Learning Failure-free PRISM Programs

Waleed Alsanie

A thesis submitted for the degree of

Doctor of Philosophy

The University of York
Department of Computer Science
United Kingdom

September 2012

بسم الله الرحمن الرحيم

To my sons Abdulaziz and Nawaf, to whom I did not give enough whilst I was working on this thesis, this thesis is dedicated to you.

Abstract

First-order logic can be used to represent relations amongst objects. Probabilistic graphical models encode uncertainty over propositional data. Following the demand of combining the advantages of both representations, probabilistic logic programs provide the ability to encode uncertainty over relational data. PRISM is a probabilistic logic programming formalism based on the distribution semantics. PRISM allows learning the parameters when the programs are known.

This thesis proposes algorithms to learn failure-free PRISM programs. It combines ideas from both areas of inductive logic programming and learning Bayesian networks. The learned PRISM programs generalise dynamic Bayesian networks by defining a halting distribution over the sampling process. Each dynamic Bayesian network models either an infinite sequential generative process or a sequential generative process of a fixed length. In both cases, only a fixed length of sequences can be sampled. On the other hand, the PRISM programs considered in this thesis represent self-terminating functions from which sequences of different lengths can be obtained. The effectiveness of the proposed algorithms on learning five programs is shown.

Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

I thank and praise Allah (God) the Almighty for giving me the courage and the ability to seek knowledge and complete this thesis.

I would like to express my gratitude to my supervisor James Cussens for his guidance and support throughout this research. James introduced me to the area of probabilistic logic programming. He shared his knowledge and expertise in inductive logic programming and probabilistic graphical models which stimulated the ideas given in this thesis. However, any shortcomings in this thesis remain my own. I would like to thank Dimitar Kazakov for providing me with comments and suggestions throughout the milestones of this research.

Thanks go to the PRISM team for answering several questions related to PRISM. Special thanks go to Taisuke Sato for giving detailed answers to different questions and detailed explanations of some scoring functions provided by PRISM.

I would like to acknowledge the funding received from King Abdulaziz City for Science and Technology (KACST) throughout the course of this thesis.

Thanks must go to Hassan Mathkour and Ameur Touir for pushing my interest of pursuing a PhD study forward.

My parents Abdullah Alsanie and Sarah Albati have alway been with me with their prayers and encouragements throughout this research. Words are not enough to express my gratitude to them.

Last but not least, I would like to express my gratitude to my wife Nouf Alruwaishid. She has always been standing shoulder to shoulder with me throughout this long journey. This thesis is a result of her love and support.

Declaration

I hereby declare that the work presented in this thesis is my own unless otherwise stated. This thesis has not been submitted for any degree other than the Doctor of Philosophy at the University of York. Some parts of this thesis were presented at the following events:

- Waleed Alsanie and James Cussens. Learning a generative failure-free PRISM clause. In *The 21st International Conference on Inductive Logic Programming (ILP 2011)*, 2011.
- Waleed Alsanie and James Cussens. Learning Recursive PRISM Programs with Observed Outcomes. In *ICML'12 workshop on Statistical Relational Learning*, 2012.

Chapter 1

Motivation and Thesis Overview

This chapter shows the significance of probabilistic logic programs to the areas of knowledge representation and reasoning. It highlights how probabilistic logic programs overcome the limitations of the *propositional* representation and reasoning with a deterministic *knowledge base*. It also shows how other modelling languages, mainly probabilistic graphical models and logic programs (LP), are generalised by such representations. These discussions are given in Section 1.1. Learning probabilistic logic programs and the difference between learning *discriminative* models and *generative* models are highlighted in Section 1.2. In Section 1.3, we go through statistical inference and discuss two approaches, the maximum likelihood approach and the Bayesian approach. We give the motivation of this thesis in Section 1.4. Finally, an overview of the thesis is given in Section 1.5.

1.1 Probabilistic Logic Programs

Logic has been used in *artificial intelligence* (AI) since McCarthy (1959) proposed it as a language of encoding common sense. This proposition is motivated by the notion that logic is a symbolic representation which can simulate verbal human common sense, and deduction can be used to reach conclusions which describe actions to be taken. Though this proposition has been of high importance to the area of AI, some challenges started to arise. McCarthy (1977) shed light on the problems facing AI. He pointed out that one of them is the *monotonicity* of deduction inference which can be problematic if the application is used in an incremental way. This issue had been a problem in knowledge representation and reasoning. McCarthy (1980) then proposed the use of *non-monotonic* reasoning to overcome this problem. He stated that human reasoning is generally non-monotonic. McCarthy indicated that *probability theory*, which had been proposed to solve this problem, cannot be taken

as a general solution because some of the problems exhibiting common sense are not probabilistic. However, Pearl (1988) showed that *probabilistic graphical models* (PGMs) are feasible alternatives and can be used to solve many problems which require reasoning under uncertainty. PGMs have been considered a revolution in AI (Darwiche, 2009) as they remedy many limitations imposed by the determinism of logic. However, to simplify the application of PGMs, the assumption that instances generated by such models are *independent and identically distributed* (i.i.d.) has become a convention. This assumption does not always apply and areas with data having complex relational structure such as computational biology (King et al., 2004; Fredouille et al., 2007), natural language processing (Mooney, 1996; Cussens et al., 1997; Cussens and Džeroski, 2000), navigation (Moyle, 2003) and knowledge discovery in databases (Mooney et al., 2002; Wrobel, 2001) require these relations to be captured. When the significance of combining relational representation and probabilistic reasoning became apparent, different models began to emerge to implement this combination. *Probabilistic relational models* (PRM) are meant to define probability distributions over the attributes in an *entity relationship diagram* (ERD) modelling a database schema (Friedman et al., 1999). Heckerman et al. (2004) generalised PRM to a compact representation and called it *directed acyclic probabilistic entity-relationship* (DAPER). *First-order logic* (FOL) is a known language for relational representation (Džeroski, 2007). A main advantage of FOL over other relational models is the ability to define relations recursively. This allows defining relations between two entities in the domain where there is a variable number (possibly many) of intermediate entities between them. It also allows expressing observations with different lengths which share a unique and repetitive internal structure. An example of the latter is a regular language expressed with Kleene star applied to some of its symbols.

The expressiveness of FOL as a relational representation became appealing and attempts have been made to combine FOL with probabilistic models (Nilsson, 1986; Poole, 1993; Ng and Subrahmanian, 1992). Different *probabilistic logic* formalisms have since been developed. For instance, *independent choice logic* (ICL) (Poole, 1997) and *Bayesian logic programs* (BLP) (Kersting and De Raedt, 2007) were stimulated by the notion of generalising *Bayesian network* (BN) (a directed acyclic graphical model) to first-order representation. *Markov logic network* (MLN) (Domingos and Richardson, 2007) also aims at lifting *Markov network* (an undirected graphical model) to a first-order level. *ProbLog* (Gutmann et al., 2008) was motivated by problems in modelling probabilistic relational databases. *Stochastic logic programs* (SLP) (Muggleton,

2000) and *PRISM* (Sato and Kameya, 1997) aim at generalising both probabilistic grammars and logic programs. Programs of such formalisms are referred to as *probabilistic logic programs* (PLPs).

PRISM has been under development since Sato (1995) formalised the *distribution semantics* on which PRISM is based. The PRISM system consists of an *inference* engine, a *sampling* engine to generate observations from a target predicate and different learning tools. We introduce the distribution semantics, probabilistic modelling in PRISM and some of the learning tools provided by PRISM in Chapter 2.

1.2 Learning Generative PLPs

A common scenario in which inductive inference is needed is as follows: we are faced with some observations and possibly some knowledge about the domain where they come from, and we are left with finding a *hypothesis* that explains these observations. Extensive work in *machine learning* has been conducted on the task of inducing a general hypothesis from such observations. These observations either represent specific examples of general concepts in the domain, such examples are referred to as *positive examples*, or they are cases which do not belong to the sought concepts, and thus are referred to as *negative examples*. Negative examples are used to prevent reaching an over-generalising hypothesis. This is because in the search for a hypothesis, simple and general hypotheses are conventionally preferred as they cope well with unforeseen instances. The aim of induction is to find regularities between the examples which belong to the same class or the same category. These regularities are used to extract features that can be relied on to categorise or classify future data. These features, say $\mathbf{x}$, are assumed to exist in the data, and thus the task of the induction is to estimate $p(y|\mathbf{x})$ where y is the class or the category. Such models are known as *discriminative models*. Inducing hypotheses in FOL has been a subject of extensive research as it allows learning discriminative models in relational domains. This area of research is referred to as *inductive logic programming* (ILP) (Muggleton, 1991).

In many applications, observations represent phenomena of some unknown model. Normally these observations could have been generated by this model. Quite commonly, this generation is non-deterministic and obeys some probability distribution. In order to learn the model from these observations, this probability distribution needs to be estimated. This is a well known problem in *statistical inference*. Models which define a joint probability distribution $p(\mathbf{y}, \mathbf{x})$ from which instances can be gener-

ated (sampled) are referred to as *generative models*. Bayesian network and *stochastic context-free grammar* (SCFG) are two typical examples of generative models.

PLPs define probability distributions over relational data encoded as logic programs. *Probabilistic inductive logic programming* (PILP) refers to the area of learning such programs. PRISM allows encoding generative models over relational data (Sato, 2009). Therefore, learning PRISM programs generalises learning generative PGMs.

1.3 Statistical Inference

Given some observations, statistical inference, also known as statistical learning, concerns inferring probability distributions from which the data could have been generated (Wasserman, 2003). PGMs define probability distributions with a qualitative part, representing conditional independence assumptions between the random variables in the model, and a quantitative part, from which probabilistic inference can be performed with respect to the given independence assumptions (Cowell et al., 2007). Therefore, the qualitative part is referred to as the *structure* of the model and the quantitative part is referred to as the *parameters* of the model. The task of learning the parameters when the structure is known is referred to as *parameter estimation*. When the structure is unknown, we are left with the problem of *structure learning*. We highlight two different learning approaches, the *maximum likelihood* (ML) approach and the *Bayesian* approach. The two approaches will be discussed in the context of multinomial distribution as we only consider discrete PGMs in this thesis.

1.3.1 The Maximum Likelihood Approach

The *maximum likelihood estimate* (MLE) $\hat{\boldsymbol{\theta}}$ of $\boldsymbol{\theta}$ is the estimate which maximises the likelihood function defined as follows

$$\mathcal{L}(\boldsymbol{\theta}) = P(D|\boldsymbol{\theta}) \tag{1.1}$$

where D is the data representing the observations. Let X be a discrete variable which can take one of k exhaustive and mutually exclusive values $\{x_1, \dots x_k\}$, according to the distribution $\boldsymbol{\theta} = \{\theta_1, \dots, \theta_k\}$ respectively. Let D represent random observations of $n_1, \dots, n_k$ outcomes of X where n_i is the number of the outcome x_i in D. Then the likelihood is given as follows

$$P(n_1, \ldots, n_k|\theta_1, \ldots, \theta_k) \propto \prod_{i=1}^{k} \theta_i^{n_i} \tag{1.2}$$

It is often more convenient to deal with the *log-likelihood* which, in this case, can be obtained from (1.2) as follows

$$\log P(n_1, \ldots, n_k|\theta_1, \ldots, \theta_k) \propto \sum_{i=1}^{k} n_i \log \theta_i \tag{1.3}$$

Maximising the log-likelihood in (1.3) leads to the MLE of θ which is (Bishop, 2006)

$$\hat{\theta}_i = \frac{n_i}{n} \tag{1.4}$$

As shown above, MLE is an observation driven estimator. For a small set of observations, MLE can lead to *overfitting*. For example, the probability of a value x_i which does not show in the drawn observations will be estimated as zero. This is an extreme estimation and is typically undesirable. There are some smoothing techniques which have been designed to deal with such situation (Chen and Goodman, 1996). For instance, *additive smoothing* is used to avoid assigning zero probability to unseen values. It adds an extra term α to n_i and then normalises the quantity. α can be set to any positive value. The additive smoothing is defined as follows

$$\hat{\theta}_i = \frac{n_i + \alpha}{n + \alpha k} \tag{1.5}$$

where k is the number of values that the random variable X can take. The formula above has also some interpretation in the Bayesian approach to estimation which will be discussed in Section 1.3.2.

1.3.1.1 Missing Values and EM

When there are missing data, the sufficient statistics required to compute the MLE cannot be obtained exactly. The *Expectation maximisation* (EM) algorithm was proposed to compute the MLE in the presence of missing data (Dempster et al., 1977). It computes the expectation of the likelihood function with respect to the probability of the missing values given the observed ones under the current setting of the parameters. This expectation is then maximised. Let Z represents the missing values, X represents the observed ones and $\boldsymbol{\theta}^{(t)}$ is the current setting of the parameters, the expectation step computes the following quantity

$$Q(\boldsymbol{\theta}|\boldsymbol{\theta}^{(t)}) = E_{Z|X,\boldsymbol{\theta}^{(t)}}[\log \mathcal{L}(\boldsymbol{\theta})]$$

Then the algorithm attempts to find a setting of the parameters $\boldsymbol{\theta}^{(t+1)}$ which maximise the above quantity as follows

$$\boldsymbol{\theta}^{(t+1)} = \arg\max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}|\boldsymbol{\theta}^{(t)})$$

The algorithm iterates with the two steps above until $\boldsymbol{\theta}$ reaches a stationary point. The algorithm starts by initialising $\boldsymbol{\theta}^{(t)}$ to some initial values. The final estimated values depend on this initial point. Therefore, the algorithm may reach a local maximum. An approach to overcome this problem is to start the algorithm from different starting points.

1.3.2 The Bayesian Approach

The Bayesian approach to estimation has a philosophical background related to the subjective interpretation of probability rather than the classic objective interpretation. This subjective interpretation states that the probability of an event is the degree of belief one has on the occurrence of this event. This is different from the classical *frequentist* interpretation of probability which defines it as the limit of the relative frequency of the event in a sequence of experiments (Koch, 2007). With this interpretation, one can have a *prior* belief on a certain event, and this belief is then updated according to some new observations. This is reflected in the Bayes theorem as follows

$$P(\boldsymbol{\theta}|D) = \frac{P(D|\boldsymbol{\theta})\ P(\boldsymbol{\theta})}{P(D)} \tag{1.6}$$

where $P(\boldsymbol{\theta})$ is the prior, $P(D|\boldsymbol{\theta})$ is the likelihood and $P(D)$ is a normalising factor which is fixed for a particular set of observations. It can be noticed that the Bayesian approach treats $\boldsymbol{\theta}$ as a random variable having a probability distribution. This is different from MLE where it is considered as an unknown fixed quantity. Bayesian estimation can be substantially affected by the choice of prior. If the prior carries significant information which influences the estimation of the posterior, this prior is called an *informative prior*; otherwise, it is a *noninformative prior*. So the prior provides a way of injecting background knowledge to the estimator, and thus it is sometimes referred to as the *bias*. It is not always the case that one has an idea of what the prior should be. The choice of prior is, in many cases, made based upon a mathematical convenience. Therefore, it is common that a prior which has the same functional form as the likelihood function is chosen so that the posterior belongs to the same family. Such prior is called a *conjugate prior*. For a multinomial distribution, the conjugate prior is the *Dirichlet distribution* defined as follows

$$\mathrm{Dir}(\alpha_1, \ldots, \alpha_k) = \frac{1}{B(\alpha_1, \ldots, \alpha_k)} \prod_{i=1}^{k} \theta_i^{\alpha_i - 1} \tag{1.7}$$

where α_i is a *hyperparameter* or a *pseudocount* and the normalising constant is the *multinomial Beta function* which is defined in term of the *gamma function* $\Gamma(\alpha_i)$ (DLMF) as follows

$$B(\alpha_1, \ldots, \alpha_k) = \frac{\Gamma(\alpha_1) \ldots \Gamma(\alpha_k)}{\Gamma(\alpha_1 + \ldots + \alpha_k)} \tag{1.8}$$

The posterior distribution of $\boldsymbol{\theta}$ is then defined as in (1.9) where $(n_1, \ldots, n_k)$ is the sufficient statistics

$$P(\boldsymbol{\theta}|D) = \mathrm{Dir}(n_1 + \alpha_1, \ldots, n_k + \alpha_k) = \frac{1}{B(n_1 + \alpha_1, \ldots, n_k + \alpha_k)} \prod_{i=1}^{k} \theta_i^{n_i + \alpha_i - 1} \tag{1.9}$$

As show above, the posterior distribution of $\boldsymbol{\theta}$ is also a Dirichlet distribution but with the parameters $n_1 + \alpha_1, \ldots, n_k + \alpha_k$.

The mean of the Dirichlet distribution $\mathrm{Dir}(\alpha_1, \ldots, \alpha_k)$ is defined as follows

$$E_{\mathrm{Dir}(\alpha_1,\ldots,\alpha_k)}[\theta_i] = \frac{\alpha_i}{\sum_{j=1}^{k} \alpha_j} \qquad \forall i : 1 \leq i \leq k \tag{1.10}$$

Thus, the mean of the posterior distribution in (1.9) is defined as follows

$$E_{\mathrm{Dir}(n_1+\alpha_1,\ldots,n_k+\alpha_k)}[\theta_i] = \frac{n_i + \alpha_i}{n + \sum_{j=1}^{k} \alpha_j} \tag{1.11}$$

It can be noticed that the additive smoothing in (1.5) is the mean of a Dirichlet distribution with the parameters $n_1+\alpha_1, \ldots, n_k+\alpha_k$. Therefore, the additive smoothing uses the hyperparameters $\alpha_1, \ldots, \alpha_k$ to *smooth* the MLE in (1.4).

1.4 Motivation

PRISM supports estimating the parameters of given programs. Sato et al. (2008) also showed a way to determine the length of a *profile-hidden Markov model* (Durbin et al., 1998) encoded as a PRISM program (we define hidden Markov models in Chapter 2). To our knowledge, no work has been conducted to learn a complete PRISM program. Muggleton (2000) proposed a method of learning stochastic logic programs. The proposed method learns the programs using ILP and then fits the parameters. Kersting and De Raedt (2007) showed how to learn Bayesian logic programs by using an ILP learning setting known as *learning from interpretations* (The ILP learning settings will be defined in Chapter 3). Learning from interpretations requires providing the learning algorithm with a large amount of data (the interpretations of the observations).

This thesis investigates learning a class of generative PRISM programs known as *failure-free*. The aim is to learn recursive PRISM programs which can be used to model stochastic processes. These programs generalise dynamic Bayesian networks by defining a halting distribution over the generative process. Dynamic Bayesian

networks model infinite or fixed length stochastic processes. Sampling an infinite process can only be done by specifying the length of sequences that the process generates. In both cases, only observations of a fixed length of sequences can be obtained. On the other hand, the recursive PRISM programs considered in this thesis are self-terminating upon some halting conditions. Thus, they generate observations of different lengths of sequences.

The direction taken by this thesis is to combine ideas from both areas of ILP and learning BNs to learn PRISM programs. It builds upon an ILP setting known as *learning from entailment*. Learning from entailment has been the most common learning setting in ILP (Kersting, 2006). Probabilistic relations in PLP can be used to encode conditional dependencies. Therefore, statistical inference will be used to induce these relations.

The learning problem is cast as an optimisation problem. This thesis follows a *search and score* approach in which a PRISM program with the highest possible score is searched for. The scoring metric that is adopted is the *Bayesian information criterion* (BIC). Different models will be represented as PRISM programs. Data of different sizes will be generated from these models. The learning algorithms will be run to learn PRISM programs from these samples and some *background knowledge* (BK). The scores of the learned programs are compared with the scores of the original programs (programs from which the data have been generated). The effectiveness of the learning algorithms to approximate a program is measured by the ratio of the score of the learned program to the score of the original one as follows

$$\frac{score_{BIC}(S_L)}{score_{BIC}(S_O)}$$

where S_L is the learned program and S_O is the original one. The closer this ratio is to 1, the better the approximation is.

1.5 Thesis Overview

This thesis is organised as follows

Chapter 2 provides the necessary background. It introduces the syntax and semantics of logic programming. It also discusses modelling probability distributions with BNs. It then introduces the PRISM formalism. First, the distribution

semantics which PRISM is built upon is explained. Then probabilistic modelling with PRISM is described. PRISM is based upon four conditions which need to be met in order to perform inference correctly. These conditions are highlighted. Finally, the chapter discusses some utilities embedded in PRISM.

Chapter 3 surveys the area of ILP. The learning settings used in ILP are defined; however, the chapter concentrates on the setting of learning from entailment. The learning approaches in ILP are explained. Finally, learning recursive logic programs is discussed. The chapter introduces the ILP system MERLIN 2.0 which is adapted in this thesis to develop one of the PRISM learning algorithms.

Chapter 4 surveys the area of learning BNs. It concentrates on the *search and score* approach. It first highlights different scoring metrics in both cases where the data is fully observed and where there are some hidden variables. Four search algorithms are discussed. The chapter concludes with a survey on learning dynamic Bayesian networks.

Chapter 5 proposes an algorithm to learn recursive PRISM programs when all outcomes of the relations are observed. The chapter shows experiments conducted with the developed algorithm on learning five programs.

Chapter 6 proposes an algorithm for learning recursive PRISM programs with some hidden outcomes. The five programs in Chapter 5 are modified so that relations with hidden outcomes are added to them. The chapter shows experiments on learning these programs with the developed algorithm.

Chapter 7 provides a conclusion and points out directions for further work.

Chapter 2

Logic Programming, Bayesian Networks and PRISM

This chapter introduces the necessary background upon which this thesis builds. Section 2.1 introduces logic programming in terms of both syntax and semantics. Section 2.2 introduces Bayesian networks and dynamic Bayesian networks. The PRISM formalism, the distribution semantics and a class of PRISM programs known as *failure-free* are introduced in Section 2.3. In Section 2.4, we show how failure-free PRISM programs generalise Bayesian networks and dynamic Bayesian networks. At the end of this chapter, Section 2.5 explains some learning utilities provided by PRISM.

2.1 Logic Programming

This section introduces the syntax and semantics of logic programs. The syntax of Prolog, which PRISM is based on, is adopted. Only *SLD-resolution* inference is considered as it is the inference procedure underlying Prolog.

2.1.1 Syntax

The alphabet of a logic program consists of *constants*, *variables*, *functors*, *predicate symbols* and *logical connectives* (Nilsson and Małuszyński, 1995). Constants are either numerics or alphanumerics starting with lowercase letters. Variables are alphanumerics starting with capital letters. All variables are assumed to be *universally quantified*. Functors are alphanumerics starting with lowercase letters and are associated with arities. A functor f with arity n is denoted by f/n. Predicate symbols are alphanumerics starting with lowercase letters and are also associated with arities. Similarly

a predicate symbol p with arity n is denoted by p/n. Logical connectives are *conjunction*, *disjunction*, *negation* and *implication*. Conjunction is denoted by $\wedge$ or ',' (comma). Disjunction is denoted by $\vee$ or ';' (semicolon). The negation of c is denoted by $\neg c$. Implication is denoted by $\leftarrow$ where the right hand side of the symbol is referred to as the *condition* and the left hand side is referred to as the *conclusion*. Constants and variables are *terms*. If f/n is a functor and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is also a term, and it is referred to as a *compound term*. If p/n is a predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atomic formula*, or simply an *atom*. A *literal* is an atom or its negation. A clause is a disjunction of literals. For example, $h_1 \vee \ldots \vee h_n \vee \neg b_1 \vee \ldots \vee \neg b_m$ is a clause. Clauses can also be written as implications where atoms are the conclusions and negated atoms are the conditions. For instance the previous clause can also be expressed as follows

$$h_1; \ldots; h_n \leftarrow b_1, \ldots, b_m$$

where the $h_1; \ldots; h_n$ part is also known as the *head* of the clause and the $b_1, \ldots, b_m$ part is known as the *body* of the clause. A *clausal theory* is a set of clauses. When there is at most one atom in the head of a clause, the clause is a *Horn clause*. If there is exactly one atom in the head, it is a *definite clause*. A definite clause with an empty body is referred to as a *fact*. A *definite logic program* is a set of only definite clauses.

