula corresponding to its link-label, which is the product of the gain and of the absolute value of weight of the formula.

5.3.2 Structure of M-GSLP

We present here the M-GSLP algorithm [Dinh *et al.* 2011b] for generative MLN structure learning, a modified version of GSLP (described in Section 5.2.2). Algorithm 13 exposes the main steps of M-GSLP.

Algorithm 13: M-GSLP(*DB*, *MLN*, *maxLength*)

Input : *DB:* a training database;
MLN: an initial (empty) Markov Logic Network;
maxLength: a positive integer;
Output: *MLN:* the final learned Markov Logic Network;

```
// Initialize
```
A set of candidate clauses $SAC = \emptyset$;
A set of candidate clauses in each iteration $CC = \emptyset$;
A set $SP = \emptyset$ of paths corresponding to clauses in the set CC;

```
// Learn unit clauses
```
Add all unit clauses to MLN and learn weights;

```
// Create graph of predicates
```
Create graph G of predicates ;
Reduce Edges of Graph (G, CC, SP) ;

```
// Generate a set of candidate clauses
```
for *length=2* **to** *maxLength* **do**
  $SAC \longleftarrow CreateCandidateClauses(DB, G, CC, SP, length, SAC)$;
end

```
// Learn the final MLN
```
Eliminate several candidate clauses(SAC) ;
Add candidate clauses into the MLN(SAC, MLN);
Prune clauses out of the MLN;

```
// Return values
```
Return(MLN);

Basically, M-GSLP differs from GSLP first in the definition of GoP hence in the reduced GoP, then in the manner of building a set of candidate clauses and learning the final MLN.

Each edge of the graph corresponds to a binary clause *bc* corresponding to its link-label. This clause *bc* is evaluated only when its number of true instantiations in the database is greater than a given threshold *noti* (number of true instantiations). This guarantees that the algorithm only spends time to evaluate good edges in terms of number of true instantiations in the database to create the GoP. Such an edge corresponding to the clause *bc* is then eliminated if the number of true instantiations of *bc* is less than *noti* or the measure of frequency of *bc* is less than or equal to a given threshold *mft* (measure of frequency). The number of edges of the GoP is thus remarkably reduced, leading to a more efficient search (*line 5-6*). Compared to the reduced GoP in GSLP (in Subsection 9), this reduced GoP is more compact consisting of more useful edges in terms of both the measure and the number of instantiations in the database. Based on this measure, the algorithm also chooses more fairly edges than the one in GSLP.

Next, the algorithm generates a set of candidate clauses based on this reduced GoP (*line 7-9*). As in GSLP, the idea is to lengthen successively every binary clause to generate longer ones. The difference is that M-GSLP considers greedily an extended clause if it leads to an improvement of measure in comparison to the clause to be extended. Once a clause reaches a given *length* or does not lead to any longer and better clause (i.e. associated with an improved WPLL), it is considered as a candidate clause. This is reasonable because from a clause c, we always want to find a clause which is longer and better than c, therefore there is no need to take into account any clause which is less than or even equal to c. This modification speeds up the process of building candidate clauses, which is exposed in more details in Subsection 5.3.2.1.

Having got a set SAC of candidate clauses, the algorithm next eliminates some clauses (*line 10*) before adding them into the MLN in turn (*line 11*). Finally, it prunes some clauses out of the final MLN (*line 12*). We present these steps in Subsection 5.3.2.2.

5.3.2.1 Building a Set of Candidate Clauses

Algorithm 14 generates at each step a set of 1-atom longer candidate clauses. The algorithm uses the same variabilization method as in GSLP (Algorithm 11). Let SP be the set of paths having the same length and CC be the set of candidate clauses. Initially, SP and CC respectively contain all *1-length* paths and binary clauses (in case of *length = 2*): each pair of $sp_i \in SP$ and $cc_i \in CC$ corresponds to an edge e_i of the reduced graph, where sp_i is the path consisting of only the single link-label of e_i and cc_i is the binary clause corresponding to this sp_i. For longer paths, we extend every *k-length* path $pk = p_1, ..., p_k$ in SP to get a longer *(k+1)-length* path $pk1 = p_1, ..., p_k, p_{k+1}$ by finding some edge p_{k+1} connected to at least one node in pk. It must be noted that by indexing the nodes of the graph we can find quickly the set of *(k+1)-length* paths, since for a node p in pk we only have to look for every edge p_{k+1} connecting p to its higher indexed node. In our algorithm, each path is variabilized to form only one clause. Assume that c_k and c_{k+1} are respectively two variabilized clauses of paths pk and $pk1$. The clause c_{k+1} is evaluated to check whether $c_{k+1}.mf$ is greater than $c_k.mf$. If it is true, the path $pk1$ is stored into the set SPG, c_{k+1} is stored into the set SCC for the next iteration, otherwise it is no longer considered. If there does not exist any clause c_{k+1} extended from the path of c_k

such that $c_{k+1}.mf > c_k.mf$, the clause c_k becomes a candidate clause stored into the set SAC. At the last iteration, all *maxLength*-atom clauses in the set SCC are added into the set SAC because as each of them reaches the maximum number of atoms, it thus becomes a candidate clause.

5.3.2.2 Creating the Final MLN

Weights are learned using the standard L-BFGS algorithm maximizing the PLL. For inference, we use the MC-SAT algorithm [Richardson & Domingos 2006] as in previous studies. However, the PLL parameters may lead to poor results when long chains of inference are required [Domingos *et al.* 2008b]. To overcome this issue, we sort candidate clauses in SAC in the decreasing order of measure of frequency and then we eliminate some of them using two criteria as follows:

- **Sub-clause:** remove every clause $SAC[j]$ when there exists a clause $SAC[i], i < j$ such that $SAC[i]$ and $SAC[j]$ have a sub-clause relation. In case *len(SAC[i] < len(SAC[j])* and there exists a sub-clause c of *SAC[j]* and a variable replacement θ such that $c\theta \equiv SAC[i]$, we have a sub-clause relation between *SAC[i]* and *SAC[j]*. Note that all candidate clauses in SAC are already sorted in the descending order of measure of frequency therefore if $i < j$ then the measure of frequency of *SAC[i]* is always greater than or equal to the one of *SAC[j]*, we eliminate *SAC[j]* in case they have a sub-clause relation.

  Example 36 *Les us consider the two clauses below:*

  $$c_1 = advisedBy(A,\ B) \vee\ !inPhase(A,\ C)$$
  $$c_2 = !hasPosition(A,\ B) \vee\ advisedBy(C,\ A) \vee\ !inPhase(C,\ D).$$

  The clause $c_3 = advisedBy(C,\ A) \vee\ !inPhase(C,\ D)$ *is a sub-clause of* c_2. *By a variable replacement* $\theta = \langle C|A, A|B, D|C\rangle$, *we have* $c_3\theta \equiv c_1$. *In this case,* c_1 *and* c_2 *have a sub-clause relation.*

- **Clauses of same predicates:** We heuristically keep only one (with the maximum measure) of clauses composed of the same set of predicates. For example, we choose only one of the 2 clauses (*student(A)* $\vee$ *advisedBy(A,B)*, *student(A)* $\vee$ *advisedBy(B,A)*) consisting of the two predicates *student* and *advisedBy*.

Clauses in SAC are added into the MLN in turn. A candidate clause is put into the MLN as long as it leads to an improvement of the performance measure (i.e. WPLL) of the MLN learned before. As adding clauses into the MLN can influence the weight of formerly added clauses, once all candidate clauses have been considered, we try to prune some of the clauses from the MLN. We use a zero-mean on each weight in order to remove clauses whose weights are less than a given threshold *mw*, as long as their elimination causes an increase of the performance measure. This step is iterated until no clause can be removed any more.

Algorithm 14: CreateCandidateClauses(*DB*, *G*, *CC*, *SP*, *length*, *SAC*)

Input : *DB:* a training database;
G(V, E): a graph of predicates;
CC: a set of *(length-1)*-atom clauses;
SP: a set of paths corresponding to clauses in *CC*;
length: a positive integer;

Output: *CC:* a set of new *length*-atom candidate clauses;
SP: a set of paths corresponding to clauses in *CC*;
SAC: a set of candidate clauses for all length;

```
// Initialize
A set of candidate clauses SCC = ∅;
A set of paths SPG = ∅;
if length==2 then
    foreach edge e_ij ∈ E do
        SCC ⟵ c_ij, c_ij is the formula corresponding to e_ij;
        SPG ⟵ the path of only e_ij;
    end
else
    foreach path pc ∈ SP corresponding to a clause c ∈ CC do
        SPC ⟵ SearchForPathInGraph(pc, G);
        cHasALongerCandidate = false;
        foreach path p ∈ SPC, p ∉ SPG do
            tc ⟵ Variabilize(p);
            if tc ∉ SCC then
                Evaluate(tc);
                if tc.mf > c.mf then
                    SCC ⟵ tc;
                    SPG ⟵ p;
                    cHasALongerCandidate = true;
                end
            end
        end
        if cHasALongerCandidate == false then // c is a candidate
            SAC ⟵ c;
        end
    end
end
CC ⟵ SCC, SP ⟵ SPG;
if length == maxLength then
    // put all maxLength-atom clauses of SCC into SAC
    SAC ⟵ SCC;
end
Return(CC, SP, SAC);
```

Datasets	ML1	ML2	MT1	MT2
Types	9	9	11	11
Constants	234	634	696	1122
Predicates	10	10	13	13
True atoms	3485	27134	5868	9058

Table 5.5: Details of datasets

5.3.3 Experiments

M-GSLP is also implemented over the Alchemy package [Kok *et al.* 2009]. We performed experiments to estimate the performance of M-GSLP following three questions:

1 How does M-GSLP compare to the state of the art generative systems for MLN structure learning?

2 How does M-GSLP improve compared to GSLP [Dinh *et al.* 2011a]?

3 Can we compare M-GSLP to a state of the art discriminative structure learning system for specific predicates?

5.3.3.1 Systems, Datasets and Methodology

To answer question 1, we choose three state-of-the-art generative systems: LSM [Kok & Domingos 2010], HGSM [Dinh *et al.* 2010b] and MBN [Khosravi *et al.* 2010]. For question 2, we compare directly M-GSLP to GSLP. For question 3, we choose the state-of-the-art Discriminative MLN Structure learning based on Propositionalization (DMSP) (described in Section 4.3).

We used the three datasets IMDB, UW-CSE and CORA presented in Subsection 4.2.4.1 because they have been used to evaluate LSM, HGSM as well as DMSP. For each domain, we performed a *5-fold cross-validation* as in previous studies. Unfortunately, this approach is not feasible for the current setting of MBN [Khosravi *et al.* 2010]. To be able to compare to MBN, we used in addition two subsample datasets (ML1, ML2) of the MovieLens database and two subsample datasets (MT1, MT2) of the Mutagenesis database. MovieLens is available at the UC Irvine machine learning repository and Mutagenesis is widely used in ILP research. MovieLens contains two entity tables and one relationship table. The table User has 3 descriptive attributes age, gender and occupation. The table Item represents information about the movies. Mutagenesis has two entity tables: Atom with 3 descriptive attributes and Mole with 5 descriptive attributes. For each of these subsample datasets, we used each system to learn the MLN on 2/3 randomly selected atoms and tested it on the remaining 1/3, as presented in [Khosravi *et al.* 2010]. Details of these datasets are reported in Table 5.5.

To evaluate the performance of the systems, we used the same methodology for evaluating HDSM and GSLP (see in Subsection 4.2.4.1).

Parameters for LSM, HGSM and DMSP were respectively set as in the papers of [Kok & Domingos 2010], [Dinh *et al.* 2010b] and [Dinh *et al.* 2010a]. We learned clauses composed of 5 literals at most for all systems and limit the number of similar predicates

per clause to 3 in order to learn the weights in a reasonable time. We set *nti = 0*, *mft = 0*, *mw = 0.001* and *penalty = 0.001* for all domains. These parameters are chosen just to estimate the performance of GSLP, we did not optimize them for any dataset yet. For each test set, we ran each system on a Dual-core AMD 2.4 GHz CPU - 4GB RAM machine.

5.3.3.2 Results

Table 5.6 reports the average values of CLLs and AUC-PRs, and Table 5.7 reports the runtime on each dataset. Higher numbers indicate better performance and NA indicates that the system was not able to return a MLN, because of either crashing or timing out after 3 days running. For MBN, we used the values published in [Khosravi *et al.* 2010].

For the largest dataset CORA, M-GSLP gives an increase of 16.9% on CLL and 10.5% on AUC-PR compared to LSM and 37.9% on CLL and 16.4% on AUC-PR compared to HGSM. It dominates MBN on CLL in the range of 20-30%. Improvement does not only occur on average values over test folds but also on most of average values over predicates in each test fold. M-GSLP runs much faster than HGSM, especially for CORA, but performs a bit slower than LSM. We can answer to question 1 that GSLP performs better, on these datasets, than the state-of-the art systems in terms of both CLL and AUC-PR.

Comparing M-GSLP to GSLP, M-GSLP gives better results than GSLP does for all datasets on both CLL and AUC-PR measures, except on the AUC-PR of the UW-CSE dataset where they are equal. M-GSLP not only produces better results than GSLP, but it also runs much faster, with 72.95% on IMDB, 33.98% on UW-CSE, 14.07% on CORA, 39.19% on Movilens2, 46.45% on MUTAGENESIS1 and 32.19% on MUTAGENESIS2. These improvements come from the modifications that we have done so far for M-GSLP. This answers question 2.

However, at the predicate level, LSM and GSLP sometimes achieve better results than M-GSLP. We provide the average CLLs, AUC-PRs for each predicate in a test-fold of the UW-CSE dataset in Table 5.8.

Comparing M-GSLP to DMSP, we used DMSP to learn discriminatively a MLN structure for predicates WorkedUnder (in the IMDB), Advisor (in the UW-CSE) and SameBib, SameTitle, SameAuthor, SameVenue (in the CORA). These predicates have been often used in previous studies for MLN discriminative learning [Dinh *et al.* 2010a], [Biba *et al.* 2008a]. Table 5.9 exposes the average CLL, AUC-PR values on those predicates over all test folds for each dataset. The results show that GSLP produces much better AUC-PRs than DMSP did while it looses a little CLLs. Indeed, it is worse 5% on CLL on predicate SameTitle while the AUC-PR is increased 18.32%. The improvement of AUC-PR reaches a maximum of 37.78% on predicate SameAuthor while the CLL is equal. GSLP gives better results than DMSP in terms of both CLL (1.87%) and AUC-PR (16.96%) on predicate SameVenue. We can conclude on question 2 that GSLP is competitive for the discriminative approach in terms of CLL and dominates it in terms of AUC-PR.