Clauses which contain no variables are called *ground clauses*; consequently, facts which do not contain variables are *ground facts*. The set of variables in a clause or a term H are denoted by $vars(H)$. A substitution $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ is an assignment of terms $t_1, \ldots, t_n$ to the variables $V_1, \ldots, V_n$. When a substitution is applied to a term or a clause H, this term or clause becomes *instantiated* with all the variables replaced by the corresponding terms. This term or clause is then denoted by $H\theta$.

2.1.2 Semantics

The semantics of a clausal logic is based upon *model theory* and *proof theory*. The model theory concerned here will be restricted to the one based on *Herbrand interpretations*. The proof theory will be restricted to the *resolution principle* proposed

by Robinson (1965) on which the SLD-resolution is based (Kowalski and Kuehner, 1971; Kowalski, 1974).

In model theory, the *Herbrand universe* consists of all ground terms built by the constants and the function symbols in the language, and the *Herbrand base* is the set of all ground atoms that can be formed by the ground terms in the universe. A *Herbrand interpretation* maps each ground term in the Herbrand universe to itself and specifies the set of *true* ground atoms in the corresponding Herbrand base. A Herbrand interpretation I is a *model* of a clause c if for all substitutions θ in which $body(c)\theta$ is true, $head(c)\theta$ must also be true. A clause C *entails* a clause c, denoted by $C \models c$, if and only if all models of C are also models of c. An interpretation I is a model of a clausal theory T if and only if it is a model of every clause in T. A clausal theory T entails a clause c ($T \models c$) if and only if $T \wedge \neg c$ is *unsatisfiable* (does not have any model). This is denoted by $T \wedge \neg c \models \square$ where $\square$ is *falsity* (De Raedt, 2008). A Herbrand model I of a theory T is referred to as a *minimal model* (also known as a *least model*) if there is no $I' \subset I$ which is also a model of T.

Proof theory concerns reasoning in logic. Given a theory T and a set of inference rules, it considers deriving new clauses from the given theory using the given rules. $T \vdash c$ denotes that the clause c is derivable from the theory T. Robinson (1965) proposed an inference rule called *resolution* which, given the two ground clauses $h_1 \leftarrow b_{11}, b_{12}$ and $h_2 \leftarrow b_{21}, h_1$, derives the new clause $h_2 \leftarrow b_{21}, b_{11}, b_{12}$ (called *resolvent*). In order to apply this rule in FOL, an operation called *unification* is used. A unifier of two terms t_1 and t_2 is a substitution θ which satisfies $t_1\theta = t_2\theta$. Robinson proposed the resolution rule along with a unification operation called the *most general unification* (mgu) to perform inference in FOL. The most general unifier U is a unifier in which for any other unifier U' there exist a substitution θ that satisfies $U\theta = U'$. To prove a clause c from a theory T, SLD-resolution uses the resolution rule and the *mgu* to derive falsity from $T \wedge \neg c$. When falsity is derivable, it is proved that $T \models c$, otherwise all derivations *fail* and $T \not\models c$. The procedure of deriving falsity from $T \not\models c$ is known as a *refutation*. Thus, proofs in an SLD-resolution are also referred to as *refutations*.

2.2 Bayesian Network

A *Bayesian network* is a probabilistic graphical model representing a probability distribution. The BN *structure* is a *directed acyclic graph* (DAG). Each node in this graph represents a random variable. Figure 2.1 shows an example of a BN with six random variables. Given an edge in a BN, the node at the end of this edge is a *child*

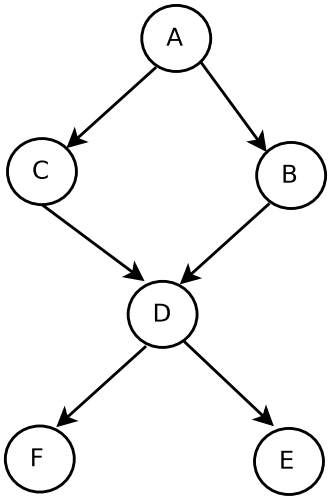


Figure 2.1: An example of Bayesian Network

of the node at the beginning of the edge which is a *parent* of this child. For example, in Figure 2.1, C is a child of A and A is the parent of C. A child and its parents constitute a *family*. The set of the parents of a child is known as the *parent set* of that child. The parent set of node D in Figure 2.1 is $\{B, C\}$, and the family is denoted by $\{B, C\} \rightarrow D$. The undirected path going between two parents in a family through their child is said to meet *head-to-head* at the child, and this child is referred to as a *head-to-head node*. For example, the undirected path $B - D - C$ meets head-to-head at D which is a head-to-head node.

A *conditional probability distribution* (CPD) is associated with each node. It defines the probabilities of the values taken by the random variable in this node given the values taken by its parents. These probabilities are the *parameters* of the model. The full joint probability distribution of the variables factorises into these CPDs. For instance, the factorisation of the joint probability distribution of the variables in the graph in Figure 2.1 is defined as follows

$$P(A, B, C, D, E, F) = P(A)P(B|A)P(C|A)P(D|C, B)P(E|D)P(F|D) \qquad (2.1)$$

Definition 1. *An undirected path between a node N_1 and another node N_2 in a BN is active given the set of nodes $\mathbf{Z}$ if and only if*

- *any head-to-head node in this path is in $\mathbf{Z}$ or one of its descendents is in $\mathbf{Z}$ and*

- *no non-head-to-head node in this path is in* $\mathbf{Z}$.

The conditional independence assumption encoded by a BN can be obtained from the DAG using the *d-separation* property (Pearl et al., 1989) (d stands for directional). d-separation can be defined as follows (Koller et al., 2007)

Definition 2. *Two sets of nodes* $\mathbf{X}$ *and* $\mathbf{Y}$ *are d-separated given the set of nodes* $\mathbf{Z}$*, denoted by* $\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z}$*, if and only if there is no* active *undirected path between some nodes in* $\mathbf{X}$ *and some nodes in* $\mathbf{Y}$ *given the nodes in* $\mathbf{Z}$.

In Figure 2.1, $D \perp\!\!\!\perp A \mid \{C, B\}$ because the two undirected paths $A - C - D$ and $A - B - D$ are not active. This is because C and B are both non-head-to-head nodes and they are all given. Also, $C \perp\!\!\!\perp B \mid \{A\}$; however, when D is added to the set of given nodes, $C \not\perp\!\!\!\perp B \mid \{A, D\}$. This is because D is a head-to-head node and it is in the undirected path $B - D - C$. Likewise, $C \not\perp\!\!\!\perp B \mid \{A, F\}$ as F is a descendent of the head-to-head node D which is in the undirected path $B - D - C$. For any node, the set consisting of its parents, children and the other parents of its children d-separates it from the other nodes in the graph. This set is referred to as the *Markov blanket* of this node (Koller et al., 2007).

Several forms of probabilistic reasoning can be performed on BN. An obvious query is computing the *marginal probability* which is defined as the probability of the instantiation of a subset of the variables in the BN. For instance let $\mathbf{X} = \mathbf{E} \cup \mathbf{F}$ be the set of the variables in a BN where $\mathbf{E}$ and $\mathbf{F}$ are mutually exclusive, then the query $P(\mathbf{E})$ is a marginal probability query. Another query is the *most probable explanation* (MPE) defined as follows

$$\arg\max_{\mathbf{f}} \; P(\mathbf{F} = \mathbf{f}, \mathbf{E} = \mathbf{e})$$

Let $\mathbf{X} = \mathbf{E} \cup \mathbf{F} \cup \mathbf{W}$ where $\mathbf{E}$, $\mathbf{F}$ and $\mathbf{W}$ are mutually exclusive; the *conditional probability* query is defined as $P(\mathbf{E}|\mathbf{F} = \mathbf{f})$. Finally, the *maximum a posteriori* (MAP) query is defined as follows

$$\arg\max_{\mathbf{f}} \sum_{\mathbf{W}} P(\mathbf{F} = \mathbf{f}, \mathbf{W}|\mathbf{E} = \mathbf{e})$$

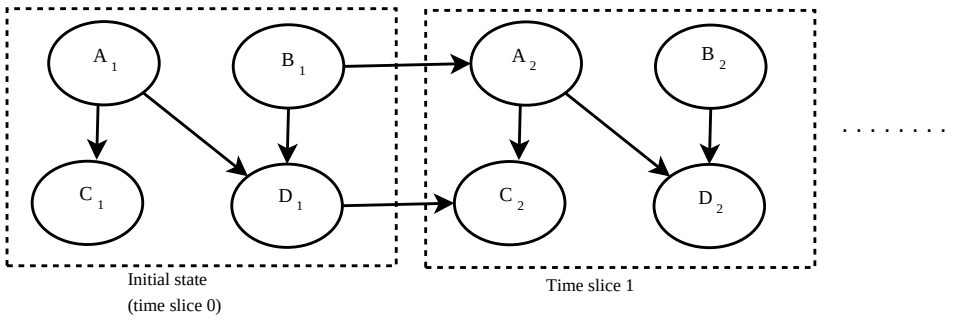


Figure 2.2: An example of a dynamic Bayesian network.

2.2.1 Dynamic Bayesian Network

A *dynamic Bayesian network* (DBN) is a probabilistic model used to represent a sequential process of data generation (Murphy, 2002). DBNs are useful to model the effects of some events on the appearance of some other events later in the process. We restrict the discussion on models which comply with the *Markov property*. A sequential process has the *Markov property* if the current state of the process depends only on the previous state. DBNs with the Markov property are useful to represent temporal models where values generated at a particular time t_i have effects on the generation of some values at t_{i+1}. Therefore, at each point of time t_i, where $i > 0$, values are generated according to a joint probability distribution of the random variables at t given the values generated at t_{i-1}. The probability distribution of a sequence of length l in a temporal process is then

$$\begin{aligned} &P(X_1^{(0)}, \ldots, X_n^{(0)}, \ldots, X_1^{(l-1)}, \ldots, X_n^{(l-1)}) = \\ &\quad P(X_1^{(0)}, \ldots, X_n^{(0)}) \prod_{t=1}^{l-1} P(X_1^{(t)}, \ldots, X_n^{(t)} | X_1^{(t-1)}, \ldots, X_n^{(t-1)}) \end{aligned} \tag{2.2}$$

The state at time 0 is the *initial state* of the process. Each state in this temporal process is referred to as a *time-slice*. Dependencies amongst variables in each time slice are known as *intra-slice* dependencies. Dependencies amongst variables between two consecutive time slices are known as *inter-slice dependencies*. Figure 2.2 shows an example of a DBN. In this DBN, the edge from A_1 to C_1, the edge from A_1 to D_1 and the edge from B_1 to D_1 represent the intra-slice dependencies. The edge from B_1 to A_2 and the edge from D_1 to C_2 represent the inter-slice dependencies.

2.2.1.1 Hidden Markov Model

A *hidden Markov model* (HMM) is defined as follows (Rabiner and Juang, 1986)

Definition 3. *A hidden Markov model is a sequential process consisting of the following elements:*

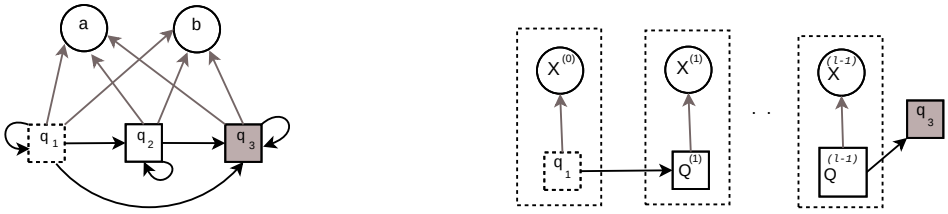
- *A set of hidden (unobserved) states* $\mathbf{q} = \{q_1, \ldots, q_n\}$
- *A transition probability distribution associated with each state defining the probabilities of moving to other states.* $p(q^{(t+1)}|q^{(t)})$ *is the probability of moving from a state* $q^{(t)}$ *at time* t *to another state* $q^{(t+1)}$ *at time* $t+1$*. The probability of moving to the next state depends only on the current state (the Markov property).*
- *Emissions generated by the states. When a state is reached, it generates an emission according to a probability distribution which depends only on this state. This probability is called the emission probability.* $p(x^{(t)}|q^{(t)})$ *is the probability that state* $q^{(t)}$ *emits* $x^{(t)}$*.*

Let $Q^{(0)}, \ldots, Q^{(l)}$ be random variables defined over the set $\mathbf{q} = \{q_1, \ldots, q_n\}$. At each time t, where $0 \leq t \leq l$, $Q^{(t)}$ takes a value depending on the value taken by $Q^{(t-1)}$ and generates an emission. An HMM can then be represented as a DBN. Therefore, the class of HMMs is a subclass of DBNs. A distribution is defined over a set $\mathbf{s} \subseteq \mathbf{q}$ of initial states. An HMM can either define a distribution over a set $\mathbf{f} \subseteq \mathbf{q}$ of final states or it can be infinite. An example of an HMM of three states and two emissions is shown in Figure 2.3a. In this example, the state from which the process starts is q_1. The process finishes on state q_3. The representation of this HMM as a DBN is shown in Figure 2.3b.

Given an HMM model M and a sequence $x^{(0)}, \ldots, x^{(l-1)}$ of emissions, the probability of observing this sequence can be computed by summing over all the states which could have generated it. This is computed as follows

$$p(x^{(0)}, \ldots, x^{(l-1)}|M) = \sum_{\mathbf{q}} \prod_{t=0}^{l-1} p(x^{(t)}|q^{(t)}) p(q^{(t+1)}|q^{(t)})$$

A common inference problem on HMMs is finding the most probable sequence of states from which a list of emissions could have been generated. The Viterbi algorithm



(a) An example of an HMM with three states and two emissions. q_1 is an initial state and q_3 is a final state.

(b) The HMM in (a) represented as a DBN. The initial state is q_1. q_3 is a final state. $X^{(t)}$ is a random variable representing an emission which can take either a or b. $Q^{(t)}$ is a random variable which takes one of the states q_1, q_2 or q_3.

Figure 2.3: An example of an HMM.

is used to find this sequence; therefore, this sequence is referred to as the *Viterbi path*. It is defined as follows

$$\arg\max_{q^{(0)},\ldots,q^{(l)}} \prod_{t=0}^{l-1} p(x^{(t)}|q^{(t)})p(q^{(t+1)}|q^{(t)})$$

2.3 The PRISM Formalism

Programming in statistical modelling (PRISM) is a probabilistic logic programming formalism. It defines a probability distribution over the space of truth values of ground facts (Sato and Kameya, 1997). This section introduces the semantics upon which PRISM is based and the syntax of the probabilistic facts.

2.3.1 Distribution Semantics

Given a logic program $DB = F \cup R$ where F is a set of ground facts and R is a set of rules (definite clauses), a probability distribution P_F is defined over all possible assignments of truth values of the ground facts in F. If $X_1, X_2, \ldots, X_n$ are the ground facts in F and $\forall i : 1 \leq i \leq n$, $x_i \in \{0, 1\}$ where 0 means *false* and 1 means *true*, the joint probability distribution $P_F(X_1 = x_1, \ldots, X_n = x_n)$ is defined such that (2.3) holds.

$$\begin{cases} 0 \leq P_F(X_1 = x_1, \ldots, X_n = x_n) \leq 1 \\ \sum_{x_1,\ldots,x_n} P_F(X_1 = x_1, \ldots, X_n = x_n) = 1 \\ \sum_{x_{n+1}} P_F(X_1 = x_1, \ldots, X_{n+1} = x_{n+1}) = P_F(X_1 = x_1, \ldots, X_n = x_n) \end{cases} \tag{2.3}$$

Because the ground facts in F are probabilistically true or false, they can be thought as random variables taking the value 0 or 1. Sampling from P_F will lead to a set $F' \subseteq F$ where the least Herbrand model of $F' \cup R$ represents a sample of true ground atoms from the Herbrand base of DB. The probability of each ground atom A is defined as follows

$$P(A) = \sum_{F' \subseteq F : F' \cup R \models A} P_F(F')$$

Thus P_F can be extended to P_{DB} which is known as the *distribution semantics* (Sato and Kameya, 1997, 2008).

2.3.2 Probabilistic Modelling

The probabilistic facts in PRISM are modelled using the built-in predicate `msw/2`. The mode of the atoms of this predicate is $\texttt{msw}(-s, +v)$. The first argument is an input argument and it is called a *switch*. Each switch is a family of i.i.d. random variables $\{S_i\}_{i \in \mathbb{N}}$, where i is a trial number. The second argument to the `msw/2` predicate is an output argument and it is a value to which a random variable from this family is instantiated (Cussens, 2012). Therefore, $\texttt{msw}(s, v)$ is represented internally in PRISM as $\texttt{msw}(s, i, v_1)$ with i being the trial number. With another fact $\texttt{msw}(s, v)$, which is also represented as $\texttt{msw}(s, j, v_2)$ where $i \neq j$, two different random variables S_i and S_j which belong to the same switch (family) are represented. The discrete sample space which an outcome of a switch is defined over is determined by the built-in predicate `values/2`. The first argument to this predicate is either a switch or a term representing a set of switches and the second argument is the sample space. An instance of this predicate is referred to as a *switch declaration*. The distribution $\boldsymbol{\theta} = \{\theta_1, \ldots, \theta_n\}$ of an outcome of a switch defined over a sample space of size n is set using the built-in predicate `set_sw/2`. The first argument to `set_sw/2` is the switch and the second argument is the distribution. Figure 2.4 shows a PRISM program modelling three random movements that need to be taken according to the sum of the numbers resulting from three rolling of two dice (`die(a)` and `die(b)`). `values(die(_),[1,2,3,4,5,6])` is a declaration of a set of switches (each switch is a ground term). `set_sw(die(a),[0.1,0.2,0.1,0.2,0.2,0.2])` is the distribution

```
values(die(_),[1,2,3,4,5,6]).

move(S):-
     move(0,S).

move(3,[]):-!.
move(N,[Z|T]):-
     msw(die(a),X),
     msw(die(b),Y),
     Z is X + Y,
     N1 is N + 1,
     move(N1,T).

set_params:-
     set_sw(die(a),[0.1,0.2,0.1,0.2,0.2,0.2]),
     set_sw(die(b),[0.2,0.1,0.2,0.1,0.2,0.1]).
```

Figure 2.4: The *dice* PRISM program

defined over the outcomes of switch `die(a)` (the distribution of the outcomes of switch `die(b)` is defined in the same way). By sampling six random variables as follows

$$\{msw(die(a), 1, X_1), msw(die(b), 1, Y_1),$$
$$msw(die(a), 2, X_2), msw(die(b), 2, Y_2),$$
$$msw(die(a), 3, X_3), msw(die(b), 3, Y_3)\}$$

we are actually sampling instances of the *target predicate* `move/1`. Sampling instances of a target predicate can be performed using the built-in predicate `get_samples/3`. The first argument to this predicate is the number of samples, the second argument is an atom of the target predicate and the third argument is a variable to which the answer is instantiated. PRISM allows performing probabilistic inference through some built-in predicates. The `prob/1` predicate is used to compute the probability of a goal. For instance, `prob(move([3,5,2]))`, is the probability of sampling three dice rolling and obtaining the three sums $3, 5$ and 2. `probf/1` is used to obtain the explanations of a successful query. Each explanation is a conjunction of atoms in a branch of the SLD-resolution used to derive the refutations of the goal. Figure 2.5 shows the result of the sampling query, the probability computation query and the explanation query (in the answer of the explanation query, `v` denotes a disjunction and `&` denotes a conjunction).

```
| ?- get_samples(10,move(_),Gs).

Gs = [move([9,7,11]),move([5,8,9]),move([6,8,5]),move([9,12,6]),
      move([9,12,11]),move([5,10,9]),move([7,6,11]),move([8,7,7]),
      move([8,6,8]),move([7,8,9])]
yes
| ?- prob(move([3,5,2])).

Probability of move([3,5,2]) is: 0.000032000000000

yes
| ?- probf(move([3,5,2])).

move([3,5,2])
   <=> move(0,[3,5,2])
move(0,[3,5,2])
   <=> move(1,[5,2]) & msw(die(a),1) & msw(die(b),2)
     v move(1,[5,2]) & msw(die(a),2) & msw(die(b),1)
move(1,[5,2])
   <=> move(2,[2]) & msw(die(a),1) & msw(die(b),4)
     v move(2,[2]) & msw(die(a),2) & msw(die(b),3)
     v move(2,[2]) & msw(die(a),3) & msw(die(b),2)
     v move(2,[2]) & msw(die(a),4) & msw(die(b),1)
move(2,[2])
   <=> move(3,[]) & msw(die(a),1) & msw(die(b),1)
move(3,[])
```

Figure 2.5: Examples of three queries on the program in Figure 2.4 by three instances of built-in PRISM predicates. The first is the sampling query, the second is the probability computation query and the third is the explanations query.

2.3.3 The Four PRISM Conditions

In order to model any problem in PRISM, four conditions need to be met (Sato and Kameya, 2001):

- **The uniqueness condition:** for any given sample F' from the ground facts F in the logic program $DB = F \cup R$, only one ground atom of the target predicate (goal) is true, others must be false. In the *dice* problem, if we sample three dice rolls and obtained `move([3,5,2])`, for instance, other movements must be false.
- **The exclusiveness condition:** the explanations of any goal are mutually exclusive. If E_1 and E_2 are explanations of a goal G, $P(E_1 \wedge E_2) = 0$, and $P(G) = p(E_1) \vee P(E_2) = P(E_1) + P(E_2)$.

- **The finite support condition:** as stated in the exclusiveness condition above that the probability of any goal is the sum of the probabilities of its explanations, the number of explanations of any goal needs to be finite in order for the summation to be computable.

- **The distribution condition:** the probability of a random variable taking two values simultaneously is 0. That is $p(\texttt{msw}(s,i,v_1)\ \&\ \texttt{msw}(s,i,v_2)) = 0$ where $v_1 \neq v_2$ and i is the trial number. Finally, the probability of any explanation is the product of the probabilities of the values that the random variables in this explanation has been instantiated to.

The *ProbLog* (Probabilistic Prolog) formalism drops the exclusiveness condition to tackle problems in graph and data mining (Gutmann et al., 2008). PRISM does not check whether or not the exclusiveness condition is met. In Appendix A, we propose a general procedure to model problems in PRISM where the exclusiveness condition does not apply.