5.3.4 Complexity of the M-GSLP Algorithm

We consider a domain of m predicates $\mathcal{P} = \{p_1, ..., p_m\}$. Without loss of generality, we assume that:

	Systems				
	Datasets	M-GSLP	GSLP	LSM	HGSM	
CLL	IMDB	**-0.099±0.03**	-0.160±0.03	-0.191±0.02	-0.183±0.03	MBN
	UW-CSE	**-0.052±0.06**	-0.058±0.06	-0.068±0.07	-0.103±0.04	
	CORA	**-0.054±0.05**	-0.059±0.05	-0.065±0.02	-0.087±0.05	
	ML1	**-0.764**	-0.824	-0.933	-1.010	-0.99
	ML2	**-0.915**	-0.947	-0.985	NA	-1.15
	MT1	**-1.689**	-1.801	-1.855	-2.031	-2.44
	MT2	**-1.754**	-1.863	NA	NA	-2.36

	Systems				
	Datasets	M-GSLP	GSLP	LSM	HGSM	
AUC-PR	IMDB	**0.790± 0.06**	0.789±0.06	0.714±0.06	0.692±0.07	MBN
	UW-CSE	**0.502± 0.07**	**0.502±0.07**	0.426±0.07	0.311±0.02	
	CORA	**0.887± 0.07**	0.836±0.07	0.803±0.08	0.762±0.06	
	ML1	**0.701**	0.683	0.683	0.611	0.64
	ML2	**0.677**	0.662	0.658	NA	0.62
	MT1	**0.766**	0.724	0.713	0.692	0.69
	MT2	**0.743**	0.615	NA	NA	0.73

Table 5.6: CLL, AUC-PR measures (generative)

Datasets	M-GSLP	GSLP	LSM	HGSM
IMDB	**0.43**	1.59	1.23	3.06
UW-CSE	3.05	4.62	**2.88**	8.68
CORA	6.72	7.82	**6.05**	64.15
MOVILENS1	**0.44**	0.44	0.46	4.02
MOVILENS2	3.18	5.23	**2.97**	NA
MUTAGENESIS1	**1.13**	2.11	1.25	4.37
MUTAGENESIS2	**8.28**	12.21	NA	NA

Table 5.7: Runtimes (hours) (generative)

	M-GSLP		GSLP		LSM	
Predicates	CLL	AUC6PR	CLL	AUC-PR	CLL	AUC-PR
AdvisedBy	-0.016	0.042	**-0.015**	**0.228**	-0.025	0.056
CourseLevel	-0.317	0.500	**-0.311**	**0.601**	-0.332	0.537
HasPosition	-0.063	0.250	**-0.057**	**0.321**	-0.059	**0.469**
InPhase	**-0.091**	**0.405**	-0.092	0.349	-0.220	0.309
Professor	**-0.0003**	**1.000**	-0.069	0.965	-0.002	**1.000**
ProjectMember	-0.071	0.0003	**-0.001**	0.001	-0.001	0.0003
Publication	**-0.071**	0.040	-0.078	**0.234**	-0.107	0.035
SameCourse	**-0.0001**	**1.000**	-0.009	0.921	-0.002	**1.000**
SamePerson	**-0.00007**	**1.000**	-0.010	0.922	-0.002	**1.000**
SameProject	**-0.00008**	**1.000**	-0.005	0.952	-0.001	**1.000**
Student	**-0.00002**	**1.000**	-0.066	0.987	-0.051	0.993
Ta	-0.004	0.001	-0.008	**0.025**	**-0.003**	0.0009
TempAdvisedBy	-0.009	**0.024**	**-0.008**	0.019	-0.009	0.003
YearsInProgram	-0.109	0.119	**-0.004**	**0.187**	-0.135	0.080
TaughtBy	**-0.008**	0.010	-0.059	**0.014**	-0.009	0.006
Average	-0.051	0.426	-0.053	0.502	-0.064	0.486

Table 5.8: CLL and AUC values in a test-fold for every predicate in UW-CSE

Algorithms →		DMSP		M-GSLP	
Datasets	Predicates	CLL	AUC-PR	CLL	AUC-PR
IMDB	WorkedUnder	**-0.022**	0.382	-0.023	**0.394**
UW-CSE	AdvisedBy	**-0.016**	0.264	**-0.016**	**0.356**
CORA	SameBib	**-0.136**	0.540	-0.139	**0.669**
	SameTitle	**-0.085**	0.624	-0.090	**0.764**
	SameAuthor	**-0.132**	0.619	-0.132	**0.995**
	SameVenue	-0.109	0.475	**-0.107**	**0.572**

Table 5.9: CLL, AUC-PR measures (discriminative)

- Every predicate $p_i, 1 \leq i \leq m$ of arity-n and its corresponding template atom is $p_i(a_{i_1}, \ldots, a_{i_n})$.
- $h(.)$ is a function of the average complexity of the weight learning algorithm.

We present here an analysis of time-complexity of M-GSLP to generate a set of candidate clauses in the worst case. This worst case occurs when *we cannot eliminate any edge of the graph* and *every path in the graph corresponds to a good clause.*

Lemma 37 *The maximum number of clauses to be evaluated by M-GSLP to generate a set of candidate clauses in the worst case is:*

$$\sum_{l=2}^{maxLength} \left(\binom{2m}{2} \times \left(n^2 + \sum_{k=2}^{n} \binom{n^2}{k} \right) \right), \tag{5.1}$$

where maxLength *is the maximum number of literals in a clause.*

Proof The number of nodes in the graph of predicates is *2m*. From a node p_i to a node p_j, $1 \leq i, j \leq 2m$ there are at most n^2 edges in which each edge has the *link-label* composed of only a *single-shared* argument (i.e. considering only one position in p_i and one position in p_j at which the arguments respectively of p_i and p_j are identical). Every edge of k-shared arguments, $1 \leq k \leq n$ connecting p_i and p_j is a k-combination of the set of edges of *single-shared* argument, thus the number of edges of k-shared arguments is $\binom{n^2}{k}$. The total number of edges connecting p_i to p_j is:

$$n^2 + \sum_{k=2}^{n} \binom{n^2}{k} \tag{5.2}$$

In the worst case, each node in the graph is connected to all the other one, thus the maximum number of pairs of nodes in this graph is $\binom{2m}{2}$, then the maximum number of edges in the graph is:

$$mne = \binom{2m}{2} \times \left(n^2 + \sum_{k=2}^{n} \binom{n^2}{k} \right) \tag{5.3}$$

A *length*-edge path in the graph is essentially a *length*-combination of the set of these *mne* edges. The number of paths of a length ranging from 2 to *maxLength* is:

$$\sum_{l=2}^{maxLength} \binom{mne}{l} = \sum_{l=2}^{maxLength} \left(\binom{2m}{2} \times \left(n^2 + \sum_{k=2}^{n} \binom{n^2}{k} \right) \right) \tag{5.4}$$

□

In the worst case, the clause variabilized from any longer extended path is new and better than the one to be extended. In other words, any path in the GoP with a length of *2* to *maxLength* is variabilized to generate a good, new clause. This clause is then evaluated to compute its weight and gain, therefore the weight learner is called once for each path (of length *2* to *maxLength*). With *h(.)* the function of average time complexity

of the weight learning algorithm, the average time for M-GSLP to get a set of candidate clauses in the worst case is:

$$h(.) \times \sum_{l=2}^{maxLength} \left(\binom{2m}{2} \times \binom{n^2 + \sum_{k=2}^{n} \binom{n^2}{k}}{l} \right).$$

However, as far as we know, none of the real-world databases contains every template atom having n arity of a same type. Besides, it hardly happens that and any longer clause is always better than the shorter one in term of the measure (of a MLN given this database). The time-complexity of M-GSLP, therefore, never reaches this upper bound. M-GSLP is usually much less complex for a real-world database.

5.4 The DSLP Algorithm

We presented in the previous section the M-GSLP algorithm, a modified version of GSLP described in section 5.2. The experiments shown that M-GSLP produces better results than the state-of-the-art approaches for generative MLN structure learning for the benchmark databases we used. This approach also takes advantage in dealing with complex domains of many predicates and associations amongst them.

Concerning discriminative learning, however, M-GSLP produced a little worse results than DMSP [Dinh *et al.* 2010a] that was introduced in Section 4.3. One reason is that M-GSLP is designed for a generative learning purpose which searches clauses for all predicates in the domain instead of for a precise predicate. Although DMSP achieved better CLL and AUC-PR values than M-GSLP did, it has to search in the space of ground atoms linked by shared constants in the database, like almost other methods for discriminative MLN structure learning do, such as HDSM [Dinh *et al.* 2010c] and ILS-DSL [Biba *et al.* 2008a], therefore a lot of time is consumed for searching. An algorithm which can produce good results in an faster manner remains necessary for the time being.

A strong advantage of M-GSLP is that it can handle a complex domain with many predicates and associations between them. By using a GoP holding information of frequency for every binary association between predicates, it can search for candidate clauses in an effective manner. We want to apply this technique in a discriminative learning purpose in order to use this advantage to overcome the limitation of current discriminative structure learners for MLNs.

Naturally, we can use M-GSLP to learn a discriminative MLN structure for some precise predicate. However, the experimental results in the previous section shown that M-GSLP only dominates DMSP in terms of runtime but did not exceed it in terms of measures (i.e. CLL and AUC-PR). One reason is that M-GSLP uses all generative tools (i.e. WPLL for measuring and L-BFGS for weight learning) which are not totally suitable for a discriminative purpose. A solution is that we should limit to discriminative tools instead of using the generative ones. Unfortunately, we can not apply a discriminative weight learning algorithm in order to create the graph of predicates as in Subsection 5.3.1 because it does not produce any result for clauses that do not contain the query predicate. In other words, for every edge of non-query predicate we can not calculate the measure of the clause corresponding to its link by using a discriminative weight learning algorithm. We propose another definition for the *num-label* of edge in the GoP in order to make it

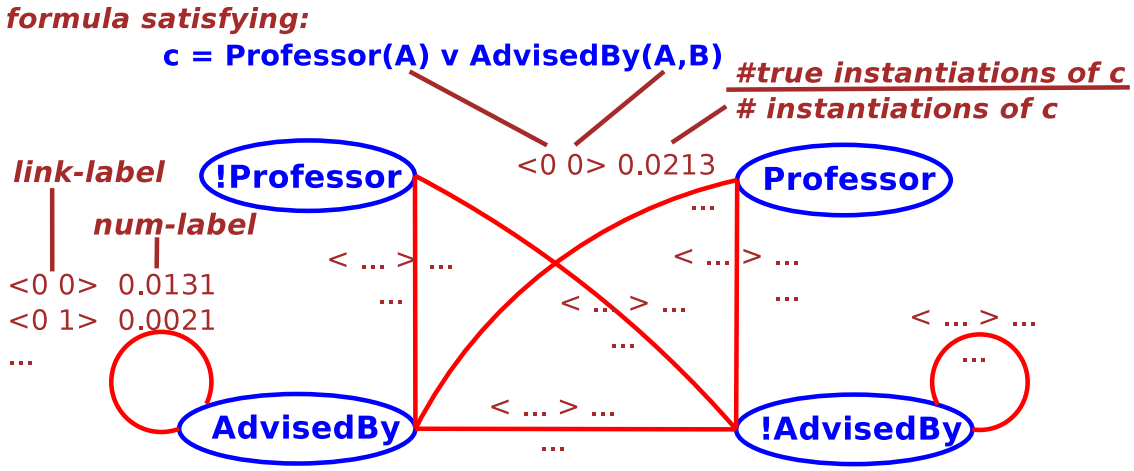


Figure 5.3: Example of graph of predicates in DSLP

applicable in our discriminative structure learning algorithm, called DSLP (Discriminative Structure Learning based on graph of Predicates for MLNs). We present this definition in Subsection 5.4.1 and the structure of DSLP in Subsection 5.4.2.

5.4.1 Graph of Predicates in DSLP

Definition 38 *Let DB be a database, $\mathcal{P}$ be a set of* m *predicates* $\{p_1, ..., p_m\}$ *occurring in DB, and D the domain (set of constants). The undirected graph of predicates* $\{p_1, ..., p_m\}$ *(GoP) is a pair* G=(V, E) *composed of a set* V *of nodes (or vertices) together with a set* E *of edges, where:*

i. A node $v_i \in V$ corresponds to a predicate p_i or its negation, $|V| = 2 \times |\mathcal{P}|$

ii. If there exists a link between two template atoms of predicates p_i (or $!p_i$) and p_j (or $!p_j$), then there exists an edge e_{ij} between the corresponding nodes v_i and v_j, and this edge is associated with two labels: link-label *which is the link itself, and* ***num-label*** **which is the proportion of the number *numTrue* of true instantiations to the total number *numTotal* of instantiations in the database of the formula corresponding to this link. We have** $e_{ij}.num\text{-}label = \frac{numTrue}{numTotal}$.

Figure 5.3 illustrates a graph of predicates for the domain in Example 31. It differs from the GoP in Figure 5.1 in the meaning of the *num-label* of an edge and the nodes without weights. It differs from the graph in Figure 5.2 in the meaning of the *num-label* of an edge.

5.4.2 Structure of DSLP

Given as inputs a training dataset DB consisting of ground atoms of predicates in the domain, a background MLN (which can be empty), *a query predicate QP* and a positive integer *maxLength*, specifying the maximum length of clauses, we aim at learning a MLN

Algorithm 15: DSLP(*DB*, *MLN*, *QP*, *maxLength*)

Input : *DB:* a training database;
MLN: an initial (empty) Markov Logic Network;
QP: a query predicate;
maxLength: a positive integer;
Output: *MLN:* the final learned Markov Logic Network;

```
// Initialize
```
A set of candidate clauses $CC = \emptyset$;
A set of paths $SP = \emptyset$;

```
// Add unit clauses
```
Add all unit clauses to MLN and learn weights;

```
// Creating graph of predicates
```
Create graph of predicates *G = (V, E)*;
Reduce Edges of Graph *G = (V, E)*;

```
// Generating candidate clauses
```
for *length=2* **to** *maxLength* **do**
  $CC \longleftarrow CreateCandidateClauses(QP, DB, G, SP)$;
end

```
// Learning the final MLN
```
Select Candidate Clauses (*CC*) ;
Add Candidate Clauses into the MLN (*QP*, *CC*, *MLN*) ;
Prune Clauses out of the MLN (*QP*, *MLN*) ;
Return(*MLN*);

that correctly discriminates between true and false groundings of the query predicate *QP*. We present here an algorithm for discriminative MLN structure learning, called DSLP (Discriminative Structure Learning based on graph of Predicates for MLN). Algorithm 15 exposes the main steps of the DSLP.

Fundamentally, we apply the structure of M-GSLP for this discriminative method, DSLP. In order to implement in a discriminative purpose, in which the system learns a MLN structure for the specific predicate *QP*, some adaptations from M-GSLP are performed. Concerning the measure to choose clauses, we propose to use the Conditional-log-likelihood (CLL) of the query predicate given the remaining predicates as evidence in order to measure directly the accuracy of the probability of the query predicate, as was done in [Richardson & Domingos 2006, Biba *et al.* 2008a] and in the others. Several modifications in the manner to evaluate a clause and in the process of reducing graph and building candidate clauses are also carried out compared to the ones in M-GSLP. We describe these modifications in the following subsections.

5.4.2.1 Evaluating Clauses

Evaluation of a clause *c* is also based on its *weight* and *gain* (similarly to M-GSLP in Subsection 5.3.2). The difference is that a discriminative learner is used here instead

of a generative one to learn the weights for a temporary MLN composed of the unit MLN plus c and then compute the measure of this temporary MLN given the training database (corresponding to the query predicate QP). The measure of frequency of c is given by: $c.mf = |c.weight| \times c.gain$, where $c.weight$ is the weight of c in the temporary MLN and $c.gain$ is the improvement of measure when c is added into the unit MLN ($c.gain = newMea - unitMea$, where $newMea$, $unitMea$ respectively are the discriminative measures (i.e. the CLL of the query predicate QP) of the temporary MLN and of the unit MLN given the training database.

5.4.2.2 Reducing Edges in the GoP

As mentioned above for GSLP and M-GSLP, the graph may contain several useless edges in term of frequency. We remove such less frequent edges by regarding all edges having a *num-label* greater than a given threshold *nlt*. Due to the definition of *num-label*, we can choose edges in the graph more "smoothly" than what was proposed in GSLP, because the decision for choosing or eliminating an edge depends directly on the ratio between the number of true instantiations and the total number of instantiations of its corresponding formula. However, for edges connecting at least one node of the query predicate, although its num-labels are less than *nlt*, there still exists some edges for which the measure of the corresponding formula is good enough (i.e is better than a given threshold). Therefore, we compute the measure for every edge connected to at least a query predicate, whose *num-label* is less than *nlt*, and also keep the good ones. This causes an increase of the number of edges in the GoP hence spreads out the search space, but for a discriminative task, it remains entirely acceptable and feasible.

5.4.2.3 Building a Set of Candidate Clauses

DSLP also searches for a set of any-length candidate clauses, then checks them in turn for addition into the final MLN. It performs in the same manner as M-GSLP but uses a discriminative measure (i.e. the CLL). However, as the MLN of arbitrary clauses might be difficult to exploit for another system, we provide an option to limit the search to Horn clauses. With the participation of both the predicate and it negation in the graph of predicates, searching for Horn clauses can be done very efficiently in DSLP. We next focus on this process more thoroughly.

Let us recall that a Horn clause is a disjunction of variable literals with at most a positive one. In DSLP, each Horn clause is variabilized from a corresponding path of edges in the GoP, two endpoints of an edge corresponding to a predicate or its negation, therefore this path also has to contain at most a negation (of some predicate in the domain). This limitation drops down the search space in the GoP. Indeed, we can now consider the GoP as a two-side graph $G = (V^+, V^-, E)$; one side V^+ for all positive predicates and the other one V^- for all negations. A path is composed from only one node $H \in V^+$ (as the head of a Horn clause) and several nodes $B_1, ..., B_m, B_i \in V^-, 1 \leq i \leq m$ (as the body of a Horn clause). The range-restriction for this Horn clause $H :- B_1, ..., B_m$ claims that every variable in the head H has to appear in the body. In the heuristic variabilization method of DSLP, a variable in the head occurs somewhere in the body if and only if its corresponding position (in the corresponding template atom of H) participates in some

incident edge of H in the path. Let us assume that the template atom of the node H is $H(h_1, ..., h_h)$ and the set of incident edges of H participating in the corresponding path of the clause $H :- B_1, ..., B_m$ is $\{e_1, ..., e_p\}$. The range-restricted condition for this Horn clause requires $\forall i, 1 \leq i \leq h, \exists j, 1 \leq j \leq p$ such that $i \subset e_j.link - label$. It means that every position i has to appear in the link-label of some edge e_j in the path. This leads to a more efficient search because we can start from such shortest paths to extend them to the longer ones by adding only negative atoms.

Example 39 *Let us assume* AdvisedBy *as the query predicate. We do not start from a path* AdvisedBy 0 0 !Professor *because its variabilized clause,* AdvisedBy(X, Y) v !Professor(X)*, is not satisfying the range-restricted condition. A possible starting path is* AdvisedBy 0 0 !Professor, AdvisedBy 1 0 !Professor *corresponding to the clause* AdvisedBy(X, Y) v !Professor(X) v !Professor(Y)*. From this path we only need to add negative predicates in order to generate a longer Horn clauses.*

5.4.3 Experiments

Like all our systems, DSLP is also implemented over the Alchemy package [Kok *et al.* 2009]. We conducted experiments to answer the following questions:

1. How does DSLP perform compared to the state-of-the-art methods for discriminative MLN structure learning?
2. How does DSLP perform for Horn clauses?
3. How does DSLP perform with various weight learning algorithms?

5.4.3.1 Systems, Datasets and Methodology

To answer question 1, we compared DSLP to DMSP [Dinh *et al.* 2010a] using the weight learning L-BFG algorithm. To answer question 2, we ran DSLP twice respectively for Horn clauses (DSLP-H) and for arbitrary clauses (DSLP-A). For question 3, we ran DSLP three times respectively with three weight learning algorithm: L-BFGS, that maximizes the PLL measure for generative weight learning, P-SCG, that maximizes the CLL measure and M3LN, that maximizes the max-margin for discriminative weight learning. Details of these weight learning algorithms were presented in chapter 3 of this dissertation. We notice that, although L-BFGS maximizing the PLL measure is for generative learning, the authors in [Biba *et al.* 2008a] and [Dinh *et al.* 2010a] shown that using them can also lead to good results for the task of discriminative MLN structure learning. We therefore chose them for this experiment.

The three datasets IMDB, UW-CSE and CORA were also used in a 5-fold cross-validation in this experiment. We also learned for the predicates: *WorkedUnder* in IMDB, *AdvisedBy* in UW-CSE and *SameBib*, *SameTitle*, *SameAuthor*, *SameVenue* in CORA. Parameters for DMSP are set as in [Dinh *et al.* 2010a]. We used all the default parameters for P-SCG as in [Lowd & Domingos 2007] and M3LN [Huynh & Mooney 2009]. We learned clauses composed of at most 5 literals for all systems and also limit the number of similar predicates per clause to 3 in order to learn the weight and to perform inference in a reasonable time. We set parameters of DSLP to: *nti = 0*, *mft = 0* for all tests. We conducted these experiments on a Dual-core AMD 2.4 GHz CPU - 16GB RAM machine.

Algorithms →		DMSP		DSLP-H		DSLP-A	
Datasets	Predicates	CLL	AUC-PR	CLL	AUC-PR	CLL	AUC-PR
IMDB	WorkedUnder	**-0.022**	0.382	-0.023	0.391	**-0.022**	**0.394**
UW-CSE	AdvisedBy	-0.016	0.264	-0.014	0.297	**-0.013**	**0.389**
CORA	SameBib	-0.136	0.540	-0.137	0.631	**-0.122**	**0.680**
	SameTitle	**-0.085**	0.624	-0.086	0.617	-0.086	**0.658**
	SameAuthor	-0.132	0.619	-0.124	0.887	**-0.115**	**0.905**
	SameVenue	-0.109	0.475	-0.101	0.523	**-0.097**	**0.571**

Table 5.10: CLL, AUC-PR measures (discriminative)

Datasets	DMSP	DSLP-H	DSLP-A
IMDB	0.40	0.23	**0.19**
UW-CSE	5.76	**2.15**	2.47
CORA	31.05	**9.12**	9.56

Table 5.11: Runtimes(hours)

5.4.3.2 Results

Table 5.10 reports the average CLL and AUC-PR values and Table 5.11 shows the average runtimes of all systems over test folds for all datasets. Concerning the CLL and AUC-PR measures, DSLP-A gives better values than DMSP except for the CLL value of predicate *SameBib* on the CORA dataset. The improvement of AUC-PR is a little better than the one of CLL. Indeed, for the predicate *WorkedUnder* (IMDB), DSLP-A and DMSP are equal on CLL but DSLP-A increases: 3.14% on AUC-PR, for the predicate *AdvisedBy* (UW-CSE), DSLP-A augments 18.75% on CLL and 47.35% on AUC-PR. For the CORA dataset, although DSLP-A decreases 1.18% on CLL, it still increases 5.45% on AUC-PR. DSLP-A augments 10.29%, 12.88%, 11.01% on CLL and 25.93%, 46.2%, 16.81% on AUC-PR respectively for the three predicates *SameBib*, *SameAuthor* and *SameVenue*. These values also show that DSLP only takes a little advantage compared to DMSP by means of the CLL and AUC-PR measures. However, from the point of view of runtime, DSLP does perform strikingly. It improves 52.5% for the predicate *WorkedUnder* (IMDB), 57.12% for the predicate *AdvisedBy* (UW-CSE) and approximately 69.21% for each predicate of the CORA dataset. We answer question 1 that DSLP dominates the state-of-the-art discriminative MLN structure learning system in the sense of runtime and achieves mostly better values by means of CLL and AUC-PR measures.

By restricting DSLP to Horn clauses, DSLP-H gives a little worse values on CLL and AUC-PR measure than DSLP-A does, while it is competitive to DMSP on these two measures. This loss comes from the fact that the space of Horn clauses is much narrower than the space of arbitrary clauses, therefore DSLP-A can find a set of more suitable clauses. However, for the predicate *WorkedUnder* (IMDB), DSLP-H spends more time than DSLP-A does to produce the final MLN. We can explain this situation by the fact that, in our method, the number of considered candidate clauses at each iteration depends on the number of candidate clauses at the previous one, therefore DSLP-H has to extend

Datasets →	UW-CSE			IMDB		
Systems	CLL	AUC-PR	RT	CLL	AUC-PR	RT
$DSLP_{L-BFGS}$	-0.013	0.389	**2.47**	-0.022	0.394	**0.19**
$DSLP_{P-SCG}$	**-0.094**	**0.451**	4.38	**-0.016**	**0.437**	0.31
$DSLP_{M3LN}$	-0.211	0.317	4.51	-0.027	0.363	0.26

Table 5.12: CLL, AUC-PR and RT (runtime in hour) results

many clauses at each iteration. The set of clauses in the final MLN of DSLP-H also justifies this remark, as it contains a lot of long Horn clauses (exactly 5-literal Horn clauses). For the medium and large datasets (UW-CSE and CORA), DSLP-H performed faster than DSLP-A. We can answer to question 2 that, in contrast to the fact that the whole space of Horn clauses is smaller than the one of arbitrary clauses, DSLP tends to consider more Horn candidate clauses at each iteration than it does with arbitrary clauses, more time is therefore consumed to evaluate all candidate clauses. An investigation to reduce the number of considered Horn clauses at each step would be an interesting point in our future work.

Table 5.12 reports the average CLL, AUC and runtime in hour (RT) of DSLP over test folds, performed with three different weight learning algorithms; L-BFGS, P-SCG and M3LN. DSLP with P-SCG gives the best CLL and AUC-PR results while DSLP with L-BFGS is the fastest version. The CLL and AUC-PR values of DSLP with L-BFGS are also acceptable compared to the ones with P-SCG despite a little loss. DSLP with M3LN produces more modest results because the M3LN was designed to maximize the F_1 score rather than CLL and AUC-PR (see in [Huynh & Mooney 2009] for detail). Based on these results we recommend to use DSLP with L-BFGS when we are looking for a balance over all the three factors (i.e. CLL, AUC-PR and runtime), otherwise to use DSLP with P-SCG to favor CLL and AUC-PR.

5.5 Related Works

The idea to encode information in database into a graph has been deployed largely in SRL [Getoor & Taskar 2007] including both the *directed* graph model and *undirected* graph model. Methods based on the directed graph model are, among others, the Relational Dependency Networks, the Probabilistic Relational Models and the Bayesian Logic Programs. Methods based on the *undirected* graph model are, among others, the Markov Logic Networks, the Relational Markov Networks. Amongst them, our Graph of Predicates is related to the class dependency directed graph in PRM, each node of which corresponds to an attribute of some class (each class can be viewed as a table in a relational database).

Concerning MLN structure learning, LHL [Kok & Domingos 2009] and LSM [Kok & Domingos 2010] also lift the database to an undirected graph, called the "lifted-graph", from which to search for paths in the lifted-graph in order to create candidate clauses. However, this graph is very different from the graph of predicates of our methods. In the lifted-graph, each node corresponds to a set of constants and each edge corresponds to a predicate. This graph is this built directly from links between ground atoms in the

database, while in the graph of predicates, is built directly, quickly from information on the types of arguments of the predicates. The size (number of nodes and edges) of the liftedgraph depends mostly on the database while it is "fixed and bounded" in our method. A path in the lifted-graph, based on relational-path-finding [Richards & Mooney 1992] or relational-cliché [Silverstein & Pazzani 1991], is really a list of shared-argument ground atoms built directly from the database and variabilized to form a clause. Our path is a list of links that contains only information about predicates and positions of shared-arguments. A heuristic variabilization technique must then be used to form a clause. Starting from binary clauses, GSLP extends a clause by considering the predicates (negative or positive) occurring in it and searching the graph for possible extensions (note that a predicate or its negation may not have the same possible extensions); therefore, it tends to add good links to a good clause to get better ones. LHL extends clauses by searching for all paths in the lifted-graph and then discards those having few true instantiations. LSM finds paths in each identified motif in the same manner as LHL does. This, more or less, still leads to a whole graph-search and counting the number of true instantiations is time-consuming when the database is large and literals have many groundings. For each path, LHL and LSM add all possible clauses with the signs of up to n literals flipped and compare their score against all possible sub-clauses. Representing also the negation of the predicates in the graph allows GSLP to handle in the same way negative or positive literals in a clause, and to extend clauses either by a positive or a negative literal depending on the corresponding gain.

5.6 Summary

The basic idea behind methods for MLN structure learning presented in this chapter is to encode information from the training database into a graph from which to search for candidate clauses. We present first the definition of a graph of predicates (GoP), each node of which corresponds to a predicate or its negation, and each edge of which corresponds to a possible link between two predicates (or a formula corresponding to this link). Depending on the purpose of learning (generative or discriminative), different labels are associated to the edges. Using this graph of predicates, an edge e is viewed as a binary clause bc and every edge incident to e is viewed as a good direction to lengthen bc, therefore every edge in the graph of predicates is extended gradually to produce longer and better candidate clauses. In other words, our methods do perform in a top-down strategy to narrow the search for candidate clauses within the graph of predicates, which is much faster than an exhaustive search in the space of clauses. The set of candidate clauses is finally considered in turn to learn the final MLN.

Based on this idea, we developed first the GSLP system for generative MLN structure learning. During the experiment process, we noticed several limiting points in GSLP, which have been overcome in the M-GSLP system. We also developed the DSLP system for the task of discriminative MLN structure learning. The experimental results showed that our systems can yield better results than the state-of-the-art algorithms for MLN structure learning. Table 5.13 resumes the different steps and components between these systems.

	GSLP	**Modified GSLP (MGSLP)**	**DSLP**
Measure (ms)	WPLL		CLL
Evaluation a clause c	c.weight, c.gain	c.gain = ms(temporary MLN) - ms(unit MLN) - penalty × |c| $c.mf = \|c.weight\| \times c.gain$	
Graph of predicates link-label	A link between two template atoms		
num-label	♯ of true instantiations	$c.mf = \|c.poids\| \times c.gain$	$\frac{\sharp \ of \ true \ instantiations}{\sharp \ of \ instantiations}$
node.weight	k times the average of its num-labels	NO	
Reduction (pruning edges)	$e(v_i, v_j).num\text{-}label < min(v_i.weight, v_j.weight)$	$e.mf \geq threshold_1$ and its number of instantiations $\geq threshold_2$	(e.num-label > nlt) or ((e.num-label < nlt) and (c.mf > mt))
Creating clauses	$(c.weight > threshold_{weight})$ and (c.gain > 0)	$c_{k+1}.mf > c_k.mf$	
Learning of the final MLN Order	Decreasing gain	Decreasing measure	
Pre-elimination	NO	sub-clauses, clauses composed of same predicates	
Candidate clauses	clauses of similar length	all clauses	

Table 5.13: A synthetic view of the different steps and components in GSLP, MGSLP and DSLP

Chapter 6

Conclusion and Future Work

Contents

Markov Logic Networks are a recent powerful SRL model that combines full first-order logic and Markov networks. MLNs attach weights to first-order clauses and view these as templates for the features of Markov networks. One main task concerning MLN is structure learning, i.e., learning a set of first-order clauses together with their weights from data. In this dissertation, we propose several methods to achieve this task.

6.1 Contributions of this Dissertation

The contributions of this dissertation consist of algorithms for MLN structure learning, which can be grouped into two classes: methods based on *propositionalization* in ILP and methods based on a *Graph of Predicates*.

MLN structure learning based on propositionalization

The basic idea in our methods is to build candidate clauses from sets of dependent variable literals, from which to learn the final MLN. In order to find such sets of dependent variable literals, we propose the use of propositionalization which transforms relational information expressed by shared ground atoms (by constants) in the database into boolean tables. These tables are then provided as contingency tables for test of dependent literals. Propositionalization is performed through two steps: building a set of variable literals corresponding to a target predicate and creating a boolean table corresponding to each target variable literal (of the target predicate).

We began by developing a first propositionalization technique, the basic idea of which is to divide the dataset into distinct groups, and then to use a variabilization technique to heuristically variabilize these groups in turn, from the largest to the smallest one. During the process of variabilization, variable literals are reused as much as possible, hence reducing the number of generated variable literals. Being given a set of variable literals, a boolean table is created for each target variable literal. Each table has a column corresponding to a variable literal, a row corresponding to a ground atom of the considering query predicate. Values in a row of a table express the relational information of ground atoms linked to the ground atom corresponding to this row. A heuristic process is used to fill values in such a table.

Based on this propositionnalization method, two algorithms for MLN structure learning have been developed: HGSM for generative learning and HDSM for discriminative learning. HGSM learns a set of weighted clauses for all predicates in the domain using the generative measure WPLL to choose clauses and the L-BFGS algorithm to learn weights. HDSM is based on a structure similar to HGSM but learns a set of weighted clauses for only a query predicate, using the CLL measure to choose clauses. Experiments show that both methods outperform the state-of-the-art structure learner for MLNs for all the datasets that we used.

We then developed the DMSP algorithm for MLN discriminative structure learning. In comparison to HDSM, DMSP brings three main differences. First, DMSP proposes a new propositionalization technique that differs from the first one in the process of creating variable literals. It is based on the idea that a lot of paths of connected ground atoms can be described by a single set of variable literals. The algorithm, therefore, variabilizes firstly a path to generate a corresponding set of variable literals, and this path is then used as a filter to ignore a lot of other paths. By this way, the set of variable literals is found much faster and we also prove that it is the smallest set to describe relations related to the query predicate in the database. Second, DMSP modifies the process of creating candidate clauses in order to generate a little more candidates. Finally, DMSP changes the order of candidate clauses to learn the final MLN.