2.3.4 Failure and Failure-free PRISM Programs

Let $DB = F \cup R$ be a definite logic program, where F is a set of probabilistic facts defining all possible instantiations of random variables, and R is a set of rules some of which define a target predicate. Let $F' \subseteq F$ be a set of probabilistic facts representing a particular instantiation of the random variables such that $R \cup F'$ do not entail any instance of the target predicate. Then, any derivation in an SLD-resolution of an instance of the target predicate formed by F' will lead to *failure*. Figure 2.6 shows an example of a PRISM program with failure. In this program, a single movement will only take place when the two dice show the same numbers. In this case, if we sample the probabilistic facts and obtain the set {`msw(die(a),2)`, `msw(die(b),5)`}, for instance, this set forms a derivation in an SLD-resolution which leads to failure. Therefore, there are some choices of probabilistic facts representing joint instantiations of the random variables which do not belong to the support set of the probability distribution represented by the definition of the target predicate.

When any set of probabilistic facts representing a joint instantiation of the random variables defined in the program, together with R, entail an instance of the target predicate, there is no set of probabilistic facts which form a derivation that leads to a failure. Such programs are known as *failure-free*.

```
values(die(_), [1,2,3,4,5,6]).

move(X):-
     msw(die(a), X),
     msw(die(b), Y),
     X = Y.

set_params:-
     set_sw(die(a), [0.1,0.2,0.1,0.2,0.2,0.2]),
     set_sw(die(b), [0.1,0.2,0.1,0.2,0.2,0.2]).
```

Figure 2.6: The *dice* PRISM program with failure.

Definition 4. *Let $DB = F \cup R$ be a definite logic program, where F is a set of probabilistic facts defining all possible instantiations of random variables, and R is a set of rules some of which define a target predicate p/n. DB is failure-free if and only if $\forall F' \subseteq F : F'$ is a set of probabilistic facts defining a joint instantiation of the random variables, $R \cup F'$ entail an instance of p/n.*

The class of failure-free PRISM programs can represent many models. In DBNs, all possible joint instantiations of the random variables belong to the target probability mass function. Therefore, any DBN can be represented by a failure-free PRISM program. Probabilistic productions of non-terminals in *stochastic context-free grammars* (SCFGs) are not conditioned and do not depend on the contexts in which the non-terminals appear. All joint choices of the probabilistic production rules belong to the probability mass function represented by the grammar. Therefore, any SCFG can be represented by a failure-free PRISM program. However, in *stochastic context-sensitive grammars* (SCSG), probabilistic productions of non-terminals may depend upon the contexts in which these non-terminals appear. Given non-terminals in a particular context represented by a SCSG, some productions of these non-terminals may not fit in this context, and thus, are not allowed by the grammar. Thus, some joint choices of these productions represent failures and they are excluded from the represented language. Therefore, not all SCSGs can be represented by failure-free PRISM programs.

2.4 PRISM and the Generalisation of DBNs

A set of switches defined over the same sample space can be represented using an unground term. The selection of a particular switch is then established by a substi-

tution of the variables in that term. This provides a way of encoding dependencies between the outcomes of the switches. For instance, given the two switches $\mathtt{s}_1$ and $\mathtt{s}_2$ whose outcomes are $\mathtt{SW}_1$ and $\mathtt{SW}_2$ respectively, the unground term $\mathtt{s}_3(\mathtt{X},\mathtt{Y})$ represents a set of switches amongst which, by the substitution $\theta = \{X/SW_1, Y/SW_2\}$, one is selected. Therefore, the outcome represented by the unground term $\mathtt{s}_3(\mathtt{X},\mathtt{Y})$ depends upon the outcomes of the two switches $\mathtt{s}_1$ and $\mathtt{s}_2$. This dependency is modelled as follows

```
.
.
msw(s_1,SW_1),
msw(s_2,SW_2),
msw(s_3(SW_1,SW_2),SW_3),
.
.
```

Given the above discussion, a BN can be encoded as a PRISM program. Sato and Kameya (2001) showed that the following PRISM program represents the same distribution represented by a BN

$$F = \{msw(parents_i(\mathbf{u}_i), x_i)\}_{i=1}^{n}$$
$$R = \{bn(X_1, \ldots, X_n) : - \textstyle\bigwedge_{i=1}^{n} msw(parents_i(\mathbf{U}_i), X_i)\}$$

where $\mathbf{U}_i$ is the parent set of X_i. $\mathbf{U}_i = \emptyset$ when X_i is a root node. $\mathbf{u}_i$ is a particular instantiation of the parent set $\mathbf{U}_i$ and x_i is a particular instantiation of their child X_i.

Let the rule $(\{bn(X_1^{(0)}, \ldots, X_n^{(0)}) \leftarrow \bigwedge_{i=1}^{n} msw(parents_i(\mathbf{U}_i), X_i)\})$ represent the dependencies between the variables $X_1^{(0)}, \ldots, X_n^{(0)}$ in the distribution given in (2.2). This rule represents the factor $P(X_1^{(0)}, \ldots, X_n^{(0)})$ in this distribution. The product of the factors corresponding to the states from 1 to ∞ can then be represented by the following recursive rule

$$rec\_def(X_1^{(t-1)}, \ldots, X_n^{(t-1)}, [X_1^{(t)}, \ldots, X_n^{(t)} | Tail]) : -$$
$$\textstyle\bigwedge_{i=1}^{n} msw(parents_i(\mathbf{U}_i^{(t)}), X_i^{(t)}),$$
$$rec\_def(X_1^{(t)}, \ldots, X_n^{(t)}, Tail).$$

where $X_1^{(t-1)}, \ldots, X_n^{(t-1)}$ are input arguments, $X_1^{(t)}, \ldots, X_n^{(t)}$ are the output arguments of the current instance of the recursion, $Tail$ is the output of the subsequent instances of the recursion and $\mathbf{U}_i^{(t)}$ is the parent set of $X^{(t)}$. Each instance of the recursion represents a factor $P(X_1^{(t)}, \ldots, X_n^{(t)} | X_1^{(t-1)}, \ldots, X_n^{(t-1)})$ in the product in (2.2). Therefore, the following PRISM program represents the same distribution represented by a DBN with an infinite sequence.

$$
\begin{aligned}
F &= \{msw(parents_i(\mathbf{u}_i^{(0)}), x_i^{(0)}), \ldots\}_{i=1}^n \\
R &= \{(\ infinite\_DBN([X_1^{(0)}, \ldots, X_n^{(0)}, X_1^{(1)}, \ldots, X_n^{(1)} | Tail]) : - \\
&\qquad \textstyle\bigwedge_{i=1}^n msw(parents_i(\mathbf{U}_i^{(0)}), X_i^{(0)}), \\
&\qquad rec\_def(X_1^{(0)}, \ldots, X_n^{(0)}, [X_1^{(1)}, \ldots, X_n^{(1)} | Tail]) \\
&\quad), \\
&\quad (\ rec\_def(X_1^{(t-1)}, \ldots, X_n^{(t-1)}, [X_1^{(t)}, \ldots, X_n^{(t)} | Tail]) : - \\
&\qquad \textstyle\bigwedge_{i=1}^n msw(parents_i(\mathbf{U}_i^{(t)}), X_i^{(t)}), \\
&\qquad rec\_def(X_1^{(t)}, \ldots, X_n^{(t)}, Tail) \\
&\quad) \\
&\quad \}
\end{aligned}
$$

Let a set of random variables $\{X_i^{(t)}\}_{t=0}^{\infty}$ for some $0 \leq i \leq n$ be designated to generate a value $x_{halt}^{(t)}$ which halts the process. Define a predicate `stop/m` such that, at the end of each time-slice t, this predicate definition halts the process when the value $x_{halt}^{(t)}$ has been generated, otherwise, it continues the process. The resulting PRISM program becomes (for clarity, we denote $X_i^{(t)}$, the random variable at time-slice t which generates the halting value, by $X_{halt}^{(t)}$)

$$
\begin{array}{ll}
F & = \{msw(parents_i(\mathbf{u}_i^{(0)}), x_i^{(0)}), \ldots\}_{i=1}^{n} \\
R & = \{(\ generalised\_DBN([X_1^{(0)}, \ldots, X_n^{(0)} | Tail]) : - \\
 & \quad \bigwedge_{i=1}^{n} msw(parents_i(\mathbf{U}_i^{(0)}), X_i^{(0)}), \\
 & \quad\quad stop(X_{halt}^{(0)}, X_1^{(0)}, \ldots, X_n^{(0)}, Tail) \\
 & \quad), \\
 & \\
 & \quad (rec\_def(X_1^{(t-1)}, \ldots, X_n^{(t-1)}, [X_1^{(t)}, \ldots, X_n^{(t)} | Tail]) : - \\
 & \quad\quad \bigwedge_{i=1}^{n} msw(parents_i(\mathbf{U}_i^{(t)}), X_i^{(t)}), \\
 & \quad\quad stop(X_{halt}^{(t)}, X_1^{(t)}, \ldots, X_n^{(t)}, Tail) \\
 & \quad), \\
 & \\
 & \quad (\ stop(x_{halt}^{(t)}, X_1^{(t)}, \ldots, X_n^{(t)}, [])), \\
 & \quad (\ stop(X_{halt}^{(t)} \neq x_{halt}^{(t)}, X_1^{(t)}, \ldots, X_n^{(t)}, Tail) : - \ rec\_def(X_1^{(t)}, \ldots, X_n^{(t)}, Tail)\ ) \\
 & \quad \}
\end{array}
$$

This program generalises a DBN by adding the additional modelling property of representing the halting probability. It defines the following distribution

$$
\begin{array}{l}
P(\{X_1^{(l)}, \ldots, X_n^{(l)}\}_{l=0}^{\infty}) = \\
\quad P(X_1^{(0)}, \ldots, X_n^{(0)}) \prod_{\substack{\forall 1 \leq t \leq \infty: \\ \nexists 1 \leq i \leq n,\, X_i^{(t-1)} = x_{halt}^{(t-1)}}} P(X_1^{(t)}, \ldots, X_n^{(t)} | X_1^{(t-1)}, \ldots, X_n^{(t-1)})
\end{array}
$$

The class of the above PRISM programs is the focus of this thesis.

In Figure 2.7, we show a PRISM program which generalises the HMM model in Figure 2.3 by defining a halting distribution based upon the outcome of the `halt(_)` set of switches. When sampling from this program, if the outcome of `halt(_)` is `'yes'` the process stops, otherwise it continues to generate another symbol (a or b). Inter-slice dependencies in DBNs are represented in PRISM programs by input arguments to the recursive predicate (variables `Next` in Figure 2.7). In the PRISM context, we will refer to these dependencies as *inter-iteration dependencies*. Likewise, intra-slice dependencies will be referred to as *intra-iteration dependencies*.

```
values(init,[q1,q2,q3]).
values(out(_),[a,b]).
values(tr(_),[q1,q2,q3]).
values(halt(_),[yes,no]).

hmm(L):-
     msw(init,S),
     hmm(S,L).

hmm(S,[Ob|Y]) :-
     msw(out(S),Ob),
     msw(tr(S),Next),
     msw(halt(Next),Decision),
     stop(Decision,Next,Y).

stop(yes,_,[]).
stop(no,Next,Y):-
     hmm(Next,Y).
```

Figure 2.7: A PRISM program generalising the distribution defined by the HMM in Figure 2.3.

2.5 Learning with PRISM

2.5.1 Parameter Estimation

Given a PRISM program DB and some observations D, PRISM allows estimating the parameters using both the MLE approach and the Bayesian approach. Parameters can be estimated in both cases when observations are fully observed and when there are some missing or hidden values. In MLE, PRISM uses an adapted version of the EM algorithm called *graphical EM* (gEM). gEM works on the SLD-resolution (explanation graph) of the observations. It uses dynamic programming to reuse the values in the iterations of the algorithm. This increases the efficiency of the gEM over a naïve application of the EM algorithm. However, in case of programs with failures, the derivations which lead to failures need to be excluded from the mass function and the probability of refutations need to be normalised. PRISM adopts the *failure-adjusted maximisation* (FAM) algorithm proposed by Cussens (2001). FAM is an instance of the EM algorithm meant to estimate the parameters of stochastic logic programs with failures. In the Bayesian approach, a MAP estimation of the parameters is supported. Given the Dirichlet prior shown in (1.7), where the hyperparameters are set by the user, and some observations $D = d_1, \ldots, d_n$, the MAP estimation finds the following quantity ($P(\boldsymbol{\theta}|D)$ is defined as in (1.9))

$$\arg\max_{\boldsymbol{\theta}} P(\boldsymbol{\theta}|D)$$

2.5.2 Structure Learning

Though PRISM does not provide any structure learning algorithm, it does support three scoring functions to score candidate programs against observations. These scoring functions are the BIC score, the variational free energy score and the Cheeseman-Stutz score. They all provide different approximations of the *log marginal likelihood* which is defined as follows (S is the structure)

$$\log P(D|S) = \log \int_{\boldsymbol{\theta}} P(\boldsymbol{\theta}|S) P(D|S, \boldsymbol{\theta})\, d\boldsymbol{\theta} \tag{2.4}$$

We discuss the BIC score and the variational free energy score in Chapter 4.

Chapter 3

Inductive Logic Programming

This chapter surveys different techniques used to learn logic programs from examples. The first section highlights the different settings of ILP. It provides an overview of the main principles that are used in the area. Section 3.2 discusses the main learning approaches used in ILP. Finally, Section 3.3 goes through one of the most challenging problems in ILP which is learning recursive clauses. It also explains MERLIN 2.0, a system which will be adapted in Chapter 5 to learn recursive PRISM programs.

3.1 Inductive Logic Programming

Inductive logic programming emerged out of the research in concept learning (Flach and Lavrač, 2002). In concept learning, the problem is to learn from specific examples of the concept a general definition to which these examples belong. This general definition can then be used to determine, with a high degree of accuracy, whether or not any further unseen example belongs to this concept. If an example e belongs to the concept H, then e is *covered* by H, otherwise H *does not cover* e. In relational data, concepts need to be defined in such a way that these relations are captured. As FOL is a relational representation, ILP concerns learning concept definitions in FOL. The learning is either performed from positive examples (those which belong to the concept) and negative examples (those which do not belong to the concept) or from only positive examples. In learning from both positive and negative examples, the problem is defined as follows (Muggleton, 1991; Flach and Lavrač, 2002; Lavrač and Džeroski, 1994)

Given:

- A set of examples $E = E^+ \cup E^-$ consisting of predicates called *foreground predicates* where E^+ is a set of positive examples and E^- is a set of negative examples.
- Background knowledge (BK) B consisting of predicates called *background predicates.*
- A declarative bias L specifying a form of clauses that can be built using the predicates in B and E.

Find (acceptance condition): a hypothesis H whose clauses are in the form L such that $\forall e^+ \in E^+ : covers(H \wedge B, e^+) = true$ (complete) and $\forall e^- \in E^- : covers(H \wedge B, e^-) = false$ (consistent).

Flach and Lavrač (2002) divide the tasks in ILP into two streams, *descriptive ILP* and *predictive ILP*. In the former, the hypotheses considered are first-order clauses which describe some correlations between objects in the domain. A typical example of this task is relational data mining and the induction of association rules. We omit the discussion of descriptive ILP and concentrate on the latter. In predictive ILP, more restrictions are imposed on the considered hypotheses. Commonly E and B are restricted to Horn clauses and the considered hypotheses are restricted to definite clauses.

Different ILP settings have been proposed based upon the definition of coverage (De Raedt, 1997). These are (a) *learning from entailment* (Muggleton, 1991), (b) *learning from interpretations* (De Raedt and Van Laer, 1995; De Raedt and Dehaspe, 1997b) and (c) *learning from satisfiability* (De Raedt and Dehaspe, 1997a). The following are definitions of coverage in these settings

Definition 5. *Let H be a hypothesis, B be background knowledge and e be an example. In learning from entailment, $covers(H \wedge B, e) = true$ if and only if $H \wedge B \models e$.*

Definition 6. *Let H be a hypothesis, B be background knowledge and e be an example. In learning from interpretations, $covers(H \wedge B, e) = true$ if and only if $I = B \cup \{e\}$ is an interpretation of H.*

Definition 7. *Let H be a hypothesis, B be background knowledge and e be an example. In learning from satisfiability, $covers(H \wedge B, e) = true$ if and only if $H \wedge B \wedge e \not\models \square$.*

In predictive ILP, learning from entailment has been the most common learning setting (Kersting, 2006). Therefore, we assume this setting in this thesis whenever ILP is discussed. Typically, in learning from entailment, examples are restricted to ground atoms. Thus, this form of examples will also be maintained throughout.

Under the *closed world assumption (CWA)*, in which literals that are not entailed by the theory under consideration are believed to be false, a theory which consists of the positive examples and none of the negative examples satisfies the acceptance condition in the above definition. However, this theory overfits the examples and does not scale to unseen instances. Therefore, it is typically the case that given two hypotheses satisfying the acceptance condition, the more general hypothesis is preferred over the less general one. The generality of two hypotheses can be defined as follows (Muggleton, 1991; De Raedt, 2008)

Definition 8. *A hypothesis H_1 is more general than, or a generalisation of, a hypothesis H_2, denoted as $H_1 \preceq H_2$, if and only if $H_1 \models H_2$. H_2 is also said to be a specialisation of H_1. H_1 is strictly more general than H_2, denoted as $H_1 \prec H_2$, if and only if $H_1 \preceq H_2$ and $H_2 \not\preceq H_1$.*

Definition 9. *A minimal generalisation H' of H is a generalisation of H such that $H' \prec H$, and there does not exist a generalisation H'' of H such that $H'' \prec H$ and $H' \prec H''$.*

Definition 10. *A maximal specialisation H' of H is a specialisation of H such that $H \prec H'$, and there does not exist a specialisation H'' of H such that $H \prec H''$ and $H'' \prec H'$.*

Searching the space of hypotheses is then performed based upon the coverage of the current hypothesis by either specialising it or generalising it. There can be a combinatorial explosion of the number of hypotheses that generalise or specialise any given one. Therefore, Muggleton and De Raedt (1994) defined two restricted forms of specialisation and generalisation operators as follows

Definition 11. *Let H be a hypothesis, a generalisation operator maps the clauses in H onto a set of minimal generalisation of H.*

Definition 12. *Let H be a hypothesis, a specialisation operator maps the clauses in H onto a set of maximal specialisation of H.*

θ-subsumption is a form of deduction in FOL proposed by Plotkin (1970) which is employed as an operational framework for specialising and generalising hypotheses. Let c_1 and c_2 be two clauses represented as the two sets of the literals they contain, then θ-subsumption is defined as follows (Muggleton and De Raedt, 1994)

Definition 13. *A clause c_1 θ-subsumes a clause c_2 if and only if there exists a substitution θ such that $c_1\theta \subseteq c_2$. In such a case, c_1 is a generalisation of c_2.*

Though Muggleton and De Raedt restricted the search operators, they also pointed out some properties of θ-subsumption which still cause problematic issues to perform search and need to be resolved. The first problem is that, starting from a clause c_1, there exist an infinite descending chain of clauses $c_1 \prec c_2 \prec \ldots \prec c_\infty \prec c_2'$ where c_i is a clause with i number of literals and c_2' is a lower bound. Following is an example given by Muggleton and De Raedt (1994) ($h(X, X) \leftarrow p(X, X)$ is the lower bound of the chain)

$$
\begin{aligned}
&h(X_1, X_2)\\
&h(X_1, X_2) \leftarrow p(X_1, X_2)\\
&h(X_1, X_2) \leftarrow p(X_1, X_2), p(X_2, X_3)\\
&h(X_1, X_2) \leftarrow p(X_1, X_2), p(X_2, X_3), p(X_3, X_4)\\
&\ldots\\
&h(X, X) \leftarrow p(X, X)
\end{aligned}
$$

With this chain, a specialisation function may not halt. The problem of generalising from the lower bound is even worse as the minimal generalisation of the lower bound is the clause c_∞ which is undefined in practice. The second problem is that there are several clauses which are equivalent under θ-subsumption. For instance, the clause $h(X_1, X_2) \leftarrow p(X_1, X_2)$ and the clause $h(X_1, X_2) \leftarrow p(X_1, X_2), p(X_1, X_3)$ θ-subsume one another. From the properties of θ-subsumption, these two clauses are also logically equivalent (Muggleton and De Raedt, 1994), and thus one of them is enough to generate. The declarative bias which defines the form of the hypothesis sought can be used to solve the former problem. The latter problem is alleviated by working only with *reduced clauses* (Plotkin, 1970). The reduced clause c' of another clause c is the clause with the minimal subset of literals in c such that c' is logically equivalent to c (Muggleton and De Raedt, 1994). Reduced clauses form a θ-subsumption lattice with a lower bound (the most specific clause) and an upper bound (the most general clause). Two specialisation operators under θ-subsumption are then defined: (a) applying a substitution θ to a clause and (b) adding a literal to a clause. Though

a generalisation of single clause under θ-subsumption may not exist due to undefinedness (as in the case above of generalising the lower bound), generalisation under θ-subsumption does exist for any two clauses. It is called the *least general generalisation* (lgg) (Plotkin, 1970). The lgg of the two terms $f(t_1, \ldots, t_n)$ and $f(x_1, \ldots, x_n)$ is the term $f(lgg(t_1, x_1), \ldots, lgg(t_n, x_n))$. The lgg of the two terms f/n and g/m where $f \neq g$ or $n \neq m$ is a variable V which represents these two terms throughout. The lgg of two atoms $p(t_1, \ldots, t_n)$ and $p(x_1, \ldots, x_n)$ is the atom $p(lgg(t_1, x_1), \ldots, lgg(t_n, x_n))$. The lgg of two atoms having different predicate symbols, different numbers of arity or different signs is undefined. The lgg of two clauses c_1 and c_2 whose sets of literals are $c_1 = \{l_i\}_{i=1}^{n}$ and $c_2 = \{l'_j\}_{j=1}^{m}$ is the clause $c_3 = \{lgg(l_i, l'_j) | 1 \leq i \leq n, 1 \leq j \leq m\}$.

Buntine (1988) showed a weakness of generalising two clauses with lgg. He stated that only these clauses are considered in the generalisation and background knowledge is not utilised. Buntine highlighted the problem of generalising the following two clauses

$$\begin{array}{l} cuddly\_pet(X) \leftarrow small(X), fluffy(X), dog(X) \\ cuddly\_pet(X) \leftarrow fluffy(X), cat(X) \end{array} \tag{3.1}$$

Given the following background knowledge,

$$\begin{array}{l} pet(X) \leftarrow cat(X) \\ pet(X) \leftarrow dog(X) \\ small(X) \leftarrow cat(X) \\ tame(X) \leftarrow pet(X) \end{array}$$

A salient generalisation would be

$$cuddly\_pet(X) \leftarrow small(X), fluffy(X), pet(X)$$

However, not considering the background knowledge and generalising by the lgg of the two clauses in (3.1) will lead to the following over-generalising clause

$$cuddly\_pet(X) \leftarrow fluffy(X)$$

Relative least general generalisation (rlgg) proposed by (Plotkin, 1970) concerns generalising two clauses c_1 and c_2 relative to a theory T under θ-subsumption. Buntine (1988) proposed a special case of rlgg where the two clauses are definite clauses and called it *generalised subsumption*. Muggleton and Feng (1990) further specialise rlgg to the case where c_1 and c_2 are ground atoms. In this case, they defined the rlgg of c_1 and c_2 relative to a background theory T as the clause c_3 which is the minimal generalisation clause in the θ-subsumption lattice for which $T \wedge c_3 \vdash c_1 \wedge c_2$. Muggleton and Feng derived a method for constructing the rlgg accordingly. The method builds the two clauses $c_1 \leftarrow a_{11}, \ldots, a_{1n}$ and $c_2 \leftarrow a_{21}, \ldots, a_{2n}$ where $a_{i1}, \ldots, a_{in}, 1 \leq i \leq 2$ are models of T. The rlgg is then the lgg of these two clauses.

In learning from only positive examples, some criterion needs to be defined to restrict the generalisation of hypotheses. Typically, a scoring function is developed to measure each candidate hypothesis in term of coverage and complexity. If the scoring function is extremely biased towards coverage, the hypothesis might overfit the examples and does not generalise well. However, a simple hypothesis might overgeneralise and instances which do not belong to the target concept might also be covered. Therefore, a trade-off between complexity and coverage is conventionally sought in developing scoring functions. We define the problem of learning from only positive examples as follows

Given:

- A set of examples E consisting of predicates called *foreground predicates*.
- Background knowledge (BK) B consisting of predicates called *background predicates*.
- A declarative bias L specifying a form of clauses that can be built using the predicates in B and E.

Find: a hypothesis H whose clauses are in the form L such that $\forall e \in E : B \cup H \models e$ and

$$\arg\max_{H} \ score(H, E)$$

Muggleton (1997) used the posterior distribution $P(H|E)$ for scoring in the ILP system Progol4.2. To derive the formula of $P(H|E)$, a prior distribution over hypotheses $P_H(H)$ is considered which decreases monotonically with H's complexity.

Complexity was measured by the number of atoms in H. By defining a prior distribution over examples $P_E(E)$, the likelihood is then defined as $P(E|H) = \prod_{i=1}^{m} \frac{P_E(e_i)}{P_E(H)}$, where $P_E(H) = \sum_{\forall e:covers(H,e)} P_E(e)$. The posterior was then derived to take the following form (C_m is a fixed constant)

$$\log P(H|E) = m \log \frac{1}{P_E(H)} + \log P_H(H) + \log C_m$$

The trade-off between complexity and generality in the above score can be noticed in the first two terms. As $P_E(H)$ (the generality factor) increases, the term $m \log \frac{1}{P_E(H)}$ decreases. In such a case, the complexity decreases (general hypotheses are normally simple) leading to an increase in $P_H(H)$ which in turn increases the second term. The opposite scenario applies when $P_E(H)$ decreases. When the number of examples m increases, more weight is put on the generality term as data carries more information. Boström (1998) uses the same idea of maximising the posterior distribution in the ILP system MERLIN 2.0. However, Boström used a different setting of prior and thus a different form of posterior was derived. We explain MERLIN 2.0 in details in Section 3.3.1.

3.2 Learning Approaches

In this section, we highlight different learning approaches used in ILP. The first three sections discuss different ways of searching the search space. Section 3.2.4 discusses an ILP approach in which an initial (incomplete or inconsistent) theory is given to the learning algorithm, and the task is to revise this theory with the light of new examples. Finally, we discuss the problem when the foreground and background predicates are not sufficient to represent the target theory and need to be extended by inventing new predicates.

3.2.1 Top-Down Approach

The top-down approach is incremental with respect to the positive examples. It starts handling a positive example by searching within the θ-subsumption lattice from the most general clause. It specialises this clause with respect to the negative examples, or a scoring function when learning from only positive examples. Specialisation stops when no further negative examples are covered or no improvement in the scoring

function is achieved. Positive examples which are covered by the induced clause are then deleted and the process is repeated with the first uncovered positive example to induce a further clause. The process stops when all positive examples are covered. The search space is restricted by the declarative bias from which a sub-lattice is defined such that the most general clause (the empty clause) and the most specific clause $\perp$, also called the *bottom clause*, are fixed. The ILP system Progol (Muggleton, 1995) and its variant Aleph[1] use the approach of mode declarations to inject the declarative bias. A mode declaration is either of the form `modeh(n,atom)`, defining the mode of the head of the clauses in the hypothesis, or `modeb(n,atom)`, defining the mode of the atoms in the body of the clauses in the hypothesis. `atom` is a ground atom whose terms are either *normal* or *place-marker*. A normal term is either a constant or a functor with some other terms. A place-marker is either of the form `#type`, in which case the term is defined as a ground term in the atom, `+type`, in which case the term is an input variable to the atom, or `-type`, in which case the term is an output variable in the atom. `type` is the type of the terms passed to the atom. The following is an example of a mode declaration for learning categorisation rules for animals[2]

$modeh(1, class(+animal, -class))$.
$modeb(1, has\_covering(+animal, \#covering))$.
$modeb(1, homeothermic(+animal))$.
$modeb(1, not(has\_gills(+animal)))$.
$modeb(*, habitat(+animal, \#habitat))$.
$modeb(1, has\_gills(+animal))$.
$modeb(1, has_legs(+animal, \#nat))$.
$modeb(1, has\_eggs(+animal))$.
$modeb(1, nhas\_gills(+animal))$.
$modeb(1, has\_milk(+animal))$.

`class/2` is defined as the head of the clauses in the sought hypothesis. The arguments to `class/2` are an input of type `animal` and an output of type `class`. The atoms in `modeb` are those which are allowed to be included in the body of any clause. Types are then defined as follows

$animal(dog)$. $animal(dolphin)\ldots$.
$class(mammal)$. $class(fish)\ldots$.
$\ldots$

Muggleton (1995) stated that given a mode declaration M, a definite clause c is in the form specified by the declarative bias L defined by the mode declaration M if and only if:

[1] http://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph.html
[2] The example is supplied with Aleph.

- The head of c is the atom of *modeh* in M with every `#type` replaced by a ground term of the type `type`, and every `-type` and `+type` replaced by variables.
- Every atom in the body of c is an atom of some `modeb` in M with every `#type` replaced by a ground term of the type `type`, and every `-type` and `+type` replaced by variables.
- Every variable corresponding to the place-marker `+type` in any atom b_i in the body of c is either in the head of c corresponding to `+type` or in some other atom b_j corresponding to `-type` where $1 \leq j < i$.

The variables in the bottom clause constructed using the mode declaration M constitute a chain such that each variable in any atom in the body is bound to the variables in the head by some depth. Muggleton defined the depth of variable V as follows

Definition 14.

$$depth(V) = \begin{cases} 0 & \textit{if } V \textit{ is in the head} \\ (max_{U \in S_V} depth(U)) + 1 & \textit{otherwise} \end{cases}$$

where S_V is the set of variables in the body atoms containing V.

The algorithm of building the bottom clause given in Muggleton (1995) adds atoms to the body by maintaining a specific depth of the variables in these atoms. For any atom to be added to the body, an assumption is made that it has some input variables (the atom is defined in mode declaration with at least one `+type` place-marker). Given this assumption and the last condition above of clauses satisfying the mode declaration, the head of any clause needs to be defined to have some input in order for the algorithm building the bottom clause to add some atoms in the body. Each atom is then part of a chain of input-output variables within the designated depth. This assumption is convenient for learning discriminative models in which some input values (predictors) are fed to the model, and this model returns some output values (predicted) accordingly.

3.2.2 Bottom-Up Approach

The bottom-up approach goes in the opposite direction of the top-down approach. It starts with the most specific clause which covers one example and generalises it as long as no negative example is covered. Golem is an early ILP system which uses a bottom-up approach (Muggleton and Feng, 1990). Golem works with extensional background knowledge. Therefore, if intensional background knowledge is given, it

is replaced with its ground model. Then the rlgg of the first two positive examples is computed with respect to the background knowledge. To compute the rlgg, two clauses whose heads are the examples and whose bodies are the ground literals in the background knowledge are built, and then the lgg of these two clauses is computed. This may lead to a clause with many literals in its body, some of which are redundant. A literal l is redundant in a clause C given the background knowledge B if $C \wedge B \vdash (C - \{l\})$ (Muggleton and Feng, 1990). These redundant literals are removed from the clause. The resulting clause can be generalised by removing some body literals. The depth of the variables can be used to prioritise literals to be removed. Negative examples, in the case of learning from both positive and negative example, or a scoring function, in the case of learning from only positive examples, can be used to restrain the search from over-generalising the clause. Positive examples which are covered by the resulting clause are then removed and the process is repeated to handle the remaining examples. Flach (1994) provides a Prolog implementation of this learning algorithm.

3.2.3 Random Search

Though top-down and bottom-up approaches have been used successfully in many domains, they are both greedy approaches which suffer the problem of being trapped in a local maximum. In greedy approaches, a consistent clause is returned once it has been found in a branch of the θ-subsumption lattice (sub-lattice). However, another consistent and more general clause might have been found could other branches have been explored. More general clauses will lead to a more general final hypothesis. The other problem with the greedy approaches is that searching the lattice in sequence may lead to testing many clauses before a consistent one is found. This process could be slow in large domains. To overcome these problems, variants of random search have been proposed. Srinivasan (2000) studied the applicability of random search algorithms in ILP. Muggleton and Tamaddoni-Nezhad (2008) proposed a genetic algorithm for searching the lattice under the system Progol4.6. Serrurier and Prade (2008) used simulated annealing search to induced more than one clause in a single step.

3.2.4 Theory Revision

Theory revision is an approach where an initial domain theory is given, and this theory is then updated according to some new observations. A main direction in which

theory revision can be applied is when one develops a theory which is complete and consistent with regards to some (possibly not enough) examples, but this theory might be incomplete or inconsistent when new examples emerge. It is therefore necessary to revise this theory according to the new examples. Adé et al. (1994) developed RUTH, a system in which an initial theory which satisfies current integrity constraints is given. When new integrity constraints emerge such that the initial theory no longer satisfies them, this theory is revised accordingly. Another area where theory revision can be applied is when the search space is very large. A possible direction to take is to start with an initial theory which is either incomplete (over-specialised) or inconsistent (over-generalised). This theory is then either specialised or generalised according to the examples. This technique reduces the search space as the initial theory, though not accepted, provides useful information to the learning algorithm on the final theory. This makes the learning algorithm start from the point of the initial theory and moves to the areas around it instead of starting from a point which is far from any acceptable hypothesis.

3.2.5 Predicate Invention

In certain cases the learning algorithm does not find a complete and consistent hypothesis that can be represented by the foreground and background predicates. It is thus convenient to design a learning algorithm which may go beyond these predicates and introduce new predicates which can be used in the final hypothesis. This process of introducing new predicates is called *predicate invention*. Introducing new predicates will extend the search space significantly and put more challenges to the learning algorithm. It is therefore convenient to limit the extended search space by limiting the predicates that can be introduced.

3.3 Learning Recursive Clauses

Learning recursive clauses is one of the challenging problems in ILP. Though the approaches discussed in Section 3.2 based on clause by clause learning are used to learn recursive clauses, they suffer from different problems which make them difficult to use in general. One of the main problems is that the examples have to come in a certain order to learn a proper recursive clause. For example, in learning the definition of the predicate `member/2`, when the examples come in the following order

$$member(a, [b, a, c, d]).$$
$$member(a, [e, f, g, a, c]).$$
$$member(a, [a, e, d]).$$
$$\ldots$$

and a clause by clause learning is used, the following hypothesis might be induced

$$member(X, [Y, X|Z]).$$
$$member(X, [Y, E|Z]) : -member(X, [E|Z]).$$
$$member(X, [X|Z]).$$

It is obvious that this hypothesis can be reduced to the following one

$$member(X, [X|Z]).$$
$$member(X, [Y|Z]) : -member(X, Z).$$

The reason that the first clause was learned unnecessarily is that, at the point of learning it, no other clause was learned to which a recursive clause can be added to cover the first example. Changing the order of the examples to the following

$$member(a, [a, e, d]).$$
$$member(a, [b, a, c, d]).$$
$$member(a, [e, f, g, a, c]).$$
$$\ldots$$

leads to the base clause being learned from the first example, and when the second example is handled, a recursive clause can be learned which together with the base clause cover it. Therefore, the desired hypothesis can be learned clause by clause having set the examples in the right order. However, in practice, examples can come in an arbitrary order and one cannot know what the right order is. In this case, we need an approach in which the recursive definition (all the clauses involved) can be learned from the same set of examples. Aha et al. (1994) proposed CRUSTACEAN, an ILP system which learns recursive definitions. CRUSTACEAN induces the base clause first from the examples. It then uses the same set of examples to learn a recursive clause which together with the base clause constitute a theory that covers the examples. The recursive clauses that can be learned by CRUSTACEAN are limited

to two literals (one literal and the recursive call). Cohen (1993) developed FORCE2 which learns one base clause and one tail recursive clause given that there is a function which can decide whether an example belongs to the base clause or to the recursive clause. The system TIM learns the same class of recursive definitions that are learned by FORCE2 but does not need the decision function (Idestam-Almquist, 1996). To induce a recursive clause, TIM analyses some regularities as well as the chains of input-output arguments in the atoms of the bottom clause. MRI is a system which adapts the idea used in TIM of analysing the regularities and learns multiple recursive clauses (Furusawa et al., 1997). MERLIN 2.0 is a theory revision system which learns a theory with recursive clauses from only positive examples by specialising an overly generalised theory. The advantage of MERLIN 2.0 is that it does not construct bottom clauses, and thus does not depend on an input-output chains of arguments. This motivates adapting MERLIN 2.0 to learn generative models where arguments can all be output to represent the joint distribution.

3.3.1 MERLIN 2.0

MERLIN 2.0 learns a recursive logic program by constructing a *deterministic finite state automaton* (DFA) of the SLD-resolutions of the positive examples from an overly generalised theory (Boström, 1998). A DFA is defined as follows

Definition 15. *A deterministic finite state automaton is a 5-tuple* $(\Sigma, \mathbf{Q}, q_0, \mathbf{F}, \delta)$*, where*

- Σ *is a finite alphabet.*
- $\mathbf{Q}$ *is a finite, non-empty, set of states.*
- $q_0 \in \mathbf{Q}$ *is an initial state.*
- $\mathbf{F} \subseteq \mathbf{Q}$ *is a set of final states.*
- $\delta : \mathbf{Q} \times \Sigma \rightarrow \mathbf{Q}$ *is a transition function*

A *nondeterministic finite state automaton* (NFA) has the state transition function defined as $\delta : \mathbf{Q} \times \Sigma \rightarrow 2^{\mathbf{Q}}$, where $2^{\mathbf{Q}}$ is the *power set* of $\mathbf{Q}$. A *finite-state automaton* (FSA) can either be a DFA or an NFA.

MERLIN 2.0 specialises the overly generalised theory by inventing predicates and maximising the posterior distribution. It adapts the *best-first model merging* algorithm of learning the structures of hidden Markov models developed by Stolcke and

Omohundro (1994). The best-first model merging algorithm is used to learn an HMM whose emissions are the clauses in the initial theory used to generate the positive examples. MERLIN 2.0 converts the HMM to a *deterministic HMM*, where no two transitions from any state q_i lead to two states which emit the same symbol. The deterministic HMM is then converted to an FSA. Because the resulting FSA comes from a deterministic HMM, it is thus a DFA.

When the DFA of the SLD-resolutions of the examples is constructed, MERLIN 2.0 represents a set of possible sequences of clauses in the initial theory which can be used in the SLD-resolutions of the examples as a *context-free grammar* (CFG) (Hopcroft et al., 2000). It then finds the intersection of the CFG and the induced DFA. This intersection is represented as a CFG. Finally, the CFG representing the intersection is converted to a logic program. Boström shows an example of learning a logic program which accepts strings represented by the regular grammar $a * b+$ (any string which starts with any number of a's, possibly none, followed by at least one b) given the following over-generalising theory

$$(c_1)\ p([]).$$
$$(c_2)\ p([a|X]) : -p(X).$$
$$(c_3)\ p([b|X]) : -p(X).$$

The above theory clearly accepts a wider range of strings that do not belong to the target language, e.g. $[a], [a, b, a], [], \ldots$. Given some examples of the target language, e.g. $[a, a, a, b, b], [a, a, b, b, b, b], [b, b], \ldots$, MERLIN 2.0 induces the DFA in Figure 3.1 from the SLD-resolutions of the examples from the initial theory. The induction of this DFA is performed by: (a) learning an HMM which represents the generation of these examples from the initial theory, (b) converting this HMM to a deterministic HMM and finally (c) converting the deterministic HMM to an FSA. A CFG representing a set of sequences of the clauses in the initial theory that can be used in the SLD-resolutions of the examples is then built as follows (in this case, the CFG is also a regular grammar)

$$P/1 \rightarrow c_1$$
$$P/1 \rightarrow c_2\ P/1$$
$$P/1 \rightarrow c_3\ P/1$$

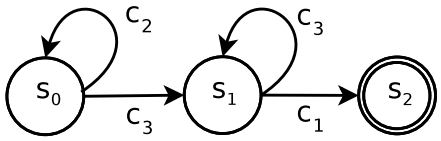


Figure 3.1: The DFA induced by MERLIN 2.0 form the SLD-resolutions of some examples, e.g. $[a, a, a, b, b], [a, a, b, b, b, b], [b, b], \ldots$ from an initial theory. c_i is clause number i in the theory.

The intersection of this grammar and the induced DFA is then constructed. The following final program is then produced from this intersection

$$
\begin{array}{l}
p([a|X]) : -p(X). \\
p([b|X]) : -p\_1(X). \\
p\_1([]). \\
p\_1([b|X]) : -p\_1(X).
\end{array}
$$

Note that MERLIN 2.0 had to invent the predicate `p_1/1` in order to represent the target theory.

In subsequent sections, we first explain the best-first model merging algorithm and then explain how MERLIN 2.0 uses it to induce the DFA.

3.3.1.1 Best-first Model Merging

Best-first model merging is an HMM learning algorithm that searches for an HMM structure S with the highest posterior probability $P(S|D)$. We show here the derivation of the posterior probability that the algorithm maximises and then show the learning algorithm which uses a hill-climbing search.

The posterior probability is averaged with respect to the posterior distribution of the parameters $\boldsymbol{\theta}$ computed with the *Viterbi approximation* in which only the most likely path is considered. First, a *Dirichlet distribution* given in (1.7) is defined as a prior for the parameters as follows

$$
P(\boldsymbol{\theta}) = \frac{1}{B(\alpha_1, \ldots, \alpha_n)} \prod_{i=1}^{n} \theta_i^{\alpha_i - 1} \tag{3.2}
$$

The normalising constant in (3.2) is the multinomial Beta function given in (1.8). The likelihood for a sample of observed i.i.d. emissions where $(c_1, \ldots, c_n)$ is the sufficient statistics (the counts of each emission in this sample) is

$$P(c_1, \ldots, c_n|\boldsymbol{\theta}) = \prod_{i=1}^{n} \theta_i^{c_i} \tag{3.3}$$

From (3.2) and (3.3) the posterior distribution of the parameters becomes

$$P(\boldsymbol{\theta}) = \frac{1}{B(c_1 + \alpha_1, \ldots, c_n + \alpha_n)} \prod_{i=1}^{n} \theta_i^{c_i + \alpha_i - 1} \tag{3.4}$$

The integral of the product of the prior (3.2) and the likelihood (3.3) can be expressed in a closed form

$$\begin{aligned} \int_{\boldsymbol{\theta}} P(\boldsymbol{\theta}) P(c_1, \ldots, c_n|\boldsymbol{\theta})\, d\boldsymbol{\theta} &= \frac{1}{B(\alpha_1, \ldots, \alpha_n)} \int_{\boldsymbol{\theta}} \prod_{i=1}^{n} \theta_i^{c_i + \alpha_i - 1}\, d\boldsymbol{\theta} \\ &= \frac{B(c_1 + \alpha_1, \ldots, c_n + \alpha_n)}{B(\alpha_1, \ldots, \alpha_n)} \end{aligned} \tag{3.5}$$

On the other hand, the *description length* is used as a prior of the HMM structure. If structure S has a code length $\ell(S)$, then a prior distribution can be

$$P(S) \propto e^{\ell(S)} \tag{3.6}$$

If there are $|\mathcal{Q}|$ possible transitions from a state q, then each one can be encoded using $\log(|\mathcal{Q}| + 1)$ bits (a special end marker is added to disallow the encoding of missing transitions explicitly). So the total description length for all transitions from a state q is $n_t^{(q)} \log(|\mathcal{Q}| + 1)$. Similarly, the description length for all emissions from q is $n_e^{(q)} \log(|\Sigma| + 1)$. The prior distribution over the HMM structures is then

$$P(S^{(q)}) \propto (|\mathcal{Q}| + 1)^{-n_t^{(q)}} (|\Sigma| + 1)^{-n_e^{(q)}} \tag{3.7}$$

The likelihood of a structure S defined as

$$P(D|S) = \int_{\boldsymbol{\theta}} P(\boldsymbol{\theta}|S)P(D|S,\boldsymbol{\theta})\, d\boldsymbol{\theta} \tag{3.8}$$

cannot be expressed in a closed form because the term $P(D|S,\boldsymbol{\theta})$ is a sum over all possible paths. By considering only the *Viterbi* path of each output and using the formula in (3.5), the posterior distribution of the structure defined as

$$P(S|D) \propto P(S)P(D|S) \tag{3.9}$$

which is the desired score to maximise can be written as follows (Boström, 1998)

$$P(S|D) \propto \prod_{q \in Q} (|\mathcal{Q}|+1)^{-n_t^{(q)}} (|\Sigma|+1)^{-n_e^{(q)}} F(t_{q_1}, \ldots, t_{q_{n_{t(q)}}}) F(e_{q_1}, \ldots, e_{q_{n_{e(q)}}}) \tag{3.10}$$

By setting $\alpha = \frac{1}{n}$, $F(t_1, \ldots, t_n)$ is defined as follows

$$\frac{B(t_1 + \frac{1}{n}, \ldots, t_n + \frac{1}{n})}{B(\frac{1}{n}, \ldots, \frac{1}{n})} \tag{3.11}$$

The best-first model merging algorithm is shown in Algorithm 1 (Stolcke and Omohundro, 1994).

3.3.1.2 Learning Deterministic HMMs

Boström pointed out that when a new sample is incorporated in the best-first model merging algorithm, a new sub-structure is created and the merging process takes place. However, the sample could be accepted (or large part of it) by the current structure without the need to create sub-structures. The creation of sub-structures will lead to non-determinism in the final induced structure. To make the structure deterministic, Boström suggested aligning any new sample with the current structure and introducing new states and transitions only for the suffix of the sample for which there is no path in the current structure. He also suggested merging two states which have transitions to two different states emitting the same symbol. These steps were implemented in MERLIN 2.0.

Algorithm 1 Best-first model merging

```
1: S_0 = the empty structure, i = 0
2: for j = 1 to k do
3:    Incorporate a new sample d_j into the structure S_i
4:    loop
5:       Compute a set of candidate merges K of the states of the current structure
         S_i
6:       For each candidate k ∈ K build the structure k(S_i) and compute its posterior
         P(k(S_i)|D)
7:       Set S_{i+1} = k*(S_i) where k*(S_i) is the structure with the merge that maximises
         the posterior
8:       if P(S_{i+1}|D) < P(S_i|D) then
9:          break
10:      else
11:         i = i + 1
12:      end if
13:   end loop
14: end for
15: return S_i
```

3.3.1.3 From an HMM to an FSA

To convert an HMM to an FSA, all the states and the transitions between the states in the HMM are mapped into the FSA as they are. Boström (1998) states that, for each state, either the transition leading to the state or the transition going from the state in the FSA is labelled with the emission of the corresponding state in the HMM. In MERLIN 2.0, Boström takes the approach of labelling the transitions leading to a state in the FSA with the emission of the corresponding state in the HMM. Therefore, the transition function of the FSA is defined such that, for each state $q^{(t)}$ in the HMM with an emission $x^{(t)}$, $\delta(q^{(t-1)}, x^{(t)}) = q^{(t)}$. All states from which there is a transition to a final state in the HMM become final states in the FSA.