MLN structure learning based on Graph of Predicates (GoP)

The basic idea under this second class consists in encoding information in the training database into a graph from which to search for candidate clauses. We introduce the new definition of Graph of Predicates, that is inspired from links between predicates in a domain and the coverage measure in ILP. For each link between two predicates, we define a formula corresponding to it. A Graph of Predicates, each node of which corresponds to a predicate or its negation and each edge of which corresponds to a possible link between two predicates, highlights the binary associations of predicates that share constants in term of the number of true instantiations in the database (of the corresponding formulas).

We implemented this idea in the GSLP algorithm to learn generatively a MLN structure from a relational database. GSLP performs a top-down strategy to narrow the search for candidate clauses within the Graph of Predicates, which is much faster than an exhaustive search in the space of clauses. During the experiment to evaluate GSLP, we realized several limitations related to the measure for choosing clauses, the strategy to reduce edges of Graph of Predicates and the methods to build a set of candidate clauses and to learn the final MLN. These limitations have been overcome in the Modified-GSLP (M-GSLP) learner.

We also developed the DSLP system for the task of discriminative MLN structure learning. In DSLP, we defined a new semantic of Graph of Predicates in order to adapt to the task of discriminative learning and accelerate the process of finding Horn clauses. Experiment results show that DSLP dominates the state-of-the-art discriminative MLN structure learners for several benchmark datasets.

6.2 Directions for Future Work

Directions for future work include the following:

- Firstly, all our methods used the closed world assumption that considers a ground atom as false when it is not in the dataset. We would like to apply the open world assumption in our algorithms in order to be able to consider only two separate sets of true and false examples, as real-world data is often collected.

- Concerning evaluation, the experimental results on several benchmark datasets show that our systems outperform the state-of-the-art learners for MLN structure learning. Our discriminative systems give also better results than ALEPH [Srinivasan 2003], the popular ILP system (experimental comparisons are reported in Appendix B). Especially, methods based on the Graph of Predicates have been more efficient when dealing with the largest of our datasets, CORA. We would like to apply them to larger and more complex domains with many relations such as the Web or medical domains in order to test their scalability and their ability to learn complex rules. We would also like to compare our learners to other systems in ILP, as well as other probabilistic, non-MLN approaches.

- Regarding propositionalization, our method implemented in DMSP is shown more suitable than the others for the task of MLN structure learning. We would like to apply it back to traditional ILP in order to exploit and evaluate this method in comparison to traditional ILP propositionalization methods. Moreover, in DMSP, creating variable literals and building tables are two separate processes. However, during the process of variable literal creation, the algorithm has to travel on a lot of g-chains which can be used in the latter process. An integration of these two processes to reuse all considered g-chains would be an interesting point to improve DMSP.

- In connection with variabilization techniques, in DMSP, a v-chain is created and then considered as a filter to ignore a lot of g-chains. However, as stated earlier, the algorithm has to search, more or less, in the space of ground atoms. We circumvented with this difficulty by limiting the number of literals per clause but this solution prevent us to catch longer relations. This difficulty is also solved by using Graph of Predicates. Unfortunately, while getting faster and longer clauses, our variabilization technique creates only one clause for each path, and thus might miss out several good clauses. We continue investigating on this problem, especially on the method to variabilize paths from the Graph of Predicates.

- Concerning performance, our generative methods rely on the L-BFGS algorithm to learn weights, which maximizes the PLL measure. This measure, however, does only capture dependencies between an atom and its Markov blanket. We plan to integrate the use of likelihood in order to deal with larger dependencies in the Markov networks. Besides, our methods are based on a generate-and-test strategy, which consumes time to evaluate clauses. A solution to discard clauses before evaluation would be useful. In addition, inference for evaluating CLL is based on MC-SAT or Lazy-MC-SAT. We plan to use new recent inference methods, as for instance

lifted inference [Milch *et al.* 2008, Singla & Domingos 2008] and *belief propagation* [Kersting *et al.* 2009, Nath & Domingos 2010] in order to save time for inferring.

- From the point of view of experiment, M-GSLP was shown to outperform the state of the art MLN structure learners on several benchmark datasets. Nevertheless, from the point of view of theory, only the complexity in the worst case was presented. This is the upper bound and M-GSLP hardly reaches this complexity for real-world datasets. Therefore, using this evaluation in the worst case to compare to other systems can not reflect exactly the performance of M-GSLP. We would like to analyze M-GSLP in an average case to be able to compare to other systems more theoretically and more thoroughly.

- Related to evaluation methodology, in our method, the discriminative CLL measure and the AUC-PR were used. These measures, as far as we know, have also been used in all the other methods for MLN generative structure learning. However, the CLL is obviously more suitable for the task of MLN discriminative structure learning. It would be useful to investigate an evaluation methodology more adapted to generative learning relying on a generative measures instead of the discriminative ones.

- Finally, lifted inference often exploits interchangeability or symmetry in the domain in order to reason as much as possible at a high-level. In our point of view, the high-level structure of Graph of Predicates could be useful when exploited by a technique of lifted inference. We would like to construct an inference algorithm combining techniques of lifted inference together with the Graph of Predicates.

Conclusion et Perspectives

Les réseaux logiques de Markov (MLNs), qui constituent l'une des approches les plus récentes de l'apprentissage statistique relationnel (SRL), reposent sur la combinaison de la logique du premier ordre avec les réseaux de Markov. Un réseau logique de Markov contient un ensemble de clauses pondérées du premier ordre servant de modèle pour construire des réseaux de Markov. L'une des tâches principales concernant les réseaux logiques de Markov est l'apprentissage de la structure, c'est-à-dire l'apprentissage d'un ensemble de clauses avec leurs poids à partir d'un jeu de données. Dans cette thèse, nous avons proposé plusieurs méthodes pour ce problème.

Contributions de cette thèse

Les contributions de ce mémoire sont des méthodes pour l'apprentissage de la structure d'un réseau logique de Markov tant générative que discriminant. Ces méthodes peuvent être divisées en deux classes : les méthodes reposant sur la propositionnalisation et celles reposant sur les Graphes des Prédicats.

Méthodes reposant sur la propositionnalization

L'idée de base de nos méthodes est de construire des clauses candidates pour apprendre le réseau logique de Markov final à partir d'ensembles de littéraux dépendants. Afin de trouver de tels ensembles de littéraux dépendants, nous proposons l'utilisation de la propositionnalisation, qui transforme l'information relationnelle représentée par des atomes clos dans la base de données en tableaux booléens, lesquels sont ensuite fournis comme tableaux de contingences à des algorithmes de test de dépendance conditionnelle. La propositionnalisation est réalisée en deux étapes: la construction d'un ensemble de littéraux dépendant liés à chaque prédicat à apprendre et la création d'un tableau booléen correspondant à chaque littéral.

Nous avons commencé par le développement d'une première technique de propositionnalisation. L'idée de base est ici de diviser le jeu de données en groupes distincts d'atomes connectés, puis d'utiliser une technique de variabilisation heuristique de ces groupes, les traitant les uns après les autres, du plus grand au plus petit. Pendant le processus de variabilisation, les littéraux sont réutilisés autant que possible, afin de réduire la taille de l'ensemble de littéraux générés. Ayant obtenu l'ensemble de littéraux, un tableau booléen est ensuite créé correspondant à chaque littéral du prédicat à apprendre. Chaque colonne correspond à un littéral, une ligne correspondant à un atome clos du prédicat cible. Les valeurs dans chaque ligne du tableau expriment des atomes clos connectés à l'atome clos de la ligne. Une technique heuristique est utilisée pour remplir les valeurs dans un tel tableau.

Deux algorithmes d'apprentissage de la structure d'un réseau logique de Markov reposant sur cette approche ont été développés : HGSM pour l'apprentissage génératif et HDSM pour l'apprentissage discriminant. HGSM cherche un ensemble de clauses pondérées pour tous les prédicats du domaine en utilisant la mesure générative WPLL,

pour choisir les clauses, et l'algorithme L-BFGS pour apprendre les poids. HDSM a la même structure que HGSM, mais cherche un ensemble de clauses pondérées pour seulement un prédicat cible, en utilisant la mesure discriminante CLL pour choisir les clauses. Les expérimentations montrent que ces deux méthodes dépassent les autres systèmes de l'état de l'art pour l'apprentissage de la structure d'un réseau logique de Markov, sur les jeux de données classiques que nous avons utilisés.

Nous avons ensuite développé l'algorithme DMSP pour l'apprentissage discriminant de la structure d'un réseau logique de Markov. En comparaison avec HDSM, DMSP a principalement trois différences. Tout d'abord, DMSP repose sur une seconde technique de propositionnalisation, qui diffère de la première dans la manière de créer l'ensemble de littéraux. Cette deuxième technique repose sur l'idée que beaucoup de chemins d'atomes connectés peuvent être décrits par un seul ensemble de littéraux avec seulement des variables. L'algorithme, par conséquent, variabilise d'abord un chemin pour générer un ensemble de littéraux avec seulement des variables, utilisé ensuite comme un filtre pour ignorer beaucoup d'autres chemins. De cette manière, l'ensemble des littéraux est trouvé beaucoup plus rapidement et nous avons également prouvé qu'il est le plus petit ensemble capable de décrire les relations issues du prédicat à apprendre dans le jeu de données sous certaines conditions. DMSP modifie le processus pour créer des clauses candidates afin d'en générer un peu plus. Enfin, DMSP change l'ordre des clauses candidates pour apprendre le réseau logique de Markov final.

Méthodes reposant sur des Graphes des Prédicates

L'idée de base est ici de coder les informations du jeu de données dans un graphe de prédicats, à partir duquel, la recherche des clauses est effectuée. Nous introduisons la définition de Graphe des Prédicats, inspirée des liens entre les prédicats et de la notion de couverture en programmation logique inductive. Un Graphe des Prédicats, dont chaque nœud correspond à un prédicat ou à sa négation et chaque arête correspond à un lien possible entre les deux prédicats, souligne les associations binaires entre les prédicats en terme de nombre d'instanciations vraies (de la formule correspondante) du jeu de données.

Nous avons ensuite développé l'algorithme GSLP pour l'apprentissage génératif de la structure d'un MLN. GSLP utilise une stratégie descendante pour limiter la recherche des clauses candidates dans le Graphe des Prédicats, ce qui est beaucoup plus rapide qu'une recherche exhaustive dans l'espace de clauses. Lors des expérimentations menées pour évaluer GSLP, nous avons détecté plusieurs limites concernant le choix des clauses, la réduction du nombre d'arêtes du Graphe des Prédicats et la construction de l'ensemble de clauses candidates. Ces problèmes ont été résolus dans une version modifiée de GSLP, intitulée M-GSLP.

Nous avons également développé le système DSLP pour la tâche d'apprentissage discriminant de la structure d'un réseau logique de Markov. Dans DSLP, nous avons défini une nouvelle sémantique du Graphe des Prédicats afin de l'adapter à la tâche d'apprentissage discriminant et d'accélérer le processus de recherche des clauses de Horn. Les résultats de l'expérimentation montrent que DSLP dépasse les systèmes de l'état de l'art pour l'apprentissage discriminant de la structure d'un réseau logique de Markov, sur les jeux de données classiques.

Perspectives

De nombreux problèmes restent ouverts et feront l'objet de nos recherches à venir.

- D'abord, toutes nos méthodes ont utilisé l'hypothèse du monde fermé qui considère un atome clos comme faux s'il n'est pas dans le jeu de données. Nous souhaitons appliquer l'hypothèse du monde ouvert dans notre algorithme afin de pouvoir considérer deux ensembles séparés d'exemples vrais et faux, plus représentatif de jeux de données réels.

- En ce qui concerne l'évaluation, les résultats des expérimentations montrent que nos systèmes dépassent les systèmes de l'état de l'art pour apprendre la structure d'un réseau logique de Markov. Nos systèmes discriminants donnent également des résultats meilleurs qu'ALEPH [Srinivasan 2003], système populaire en ILP (les comparaisons sont données dans l'annexe B). Les méthodes reposant sur les Graphes des Prédicats sont avantageuses pour des jeux de données de grande taille, comme CORA. Nous souhaitons les appliquer dans des domaines plus grands et plus complexes avec de nombreuses relations tels que le Web ou le domaine médical, afin de tester leur capacité à apprendre des clauses complexes. Nous souhaitons également comparer nos systèmes avec d'autres systèmes d'ILP ainsi que des approches probabilistes autres que les réseaux logiques de Markov.

- Au niveau de la propositionnalisation, la méthode mise en œuvre dans DMSP s'est montrée plus adaptée que les autres à l'apprentissage de la structure d'un réseau logique de Markov. Nous souhaitons l'appliquer à l'ILP classique afin d'exploiter et d'évaluer cette méthode par rapport aux méthodes traditionnelles de propositionnalisation. De plus, dans DMSP, la construction d'un ensemble de littéraux et la création des tableaux sont deux processus séparés. Cependant, durant le processus de construction des littéraux, l'algorithme doit parcourir beaucoup de g-chaînes qui peuvent être utilisées dans le second processus. Une intégration de ces deux processus pour réutiliser toutes les g-chaînes parcourues pourrait améliorer la performance de DMSP.

- En liaison avec les techniques de variabilisation dans DMSP, une v-chaîne est créée puis considérée comme filtre pour ignorer plusieurs g-chaînes. Toutefois, comme indiqué précédemment, l'algorithme peut parcourir une zone assez vaste de l'espace des atomes clos. Nous avons contourné cette difficulté en limitant le nombre de littéraux par clause, mais ce n'est plus applicable idée si nous voulons extraire des relations plus longues. Cette difficulté est également résolue en utilisant le Graphe des Prédicats. Malheureusement, bien que des clauses plus longues soient trouvées plus rapidement, notre technique de variabilisation crée seulement une clause par chemin, et pourrait ainsi manquer plusieurs bonnes clauses. Nous continuons d'étudier ce problème et notamment la méthode de variabilisation dans le Graphe des Prédicats.

- En ce qui concerne la performance, nos méthodes génératives utilisent l'algorithme L-BFGS pour apprendre les poids, par maximisation de la mesure PLL. Cette mesure, cependant, ne tient compte que des dépendances entre un atome et sa couverture de

Markov. Nous envisageons d'intégrer l'utilisation de la vraisemblance (likelihood) afin de pouvoir traiter des dépendances plus larges dans les réseaux de Markov. Nos méthodes sont basées sur la stratégie générer-et-tester, qui nécessite beaucoup de temps pour évaluer les clauses. Il faudrait des outils pour exclure les clauses avant de les évaluer. De plus, nous prévoyons d'utiliser d'autres méthodes d'inférence récentes au lieu de MC-SAT ou Lazy-MC-SAT, par exemple, *inférence liftée* (lifted inference) [Milch *et al.* 2008, Singla & Domingos 2008] ou la *propagation de croyance* (belief propagation) [Kersting *et al.* 2009, Nath & Domingos 2010] afin d'accélérer l'inférence.

- Du point de vue de l'expérimentation, M-GSLP surpasse les systèmes de l'état de l'art pour apprendre la structure d'un réseau logique de Markov, sur les jeux de données classiques. Néanmoins, du point de vue de la théorie, la seule complexité au pire a été présentée. Il s'agit d'une borne supérieure que M-GSLP n'atteint sans doute jamais pour des jeux de données réels. Par conséquent, ce outére de comparaison ne reflète pas exactement de la performance de M-GSLP. Nous souhaitons analyser la complexité en moyenne de nos méthodes, notamment de M-GSLP, pour pouvoir les comparer avec d'autres systèmes plus théoriquement et plus profondément.

- Pour l'évaluation, dans notre méthode, comme dans toutes les autres méthodes génératives d'apprentissage de la structure d'un réseau logique de Markov, la mesure discriminante CLL et AUC-PR ont été utilisés. Ces mesures, évidemment, sont plus appropriées pour la tâche d'apprentissage discriminant de la structure de réseau logique de Markov. Il serait utile d'envisager une méthodologie d'évaluation générative reposant sur une mesure générative.

- Enfin, l'inférence liftée (lifted inference) exploite souvent l'interchangeabilité ou la symétrie dans un domaine afin de raisonner autant que possible à haut niveau. De notre point de vue, la structure de haut niveau du Graphe des Prédicats pourrait être utile pour une technique d'inférence liftée. Nous souhaitons construire un algorithme d'inférence combinant ces deux techniques.

Appendix A

Evaluation Metrics

This appendix surveys the metrics of Receiver Operating Characteristic (ROC) curves and Precision-Recall (PR) curves.

A.1 Classifier Performance

We begin by considering binary classification problems using only two classes *true* and *false*. A discrete (binary) classifier labels examples as either positive or negative. The decision made by the classifier can be represented in a confusion matrix or contingency table. The confusion matrix has four categories: True positives (TP) are examples correctly labeled as positives. False positives (FP) refer to negative examples incorrectly labeled as positive. True negatives (TN) correspond to negatives correctly labeled as negative. Finally, false negatives (FN) refer to positive examples incorrectly labeled as negative.

Figure A.1(a) shows a confusion matrix. Given the confusion matrix, metrics are defined as shown in Figure A.1(b). *Accuracy* is the percentage of classifications that are correct. The True Positive Rate (TPR) measures the fraction of positive examples that are correctly labeled. The False Positive Rate (FPR) measures the fraction of negative examples that are misclassified as positive. Recall is the same as TPR, whereas Precision measures the fraction of examples classified as positive that are truly positive.

A.2 ROC and PR Curves

The confusion matrix can be used to construct a point in either ROC space or PR space. In ROC space, one plots the False Positive Rate (FPR) on the *x-axis* and the True Positive

	actual positive	actual negative
predicted positive	TP	FP
predicted negative	FN	TN

(a) Confusion Matrix

Recall	=	$\frac{TP}{TP+FN}$
Precision	=	$\frac{TP}{TP+FP}$
True Positive Rate	=	$\frac{TP}{TP+FN}$
False Positive Rate	=	$\frac{FP}{FP+TN}$
Accuracy	=	$\frac{TP+TN}{TP+FP+TN+FN}$

(b) Definitions of metrics

Table A.1: Common machine learning evaluation metrics

Rate (TPR) on the *y-axis*. In PR space, one plots Recall on the *x-axis* and Precision on the *y-axis* (or inverse).

Some probabilistic classifiers, such as Naive Bayes classifiers or neural networks, give probabilities (degrees) of examples, where the bigger the probability, the more likely the example to be true. Such a probabilistic classifier can be used with a threshold to produce a binary classifier: if an output of the classifier is above the threshold, the classifier decides true, otherwise false. Each threshold value produces a point in ROC space or PR space. Conceptually, we can vary a threshold from $-\infty$ to $+\infty$ and trace a curve through ROC space or PR space. Computationally, this is a poor way of generating a ROC curve or PR curve. We refer the reader to [Fawcett 2003] for a more efficient and careful computational method.

Precision-Recall curves have been cited as an alternative to ROC curves for tasks with a large skew in the class distribution [Bockhorst & Craven 2005, Singla & Domingos 2005, Kok & Domingos 2005, Davis & Goadrich 2006], which captures the effect of the large number of negative examples on the performance of algorithm. An important difference between ROC space and PR space is the visual representation of the curves. The ideal in ROC space is to be in the upper-left-hand corner while the ideal in PR space is to be in the upper-right-hand corner.

A.3 Area Under the Curve

A ROC curve or a PR-curve describes the performance of a classifier in a two-dimensional space. To be more intuitive, we may want to reduce ROC performance to a single scalar to compare classifiers. Often, the Area Under the Curve (AUC) over either ROC or PR space [Bradley 1997] is used for this purpose.

The area under the ROC curve, AUC-ROC (respectively AUC-PR), can be calculated by using the trapezoidal areas created between each ROC point (respectively each precision point). Since the AUC-ROC or AUC-PR is a portion of the area of the unit square, their values will always be between 0 and 1.

By including the intermediate interpolated PR points, [Davis & Goadrich 2006] can use the composite trapezoidal method to approximate the area under the PR curve (AUC-PR). The AUC-PR value for a random classifier is equal to $\frac{TP+FN}{TP+FP+TN+FN}$, as this is the expected precision for classifying a random sample of examples as positive. We use a tool provided in [Davis & Goadrich 2006] to compute AUC-PR values in our experiments.

Appendix B

Experimental Comparison to ILP

Contents

We have presented so far our methods for MLN structure learning in which each system improves the performance compared to the previous developed ones. However, for most of methods, we have just evaluated on the CLL and AUC-PR measures and compared to other MLN learners. It is necessary to see how our methods perform in comparison to at least one non-MLN system. In this chapter, we thus present an experimental comparison between our methods and an ILP system.

B.1 Systems and Datasets

Concerning comparisons between MLN methods and ILP methods, as far as we know, there exists:

- The work of [Richardson & Domingos 2006] in which they compare a MLN structure learning algorithm to the CLAUDIEN [De Raedt & Dehaspe 1997] system.

- The work of [Huynh & Mooney 2008] in which they compare an Aleph-based method for discriminative MLN structure learning against ALEPH and BUSL. This method uses ALEPH to learn a set of initial clauses and then learns a discriminative MLN from this set using the L_1-regulation.

These experiment results show that generative MLN structure learners outperformed CLAUDIEN. However, for discriminative MLN structure learning, the result in [Huynh & Mooney 2008] is not completely convincing because their discriminative MLN learner is an ALEPH-based system. In other words, the system in [Huynh & Mooney 2008] performs refinement of clauses output by ALEPH, hence it obviously improves the performance of ALEPH. In this chapter, we conduct experiments to compare our discriminative MLN structure learners against ALEPH.

We used the classical ALEPH command **induce** which scores a clause based only on its coverage of currently uncovered positive examples. Besides, Aleph also has a wide variety of learning parameters, amongst which we used some major ones as follows:

- **Search strategy:** Aleph provides several search strategies including the breadth-first search, depth-first search, iterative beam search, iterative deepening, and heuristic methods requiring an evaluation function. We used the default breadth-first search.
- **Evaluation function:** We used the default evaluation function in Aleph is the coverage, which is defined as the number of positive examples covered by the clause minus the number of negations.
- **Clause length:** This parameter defines the size of a particular clause. We limited this length to 5, which is similar to the maximum number of literals, given as parameters of our discriminative systems.
- **Minimum accuracy:** This is used to put a lower bound on the accuracy of the clauses learned by Aleph. The accuracy of a clause has the same meaning as precision. We set this parameter to *0.5*.
- **Minimum positive:** The minimum number of positive examples covered by an acceptable clause was set to *1*.

For the other parameters, we used the default values in ALEPH.

In order to find out how ALEPH performs against our discriminative systems, we reused the three dataset IMDB, UW-CSE and CORA (described in Subsection 4.2.4.1) for the following reasons:

- They have been used to evaluate all our discriminative systems for learning both Horn and arbitrary clauses. IMDB is a small dataset, UW-CSE is medium and CORA is a quite large dataset.
- CORA contains both true and false ground atoms which is suitable for ALEPH to find a set of Horn clauses that covers the most number of positive examples and the least number of negative examples. Four predicates *SameAuthor*, *SameTitle*, *SameVenue* and *SameBib* (with an increasing number of true ground atoms) are learned separately; the number of true ground atoms in the dataset of *SameBib* is the biggest (approximately 50000 examples).
- The closed-world assumption is used for IMDB and UW-CSE. ALEPH learns clauses for the predicates *WorkedUnder* (IMDB) and *AdvisedBy* (UW-CSE).

It must be noticed that the above configuration of ALEPH comes from the fact that we first performed ALEPH 5 times with different configurations for the predicate *SameBib* (CORA) on only one fold, then chose the one that gives the best accuracy. We compare the results obtained by ALEPH to only the Horn-clause versions of HDSM, DMSP and DSLP.

B.2 Methodology

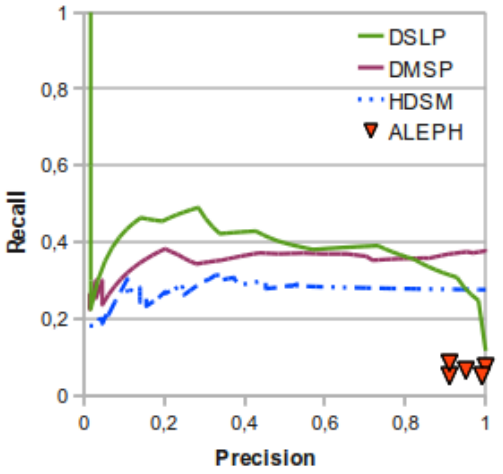
A 5-fold cross-validation is used to evaluate HDSM, DMSP and DSLP for all these datasets and the average CLL and AUC-PR values over test folds are reported for each predicate

(see results in Subsections 4.2.6.2, 4.3.4.2 and 5.4.3.2). Note that we applied our discriminative learners to perform with both Horn clauses and arbitrary clauses. Because ALEPH only looks for Horn clauses, we take only results of Horn clause versions for comparisons. Unfortunately, it is not easy to compute AUC-PR values as well as to plot PR-curves for ALEPH because the output of ALEPH is just the numbers of true positives, false positives, true negatives and false negatives, without any probability information for each true/false ground atom. To plot PR curves for ALEPH and to compute AUC-PR values, in our knowledge, there exists the method based on the use of bagging for ILP investigated by [Breiman 1996], and then the method based on ensembles with different seeds [Dutra *et al.* 2002], which has been shown much better than the former. The idea underlying these methods is to use different classifiers to learn and to combine their results by a *voting threshold*. ALEPH, performing with different seeds, will give different results; that can be viewed as results of different classifiers. The number of classifiers is recommended to be greater than 25 and [Dutra *et al.* 2002] used 100 in their experiments. They then had to use a *turning phase* to find out the best threshold for voting and the best value of parameter *minacc* for ALEPH. These learned parameters (*minacc* and *voting threshold*) were then used for evaluating on the test fold. These methods, thereby, have to create many theories (hence launch ALEPH many times). For example, in the experiments using "different seeds" of [Dutra *et al.* 2002], for a fold, they have to create *6000* theories for turning and *300* for evaluating. Applying this method to compute a PR curve for ALEPH on IMDB, UW-VSE and CORA requires a lot of times because of the fact that it takes approximately a half day each time on CORA or approximately *15* minutes each time on IMDB. For the time being, we therefore only run ALEPH *5* times for each fold of cross-validation, each time with *different seeds* (by changing the order of training examples) so that ALEPH can generates a new theory. Result of each run will be used to plot a point in the space of precision recall. We compare relatively these points to PR curves of our methods.

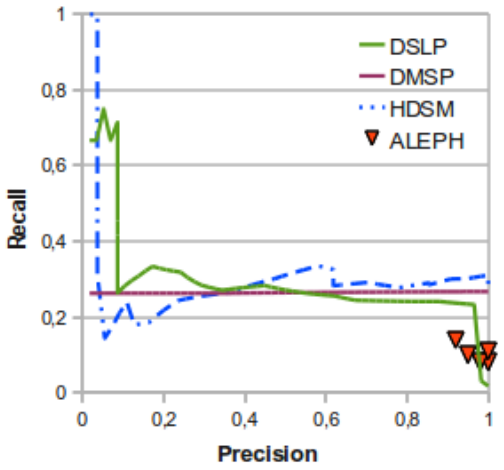
B.3 Results

For each predicate and for each test fold we plot the PR curves generated respectively by HDSM, DMSP and DSLP. Concerning ALEPH, for each fold (learning on *4* parts and testing on the remain), we draw 5 points, each point corresponds to a pair of precision, recall output by ALEPH. Figures B.1, B.2, B.3, B.4, B.5 and B.6 shows respectively the PR-curves output by HDSM, DMSP and DSLP and PR-points output by ALEPH for the query predicates on all *5* test folds. We can see that all the PR points from ALEPH lie entirely under every curve and ALEPH often gives the same PR points for several different seeds. The PR points from ALEPH distribute near the value corresponding to *precision* = *1* because ALEPH tries to find a set of clauses that covers most of positive ground atoms and few of negative ones. These results mean that the performance of ALEPH in terms of precision-recall is not better than the performance of our discriminative MLN structure learners. It must be noticed that two propositionalization methods were respectively implemented for HDSM and DMSP. All the PR-curves from DMSP are over the ones from HDSM. It means that our propositionalization method in DMSP is much better than the first one in HDSM. We will further investigate in the properties of this method.

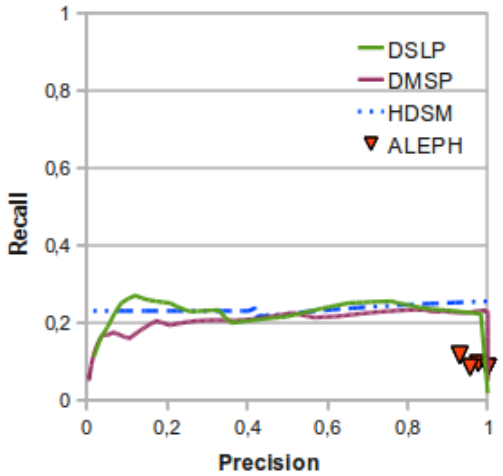
Concerning the number of learned clauses, ALEPH always brings more clauses than the others. For example, for the predicate *SameBib* on the first test fold, ALEPH returned *68* clauses while DSLP gave 10 clauses and DMSP generated *8* clauses. Detail of these clauses can be found in Appendix C of this dissertation. One reason for that is because ALEPH always tries to generate new clauses in order to cover more positive clauses while we try to prune some clauses at the last step of our methods in order to improve the measure (i.e. the CLL measure).



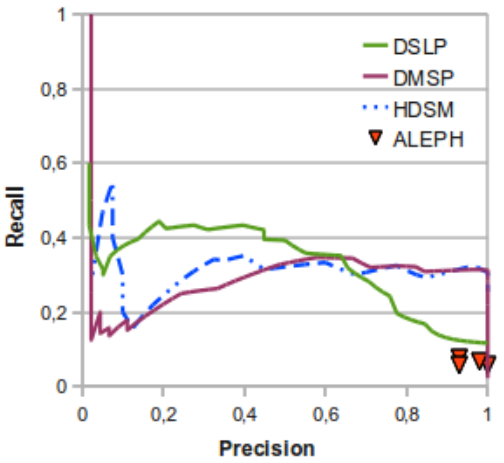
(a) Fold 1



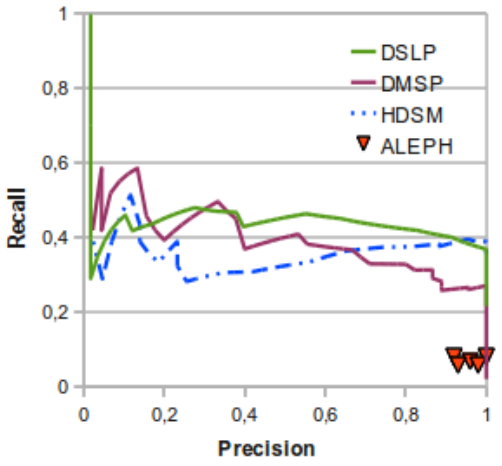
(b) Fold 2



(c) Fold 3

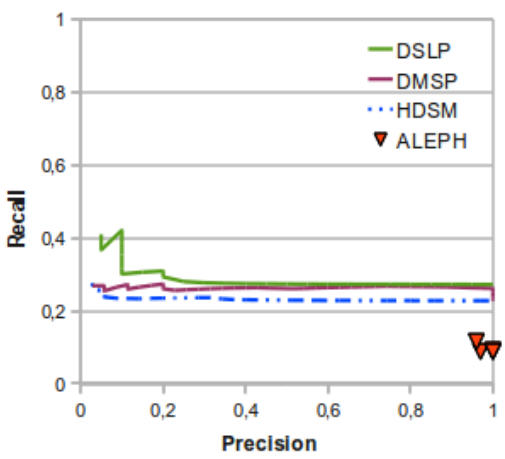


(d) Fold 4

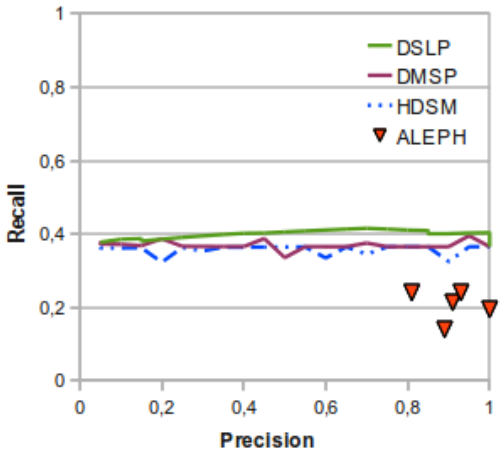


(e) Fold 5

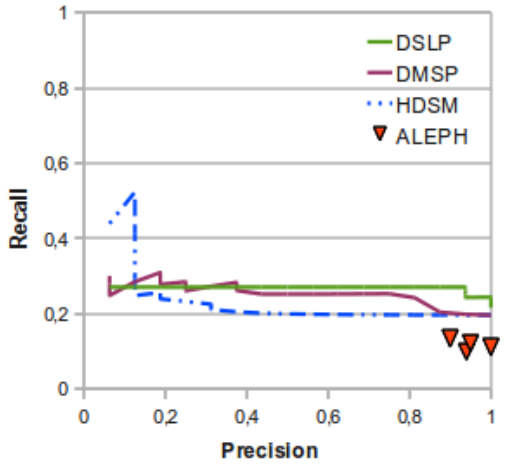
Figure B.1: Results for the predicate *WorkedUnder* in IMDB