Chapter 4

Learning Bayesian Networks

Two different approaches have been followed in learning BNs. The first is the *constraints based* approach and the second is the *search and score* approach. In the constraint based approach, d-separation constraints are given to the learning algorithm and the task is to learn a BN that satisfies these constraints (Jensen and Nielsen, 2007). The *search and score* approach is an optimisation problem where the tasks is to find a BN that maximises a given scoring function. This chapter discusses the latter. Section 4.1 introduces the search and score problem and highlights its hardness. Section 4.2 goes through different scoring functions for BNs with all the variables observed as well as for BNs with some hidden variables. Section 4.3 explains four search strategies. Finally, Section 4.4 discusses learning DBNs.

4.1 Introduction

The task of learning the structure of a BN using a search and score approach can be cast as follows: given some observations D which are assumed to be generated by some BN whose structure is S, the task is to learn a BN structure S' which approximates S by optimising some scoring function $f(S, D)$. The space of all possible BNs of n random variables grows more than exponentially in n (Jensen and Nielsen, 2007). For any number of nodes $n \geq 1$, the number of all possible DAGs that can be constructed with these n nodes is (Robinson, 1973)

$$f(n) = \sum_{i=1}^{n} (-1)^{i+1} \frac{n!}{(n-i)!i!} 2^{i(n-i)} f(n-i) \tag{4.1}$$

with $f(0) = 1$. Chickering (1996) showed that finding a BN with the optimal BDe score is $\mathcal{NP}$-hard even if the size of the parent set for each variable is restricted (we discuss the BDe score in Section 4.2.1). Chickering et al. (2004) showed that the problem is $\mathcal{NP}$-hard with any consistent scoring function. Chickering et al. define consistent scoring functions as those which, given a large dataset, favour the one that represents the distribution reflected by this dataset, and given two structures representing this distribution, they favour the one with the smallest number of independent parameters. Therefore, as normally the case in solving hard problems, heuristic search strategies have been used. However, exact search strategies have also been used to solve problems with limited parent set sizes and limited number of random variables (Koivisto and Sood, 2004; Silander and Myllymäki, 2006; Cowell, 2009; Cussens, 2011). In Section 4.2 we go through different scoring functions in both cases where the data is complete and where there are some hidden variables. In Section 4.3 we highlight three heuristic search strategies developed in the literature as well as one exact search strategy developed by Cussens (2011) which will be used later in Chapter 5. Finally, we conclude by going through the literature of learning dynamic Bayesian networks.

4.2 Scoring Bayesian Networks

Given a BN structure S parameterised by $\boldsymbol{\theta}$, the likelihood $P(D|S, \boldsymbol{\theta})$ increases as the complexity (number of independent parameters) of S increases. This can be explained as follows: given two structures S_1 and S_2 whose sets of edges are E_1 and E_2 respectively where $E_1 \subseteq E_2$, for each settings of the parameters $\boldsymbol{\theta}_1$ for S_1 there exist some settings of the parameters $\boldsymbol{\theta}_2$ for S_2 in which S_2 encodes the same distribution encoded by S_1. This is because when the complexity of a model increases, it gives more freedom to set the parameters to fit the observations, thus maximising a point estimate (e.g. the likelihood). Therefore, the marginal likelihood defined as

$$P(D|S) = E_{\boldsymbol{\theta}|S}[P(D|S, \boldsymbol{\theta})] = \int_{\boldsymbol{\theta}} P(\boldsymbol{\theta}|S) P(D|S, \boldsymbol{\theta})\, d\boldsymbol{\theta} \tag{4.2}$$

has widely been used in the literature because it computes the expectation of the likelihood with respect to all settings of the parameters. This expectation penalises complex models. However, the integral in (4.2) cannot always be expressed in closed form and needs to be approximated. In the following two sections, we discuss scoring

BNs with the log marginal likelihood score expressed in (2.4) which is the logarithmic form of (4.2). Section 4.2.1 discusses scoring BNs with the log marginal likelihood score when the outcomes of all variables are observed. Section 4.2.2 treats the case when there are some hidden variables.

4.2.1 Scoring BN with Fully Observed Variables

Heckerman et al. (1995) reported that when the data is fully observed, the log marginal likelihood can be computed in closed form subject to the following two assumptions: (a) the parameters are independent and (b) Dirichlet prior is chosen. Heckerman et al. derived the *Bayesian Dirichlet equivalent* (BDe) score in which the score of the BN is decomposable into the sum of the family scores. The score of family i is defined as follows

$$Score_i(S) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(n_{ij} + \alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(n_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})} \tag{4.3}$$

where q_i is the number of joint instantiation of the parents in the family and r_i is the number of values that the child can take. n_{ijk} is the number of times the child takes the kth value when the parents are in their jth instantiation and α_{ijk} is the corresponding Dirichlet prior. $n_{ij} = \sum_{k=1}^{r_i} n_{ijk}$ and $\alpha_{ij} = \sum_{k=1}^{r_i} \alpha_{ijk}$. When each α_{ijk} is set to $\frac{\alpha}{r_i q_i}$ for some choice of α, we have the BDeu score (u stands for uniform). Therefore the final score of a BN with n families is defined as follows

$$\log P(D|S) = \sum_{i=1}^{n} \log Score_i(S) \tag{4.4}$$

There has also been some approximations to the log marginal likelihood (Chickering and Heckerman, 1996; Maxwell Chickering and Heckerman, 1997). Laplace, also called Gaussian, approximation is based on the idea that the posterior $P(\boldsymbol{\theta}|D, S) \propto P(D|\boldsymbol{\theta}, S)P(\boldsymbol{\theta}|S)$ is asymptotically (with a very large sample) Gaussian. So let

$$g(\boldsymbol{\theta}) \equiv \log(P(D|\boldsymbol{\theta}, S)P(\boldsymbol{\theta}|S)) \tag{4.5}$$

and let $\tilde{\boldsymbol{\theta}}$ be the *maximum a posteriori (MAP)* of $\boldsymbol{\theta}$

$$\tilde{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}}\{P(\boldsymbol{\theta}|D,S)\} = \arg\max_{\boldsymbol{\theta}}\{g(\boldsymbol{\theta})\} \tag{4.6}$$

by expanding $g(\boldsymbol{\theta})$ (which is approximated as Gaussian) with respect to $\tilde{\boldsymbol{\theta}}$ we get

$$g(\boldsymbol{\theta}) \approx g(\tilde{\boldsymbol{\theta}}) + -\frac{1}{2}(\boldsymbol{\theta}-\tilde{\boldsymbol{\theta}})^t A(\boldsymbol{\theta}-\tilde{\boldsymbol{\theta}}) \tag{4.7}$$

where A is the negative Hessian of $g(\boldsymbol{\theta})$ computed at $\tilde{\boldsymbol{\theta}}$. Substituting (4.7) into the integral $\int P(\boldsymbol{\theta}|S)P(D|S,\boldsymbol{\theta})\,d\boldsymbol{\theta}$ and taking the log results in the following approximation

$$\log P(D|S) \approx \log P(D|S,\tilde{\boldsymbol{\theta}}) + \log P(\tilde{\boldsymbol{\theta}}|S) + \frac{dim}{2}\log(2\pi) - \frac{1}{2}\log|A| \tag{4.8}$$

where *dim* is the number of independent parameters.

Another approximation, which is according to Heckerman et al. (1995) more efficient than the Laplace approximation, is the *Bayesian information criterion* (BIC) (Schwarz, 1978)

$$\log P(D|S) \approx \log P(D|S,\hat{\boldsymbol{\theta}}) - \frac{dim}{2}\log N \tag{4.9}$$

where $\hat{\boldsymbol{\theta}}$ is the *maximum likelihood* value of $\boldsymbol{\theta}$ and N is the sample size.

Draper (1995) adds the term $\frac{dim}{2}\log(2\pi)$ in (4.8) to (4.9) to obtain the following approximation

$$\log P(D|S) \approx \log P(D|S,\hat{\boldsymbol{\theta}}) - \frac{dim}{2}\log N + \frac{dim}{2}\log(2\pi) \tag{4.10}$$

4.2.2 Scoring BN with Hidden Variables

In scoring BNs with hidden variables, the approximations (4.8), (4.9) and (4.10) can be used with the EM algorithm (Section 1.3.1.1) to find an estimate $\hat{\boldsymbol{\theta}}$ of the parameters.

The *variational Bayesian* (VB) approach has also been used to score models with hidden variables (Attias, 1999, 2000; Ghahramani and Beal, 2000). Let $\mathbf{X}$ be the observed variables, $\mathbf{Z}$ be the hidden variables and $\boldsymbol{\theta}$ be the parameters. As the log marginal likelihood

$$\log P(\mathbf{X}|S) = \log \int \int P(\mathbf{X}, \mathbf{Z}, \boldsymbol{\theta}|S)\, d\mathbf{Z}\, d\boldsymbol{\theta}$$

is intractable, another distribution $Q(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S)$ is introduced and the log marginal likelihood can then be written as follows

$$\begin{aligned} \log P(\mathbf{X}|S) &= \log \int \int Q(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S) \frac{P(\mathbf{X}, \mathbf{Z}, \boldsymbol{\theta}|S)}{Q(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S)}\, d\mathbf{Z}\, d\boldsymbol{\theta} \\ &\geq \int \int Q(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S) \log \frac{P(\mathbf{X}, \mathbf{Z}, \boldsymbol{\theta}|S)}{Q(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S)}\, d\mathbf{Z}\, d\boldsymbol{\theta} = \mathcal{F} \end{aligned} \tag{4.11}$$

The last inequality is due to Jensen's inequality. It is a lower bound to the log marginal likelihood and known as the *variational free energy* (VFE). Note that setting $Q(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S) = P(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S)$ in (4.11) puts the lower bound into its maximum and it will be equal to the log marginal likelihood. However, computing $P(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S)$ requires computing its normalising factor which is the log marginal likelihood itself. Therefore, the problem is simplified by assuming the following factorisation $Q(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S) \approx Q(\mathbf{Z}|\mathbf{X}, S)\ Q(\boldsymbol{\theta}|\mathbf{X}, S)$. By substituting into (4.11), we obtain the following form of VFE

$$\log P(\mathbf{X}|S) \geq \int Q(\mathbf{Z}|\mathbf{X}, S)\ Q(\boldsymbol{\theta}|\mathbf{X}, S) \log \frac{P(\mathbf{X}, \mathbf{Z}, \boldsymbol{\theta}|S)}{Q(\mathbf{Z}, \boldsymbol{\theta}|\mathbf{X}, S)}\, d\mathbf{Z}\, d\boldsymbol{\theta} = \mathcal{F} \tag{4.12}$$

The log marginal likelihood is then approximated by maximising the VFE. *Variational Bayes-EM* (VB-EM) is an iterative algorithm which maximises the VFE with respect

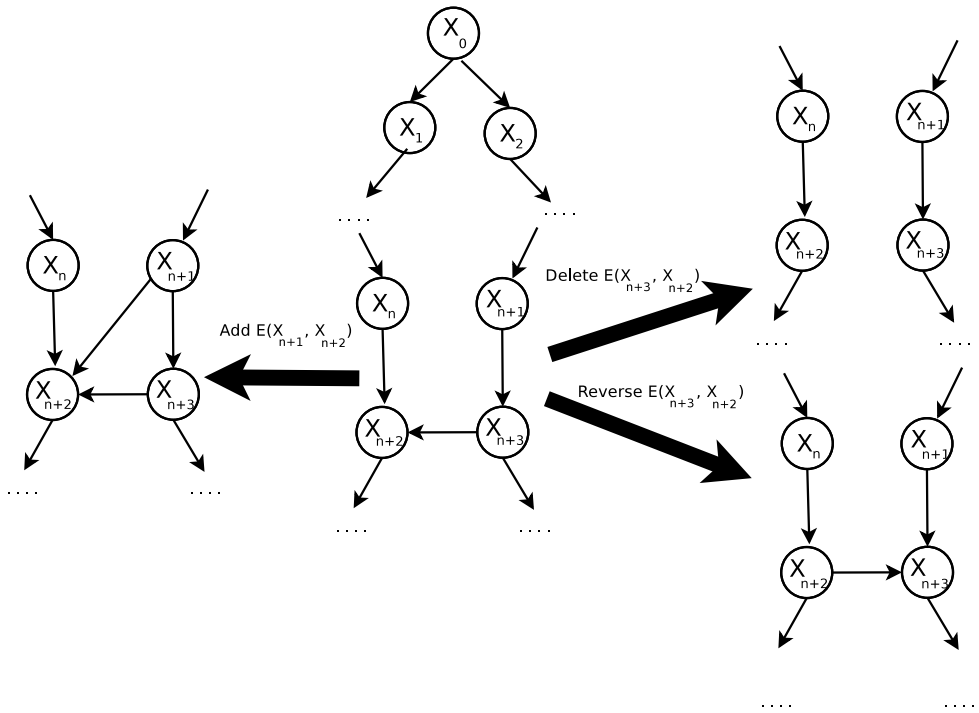


Figure 4.1: An example of deleting, reversing and adding an edge in a local search strategy.

to the free distributions $Q(\mathbf{Z}|\mathbf{X}, S)$ and $Q(\boldsymbol{\theta}|\mathbf{X}, S)$ (Beal and Ghahramani, 2003; Kurihara and Sato, 2004, 2006; Sato et al., 2008).

4.3 Search

In searching for a structure that maximises a given score, three operations are used to move in the search space. These are: (a) deleting an edge, (b) adding an edge or (c) reversing an edge. Any move can be accepted subject to the condition that the resulting structure is a valid BN (DAG). We highlight four search strategies here. The first is a simple local search which is a plain hill-climbing. The second is K3 which is a hill-climbing search that works on a total ordering of the random variables. The third is the structural-EM which is aimed at learning BNs with hidden variables. The final strategy is an exact search, referred to as the *cutting planes* algorithm, based on integer programming. The cutting planes algorithm guarantees finding the optimal structure when the family scores are given.

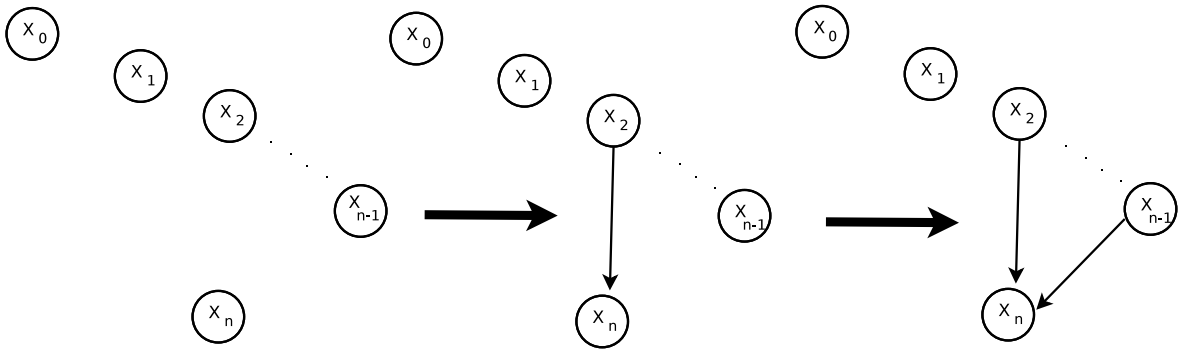


Figure 4.2: K3 search strategy searching for a parent set for X_n. It starts from the empty parent set and then find that adding X_2 as a parent of X_n has the highest score amongst the scores obtained by adding any of the other variables. It then finds that adding X_{n-1} to X_2 in the parent set has the highest score amongst the score obtained by adding any of the remaining variables.

4.3.1 Local Search

In local search, the search can start from any initial randomly selected structure. At each point, an edge is either added, deleted or reversed (Figure 4.1). The search takes the move that most increases the score. It stops when it does not find a local move which increases the score. It is obvious that this strategy may become trapped in a local maximum. However, the advantage of this strategy is that when it is used with a decomposable score, only the new families that emerge as the result of the change need to be scored. The scores of the rest of the families are preserved in the final score of the full structure in (4.4). For example, in Figure 4.1, by deleting the edge $E(X_{n+3}, X_{n+2})$, only the emerged family $\{X_n\} \rightarrow X_{n+2}$ needs to be scored. The score of the new structure can be computed by using the previously computed scores of all other families and replacing the score of the family $\{X_n, X_{n+3}\} \rightarrow X_{n+2}$ with the score of the new family $\{X_n\} \rightarrow X_{n+2}$. The same scenario happens when adding or reversing an edge.

4.3.2 K3

Though K3 (Bouckaert, 1993) is a complete framework of search and score approach where the score is a *minimum description length* (MDL), we will discuss only the search strategy here. Due to the massive search space, K3 reduces it by assuming a total order amongst the random variables. Given a total order $X_0 \prec \ldots \prec X_n \prec \ldots \prec X_{n+m}$, a random variable X_n is allowed to have as parents a set $S' = \{X_0, \ldots, X_{n-1}\} \subseteq S$. The search starts with each random variable having

Algorithm 2 Structural-EM

1: $t = 0$
2: $random(S^0_{\mathbf{X},\mathbf{Z}})$
3: $\boldsymbol{\theta}^0 = initialise\_parameters(S^0_{\mathbf{X},\mathbf{Z}})$
4: **repeat**
5: $\quad \boldsymbol{\theta}^{t+1}_{EM} = EM(S^t_{\mathbf{X},\mathbf{Z}}, \boldsymbol{\theta}^t, D)$
6: $\quad$ From $S^t_{\mathbf{X},\mathbf{Z}}$, move to $S^{t+1}_{\mathbf{X},\mathbf{Z}}$ such that:
$\qquad S^{t+1}_{\mathbf{X},\mathbf{Z}} = \arg\max_{S_{\mathbf{X},\mathbf{Z}}} score(S_{\mathbf{X},\mathbf{Z}}, \boldsymbol{\theta}^{t+1}_{EM}, D)$
7: $\quad \boldsymbol{\theta}^{t+1} = \arg\max_{\boldsymbol{\theta}} P(D|S^{t+1}_{\mathbf{X},\mathbf{Z}}, \boldsymbol{\theta})$
8: $\quad t = t + 1$
9: **until** convergence
10: **return** $S^t_{\mathbf{X},\mathbf{Z}}$

the empty set as its parent set. It adds variables one by one to the parent set of this variable from the set of variables preceding it in the total order until the score no longer improves. For instance, for variable X_n, it starts by first scoring the family $\emptyset \rightarrow X_n$. It then adds the variable $X_i \in S'$ with the highest score as a parent of X_n. This results in the family $\{X_i\} \rightarrow X_n$. The next step is to find a variable in $S' \setminus \{X_i\}$ which when added to X_i results in a parent set of X_n with the highest score amongst the parent sets that are obtained by adding any other variable in $S' \setminus \{X_i\}$ to X_i. The process continues adding variables until no further improvement is achieved. It then moves to another random variable and starts the same process until all variables are covered. This process is depicted in Figure 4.2 for finding one family.

4.3.3 Structural-EM

The *structural-EM* algorithm is aimed at learning BNs with hidden variables or missing values (Friedman, 1997, 1998). Given a BN with $\mathbf{X}$ representing the observed variables and $\mathbf{Z}$ representing the hidden variables, the algorithm works by, initially, selecting a random structure $S^0_{\mathbf{X},\mathbf{Z}}$ and then estimates its parameters $\boldsymbol{\theta}^0$ using the EM algorithm (Section 1.3.1.1). Then the estimated parameters are used to generate a complete data from which an expected sufficient statistics can be obtained. This sufficient statistics is then used for scoring candidate structures that will be moved to in the next step. This process is repeated until convergence (no considerable improvement is achieved). When BIC is used as a scoring function, the expected BIC score is defined as follows

$$score_{BIC}(S_{\mathbf{X},\mathbf{Z}}, \boldsymbol{\theta}', D) = E_{\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}'}[\log P(D|S_{\mathbf{X},\mathbf{Z}}, \boldsymbol{\theta})] - \frac{dim}{2} \log N$$

where $\boldsymbol{\theta}'$ represents the parameters estimated by the EM algorithm given a previous structure $S^t_{\mathbf{Z},\mathbf{X}}$. Friedman (1997) showed that when the BIC score is used, the algorithm monotonically improves the score at each iteration. The algorithm is shown in Algorithm 2.

4.3.4 Cutting Planes

This section highlights the algorithm developed by Cussens (2011) in which the problems of learning BNs is cast as an *integer programming* (IP) problem. Integer programming problems are sub-classes of *linear programming* (LP) problems. Linear programming is briefly defined as maximising or minimising a linear objective function given some linear constraints (Vanderbei, 2007). For example, given $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$, $\mathbf{b} = \langle b_1, \ldots, b_m \rangle$ and $\mathbf{A} = \langle \mathbf{a}_1, \ldots, \mathbf{a}_m \rangle$ where $\mathbf{a}_j = \langle a_{j1}, \ldots, a_{jn} \rangle$ and $1 \leq j \leq m$, the following defines a linear programming problem

$$\begin{aligned} &\text{maximise} \quad \sum_{i=1}^{n} x_i c_i, \qquad \text{subject to:} \\ &\forall 1 \leq j \leq m : \sum_{i=1}^{n} a_{ji} x_i \geq b_j \end{aligned} \tag{4.13}$$

When an additional constraint is employed that some or all of the variables in the LP problem are integers, then we have an IP problem. It is sometimes hard to solve the IP problem itself, so the problem is simplified by transforming it into an LP problem by dropping off the integrality constraints. This transformation is called the *linear programming relaxation* (LP-relaxation). As the transformed problem has fewer constraints, its solution provides an upper bound to the solution of the IP problem (Vanderbei, 2007). The solution to the IP problem can then be found based upon this upper bound.

Let $c(X, \mathbf{Pa})$ represents the score of the family $\mathbf{Pa} \to X$, Cussens (2011) defined an integer variable $I(\mathbf{Pa} \to X) \in \{0, 1\}$ as follows

$$I(\mathbf{Pa} \to X) = \begin{cases} 1 & \text{if and only if } \mathbf{Pa} \text{ is the parent set of the} \\ & \text{variable } X \text{ in an optimal BN} \\ 0 & \text{otherwise} \end{cases}$$

Let $\mathcal{P}_X$ be the set of all possible parent sets of a variable X and $\mathcal{X}$ be the set of all the variables in the BN, Cussens then defined the BN learning problem as follows

$$\begin{aligned} &\arg\max_{I(\mathbf{Pa}\to X)} \sum_{X\in\mathcal{X}} \sum_{\mathbf{Pa}\in\mathcal{P}_X} c(X,\mathbf{Pa}) I(\mathbf{Pa}\to X) \qquad \text{subject to:} \\ &\text{the graph represented by the families } I(\mathbf{Pa}\to X)=1 \text{ is a DAG.} \end{aligned} \tag{4.14}$$

The constraint above needs to be encoded as a set of linear constraints. But first, the solver needs to choose only one parent set from all possible parent sets for any variable X. This is also imposed as a linear constraint and called the *convexity constraint* which is defined as follows

$$\forall X : \sum_{\mathbf{Pa}\in\mathcal{P}_X} I(\mathbf{Pa}\to X) = 1$$

Cussens adopts the linear constraints proposed by Jaakkola et al. (2010) to rule out the integer solutions which do not represent DAGs. These constraints are based upon the observation that in any subset $\mathbf{C}$ of nodes in the DAG, there is at least one node which has no parents in $\mathbf{C}$. The linear constraints representing this observation are called the *cluster-based constraints*. Cussens referred to them as the *1-cluster-based constraints*[1]. These constraints are defined as follows

$$\forall \mathbf{C} \subseteq \mathcal{X} : \sum_{X\in\mathbf{C}} \sum_{\mathbf{Pa}:\mathbf{Pa}\cap\mathbf{C}=\emptyset} I(\mathbf{Pa}\to X) \geq 1$$

[1]Cussens proposed a generalisation of the constraints proposed by Jaakkola et al. and called it *k-cluster-based constraints*. As the cluster-based constraints are special cases of the k-cluster-based constraints when $k = 1$, Cussens referred to the cluster-based constraints as the 1-cluster-based constraints.