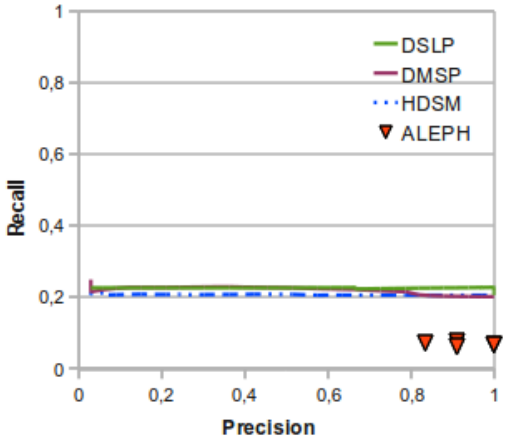
(a) Fold 1



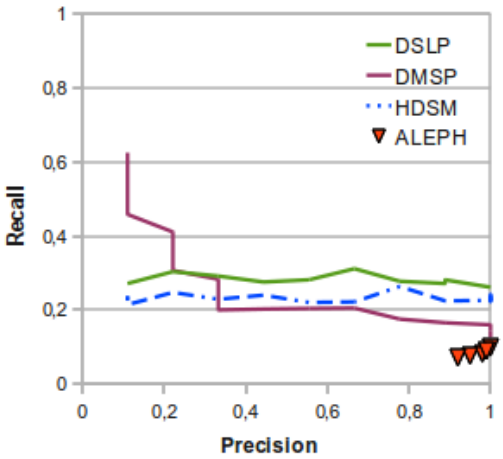
(b) Fold 2



(c) Fold 3



(d) Fold 4



(e) Fold 5

Figure B.2: Results for the predicate *AdvisedBy* in UW-CSE

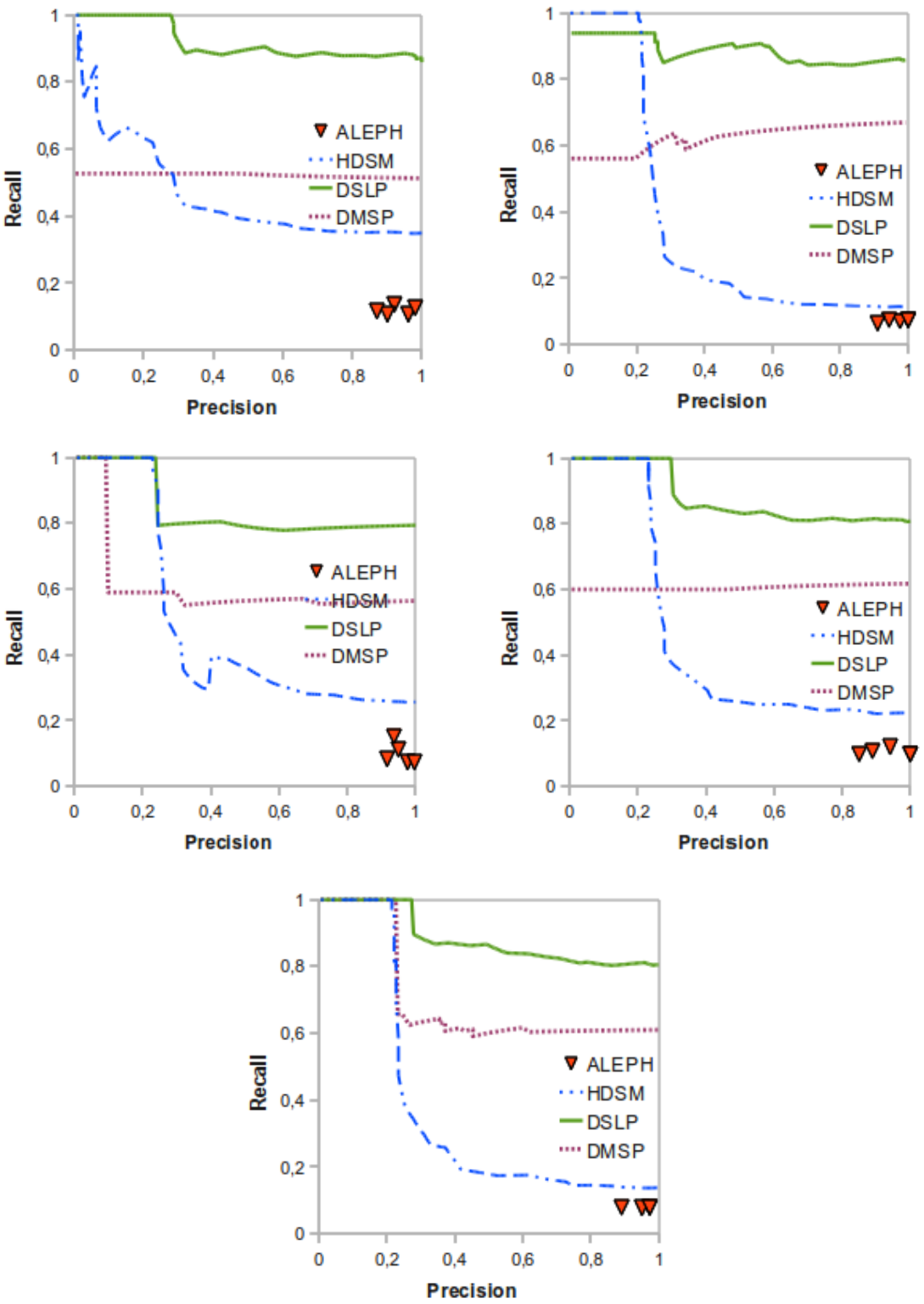


Figure B.3: Results for the predicate *SameAuthor* in CORA

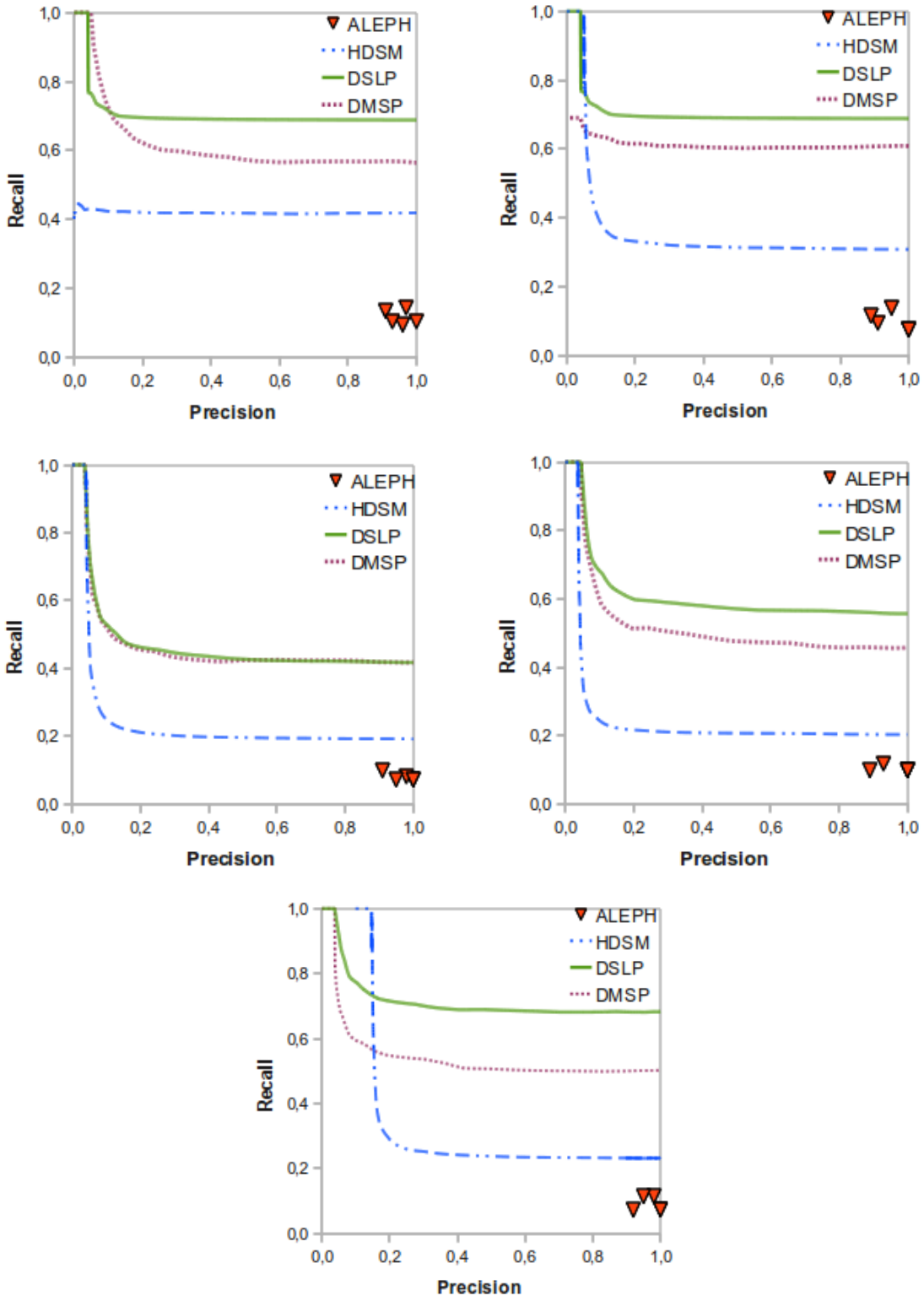
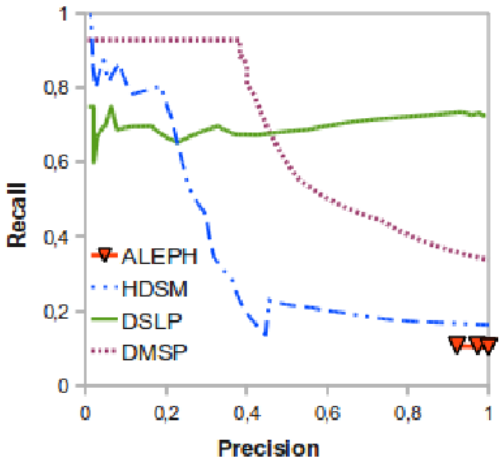
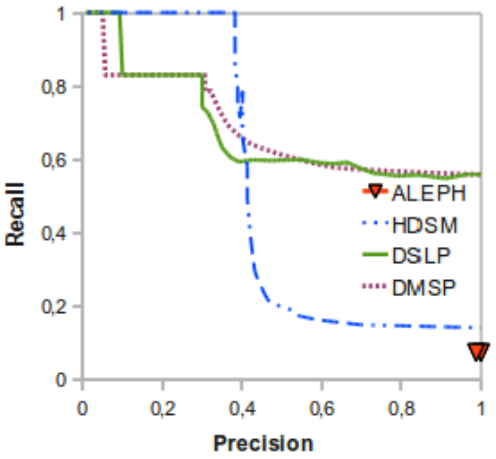


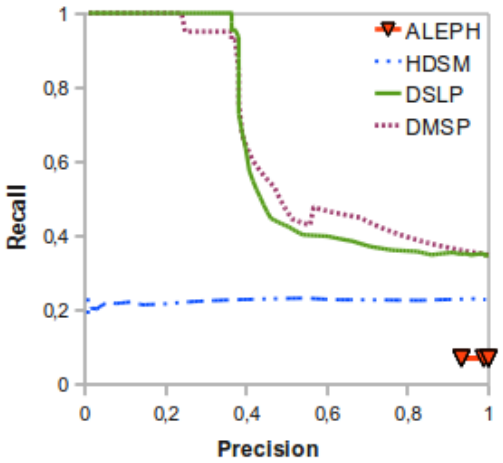
Figure B.4: Results for the predicate *SameBib* in CORA



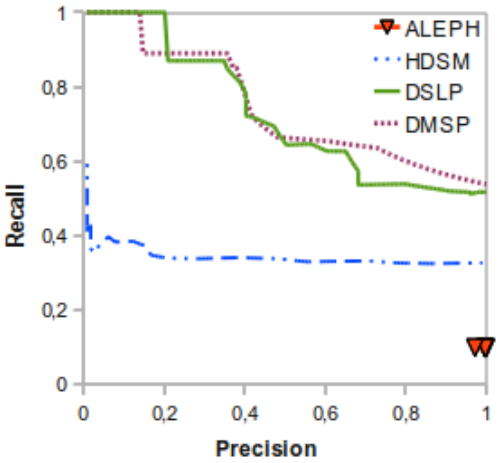
(a) Fold 1



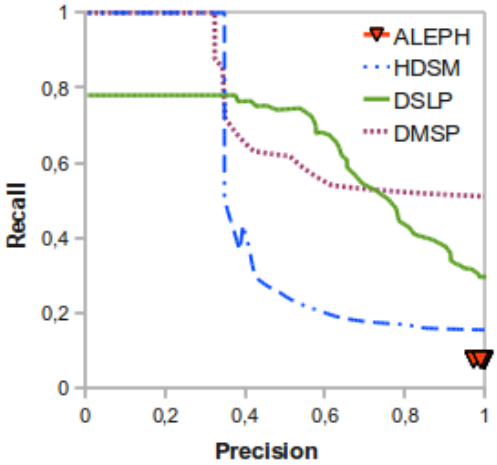
(b) Fold 2



(c) Fold 3



(d) Fold 4



(e) Fold 5

Figure B.5: Results for the predicate *SameTitle* in CORA

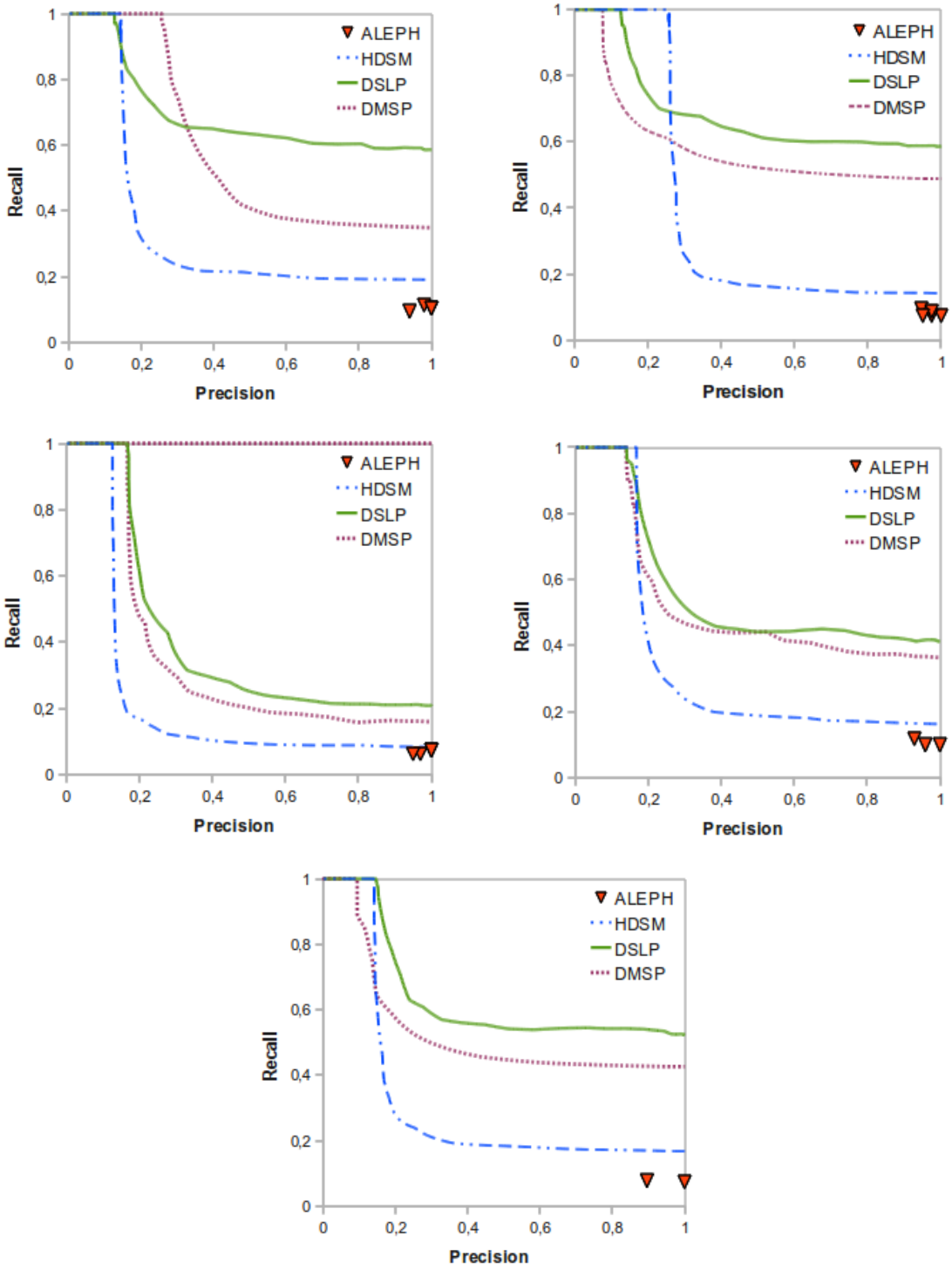


Figure B.6: Results for the predicate *SameVenue* in CORA

Appendix C

Clauses Learned by Discriminative Systems

Clauses learned by discriminative systems ALEPH, DSLP, DMSP, HDSM for the predicate *SameBib* (CORA dataset) on one fold.

ALEPH

samebib(A,B) :- author(B,C), title(B,D).
samebib(A,B) :- title(B,C), title(A,C).
samebib(A,B) :- title(B,C), title(D,C), author(D,E).
samebib(A,B) :- author(B,C), venue(B,D).
samebib(A,B) :- author(B,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(D,C), title(D,E), venue(D,F).
samebib(A,B) :- author(A,C), venue(A,D), venue(E,D), title(E,F).
samebib(A,B) :- title(B,C), title(A,C).
samebib(A,B) :- title(A,C), title(D,C), author(D,E), venue(A,F).
samebib(A,B) :- author(A,C), author(D,C), title(D,E), haswordauthor(C,F).
samebib(A,B) :- author(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- title(B,C), title(A,C), title(D,C), venue(D,E).
samebib(A,B) :- venue(B,C), haswordvenue(C,D), haswordtitle(E,D), venue(A,C).
samebib(A,B) :- author(B,C), venue(B,D), haswordvenue(D,E), haswordtitle(F,E).
samebib(A,B) :- author(B,C), haswordauthor(C,D), haswordtitle(E,D).
samebib(A,B) :- author(A,C), venue(A,D), haswordvenue(D,E), haswordtitle(F,E).
samebib(A,B) :- title(B,C), title(A,C), venue(A,D).
samebib(A,B) :- venue(B,C), venue(A,C), haswordvenue(C,D), haswordtitle(E,D).
samebib(A,B) :- author(A,C), title(B,D), title(A,D).
samebib(A,B) :- venue(B,C), venue(A,C), haswordvenue(C,D), haswordtitle(E,D).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D).
samebib(A,B) :- author(B,C), title(B,D), title(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- venue(B,C), venue(A,C), haswordvenue(C,D), haswordtitle(E,D).
samebib(A,B) :- title(B,C), title(A,C), haswordtitle(C,D), haswordauthor(E,D).
samebib(A,B) :- venue(B,C), venue(A,C), haswordvenue(C,D), haswordtitle(E,D).
samebib(A,B) :- venue(B,C), venue(A,C), haswordvenue(C,D), haswordtitle(E,D).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), haswordtitle(C,D), haswordauthor(E,D).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).

samebib(A,B) :- author(A,C), haswordauthor(C,D), haswordtitle(E,D), haswordvenue(F,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- author(B,C), author(A,C), title(B,D), title(A,D).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), haswordtitle(C,D), haswordauthor(E,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), haswordtitle(C,D), haswordauthor(E,D).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,C), title(B,D), title(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- author(B,C), author(A,C), title(B,D), title(A,D).
samebib(A,B) :- author(B,C), author(A,C), title(B,D), title(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- author(B,C), author(A,C), title(B,D), title(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- author(B,C), author(A,C), title(B,D), title(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).
samebib(A,B) :- author(B,C), author(A,C), title(B,D), title(A,D).
samebib(A,B) :- author(B,C), author(A,D), title(B,E), title(A,E).
samebib(A,B) :- title(B,C), title(A,C), venue(B,D), venue(A,D).

DSLP

SameBib(a1,a1) :- Author(a1,a2), Author(a1,a3)
SameBib(a1,a1) :- Title(a1,a2), Title(a1,a3)
SameBib(a1,a1) :- Venue(a1,a2), Venue(a1,a3)
SameBib(a1,a3) :- Title(a1,a2), Title(a3,a4), Venue(a1,a5)
SameBib(a3,a1) :- Author(a1,a2), Venue(a3,a4)
SameBib(a1,a4) :- Author(a1,a2), Title(a1,a3), Title(a4,a5)
SameBib(a1,a3) :- Author(a1,a2), Title(a3,a4), Venue(a1,a5)

SameBib(a1,a3) :- Author(a1,a2), Author(a3,a4), Title(a3,a5)
SameBib(a1,a3) :- Title(a1,a2), Venue(a3,a4)
SameBib(a3,a1) :- Title(a1,a2), Venue(a1,a5), Venue(a3,a4)

DMSP
SameBib(a1,a1) :- Author(a1,a2), Title(a1,a3)
SameBib(a1,a1) :- Venue(a1,a2), Venue(a1,a3)
SameBib(a1,a1) :- Author(a1,a2), Author(a1,a3)
SameBib(a1,a3) :- Title(a1,a2), Title(a3,a4)
SameBib(a1,a3) :- Author(a1,a2), Title(a3,a4), Venue(a1,a5)
SameBib(a1,a4) :- Author(a1,a2), Venue(a1,a3), Venue(a4,a5)
SameBib(a3,a1) :- Title(a1,a2), Venue(a1,a5), Venue(a3,a4)
SameBib(a1,a3) :- Author(a1,a2), Author(a3,a4), Title(a3,a5)

HDSM
SameBib(a1,a1) :- Venue(a1,a2)
SameAuthor(a1,a1) :- HasWordAuthor(a1,a2)
SameAuthor(a2,a3) :- Author(a1,a2), SameAuthor(a3,a2)
SameTitle(a2,a3) :- Title(a1,a2), SameTitle(a3,a2)

Bibliography

[Agresti 2002] A. Agresti. Categorical data analysis. Wiley Series in Probability and Statistics. Wiley-Interscience, 2nd édition, 2002.

[Alphonse & Rouveirol 1999] Érick Alphonse and Céline Rouveirol. *Selective Propositionalization for Relational Learning.* In PKDD, pages 271–276, 1999.

[Alphonse & Rouveirol 2000] Érick Alphonse and Céline Rouveirol. *Lazy Propositionalisation for Relational Learning.* In ECAI, pages 256–260, 2000.

[Anderson *et al.* 2002] Corin R. Anderson, Pedro Domingos and Daniel S. Weld. *Relational Markov Models and their Application to Adaptive Web Navigation.* In Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery and data mining, pages 143–152, New York, NY, USA, 2002. ACM.

[Arias *et al.* 2007] Marta Arias, Roni Khardon and Jérôme Maloberti. *Learning Horn Expressions with LOGAN-H.* J. Mach. Learn. Res., vol. 8, pages 549–587, December 2007.

[Besag 1975] Julian Besag. *Statistical Analysis of Non-Lattice Data.* Journal of the Royal Statistical Society. Series D (The Statistician), vol. 24, no. 3, pages 179–195, 1975.

[Biba *et al.* 2008a] Marenglen Biba, Stefano Ferilli and Floriana Esposito. *Discriminative Structure Learning of Markov Logic Networks.* In ILP '08: Proceedings of the 18th international conference on Inductive Logic Programming, pages 59–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[Biba *et al.* 2008b] Marenglen Biba, Stefano Ferilli and Floriana Esposito. *Structure Learning of Markov Logic Networks through Iterated Local Search.* In Proceeding of the 2008 conference on ECAI 2008, pages 361–365, Amsterdam, The Netherlands, 2008. IOS Press.

[Bishop 2006] Christopher M. Bishop. Pattern recognition and machine learning (information science and statistics). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Bockhorst & Craven 2005] Joseph Bockhorst and Mark Craven. *Markov networks for detecting overlapping elements in sequence data.* In Neural Information Processing Systems 17 (NIPS. MIT Press, 2005.

[Bradley 1997] Andrew P. Bradley. *The use of the area under the ROC curve in the evaluation of machine learning algorithms.* Pattern Recognition, vol. 30, pages 1145–1159, 1997.

[Braz *et al.* 2005] Rodrigo De Salvo Braz, Eyal Amir and Dan Roth. *Lifted first-order probabilistic inference.* In In Proceedings of IJCAI-05, 19th International Joint Conference on Artificial Intelligence, pages 1319–1325. Morgan Kaufmann, 2005.

[Breiman 1996] Leo Breiman. *Bagging Predictors.* In Machine Learning, pages 123–140, 1996.

[Bromberg *et al.* 2006] Facundo Bromberg, Alicia Carriquiry, Vasant Honavar, Giora Slutzki and Leigh Tesfatsion. *Efficient Markov Network Structure Discovery using Independence Tests.* In In SIAM International Conference on Data Mining, 2006.

[Collins 2002] Michael Collins. *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, EMNLP '02, pages 1–8, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[Davis & Goadrich 2006] Jesse Davis and Mark Goadrich. *The Relationship between Precision-Recall and ROC Curves.* In ICML '06: Proceedings of the 23rd international conference on Machine learning, pages 233–240, New York, NY, USA, 2006. ACM.

[De Raedt & Dehaspe 1997] Luc De Raedt and Luc Dehaspe. *Clausal Discovery.* Machine Learning, vol. 26, no. 2-3, pages 99–146, 1997.

[De Raedt *et al.* 2008] Luc De Raedt, Paolo Frasconi, Kristian Kersting and Stephen Muggleton, editeurs. Probabilistic inductive logic programming - theory and applications, volume 4911 of *Lecture Notes in Computer Science*. Springer, 2008.

[De Raedt 2008] Luc De Raedt. Logical and relational learning. Springer, Secaucus, NJ, USA, 2008.

[Dehaspe 1997] Luc Dehaspe. *Maximum Entropy Modeling with Clausal Constraints.* pages 109–124, 1997.

[Della Pietra *et al.* 1997] Stephen Della Pietra, Vincent Della Pietra and John Lafferty. *Inducing Features of Random Fields.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 19, no. 4, pages 380–393, 1997.

[Dempster *et al.* 1977] A. P. Dempster, N. M. Laird and D. B. Rubin. *Maximum Likelihood from Incomplete Data via the EM Algorithm.* Journal of the Royal Statistical Society. Series B (Methodological), vol. 39, no. 1, pages 1–38, 1977.

[Dinh *et al.* 2010a] Quang-Thang Dinh, Matthieu Exbrayat and Christel Vrain. *Discriminative Markov Logic Network Structure Learning Based on Propositionalization and χ^2-test.* In ADMA (1), pages 24–35, 2010.

[Dinh *et al.* 2010b] Quang-Thang Dinh, Matthieu Exbrayat and Christel Vrain. *Generative Structure Learning for Markov Logic Networks.* In Proceeding of the 2010 conference on STAIRS 2010: Proceedings of the Fifth Starting AI Researchers' Symposium, pages 63–75, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press.

[Dinh *et al.* 2010c] Quang-Thang Dinh, Matthieu Exbrayat and Christel Vrain. *Heuristic Method for Discriminative Structure Learning of Markov Logic Networks.* In ICMLA, pages 163–168, 2010.

[Dinh *et al.* 2011a] Quang-Thang Dinh, Matthieu Exbrayat and Christel Vrain. *Apprentissage génératif de la structure de réseaux logiques de Markov à partir d'un graphe des prédicats.* In EGC, pages 413–424, 2011.

[Dinh *et al.* 2011b] Quang-Thang Dinh, Matthieu Exbrayat and Christel Vrain. *Generative Structure Learning for Markov Logic Networks Based on Graph of Predicates.* In IJCAI, pages 1249–1254, 2011.

[Domingos & Lowd 2009] Pedro Domingos and Daniel Lowd. Markov logic: An interface layer for artificial intelligence. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009.

[Domingos & Richardson 2007] Pedro Domingos and Matthew Richardson. Markov logic: A unifying framework for statistical relational learning, pages chapter 12, 339–371. In Introduction to Statistical Relational Learning, 2007.

[Domingos *et al.* 2008a] Pedro Domingos, Stanley Kok, Daniel Lowd, Hoifung Poon, Matthew Richardson and Parag Singla. *Markov Logic.* In Probabilistic Inductive Logic Programming, pages 92–117, 2008.

[Domingos *et al.* 2008b] Pedro Domingos, Daniel Lowd, Stanley Kok, Hoifung Poon, Matthew Richardson and Parag Singla. *Uncertainty Reasoning for the Semantic Web I.* chapitre Just Add Weights: Markov Logic for the Semantic Web, pages 1–25. Springer-Verlag, Berlin, Heidelberg, 2008.

[Dutra *et al.* 2002] Ines De Castro Dutra, Vitor Santos Costa and Jude Shavlik. *An Empirical Evaluation of Bagging in Inductive Logic Programming.* In In Proceedings of the Twelfth International Conference on Inductive Logic Programming, pages 48–65. Springer-Verlag, 2002.

[Dzeroski 2007] Saso Dzeroski. Inductive logic programming in a nutshell, chapitre 2. In the book: Introduction to Statistical Relational Learning. The MIT Press, 2007.

[Edwards 2000] David Edwards. Introduction to graphical modelling. Springer, June 2000.

[Fawcett 2003] Tom Fawcett. *ROC graphs: Notes and practical considerations for data mining researchers.* Rapport technique HPL-2003-4, HP Laboratories, Palo Alto, CA, USA, January 2003.

[Friedman *et al.* 1999] Nir Friedman, Lise Getoor, Daphne Koller and Avi Pfeffer. *Learning Probabilistic Relational Models.* pages 1300–1309, 1999.

[Genesereth & Nilsson 1987] Michael Genesereth and Nils Nilsson. Logical foundations of artificial intelligence. Morgan Kaufmann, San Mateo, CA, 1987.

[Getoor & Taskar 2007] Lise Getoor and Ben Taskar. Introduction to statistical relational learning (adaptive computation and machine learning). The MIT Press, November 2007.

[Gilks & Spiegelhalter 1999] W.R. Gilks and DJ Spiegelhalter. Markov chain monte carlo in practice. Chapman and Hall/CRC, 1999.

[Hoos & Stutzle 2004] Holger H. Hoos and Thomas Stutzle. Stochastic local search: Foundations & application. Morgan Kaufmann, 1 édition, September 2004.

[Huynh & Mooney 2008] Tuyen N. Huynh and Raymond J. Mooney. *Discriminative Structure and Parameter Learning for Markov Logic Networks*. In ICML '08: Proceedings of the 25th international conference on Machine learning, pages 416–423, New York, NY, USA, 2008. ACM.