Because there are too many cluster-based constraints to be included in an IP problem, the IP problem is first constructed by (4.14) and the convexity constraints. The solution to the LP problem obtained by the LP-relaxation of this IP problem corresponds to finding the best family scores. However, in most cases these families constitute a graph which contains cycles. The solutions to the LP problem represent vertices of a convex polytope. The algorithm then starts ruling out cycles by introducing cluster-based constraints. When a cluster-based constraint is introduced, a new LP problem is constructed. These cluster-based constraints represent planes which cut off the invalid (as they do not represent DAGs) solutions to the original LP problem from the polytope, hence the name *cutting planes*. When a solution after adding a cutting plane represents a DAG, the algorithm returns this DAG as an optimal BN, otherwise the algorithm introduces another cutting plane. If no further cutting planes can be added, the algorithm selects a non-integer variable in the final solution and *branches on* it by creating two sub-problems, one with this variable being 1 and the other with the variable being 0. It then solves these two sub-problems in the same way as the original one. It returns the BN with the optimal score amongst the ones found in these two sub-problems.

4.4 Learning DBNs

One of the main tasks in which DBNs have been applied is representing gene regulatory networks modelling temporal microarray data (Murphy and Mian, 1999). This has motivated learning these networks in the form of DBNs from gene expression data (Segal et al., 2003; Cantone et al., 2009; Vinh et al., 2011). However, DBNs can also be applied to solve a wide range of problems in different domains. Therefore, an obvious research direction has also been taken to learn general purpose DBNs. The *Bayes net toolbox* (BNT)[2] facilitates learning general purpose DBNs with fully observed outcomes and with no intra-slice dependencies. DBmcmc[3] is aimed at learning the same class of DBNs as those learned by BNT using *Markov chain Monte Carlo* (MCMC). Friedman et al. (1998) proposed a method to learn DBNs with hidden variables using the structural-EM algorithm. Robinson and Hartemink (2010) proposed learning DBNs where dependencies change over time and call this class of DBNs *non-stationary* DBNs. Learning HMMs, which are special instances of DBNs, has also been considered due to the significance and early use of HMMs in natural language

[2] `https://code.google.com/p/bnt/`
[3] `http://www.bioss.ac.uk/~dirk/software/DBmcmc/`

processing (Manning and Schüetze, 1999), bioinformatics (Durbin et al., 1998) and speech recognition (Rabiner, 1989). We highlighted the best-first model merging algorithm of learning HMMs in Section 3.3.1.1. Au and Cheung (2004) proposed the same idea of the best-first model merging algorithm to learn HMMs by merging states starting from very specific structures. However, the decision to merge two states differs from that in the best-first model merging in the sense that it is based on the KL-divergence between the two structures before and after the merge. Au and Cheung applied this technique to solve problems in information extraction. Seymore et al. (1999) also discussed learning HMMs for information extraction. Won et al. (2007) used genetic algorithms to learn HMMs for the task of predicting protein secondary structures. However, the algorithms proposed for learning HMMs were developed with respect to the specific topology imposed by any HMM (Section 2.2.1.1). For instance, in the best-first model merging, the operation to move in the search space is merging states. This is because in the topology of any HMM, each hidden variable has parents of only hidden variables. Therefore, the algorithm was tailored to process sequences of hidden variables. However, this is not the case in a DBN which is not an HMM where the parents of hidden variables are not necessarily hidden variables. Therefore, developing algorithms to learn any DBN faces the challenge of dealing with a wider class of topologies (we assume only stationary DBNs). The general DBN learning algorithms discussed above considered only inter-slice dependencies where the aim is to model the relationships between random variables over time.

Chapter 5

Learning Recursive PRISM Programs with Fully Observed Outcomes

This chapter presents an algorithm to learn recursive PRISM programs with all the outcomes of the probabilistic atoms observed. These programs generalise DBNs by defining a halting distribution over the generative process. The algorithm learns the dependencies between the different iterations as well as the dependencies amongst the outcomes within each iteration. Section 5.1 introduces the problem and provides a formalisation of the learning task. Section 5.2 describes the steps that the learning algorithm goes through. Experiments conducted on learning five programs are given in Section 5.3.

5.1 Introduction

It was shown in Section 2.4 how failure-free recursive PRISM programs can generalise DBNs by defining a halting distribution over the generative process. In this chapter, we present an algorithm to learn this class of PRISM programs.

By defining a halting distribution, the model allows sampling observations with different lengths of sequences. The sampling process is self-terminating based upon some halting condition. This is an additional modelling property over those in DBNs where each DBN is either an infinite generative process or a generative process of a fixed length. The importance of modelling a self-terminating process is that it represents a halting function in which, based upon its termination, subsequent functions can take over. This is useful in modelling machines. For instance, in a stochastic context-free grammar (SCFG), an observation of the modelled language is generated

by a sequence of terminal and non-terminal symbols. Each non-terminal symbol represents a halting function. Though in this thesis only programs of a single tail recursive definition are addressed, it provides a building block in which further work can be carried out to learn a model of multiple tail recursive definitions, e.g. a SCFG. The algorithm presented in this chapter learns both *inter-iteration* and *intra-iteration* dependencies.

Kersting and De Raedt (2007) adopt learning from interpretations to learn Bayesian logic programs. The setting that will be adopted here is learning from entailment. Ideas from both areas of ILP and learning BNs are built upon. The learning algorithm is given: (a) observations D of a target predicate, (b) BK consisting of PRISM switch declarations and (c) a declarative bias specifying a term representing some switches which generate a value that halts the generative process. It aims at learning a PRISM program S that maximises the log marginal likelihood $\log P(D|S)$. PRISM does not support computing the exact log marginal likelihood. However, as mentioned in Section 2.5.2, PRISM supports three approximations to the log marginal likelihood. Amongst these approximations is the BIC score defined in (4.9) which is adopted here as the objective score to maximise. Formally, the learning problem is defined as follows

Given:

- A set D of observations in FOL.
- Background knowledge B in FOL consisting of
  - A set F of k functions where each function represents a set of switches.
  - A set of ground terms G representing the values that the switches may generate.
  - Sets $l_1, \ldots, l_k$ such that $\forall i : 1 \leq i \leq k$, $l_i \subseteq G$ and $\bigcap_{i=1}^{k} l_i = \emptyset$.
  - A set SD of atoms $\texttt{values}(f_i, l_i)$ where $f_i \in F$ and $1 \leq i \leq k$ declaring the switches.
  - A set of probabilistic atoms $\texttt{msw}(f, v)$ where $f \in F$ and $\exists l_i : v \in l_i$ such that $\texttt{values}(f, l_i) \in SD$.
- A declarative bias HT specifying the function f representing the set of switches that halt the generative process. Let these switches generate a value `t` to halt the process, then the bias is given as follows:

Observations:

```
sentence(['Hello!',
          person1,studies,playing,',',
          person1,studies,playing,',',
          person3,goes,playing,'.']).
sentence(['Hello!',
          person3,likes,playing,'.']).
sentence(['Welcome!',
          person3,goes,walking,'.']).
sentence(['Hello!',
          person1,likes,playing,',',
          person2,goes,walking,',',
          person3,studies,playing,',',
          person3,studies,playing,'.']).
.
.
.
```

BK:

```
values(start,['Hello!','Welcome!']).
values(subject,[person1,person2,
                person3]).
values(verb,[likes,goes,studies]).
values(object(_),[playing,walking,
                  shopping]).
values(punc(_),[',','.']).

% The bias specifying the halt (HT)
stop:-
     msw(punc(_),'.').
```

Figure 5.1: On the left are observations of the target predicate `sentence/1` whose definition needs to be learned according to the BK and the bias on the right.

`stop:- msw(f,t).`
where $f \in F$ and $\exists l_i : t \in l_i$ such that $\texttt{values}(f, l_i) \in SD$.

Find: a PRISM program S represented by the predicates in $D \cup B \cup \{\texttt{rec\_def/n}, \texttt{stop/m}\}$ modelling a generative process which halts as specified by HT such that $\forall d \in D : B \cup S \models d$ and

$$S = \arg\max_{S'} \log P(D|S', \hat{\boldsymbol{\theta}}) - \frac{dim}{2} \log N$$

The definition above states that the sets of switch outcomes defined by the switch declarations are mutually exclusive. Note that the learning algorithm goes beyond the predicates in the observations and the background knowledge. It invents the new predicates `rec_def/n` and `stop/m` where $m > 0$. Figure 5.1 shows examples of some observations of the target predicate `sentence/1` whose definition needs to be learned given the shown BK and bias. Each switch declaration in the BK implicitly encodes a set of probabilistic atoms. For instance, the switch declaration `values(start,['Hello!','Welcome!'])` implicitly encodes the set

```
{msw(start,'Hello!'), msw(start,'Welcome!')}
```

Program 1:

```
values(start,['Hello!','Welcome!']).
values(subject,[person1,person2,
                person3]).
values(verb,[likes,goes,studies]).
values(object(_),[playing,walking,
                  shopping]).
values(punc(_),[',','.']).

sentence([B,Person,VB,Do,Pun|Tail]):-
   msw(start,B),
   rec_def([Person,VB,Do,Pun|Tail]).

rec_def([Person,VB,Do,Pun|Tail]):-
   msw(subject,Person),
   msw(verb,VB),
   msw(object(Person),Do),
   msw(punc(VB),Pun),
   stop(Pun,Tail).

stop('.',[]):-!.
stop(_,Tail):-
   rec_def(Tail).
```

Program 2:

```
values(start,['Hello!','Welcome!']).
values(subject,[person1,person2,
                person3]).
values(verb,[likes,goes,studies]).
values(object(_),[playing,walking,
                  shopping]).
values(punc(_),[',','.']).

sentence([B,Person,VB,Do,Pun|Tail]):-
   msw(start,B),
   rec_def(B,[Person,VB,Do,Pun|Tail]).

rec_def(B,[Person,VB,Do,Pun|Tail]):-
   msw(subject,Person),
   msw(verb,VB),
   msw(object(B),Do),
   msw(punc(VB),Pun),
   stop(Pun,Do,Tail).

stop('.',_,[]):-!.
stop(_,Par,Tail):-
   rec_def(Par,Tail).
```

Figure 5.2: Two PRISM programs from which the observations in Figure 5.1 could have been generated.

.

The clause `stop:- msw(punc(_),'.')` is the declarative bias. It states that when the switches represented by the term `punc(_)` generate the value `'.'`, the generative process must halt.

Figure 5.2 shows two programs from which the observations in Figure 5.1 could have been generated. Note that the difference between the two programs is in the dependencies between outcomes. As an unground term represents a set of switches, grounding a term in a probabilistic atom by the outcomes of some preceding atoms amounts to selecting a switch amongst this set. Therefore, in `Program 1`, the selection of a switch amongst the set represented by the term `object/1` depends upon the outcome of the switch `subject/0`. Thus, the probabilistic outcome `Do` depends on the probabilistic outcome `Person`. However, in `Program 2`, the selection of a switch amongst the set `object/1` in the first iteration depends upon the outcome of the switch `start/0`. The selection of a switch amongst this set in subsequent iterations depends upon the outcome of the switch selected amongst the

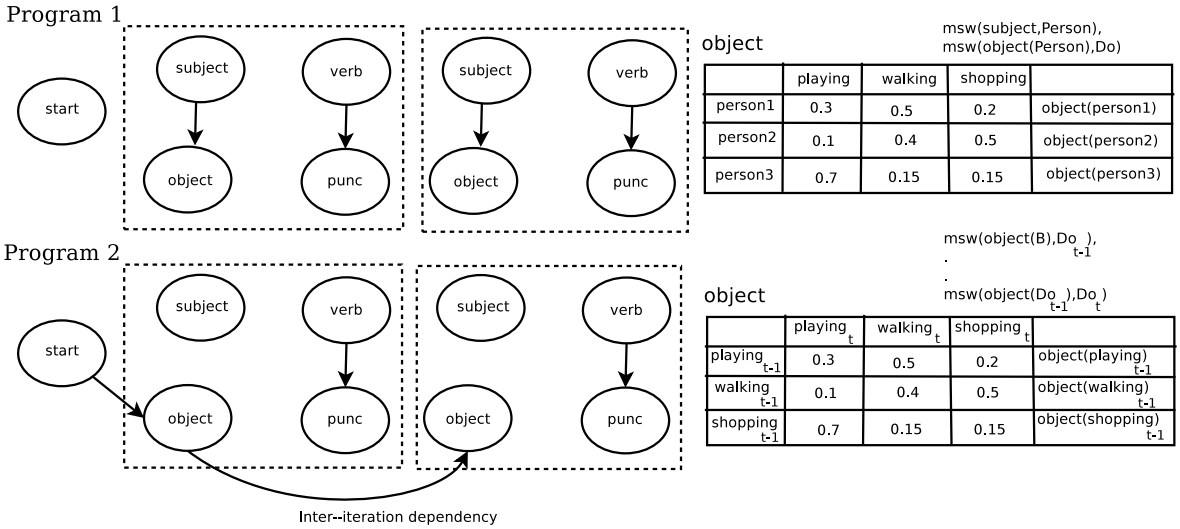


Figure 5.3: Two DBNs representing the dependencies modelled in the programs in Figure 5.2. The tables on the right show that each unground term (here `object(_)`) in a probabilistic atom corresponds to a CPT, and each particular grounding of this term (a PRISM switch) corresponds to a row in the CPT.

set `object/1` in the previous iteration. Therefore, in `Program 2`, there is an inter-iteration dependency which does not exist in `Program 1`. Figure 5.3 depicts two DBNs representing the two programs without the halting conditions. As shown in the figure, the set of switches represented by the unground term `object/1` corresponds to a *conditional probability table* (CPT). Each switch of this set achieved by grounding the term corresponds to a row in the CPT. The substitution of the variables in the term corresponds to the evidence in the conditional probability of the outcome associated with this term given these variables. For instance, in `Program 1`, the dependency {`msw(subject,Person), msw(object(Person),Do)`} represents the conditional probability distribution $P(Do|Person)$, and the substitution $\theta = \{Person/person1\}$ is an instantiation of the evidence to result in $P(Do|Person = person1)$. In `Program 2`, it can be noticed that the inter-iteration dependency corresponds to an argument to the recursive predicate which is used to ground the term `object/1` in the next iteration.

Given the above discussion, the learning algorithm needs to induce the following

- The dependencies between the outcomes of the probabilistic atoms.
- The atoms that occur once in the generative process (atom `msw(start,B)` in the programs in Figure 5.2), we will refer to these atoms as the *initial atoms* and their outcomes as the *initial outcomes*, and the atoms involved in the iteration,

we will refer to these atoms as the *iteration atoms* and their outcomes as the *iteration outcomes*. Initial atoms need to be separated from the iteration atoms. A predicate `rec_def/m` needs to be invented whose definition consists of the iteration atoms.

- The inter-iteration dependencies (if any).
- The definition of the halting condition by inventing a predicate `stop/m` where $m > 0$.

Section 5.2 addresses the points above and Section 5.3 shows some experiments conducted with the developed learning algorithm.

5.2 Learning

The algorithm needs to learn the dependencies between the outcomes of the probabilistic atoms. These are the dependencies amongst the initial outcomes, the dependencies between the initial outcomes and the first iteration outcomes and the dependencies amongst the iteration outcomes. For instance, in `Program 2` in Figure 5.2, the first case does not apply as there is only one initial outcome which is `B`. However, there is a dependency between the initial outcome `B` and the iteration outcome `Do` and there is a dependency between the iteration outcomes `VB` and `Pun`. Learning the dependencies amongst the iteration outcomes is a dilemma of whether to learn them from the statistics obtained from a single iteration (e.g. the first iteration) or to learn them from the statistics obtained from all the iterations. Learning from the statistics obtained from a single iteration amounts to taking a snap shot of the observations at a particular time and learn the dependencies from only this instance. This snap shot carries less information than the information that can be obtained from the statistics in all the iterations. However, learning from all the iterations can be significantly more computationally demanding. A single observation might have been generated by a large number of iterations which is more than the elements of the list obtained by taking a snap shot of a single iteration in all the observations. Therefore, we go with the former choice and learn these dependencies from only the statistics obtained from the first iteration. Learning from the first iteration guarantees that the set of elements obtained from the snap shot is greater than or equal to the set of elements obtained from a snap shot of any other iteration. This is due to the fact that in some observations the generative process might have generated the outcomes of the first iteration and halted.

Algorithm 3 Build a bottom clause using rlgg

1: $D = \{d_1, d_2, \ldots, d_n\}$ % n observations
2: G_{msw} = the set of all ground probabilistic atoms in BK
3: $C_1 = (d_1 \leftarrow G_{msw})$
4: $C_2 = (d_2 \leftarrow G_{msw})$
5: $BC' = lgg(C_1, C_2)$
6: $BC = reduced(BC')$
7: $Expl$ = all observations in D explained by BC
8: $D' = D \setminus Expl$
9: **while** $D' \neq \emptyset$ **do**
10: Select any $d' \in D'$
11: $C = (d' \leftarrow G_{msw})$
12: $BC' = lgg(BC, C)$
13: $BC = reduced(BC')$
14: $Expl$ = all observations in D' explained by BC
15: $D' = D' \setminus Expl$
16: **end while**
17: **return** BC

The initial outcomes and the first iteration outcomes need to be identified. The algorithm then builds a bottom clause which consists of all possible dependencies amongst these outcomes. From the body of the bottom clause, a set of atoms encoding outcome dependencies is built with the aim that this set contributes to maximising the BIC score. When this step is finished, this set needs to be split into two sets to separate the initial atoms from the iteration atoms. When this split is performed, the algorithm needs to decide the iteration outcomes that need to be passed as arguments to the next iteration (learning the inter-iteration dependencies). Finally, it forms the definition of the halting condition accordingly. These steps will be explained in detail in the following sections. Section 5.2.1 discusses building the bottom clause. Section 5.2.2 discusses three search strategies for finding dependencies amongst the outcomes. Section 5.2.3 explains how the initial atoms are split from the iteration atoms. Finally, Section 5.2.4 explains how the algorithm designates the arguments that will be passed to the next iteration and the construction of the halting condition.

5.2.1 Bottom Clause

The head h of the bottom clause $BC = (h \leftarrow B)$ is an atom of the target predicate whose arguments are the output of the generative process. The variables $vars(h)$ in h are the initial outcomes, the first iteration outcomes and a variable representing the output of subsequent iterations. The body B of the bottom clause is the set

of probabilistic atoms which encode all possible dependencies amongst the outcomes. Algorithm 3 builds bottom clauses using the rlgg. It first grounds all the probabilistic atoms in BK. For instance given the BK in Figure 5.1 whose probabilistic atoms are

```
msw(start,'Hello!'). msw(start,'Welcome!').
msw(subject,person1). msw(subject,person2). msw(subject,person3).
msw(verb,likes). msw(verb,goes). msw(verb,studies).
msw(object(_),playing). msw(object(_),walking). msw(object(_),shopping).
msw(punc(_),','). msw(punc(_),'.').
```

all possible groundings of the terms `object(_)` and `punc(_)` by the outcomes of all other switches defined by the switch declarations are considered. i.e.

```
msw(object('Hello!'),playing). ... msw(object('.'),shopping).
msw(punc('Hello!'),','). ... msw(punc(shopping),'.').
```

Note that a term is not grounded by the outcomes of the switches it represents. e.g. `msw(object(walking),playing)` is not considered. This is because in the final PRISM program, each term representing a set of switches is assumed to exist once in the program. Therefore, a term can only be grounded by the outcomes of the switches represented by the other terms. The algorithm then builds two clauses C_1 and C_2 whose heads are the first and second observations d_1 and d_2 and whose bodies are these ground atoms. It computes the lgg of these two clauses which results in a clause with possibly both redundant atoms and atoms with hidden (unobserved) variables. Hidden variables are those whose values do not appear in the observations. Thus those variables are not shown in the head of the resulting clause. Both kind of atoms are deleted and a reduced clause is obtained. All observations which are explained by the reduced clause are deleted. If there are some observations which are not explained, one of these observations is selected and a clause is built in the same way as C_1 and C_2. Then the lgg of this clause and the reduced clause resulting from the previous step is computed and the resulting clause is reduced. Similarly, all observations explained by this clause are then deleted. The process continues until all observations are explained.

The problem of this algorithm is that it constructs, possibly many, atoms with hidden variables which the algorithm eliminates at each step. This motivated designing Algorithm 4 for bottom clause construction in which no atoms with hidden variables are generated. This algorithm works by first computing the lgg of all the observations to construct the head of the clause. It then constructs a set of all the variables in this head that represent the outcomes of probabilistic atoms. This set consists of all the variables in the head except one which represents the output of the

Algorithm 4 Build a bottom clause

1: $D = \{d_1, d_2, \ldots, d_n\}$ % n observations
2: $h = d_1$
3: **for** $i = 2$ to n **do**
4: $\quad h = lgg(h, d_i)$
5: **end for**
6: $Var = outcomes\_of\_msw\_atoms(h)$
7: $B = \emptyset$
8: **for all** `values`(t, O) in BK **do**
9: $\quad functor(t, f, k)$ % f is the functor and k is the arity
10: $\quad$ **for all** $V' \in Var$ **do**
11: $\quad\quad$ **if** the set of values instantiated to V' in D is a subset of O **then**
12: $\quad\quad\quad V = V'$
13: $\quad\quad\quad$ break
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: $\quad Var' = Var \setminus \{V\}$
17: $\quad C = combinations(k, Var')$ % C is a set of combinations
18: $\quad$ **for all** $C' \in C$ **do**
19: $\quad\quad t_{new} = construct\_term(f, C')$
20: $\quad\quad B = B \cup \{msw(t_{new}, V)\}$
21: $\quad$ **end for**
22: **end for**
23: **return** $BC = (h \leftarrow B)$

subsequent iterations. For instance, in Figure 5.2, the variable `Tail` represents the output of the subsequent iterations and will not be included, so the constructed set is {`B,Person,VB,Do,Pun`}. This set of variables, call it Var, is used to construct the atoms in the body of the bottom clause. For each term representing a set of switches, the arity k of this term is obtained. If V is the variable in Var that represents the outcomes of this set of switches, then by letting $Var' = Var \setminus \{V\}$, the sets $C_1, \ldots, C_m$ of the k-combinations of Var' where $m = \binom{|Var'|}{k}$ are used to construct m terms whose sets of arguments are $C_1, \ldots, C_m$. Probabilistic atoms are then constructed for each of these terms with V as the outcomes of these atoms. For example, for the term `object/1` in the BK in Figure 5.1, the variable `Do` is excluded from the set {`B,Person,VB,Do,Pun`} as it represents the outcome of the set of switches represented by `object/1`. The 1-combinations of the set of the remaining elements are {`B`}, {`Person`}, {`VB`} and {`Pun`}. The algorithm then constructs the following atoms accordingly

```
{msw(object(B),Do), msw(object(Person),Do),
```

```
msw(object(VB),Do), msw(object(Pun),Do)}
```

It is obvious from the atoms above that their outcomes are unified. Therefore, the bottom clause built from the observations and BK in Figure 5.1 is a clause with failure. In general, any bottom clause generated by Algorithm 3 or Algorithm 4 from a BK which contains parameterised terms in the switch declarations is a clause with failure.

5.2.2 Searching for Outcome Dependencies

When the bottom clause is built, the learning algorithm searches for a set of atoms in the body of the bottom clause that encodes the three types of dependencies discussed at the beginning of Section 5.2. The search aims at finding the set of atoms which may contribute to maximising the final BIC score. Three search strategies will be discussed. The first is a greedy generalisation of a clause whose body is the same as the body of the bottom clause. The second, is a random search based upon an ordering of the atoms in the body of the bottom clause. These two strategies do not guarantee finding the optimal dependencies with respect to the log marginal likelihood. This has motivated adopting the cutting planes search, explained in Section 4.3.4, which finds the dependencies with the optimal log marginal likelihood. The cutting planes search will thus be used in the final learning algorithm.

As learning the outcome dependencies is based upon the initial outcomes and the first iteration outcomes, the values generated by all other iterations in each observation are not required at this step. The first two strategies are based upon scoring PRISM clauses after each move in the search space. As these clauses define generative processes of only the initial and first iteration outcomes, they need to be scored against only the values of these outcomes. Therefore, a new set of observations D' needs to be built in which the output of all the iterations apart from the first one is trimmed from the original set of observations D. Let this be performed by the following function

$$D' = trim\_obs\_from\_second\_to\_end(D)$$

Figure 5.4 shows the result of applying this function on the observations in Figure 5.1. The variable representing the trimmed values needs not be included in the head of

```
sentence(['Hello!',person1,studies,playing,',']).
sentence(['Hello!',person3,likes,playing,'.']).
sentence(['Welcome!',person3,goes,walking,'.']).
sentence(['Hello!',person1,likes,playing,',']).
.
.
.
```

Figure 5.4: The result of applying the trimming function on the observation in Figure 5.1.

these clauses. This is performed by deleting this variable from the head of the bottom clause using the following function

$$h' = delete\_var\_of\_subsequent\_iter(h)$$

The result of applying this function on the head of the programs in Figure 5.2 is `sentence([B,Person,VB,Do,Pun])`.

5.2.2.1 Greedy Generalisation

The greedy generalisation search takes the bottom clause $BC = (h \leftarrow B)$ and the set of observations D and applies the two functions

$$D' = trim\_obs\_from\_second\_to\_end(D)$$
$$h' = delete\_var\_of\_subsequent\_iter(h)$$

It then starts with the clause $BC' = (h' \leftarrow B)$ as an initial state in the search space and moves in the space by choosing the best candidate generalisation. The candidate generalisations of a clause are clauses whose head is the same as the head of the generalised clause and whose bodies are the set of atoms in the body of the generalised clause with one atom deleted. The search starts from the clause BC' by deleting the first atom in its body and then scores it. It ends up with a clause C_1 whose score is $Score_1$. It then goes back to BC' and deletes the second atom in its body and scores it. This results in a clause C_2 whose score is $Score_2$. If $Score_1 > Score_2$ then C_2 is removed and C_1 is preserved, otherwise the opposite is

Algorithm 5 Greedy generalisation

```
1: D = {d_1, d_2, ..., d_n} % n observations
2: BC = (h ← B) % the bottom clause
3: D' = trim_obs_from_second_to_end(D)
4: h' = delete_var_of_subsequent_iter(h)
5: C = (h' ← B)
6: R = score_BIC(C, D')
7: while C is a clause with failure do
8:    for all b ∈ B do
9:       B' = B \ {b}
10:      C' = (h' ← B')
11:      R' = score_BIC(C', D')
12:      if R' ≥ R then
13:         C = C'
14:         R = R'
15:      end if
16:   end for
17:   B = body_of(C)
18: end while
19: return B
```

applied. It then continues by deleting the third atom in the body of BC' to generate C_3 and compares its score $Score_3$ with the score of the clause preserved in the previous step. The process continues until all the atoms in the body of BC' are processed. The result of this step is a clause generalising BC' by one atom dropped off. The search continues by generalising clauses until it reaches a failure-free clause whose body is then returned. This process is shown in Algorithm 5. Given n atoms in the body of the bottom clause, BIC score is run n times to delete one atom from this clause. It is then run $n-1$ times to delete an atom from the clause resulting from the first step. Therefore, the number of times BIC score is run is $n+n-1+n-2+\ldots+n-m$ where $n-m$ is the number of atoms in the final failure-free clause. Thus BIC score is run $\approx n*(n+1)/2$ times. As BIC runs the EM algorithm, the cost of the BIC score includes the cost of the EM algorithm. The cost of the gEM (the PRISM instance of EM) grows linearly with the nodes in the explanation graph of any observation (Sato and Kameya, 2001). The explanation graph of any observation of a predicate grows with the number of atoms in the body of the predicate definition. The time for scoring clauses in this strategy decreases as the number of atoms in the body is reduced at each step. In some problems, the bottom clause may contain a large number of atoms (e.g. >1000 atoms) in its body. In such cases, this strategy can be problematic and

the search may take considerable time in scoring the bottom clause and subsequent clauses until it reaches a significantly reduced clause. This would affect the overall performance. It is also obvious that this strategy may become trapped in a local maximum.

5.2.2.2 Random Search

This search strategy avoids the problem of the greedy generalisation search in which clauses with large number of atoms in their bodies are scored. It scores only possible failure-free clauses. The search generates failure-free clauses randomly and returns the body of the one with the maximum score. As shown in Algorithm 6, it first generates D' and BC' in the same way as the greedy generalisation does. It then clusters the atoms in the body of BC' based upon the arity of the terms representing the switches. It builds a total order $C_0 \prec \ldots \prec C_k$ of these clusters where $\forall i : 0 \leq i \leq k$, C_i is a cluster of atoms in which the arity of the terms in this cluster is i. For instance, given the following clause

```
sentence([B,Person,VB,Do,Pun]):-
    msw(start,B),
    msw(subject,Person),
    msw(verb,VB),
    msw(object(B),Do), msw(object(Person),Do),
    msw(object(VB),Do), msw(object(Pun),Do),
    msw(punc(B),Pun), msw(punc(Person),Pun),
    msw(punc(VB),Pun), msw(punc(Do),Pun).
```

The following two clusters are built

$C_0 = \{$`msw(start,B), msw(subject,Person), msw(verb,VB)`$\}$
$C_1 = \{$`msw(object(B),Do), msw(object(Person),Do),`
`msw(object(VB),Do), msw(object(Pun),Do),`
`msw(punc(B),Pun), msw(punc(Person),Pun),`
`msw(punc(VB),Pun), msw(punc(Do),Pun)`$\}$

Let B be initialised to the empty set. The cluster order is then traversed starting from C_0. All the atoms in C_0 are added to B. For each other cluster C_i, it deletes all atoms whose terms are not grounded by a subset of the outcomes of the atoms in B or whose outcomes are generated by some atoms in B. When these atoms are deleted, a set C_i' is generated. It then selects an atom randomly from C_i' and adds it to B. These two steps are repeated until the set $C_i' = \emptyset$ is generated. It then moves

Algorithm 6 Random search

1: $D = \{d_1, d_2, \ldots, d_n\}$ % n observations
2: $BC = (h \leftarrow B)$ % the bottom clause
3: $D' = trim\_obs\_from\_second\_to\_end(D)$
4: $h' = delete\_var\_of\_subsequent\_iter(h)$
5: $Score = -\infty$
6: $\{C\}_{\prec_0^k}$ = cluster the atoms in B based upon the arity of the terms in these atoms that represent the switches and return these cluster
7: $B = C_0$
8: $\{C\}_{\prec} = \{C\}_{\prec_0^k} \setminus \{C_0\}$
9: m = number of clauses to generate
10: **for** $j = 1$ to m **do**
11: **while** $\{C\}_{\prec} \neq \emptyset$ **do**
12: C_i = the first element in $\{C\}_{\prec}$
13: **loop**
14: O = the set of outcomes generate by the atoms in B
15: $C_i' = C_i \setminus \{msw(t,v) \in C_i | \nexists \theta \subseteq O : vars(t\theta) = \emptyset\}$
16: $C_i'' = C_i' \setminus \{msw(t_1,v) \in C_i' | \exists msw(t_2,v) \in B\}$
17: **if** $C_i'' = \emptyset$ **then**
18: break
19: **else**
20: $b = random\_atom(C_i'')$
21: $B = B \cup \{b\}$
22: **end if**
23: **end loop**
24: $\{C\}_{\prec} = \{C\}_{\prec} \setminus \{C_i\}$
25: **end while**
26: $Clause' = (h' \leftarrow B)$
27: $Score' = score_{BIC}(Clause', D')$
28: **if** $Score' \geq Score$ **then**
29: $Clause = Clause'$
30: $Score = Score'$
31: **end if**
32: $j = j + 1$
33: **end for**
34: **return** $B = body\_of(Clause)$

to the next cluster in the order and processes it in the same way. For instance, from the above two clusters, the atoms in C_0 are added to the empty set B as follows

$B = \{$`msw(start,B), msw(subject,Person), msw(verb,VB)`$\}$

All atoms which are not grounded by a subset of {B,Person,VB} or whose outcomes are members of this set are deleted from C_1 to generate C_1' as follows

$C_1' = \{$msw(object(B),Do), msw(object(Person),Do),
msw(object(VB),Do), msw(punc(B),Pun),
msw(punc(Person),Pun), msw(punc(VB),Pun)$\}$

Assume that the atom msw(object(Person),Do) was randomly selected from C_1', then

$B = \{$msw(start,B), msw(subject,Person),
msw(verb,VB), msw(object(Person),Do)$\}$

The cluster C_1 is processed again with the new B. By deleting the atoms which are not grounded by {B,Person,VB,Do} and those whose outcomes are members of this set, the following set is then obtained

$C_1' = \{$msw(punc(B),Pun), msw(punc(Person),Pun),
msw(punc(VB),Pun),msw(punc(Do),Pun)$\}$

Assuming that msw(punc(Person),Pun) was randomly selected, B then becomes

$B = \{$msw(start,B), msw(subject,Person),
msw(verb,VB), msw(object(Person),Do),msw(punc(Person),Pun)$\}$

Generating C_1' from C_1 according to {B,Person,VB,Do,Pun} then leads to $C_1' = \emptyset$. B is then given as a candidate body. The clause that is randomly generated then becomes

```
sentence([B,Person,VB,Do,Pun]):-
    msw(start,B),
    msw(subject,Person),
    msw(verb,VB),
    msw(object(Person),Do),
    msw(punc(Person),Pun).
```

Algorithm 7 Cutting planes for selecting PRISM probabilistic atoms

1: $D = \{d_1, d_2, \ldots, d_n\}$ % n observations
2: $BC = (h \leftarrow B)$ % the bottom clause
3: $D' = trim\_obs\_from\_second\_to\_end(D)$
4: $all\_atoms\_scores = \emptyset$
5: **for all** $b \in B$ **do**
6: $\quad atom\_score = family\_score(b, D')$ % use equation (4.3) and return $\{b, Score\}$
7: $\quad all\_atoms\_scores = all\_atoms\_scores \cup \{atom\_score\}$
8: **end for**
9: $final\_atoms\_with\_scores = bn\_cutting\_planes(all\_atoms\_scores)$
10: **return** $atoms = atoms\_only(final\_atoms\_with\_scores)$ % drop off scores

When all clusters are visited, the randomly generated clause $(h' \leftarrow B)$ is obtained. Each generated clause is scored. The search generates a designated number of clauses and returns the body of the one with the maximum score.

5.2.2.3 Cutting Planes

The aim of using the cutting planes algorithm is to contribute to maximising the final BIC score by selecting the set of atoms from the body of the bottom clause which encodes the outcome dependencies that has the optimal log marginal likelihood score. This is achieved by finding the atoms with the optimal BDeu score based upon the statistics obtained from the values generated by the initial outcomes and the first iteration outcomes. Each atom in the body of the bottom clause represents a candidate family. For instance, `msw(object(Person),Do)` represents the family $\{Person\} \rightarrow Do$ and `msw(start,B)` represents the root node $\emptyset \rightarrow B$. As the BDeu score is decomposable, each atom is scored individually using (4.3). The score of each atom `msw(t,X)` is then represented in the cutting planes formalisation in (4.14) by $c(X, \mathbf{Pa})$ where $\mathbf{Pa} = vars(t)$. Following Cussens (2011), the SCIP (solving constraint integer programs) (Achterberg, 2007) framework is used to solve this IP problem. The pseudocode of this strategy is given in Algorithm 7.

5.2.3 Recursive Definition

When the atoms encoding the outcome dependencies are selected, they need to be split into two sets. The first set includes the initial atoms whose outcomes are used once in the generative process. The other set includes the iteration atoms which are used repetitively in the generative process. A predicate `rec_def/n` needs to be invented and defined by the iteration atoms.

```
[start/0,
 subject/0,verb/0,object/1,punc/1,subject/0,verb/0,object/1,punc/1
 subject/0,verb/0,object/1,punc/1]
[start/0,subject/0,verb/0,object/1,punc/1]
[start/0,subject/0,verb/0,object/1,punc/1]
[start/0,
 subject/0,verb/0,object/1,punc/1,subject/0,verb/0,object/1,punc/1
 subject/0,verb/0,object/1,punc/1,subject/0,verb/0,object/1,punc/1]
.
.
.
```

Figure 5.5: The Lists of the terms representing the switches that were used to generate the observations in Figure 5.1.

For each observation d_i, let Str_i be a list of the terms representing the switches that has generated the values in d_i. Let $Strings = \{Str_1, \ldots, Str_n\}$ be the set of the lists created from all observations. Figure 5.5 shows the set $Strings$ created from the observations in Figure 5.1. MERLIN 2.0 explained in Section 3.3.1 can then be used to construct a DFA from $Strings$. Recursion is then identified from the DFA using the pattern in Figure 5.6. Atoms whose terms are involved in the pattern are included in the set of iteration atoms. These represent the definition of the `rec_def/n` predicate. The output argument of this predicate (`-`) is the list containing the outcomes of the atoms defining it and a variable representing subsequent iterations. The input arguments to this predicate (`+`) are the outcomes of the initial atoms which some atoms in its body (iteration atoms) depend upon. For instance, in `Program 1` in Figure 5.2, there is no input to `rec_def/1` as none of the atoms in its body depend on the outcome of the initial atom `msw(start,B)`. However, in `Program 2`, `msw(object(B),Do)` depends on the outcome of `msw(start,B)`, thus `B` needs to be passed as an input argument to `rec_def/2`.

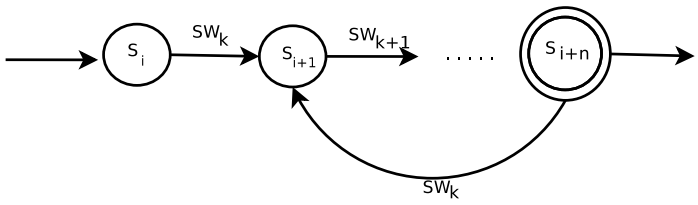


Figure 5.6: The Pattern of recursion in a DFA.

```
sentence([person1,studies,playing,',',person1,studies,playing,',']).
sentence([person1,likes,playing,',',person2,goes,walking,',']).
.
.
.
```

Figure 5.7: The values in the observations in Figure 5.1 generated by the first and second iterations of the sought program S.

5.2.4 Inter-Iteration Dependencies

The number of inter-iteration dependencies is the number of input arguments to the recursive predicate. This number has already been determined when the iteration atoms were identified. Let this number be k. The task is to designate the k outcomes of the iteration atoms that the atoms in the next iteration depend upon. The order of these outcomes needs also to be determined. For instance, if `X` and `Y` are two input arguments to `rec_def(A,B,[H|Tail])`, the two calls `rec_def(X,Y,Z)` and `rec_def(Y,X,Z)` are different. Therefore, given k-permutations of the outcomes of the iteration atoms, the problem of learning the iteration-dependencies amounts to finding a particular permutation.

These dependencies are learned based upon the statistics obtained from the values generated by the first and second iterations. A new set of observations is constructed whose elements are observations of these values. Let this be performed by the following function

$$D' = trim\_obs\_from\_initial\_and\_from\_third\_to\_end(D)$$

The set D' obtained from applying this function on the set of observations in Figure 5.1 is shown in Figure 5.7. Two instances of the body of the recursive predicate definition are created. Let these two instances be B_1 and B_2 respectively. B_1 represents the atoms generating the values of the first iteration and B_2 represents the atoms generating the values in the second iteration. Let the set of outcomes generated by B_1 be Var. Let the set of iteration atoms that depend upon the input arguments of the recursive predicate in B_2 be D_2. The score of each permutation is the sum of the scores of the atoms in the set D_2 with respect to this permutation. The task

is then to find the permutation with the maximum score. Let $C_{\prec 1}, \ldots, C_{\prec e}$ be the k-permutations of the set Var where $e = \frac{|Var|!}{(|Var|-k)!}$. For each atom $d_j \in D_2$, a set of the indices $In_{\prec j}$ of the input arguments that this atoms depends upon is created. The indices are the positions in the list of input arguments to the recursive predicate. They are ordered according to the appearance of these arguments in the term. For instance, given the following definition of a recursive predicate

```
rec_def(A,B,C,[X,Y,Z|Tail]):-
    .
    .
    msw(t(C,F,A),Y),
    .
    .
```

the set $In_{\prec}$ for the atom `msw(t(C,F,A),Y)` is $In = \{3, 1\}$. 3 is the position of the variable `C` in the list of input arguments for `rec_def(A,B,C,[X,Y,Z|Tail])` and 1 is the position of `A`. The score of each atom $d_j \in D_2$ with respect to the permutation $C_{\prec i}$ is then the score of the family represented by the atom when the input arguments in the term representing the switch are replaced by the variables in $C_{\prec i}$ according to the indices in $In_{\prec j}$. The atoms in D_2 are then scored with respect to each permutation. The permutation with the maximum score is then returned as the inter-iteration dependencies.

Halting the process is determined by a value generated by the atom specified by the bias. The outcome of this atom needs to be checked to see if it has been instantiated to this value (the atom `msw(punc(_),'.')` in the BK in Figure 5.1, and the outcome of this atom in the programs in Figure 5.2 is `Pun`). Therefore, this outcome is passed from `rec_def/n` to `stop/m` which halts the process if this unification has occurred; otherwise, `stop/m` calls `rec_def/n` back to continue the process. Thus, if the outcome is already in the designated permutation then $m = n$, otherwise the outcome is added to the permutation and $m = n + 1$.

5.3 Experiments

To evaluate the learning algorithm shown in Algorithm 8, five problems were modelled as PRISM programs. The first is a *small language* model where each sentence consists of a subject, a verb, an object and a punctuation mark. Therefore, this model has 4 outcomes in each iteration. The second is an adaptation of the *Asia* BN given in Cowell et al. (2007) which has 8 nodes. The BN was extended to a DBN by repeating the distribution in each slice and then the DBN was generalised to a

Algorithm 8 Learning PRISM programs with fully observed outcomes

1: $D = \{d_1, d_2, \ldots, d_n\}$ % n observations
2: BK = background knowledge
3: $(h \leftarrow B')$ = the bottom clause built using Algorithm 4.
4: B = the set of atoms selected from B' representing the outcome dependencies returned by Algorithm 7.
5: $\{TargetPred, RecPred\}$ = split the clause $(h \leftarrow B)$ into a target predicate definition and a recursive predicate definition as explained in Section 5.2.3.
6: $\{RecDef, STOP\}$ = find inter-iteration dependencies as explained in Section 5.2.4 and return the new body of the definition of the recursive predicate $RecPred$ found in the previous step as well as the definition of the halting condition $STOP$.
7: $S = \{TargetPred, RecDef, STOP\}_{\prec}$
8: **return** S

recursive PRISM program by defining a halting distribution. Thus, each iteration has 8 outcomes. The *small language* program and the *Asia* program were modelled such that there are no inter-iteration dependencies. The third program is an adaptation of a dynamic Bayesian network used in a *cervical cancer diagnosis* model (Agnieszka et al., 2009). It has 4 outcomes in each iteration and 3 inter-iteration dependencies. The fourth is an adaptation of a dynamic Bayesian network for maintenance decision making (McNaught and Zagorecki, 2009). It has 5 outcomes in each iteration and 3 inter-iteration dependencies. These two DBN models were adapted so that an iteration depends only on the previous one (the Markov property) and a distribution over halting the generative process was added. The last program is an adaptation of a subset of the alarm BN in the Bayesian network repository[1]. 10 nodes from the network in the repository were modelled such that two are initial outcomes and 8 are iteration outcomes with 2 inter-iteration dependencies.

Five sets of observations with each of the sizes 500, 1000, 1500 and 2000 where sampled from the programs. The algorithm was run to learn PRISM programs from each set of observations with the corresponding BK and bias. For each sample size, the ratios of the five programs learned to the original program from which these sets where generated were computed, and the average ratio were obtained. The results of the scores and the time taken by the learning algorithm in each run for the four programs is shown in the tables 5.1, 5.2, 5.3, 5.4 and 5.5. The learning time could almost certainly be reduced with more effort on code optimisation. Figure 5.8 shows the average ratio of the BIC scores of the learned programs to the original ones for each

[1] `http://www.cs.huji.ac.il/labs/compbio/Repository/`

sample size for the four programs. It can be noticed that the learned *small language* programs for the four set sizes have, on average, the same scores as the original one. The learned *Asia* programs for the set sizes of 500 and 1000 observations scored, on average, less than the original one. The scores of the learned programs for the set sizes of 1500 and 2000 observations attained the same scores as the original program. The *cervical cancer diagnosis* learned programs scored, on average, less than the original program for the four set sizes. However, the curve shows that the scores of the learned programs converge to the scores of the original program with the increase of the size of observations. The programs learned for *maintenance decision making* scored, on average, higher than the original program in the four set sizes. The scores also converge to the scores of the original program as the size of the observations increases. The *alarm* learned programs scored slightly lower, on average, than the original one for the two set sizes of 500 and 1000 and attained the same scores as the original one in the two sets of 1500 and 2000. The graphs in Figure 5.8 indicate that, in the limit of the size of observations, the scores of the learned programs, converge to the scores of the original.