[Huynh & Mooney 2009] Tuyen N. Huynh and Raymond J. Mooney. *Max-Margin Weight Learning for Markov Logic Networks*. In ECML PKDD '09: Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases, pages 564–579, Berlin, Heidelberg, 2009. Springer-Verlag.

[Jordan *et al.* 1999] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola and Lawrence K. Saul. *An Introduction to Variational Methods for Graphical Models*. Mach. Learn., vol. 37, pages 183–233, November 1999.

[Kautz *et al.* 1996] H. Kautz, B. Selman and Y. Jiang. *A general stochastic approach to solving problems with hard and soft constraints*, 1996.

[Kersting & De Raedt 2007] Kristian Kersting and Luc De Raedt. Bayesian logic programming: Theory and tool. In Introduction to Statistical Relational Learning, 2007.

[Kersting *et al.* 2009] Kristian Kersting, Babak Ahmadi and Sriraam Natarajan. *Counting belief propagation*. In Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI '09, pages 277–284, Arlington, Virginia, United States, 2009. AUAI Press.

[Khosravi *et al.* 2010] Hassan Khosravi, Oliver Schulte, Tong Man, Xiaoyuan Xu and Bahareh Bina. *Structure Learning for Markov Logic Networks with Many Descriptive Attributes*. In Proceedings of the 27th International Conference on Machine Learning (ICML-10), 2010.

[Knobbe *et al.* 2001] Arno J. Knobbe, Marc De Haas and Arno Siebes. *Propositionalisation and Aggregates*. In In Proceeding of the 5th PKDD, pages 277–288. Springer-Verlag, 2001.

[Kok & Domingos 2005] Stanley Kok and Pedro Domingos. *Learning the Structure of Markov Logic Networks*. In ICML '05: Proceedings of the 22nd international conference on Machine learning, pages 441–448, New York, NY, USA, 2005. ACM.

[Kok & Domingos 2009] Stanley Kok and Pedro Domingos. *Learning Markov Logic Network Structure via Hypergraph Lifting.* In ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning, pages 505–512, New York, NY, USA, 2009. ACM.

[Kok & Domingos 2010] Stanley Kok and Pedro Domingos. *Learning Markov Logic Networks Using Structural Motifs.* In Johannes Fürnkranz and Thorsten Joachims, editeurs, Proceedings of the 27th International Conference on Machine Learning (ICML-10), pages 551–558, Haifa, Israel, June 2010. Omnipress.

[Kok *et al.* 2009] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, J. Wang and P. Domingos. *The Alchemy system for statistical relational AI.* Rapport technique, University of Washington., 2009.

[Krogel & Wrobel 2001] Mark-A. Krogel and Stefan Wrobel. *Transformation-Based Learning Using Multirelational Aggregation.* In Proceedings of the 11th International Conference on Inductive Logic Programming, ILP '01, pages 142–155, London, UK, 2001. Springer-Verlag.

[Krogel *et al.* 2003] Mark-A. Krogel, Simon Rawles, Filip Zelezny, Peter Flach, Nada Lavrac; and Stefan Wrobel. *Comparative evaluation of approaches to propositionalization.* In Tamas Horvath and Akihiro Yamamoto, editeurs, Proceedings of the 13th International Conference on Inductive Logic Programming (ILP 2003), pages 194–217, Heidelberg, 2003. Springer.

[Kuželka & Železný 2008] Ondřej Kuželka and Filip Železný. *HiFi: Tractable Propositionalization through Hierarchical Feature Construction.* In Filip Železný and Nada Lavrač, editeurs, Late Breaking Papers, the 18th International Conference on Inductive Logic Programming, 2008.

[Lavrac & Dzeroski 1994] Nada Lavrac and Saso Dzeroski. Inductive logic programming: Techniques and applications. Ellis Horwood, New York, 1994.

[Lavrac *et al.* 2002] Nada Lavrac, Filip Zelezny and Peter Flach. *RSD: Relational subgroup discovery through first-order feature construction.* In In 12th International Conference on Inductive Logic Programming, pages 149–165. Springer, 2002.

[Lee *et al.* 2007] Su I. Lee, Varun Ganapathi and Daphne Koller. *Efficient Structure Learning of Markov Networks using L1-Regularization.* In B. Schölkopf, J. Platt and T. Hoffman, editeurs, Advances in Neural Information Processing Systems 19, pages 817–824. MIT Press, Cambridge, MA, 2007.

[Lesbegueries *et al.* 2009] Julien Lesbegueries, Nicolas Lachiche and A Braud. A propositionalisation that preserves more continuous attribute domains. 2009.

[Liang & Jordan 2008] Percy Liang and Michael I. Jordan. *An asymptotic analysis of generative, discriminative, and pseudolikelihood estimators.* In Proceedings of the 25th international conference on Machine learning, ICML '08, pages 584–591, New York, NY, USA, 2008. ACM.

[Liu *et al.* 1989] Dong C. Liu, Jorge Nocedal and Dong C. *On the Limited Memory BFGS Method for Large Scale Optimization.* Mathematical Programming, vol. 45, pages 503–528, 1989.

[Lovasz 1996] Laszlo Lovasz. *Random walks on graphs: A survey.* Combinatorics, vol. 2, pages 353–398, 1996.

[Lowd & Domingos 2007] Daniel Lowd and Pedro Domingos. *Efficient Weight Learning for Markov Logic Networks.* In PKDD 2007: Proceedings of the 11th European conference on Principles and Practice of Knowledge Discovery in Databases, pages 200–211, Berlin, Heidelberg, 2007. Springer-Verlag.

[McDonald 2009] John H. McDonald. Handbook of biological statistics. Sparky House Publishing, Baltimore, Maryland, USA, second édition, 2009.

[Mihalkova & Mooney 2007] Lilyana Mihalkova and Raymond J. Mooney. *Bottom-up Learning of Markov Logic Network Structure.* In ICML '07: Proceedings of the 24th international conference on Machine learning, pages 625–632, New York, NY, USA, 2007. ACM.

[Milch *et al.* 2008] Brian Milch, Luke S. Zettlemoyer, Kristian Kersting, Michael Haimes and Leslie P. Kaelbling. *Lifted probabilistic inference with counting formulas.* In AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence, pages 1062–1068. AAAI Press, 2008.

[Moller 1993] M. Moller. *A scaled conjugate gradient algorithm for fast supervised learning.* Neural Networks, vol. 6, no. 4, pages 525–533, 1993.

[Muggleton & Feng 1990] Stephen Muggleton and Cao Feng. *Efficient Induction Of Logic Programs.* In New Generation Computing. Academic Press, 1990.

[Muggleton & Feng 1992] Stephen Muggleton and C. Feng. *Efficient Induction in Logic Programs.* In Stephen Muggleton, editeur, Inductive Logic Programming, pages 281–298. Academic Press, 1992.

[Muggleton 1995] Stephen Muggleton. *Inverse entailment and progol.* New Generation Computing, vol. 13, pages 245–286, 1995. 10.1007/BF03037227.

[Nath & Domingos 2010] Aniruddh Nath and Pedro Domingos. *Efficient Belief Propagation for Utility Maximization and Repeated Inference.* In AAAI, 2010.

[Neville & Jensen 2004] Jennifer Neville and David Jensen. *Dependency Networks for Relational Data.* In ICDM '04: Proceedings of the Fourth IEEE International Conference on Data Mining, pages 170–177, Washington, DC, USA, 2004. IEEE Computer Society.

[Nocedal & Wright 1999] Jorge Nocedal and Stephen J. Wright. Numerical optimization. Springer, August 1999.

[Pearl 1988] Judea Pearl. Probabilistic reasoning in intelligent systems: Networks of plausible inference. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[Plotkin 1970] Gordon D. Plotkin. *A Note on Inductive Generalization.* Machine Intelligence, vol. 5, pages 153–163, 1970.

[Poole 2003] David Poole. *First-order probabilistic inference.* In Proceedings of the 18th international joint conference on Artificial intelligence, pages 985–991, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[Poon & Domingos 2006] Hoifung Poon and Pedro Domingos. *Sound and Efficient Inference with Probabilistic and Deterministic Dependencies.* In AAAI'06: Proceedings of the 21st national conference on Artificial intelligence, pages 458–463. AAAI Press, 2006.

[Poon & Domingos 2008] Hoifung Poon and Pedro Domingos. *Joint unsupervised coreference resolution with Markov logic.* In Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP '08, pages 650–659, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[Poon *et al.* 2008] Hoifung Poon, Pedro Domingos and Marc Sumner. *A general method for reducing the complexity of relational inference and its application to MCMC.* In Proceedings of the 23rd national conference on Artificial intelligence - Volume 2, pages 1075–1080. AAAI Press, 2008.

[Quinlan 1990] J. R. Quinlan. *Learning Logical Definitions from Relations.* Mach. Learn., vol. 5, pages 239–266, September 1990.

[Rabiner 1990] Lawrence R. Rabiner. *Readings in speech recognition.* pages 267–296, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[Richards & Mooney 1992] B. L. Richards and R. J. Mooney. *Learning Relations by Pathfinding.* In Proc. of AAAI-92, pages 50–55, San Jose, CA, 1992.

[Richards & Mooney 1995] Bradley L. Richards and Raymond J. Mooney. *Automated Refinement of First-Order Horn-Clause Domain Theories.* Machine Learning, vol. 19, pages 95–131, May 1995.

[Richardson & Domingos 2004] Matthew Richardson and Pedro Domingos. *Markov Logic: A Unifying Framework for Statistical Relational Learning.* In In Proceedings of the ICML-2004 Workshop on SRL and its Connections to Other Fields, pages 49–54, 2004.

[Richardson & Domingos 2006] Matthew Richardson and Pedro Domingos. *Markov Logic Networks.* Mach. Learn., vol. 62, no. 1-2, pages 107–136, 2006.

[Saitta & Vrain 2008] Lorenza Saitta and Christel Vrain. *A Comparison between Two Statistical Relational Models.* In Proceedings of the 18th international conference on Inductive Logic Programming, ILP '08, pages 244–260, Berlin, Heidelberg, 2008. Springer-Verlag.

[Sarkar *et al.* 2008] Purnamrita Sarkar, Andrew W. Moore and Amit Prakash. *Fast incremental proximity search in large graphs.* In Proceedings of the 25th international

conference on Machine learning, ICML '08, pages 896–903, New York, NY, USA, 2008. ACM.

[Sato & Kameya 1997] Taisuke Sato and Yoshitaka Kameya. *PRISM: A Language for Symbolic-statistical Modeling.* In In Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97, pages 1330–1335, 1997.

[Sha & Pereira 2003] Fei Sha and Fernando Pereira. *Shallow Parsing with Conditional Random Fields.* In NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, pages 134–141, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

[Shavlik & Natarajan 2009] Jude Shavlik and Sriraam Natarajan. *Speeding up Inference in Markov Logic Networks by Preprocessing to Reduce the Size of the Resulting Grounded Network.* In IJCAI'09: Proceedings of the 21st international jont conference on Artifical intelligence, pages 1951–1956, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[Shewchuk 1994] Jonathan R. Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.* Rapport technique, Pittsburgh, PA, USA, 1994.

[Silverstein & Pazzani 1991] Glenn Silverstein and Michael J. Pazzani. *Relational Clichés: Constraining Induction During Relational Learning.* In ML, pages 203–207, 1991.

[Singla & Domingos 2005] Parag Singla and Pedro Domingos. *Discriminative Training of Markov Logic Networks.* In In Proc. of the Natl. Conf. on Artificial Intelligence, 2005.

[Singla & Domingos 2008] Parag Singla and Pedro Domingos. *Lifted first-order belief propagation.* In Proceedings of the 23rd national conference on Artificial intelligence - Volume 2, pages 1094–1099. AAAI Press, 2008.

[Spirtes *et al.* 2001] Peter Spirtes, Clark Glymour and Richard Scheines. Causation, prediction, and search, second edition (adaptive computation and machine learning). The MIT Press, 2 édition, January 2001.

[Srinivasan 2003] Ashwin Srinivasan. *The Aleph Manual.* http://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph, 2003.

[Srinivasan 2007] Ashwin Srinivasan. *The Aleph Manual.* Rapport technique, http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/aleph, 2007.

[Taskar *et al.* 2007] Ben Taskar, Pieter Abbeel, Ming-Fai Wong and Daphne Koller. *Relational Markov Networks.* In L. Getoor and B. Taskar, editeurs, Introduction to Statistical Relational Learning. MIT Press, 2007.

[Tasker *et al.* 2002] Ben Tasker, Abbeel Pieter and Daphne Koller. *Discriminative Probabilistic Models for Relational Data.* In Proceedings of the 18th Annual Conference

on Uncertainty in Artificial Intelligence (UAI-02), pages 485–49, San Francisco, CA, 2002. Morgan Kaufmann.

[Wei *et al.* 2004] Wei Wei, Jordan Erenrich and Bart Selman. *Towards efficient sampling: exploiting random walk strategies.* In Proceedings of the 19th national conference on Artifical intelligence, AAAI'04, pages 670–676. AAAI Press, 2004.

[Zelle *et al.* 1994] John M. Zelle, Raymond J. Mooney and Joshua B. Konvisser. *Combining top-down and bottom-up techniques in inductive logic programming.* In in Proceedings of the Eleventh International Conference on Machine Learning ML-94, Morgan-Kaufmann, pages 343–351. Morgan Kaufmann, 1994.

Quang-Thang DINH

Apprentissage Statistique Relationnel : Apprentissage de Structures de Réseaux de Markov Logiques

Résumé :
Un réseau logique de Markov est formé de clauses en logique du premier ordre auxquelles sont associés des poids. Cette thèse propose plusieurs méthodes pour l'apprentissage de la structure de réseaux logiques de Markov (MLN) à partir de données relationnelles. Ces méthodes sont de deux types, un premier groupe reposant sur les techniques de propositionnalisation et un second groupe reposant sur la notion de Graphe des Prédicats. L'idée sous-jacente aux méthodes à base de propositionnalisation consiste à construire un jeu de clauses candidates à partir de jeux de littéraux dépendants. Pour trouver de tels jeux, nous utilisons une méthode de propositionnalisation afin de reporter les informations relationnelles dans des tableaux booléens, qui serviront comme tables de contingence pour des test de dépendance. Nous avons proposé deux méthodes de propositionnalisation, pour lesquelles trois algorithmes ont été développés, qui couvrent les problèmes d'appprentissage génératif et discriminant. Nous avons ensuite défini le concept de Graphe des Prédicats qui synthétise les relations binaires entre les prédicats d'un domaine. Des clauses candidates peuvent être rapidement et facilement produites en suivant des chemins dans le graphe puis en les variabilisant. Nous avons développé deux algorithmes reposant sur les Graphes des Prédicats, qui couvrent les problèmes d'appprentissage génératif et discriminant.
Mots clés : Réseaux Logiques de Markov, Apprentissage de Structure, Propositionnalisation, Apprentissage Statistique Relationnel.

Statistical Relational Learning: Structure Learning for Markov Logic Networks

Abstract:
A Markov Logic Network is composed of a set of weighted first-order logic formulas. In this dissertation we propose several methods to learn a MLN structure from a relational dataset. These methods are of two kinds: methods based on *propositionalization* and methods based on *Graph of Predicates*. The methods based on propositionalization are based on the idea of building a set of candidate clauses from sets of dependent variable literals. In order to find such sets of dependent variable literals, we use a propositionalization technique to transform relational information in the dataset into boolean tables, that are then provided as contingency tables for tests of dependence. Two propositionalization methods are proposed, from which three learners have been developed, that handle both generative and discriminative learning. We then introduce the concept of Graph of Predicates, which synthethises the binary relations between the predicates of a domain. Candidate clauses can be quickly and easily generated by simply finding paths in the graph and then variabilizing them. Based on this Graph, two learners have been developed, that handle both generative and discriminative learning.
Keywords: Markov Logic Networks, Structure Learning, Propositionalization, Statistical Relational Learning.

Laboratoire d'Informatique Fondamentale d'Orléans
Bat. 3IA, Université d'Orléans
Rue Léonard de Vinci, B.P. 6759
F-45067 ORLEANS Cedex 2