Figure 5.11 shows the original *maintenance decision making* program from which 1000 observations were sampled and a program learned by Algorithm 8 from these observations and the BK and bias in Figure 5.10. The generalised *maintenance decision making* DBN is shown in Figure 5.9. It can be noticed that, in the learned program, the dependencies amongst the initial outcomes are the same as the ones in the original program. The arguments passed from the body of the target predicate definition to the recursive predicate which represent the dependencies between the initial outcomes and the first iteration outcomes are also the same as the ones in the original program. The dependencies amongst the iteration outcomes are the same in both programs. However, the two programs differ in the inter-iteration dependencies.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Small Language	500	1	**Learned**	−5176.4	1	3 sec	< 1 sec
			Original	−5176.4			< 1 sec
		2	**Learned**	−4863.8	1	2 sec	< 1 sec
			Original	−4863.8			< 1 sec
		3	**Learned**	−5077.5	1	2 sec	< 1 sec
			Original	−5077.5			< 1 sec
		4	**Learned**	−5061.2	1	4 sec	< 1 sec
			Original	−5061.2			< 1 sec
		5	**Learned**	−5114.7	1	2 sec	< 1 sec
			Original	−5114.7			< 1 sec
		Average ratio			1		
		Standard deviation			0		
	1000	1	**Learned**	−9707.5	1	3 sec	< 1 sec
			Original	−9707.5			< 1 sec
		2	**Learned**	−9954.7	1	4 sec	< 1 sec
			Original	−9954.7			< 1 sec
		3	**Learned**	−9545.0	1	3 sec	< 1 sec
			Original	−9545.0			< 1 sec
		4	**Learned**	−9719.5	1	3 sec	< 1 sec
			Original	−9719.5			< 1 sec
		5	**Learned**	−9374.0	1	3 sec	< 1 sec
			Original	−9374.0			< 1 sec
		Average ratio			1		
		Standard deviation			0		
	1500	1	**Learned**	−14735.0	1	5 sec	< 1 sec
			Original	−14735.0			< 1 sec
		2	**Learned**	−14368.2	1	5 sec	< 1 sec
			Original	−14368.2			< 1 sec
		3	**Learned**	−14537.4	1	5 sec	< 1 sec
			Original	−14537.4			< 1 sec
		4	**Learned**	−14014.0	1	5 sec	< 1 sec
			Original	−5524.2			< 1 sec
		5	**Learned**	−5549.2	1	5 sec	< 1 sec
			Original	−14513.7			< 1 sec
		Average ratio			1		
		Standard deviation			0		
	2000	1	**Learned**	−19526.3	1	7 sec	< 1 sec
			Original	−19526.3			< 1 sec
		2	**Learned**	−20040.8	1	7 sec	< 1 sec
			Original	−20040.8			< 1 sec
		3	**Learned**	−19778.1	1	7 sec	< 1 sec
			Original	−19778.1			< 1 sec
		4	**Learned**	−19608.9	1	7 sec	< 1 sec
			Original	−19608.9			< 1 sec
		5	**Learned**	−19263.7	1	6 sec	< 1 sec
			Original	−19263.7			< 1 sec
		Average ratio			1		
		Standard deviation			0		

Table 5.1: The result of the experiments on learning the *small language* program with fully observed outcomes.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Asia	500	1	**Learned**	−8677.5	1	23 sec	< 1 sec
			Original	−8677.5			< 1 sec
		2	**Learned**	−9560.9	1.0003	30 sec	< 1 sec
			Original	−9557.3			< 1 sec
		3	**Learned**	−9237.4	1.0002	6 sec	< 1 sec
			Original	−9234.8			< 1 sec
		4	**Learned**	−9176.6	1.0002	7 sec	< 1 sec
			Original	−9174.6			< 1 sec
		5	**Learned**	−8731.6	1.0012	5 sec	< 1 sec
			Original	−8720.9			< 1 sec
		Average ratio			1.00041		
		Standard deviation			0.00042		
	1000	1	**Learned**	−17896.9	1.0017	27 sec	< 1 sec
			Original	−17864.9			< 1 sec
		2	**Learned**	−19082.5	1	12 sec	< 1 sec
			Original	−19082.5			< 1 sec
		3	**Learned**	−18871.5	1	24 sec	< 1 sec
			Original	−18871.5			< 1 sec
		4	**Learned**	−19104.2	1	12 sec	< 1 sec
			Original	−19104.2			< 1 sec
		5	**Learned**	−18811.0	1	10 sec	< 1 sec
			Original	−18811.0			< 1 sec
		Average ratio			1.0003		
		Standard deviation			0.000715		
	1500	1	**Learned**	−25842.2	1	49 sec	1 sec
			Original	−25842.2			1 sec
		2	**Learned**	−24825.2	1	15 sec	1 sec
			Original	−24825.2			1 sec
		3	**Learned**	−24908.9	1	14 sec	1 sec
			Original	−24908.9			1 sec
		4	**Learned**	−25427.2	1	15 sec	1 sec
			Original	−25427.2			1 sec
		5	**Learned**	−25003.2	1	24 sec	1 sec
			Original	−25003.2			1 sec
		Average ratio			1		
		Standard deviation			0		
	2000	1	**Learned**	−33724.7	1	56 sec	1 sec
			Original	−33724.7			1 sec
		2	**Learned**	−39716.2	1	39 sec	1 sec
			Original	−39716.2			1 sec
		3	**Learned**	−38677.0	1	40 sec	1 sec
			Original	−38677.0			1 sec
		4	**Learned**	−40237.2	1	42 sec	1 sec
			Original	−40237.2			1 sec
		5	**Learned**	−39737.2	1	49 sec	1 sec
			Original	−39737.2			1 sec
		Average ratio			1		
		Standard deviation			0		

Table 5.2: The result of the experiments on learning the *Asia* program with fully observed outcomes.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Cervical	500	1	**Learned**	−34455.1	1.236	33 sec	1 sec
			Original	−27870.7			1 sec
		2	**Learned**	−35634.8	1.264	9 sec	1 sec
			Original	−28179.8			1 sec
		3	**Learned**	−34760.7	1.221	58 sec	1 sec
			Original	−28448.5			1 sec
		4	**Learned**	−35766.6	1.284	8 sec	1 sec
			Original	−27837.6			1 sec
		5	**Learned**	−35393.1	1.279	36 sec	1 sec
			Original	−27655.5			1 sec
		Average ratio			1.257		
		Standard deviation			0.0245		
	1000	1	**Learned**	−60541.7	1.195	1 min & 24 sec	1 sec
			Original	−50627.0			1 sec
		2	**Learned**	−64392.9	1.247	16 sec	1 sec
			Original	−51611.0			1 sec
		3	**Learned**	−61654.0	1.240	1 min & 9 sec	1 sec
			Original	−49720.4			1 sec
		4	**Learned**	−49927.3	1	39 sec	1 sec
			Original	−49927.3			1 sec
		5	**Learned**	−52010.7	1	22 sec	1 sec
			Original	−52010.7			1 sec
		Average ratio			1.136		
		Standard deviation			0.113		
	1500	1	**Learned**	−72276.2	1	1 min & 48 sec	1 sec
			Original	−72276.2			1 sec
		2	**Learned**	−90342.9	1.240	28 sec	1 sec
			Original	−72856.2			1 sec
		3	**Learned**	−89019.4	1.213	48 sec	1 sec
			Original	−73359.1			1 sec
		4	**Learned**	−74438.5	1	35 sec	1 sec
			Original	−74438.5			1 sec
		5	**Learned**	−88348.9	1.223	28 sec	1 sec
			Original	−72191.6			1 sec
		Average ratio			1.135		
		Standard deviation			0.110		
	2000	1	**Learned**	−93055.0	1	2 min & 8 sec	2 sec
			Original	−93055.0			2 sec
		2	**Learned**	−114488.6	1.183	38 sec	2 sec
			Original	−96742.4			2 sec
		3	**Learned**	−93387.6	1	47 sec	2 sec
			Original	−93387.6			2 sec
		4	**Learned**	−96181.1	1	48 sec	2 sec
			Original	−96181.1			2 sec
		5	**Learned**	−95927.4	1	1 min & 14 sec	2 sec
			Original	−95927.4			2 sec
		Average ratio			1.036		
		Standard deviation			0.073		

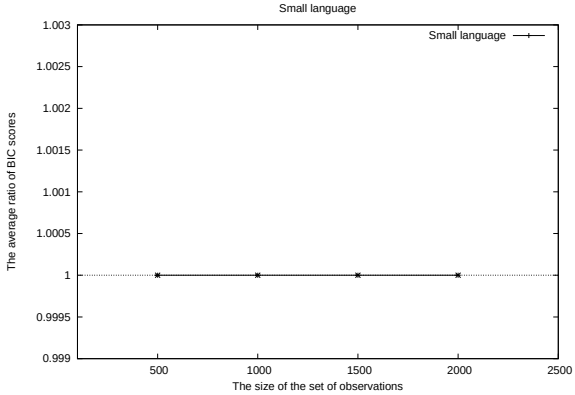
Table 5.3: The result of the experiments on learning the *cervical cancer diagnosis* program with fully observed outcomes.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Maintenance	500	1	**Learned**	−12672.0	0.972	23 sec	< 1 sec
			Original	−13029.5			< 1 sec
		2	**Learned**	−11964.8	1.014	22 sec	< 1 sec
			Original	−11789.7			< 1 sec
		3	**Learned**	−11348.6	0.967	22 sec	< 1 sec
			Original	−11725.7			< 1 sec
		4	**Learned**	−12153.1	0.969	21 sec	< 1 sec
			Original	−12530.8			< 1 sec
		5	**Learned**	−11317.9	0.963	22 sec	< 1 sec
			Original	−11752.5			< 1 sec
		Average ratio			0.977		
		Standard deviation			0.0188		
	1000	1	**Learned**	−23189.6	0.978	42 sec	< 1 sec
			Original	−23688.5			< 1 sec
		2	**Learned**	−23133.6	0.978	41 sec	< 1 sec
			Original	−23653.9			< 1 sec
		3	**Learned**	−22818.3	0.977	41 sec	< 1 sec
			Original	−23332.5			< 1 sec
		4	**Learned**	−23446.8	0.984	41 sec	< 1 sec
			Original	−23827.9			< 1 sec
		5	**Learned**	−22719.6	0.977	42 sec	< 1 sec
			Original	−23233.4			< 1 sec
		Average ratio			0.979		
		Standard deviation			0.0023		
	1500	1	**Learned**	−33911.4	0.983	1 min & 2 sec	< 1 sec
			Original	−34463.5			< 1 sec
		2	**Learned**	−33759.1	0.983	1 min & 1 sec	< 1 sec
			Original	−34318.1			< 1 sec
		3	**Learned**	−34317.6	0.983	1 min & 1 sec	< 1 sec
			Original	−34885.9			< 1 sec
		4	**Learned**	−33462.5	0.983	1 min	< 1 sec
			Original	−34035.3			< 1 sec
		5	**Learned**	−34232.7	0.983	1 min & 2 sec	< 1 sec
			Original	−34792.1			< 1 sec
		Average ratio			0.983		
		Standard deviation			0		
	2000	1	**Learned**	−44624.2	0.986	1 min & 22 sec	< 1 sec
			Original	−45215.6			< 1 sec
		2	**Learned**	−45022.1	0.987	1 min & 23 sec	< 1 sec
			Original	−45605.0			< 1 sec
		3	**Learned**	−45008.2	0.987	1 min & 21 sec	< 1 sec
			Original	−45565.7			< 1 sec
		4	**Learned**	−44694.0	0.986	1 min & 23 sec	< 1 sec
			Original	−45289.0			< 1 sec
		5	**Learned**	−45052.7	0.987	1 min & 22 sec	< 1 sec
			Original	−45609.6			< 1 sec
		Average ratio			0.987		
		Standard deviation			0		

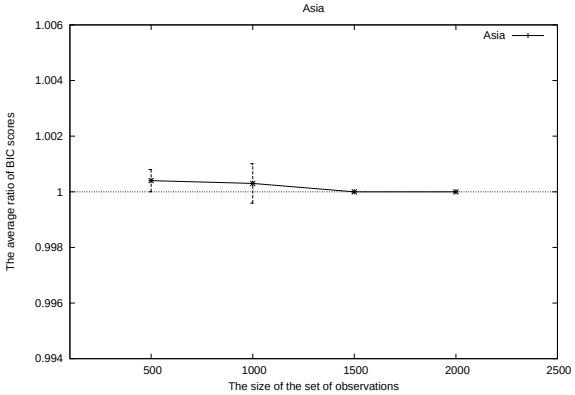
Table 5.4: The result of the experiments on learning the *maintenance decision making* program with fully observed outcomes.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Alarm	500	1	**Learned**	−21730.7	1.006	28 sec	< 1 sec
			Original	−21587.6			< 1 sec
		2	**Learned**	−16539.7	1.002	25 sec	< 1 sec
			Original	−16502.9			< 1 sec
		3	**Learned**	−17510.3	1.015	26 sec	< 1 sec
			Original	−17248.2			< 1 sec
		4	**Learned**	−17028.4	1	26 sec	< 1 sec
			Original	−17028.4			< 1 sec
		5	**Learned**	−17261.6	1.003	26 sec	< 1 sec
			Original	−17194.4			< 1 sec
		Average ratio			1.005		
		Standard deviation			0.005		
	1000	1	**Learned**	−45001.7	1.006	51 sec	< 1 sec
			Original	−44709.6			< 1 sec
		2	**Learned**	−38450.9	1	47 sec	< 1 sec
			Original	−38450.9			< 1 sec
		3	**Learned**	−37659.8	1	46 sec	< 1 sec
			Original	−37659.8			< 1 sec
		4	**Learned**	−37790.0	1	47 sec	< 1 sec
			Original	−37790.0			< 1 sec
		5	**Learned**	−38760.8	1	47 sec	< 1 sec
			Original	−38760.8			< 1 sec
		Average ratio			1.001		
		Standard deviation			0.002		
	1500	1	**Learned**	−63610.8	1	47 sec	< 1 sec
			Original	−63610.8			< 1 sec
		2	**Learned**	−67526.5	1	47 sec	< 1 sec
			Original	−67526.5			< 1 sec
		3	**Learned**	−67434.8	1	47 sec	< 1 sec
			Original	−67434.8			< 1 sec
		4	**Learned**	−66961.3	1	47 sec	< 1 sec
			Original	−66961.3			< 1 sec
		5	**Learned**	−66149.8	1	47 sec	< 1 sec
			Original	−66149.8			< 1 sec
		Average ratio			1		
		Standard deviation			0		
	2000	1	**Learned**	−85376.9	1	1 min & 26 sec	1 sec
			Original	−85376.9			1 sec
		2	**Learned**	−73925.0	1	1 min & 30 sec	1 sec
			Original	−73925.0			1 sec
		3	**Learned**	−73136.0	1	1 min & 31 sec	1 sec
			Original	−73136.0			1 sec
		4	**Learned**	−72085.3	1	1 min & 27 sec	1 sec
			Original	−72085.3			1 sec
		5	**Learned**	−73415.3	1	1 min & 28 sec	1 sec
			Original	−73415.3			1 sec
		Average ratio			1		
		Standard deviation			0		

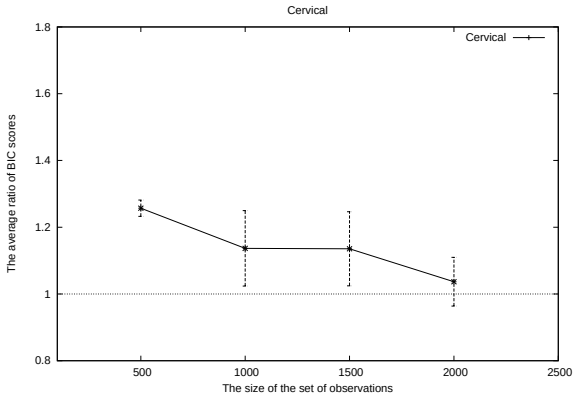
Table 5.5: The result of the experiments on learning the *alarm* program with fully observed outcomes.



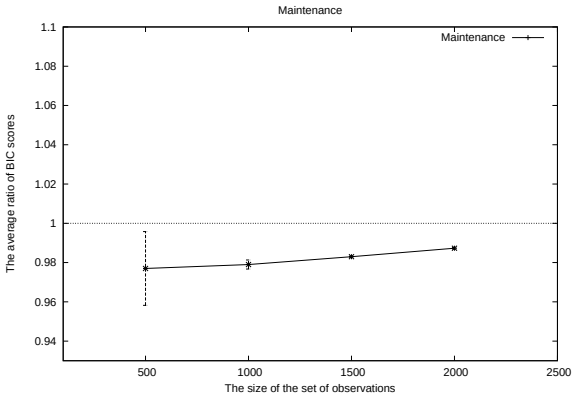
(a) The *small language* program.



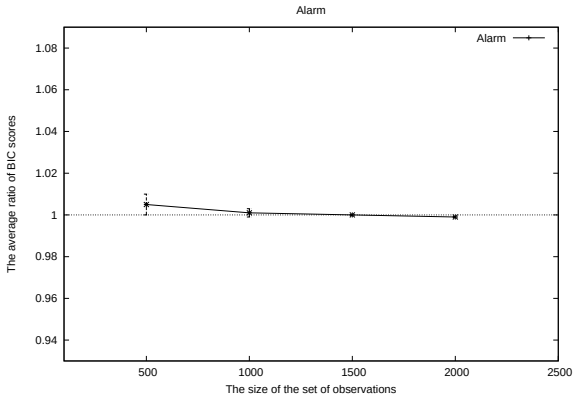
(b) The *Asia* program.



(c) The *cervical cancer diagnosis* program.



(d) The *maintenance decision making* program.



(e) The *alarm* program.

Figure 5.8: The ratios of the BIC scores of the learned programs to the original programs. Each point represents an average of 5 ratios of BIC scores of programs learned from 5 different and independent samples to the BIC scores of the original programs from which these samples were generated. The bars represent the standard deviations.

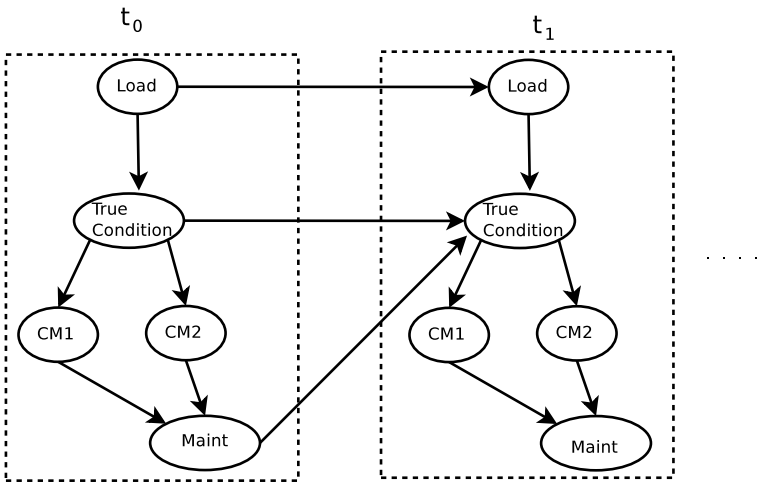


Figure 5.9: The *maintenance decision making* DBN.

```
values(load,[normal,abnormal]).
values(true_condition(_),[good,wear_1,wear_2,wear_3,failure_mode1,failure_mode2]).
values(cm1(_),[low1,medium1,high1]).
values(cm2(_),[low2,medium2,high2]).
values(maintenance(_,_),[none,reset,replace]).
values(load_t(_),[t_normal,t_abnormal,t_halt]).
values(true_condition_t(_,_,_),[t_good,t_wear_1,t_wear_2,t_wear_3,t_failure_mode1,
                                t_failure_mode2]).
values(cm1_t(_),[t_low1,t_medium1,t_high1]).
values(cm2_t(_),[t_low2,t_medium2,t_high2]).
values(maintenance_t(_,_),[t_none,t_reset,t_replace]).

% The bias specifying the halt (HT)
stop:- msw(load_t(_),t_halt).
```

Figure 5.10: The BK and the bias used to learn the *maintenance decision making* PRISM programs.

Original:

```
% To generate the initial outcomes
values(load,[normal,abnormal]).
values(true_condition(_),
      [good,wear_1,wear_2,wear_3,
      failure_mode1,failure_mode2]).
values(cm1(_),[low1,medium1,high1]).
values(cm2(_),[low2,medium2,high2]).
values(maintenance(_,_),
       [none,reset,replace]).
% To generate the iteration outcomes
values(load_t(_),
       [t_normal,t_abnormal,t_halt]).
values(true_condition_t(_,_,_),
       [t_good,t_wear_1,t_wear_2,
        t_wear_3,t_failure_mode1,
        t_failure_mode2]).
values(cm1_t(_),
       [t_low1,t_medium1,t_high1]).
values(cm2_t(_),
       [t_low2,t_medium2,t_high2]).
values(maintenance_t(_,_),
       [t_none,t_reset,t_replace]).

decision([L,TC,CM1,CM2,MAINT|Tail]):-
    msw(load,L),
    msw(true_condition(L),TC),
    msw(cm1(TC),CM1),
    msw(cm2(TC),CM2),
    msw(maintenance(CM1,CM2),
        MAINT),
    decision1(L,TC,MAINT,Tail).

decision1(L,TC,MAINT,
          [L_t,TC_t,CM1_t,CM2_t,
          MAINT_t|Tail]):-
    msw(load_t(L),L_t),
    msw(true_condition_t(L_t,TC,MAINT),
        TC_t),
    msw(cm1_t(TC_t),CM1_t),
    msw(cm2_t(TC_t),CM2_t),
    msw(maintenance_t(CM1_t,CM2_t),
        MAINT_t),
    stop(L_t,TC_t,MAINT_t,Tail).

stop(t_halt,_,_,[]):- !.
stop(L_t,TC_t,MAINT_t,Tail):-
    decision1(L_t,TC_t,MAINT_t,Tail).
```

Learned:

```
values(load,[normal,abnormal]).
values(true_condition(_),
       [good,wear_1,wear_2,wear_3,
        failure_mode1,failure_mode2]).
values(cm1(_),[low1,medium1,high1]).
values(cm2(_),[low2,medium2,high2]).
values(maintenance(_,_),
       [none,reset,replace]).

values(load_t(_),
       [t_normal,t_abnormal,t_halt]).
values(true_condition_t(_,_,_),
       [t_good,t_wear_1,t_wear_2,
        t_wear_3,t_failure_mode1,
        t_failure_mode2]).
values(cm1_t(_),
       [t_low1,t_medium1,t_high1]).
values(cm2_t(_),
       [t_low2,t_medium2,t_high2]).
values(maintenance_t(_,_),
       [t_none,t_reset,t_replace]).

decision([A,B,C,D,E,F,G,H,I,J|K]):-
      msw(load,A),
      msw(true_condition(A),B),
      msw(cm1(B),C),
      msw(cm2(B),D),
      msw(maintenance(C,D),E),
      rec_def(A,B,E,[F,G,H,I,J|K]).

rec_def(A,B,E,[F,G,H,I,J|K]):-
      msw(load_t(A),F),
      msw(true_condition_t(B,E,F),G),
      msw(cm1_t(G),H),
      msw(cm2_t(G),I),
      msw(maintenance_t(H,I),J),
      stop(I,F,H,K).

stop(A,t_halt,B,[]):-
      !.
stop(I,F,H,K):-
      rec_def(I,F,H,K).
```

Figure 5.11: Two PRISM programs representing the original *maintenance decision making* program from which 1000 observations were sampled, and the program which was learned from these observations and the BK and bias given in Figure 5.10.

Chapter 6

Learning Recursive PRISM Programs with Hidden Outcomes

Algorithm 8 given in Chapter 5 is no longer useful in the presence of hidden outcomes. This is because the log marginal likelihood no longer decomposes into the sum of the family scores. Thus, algorithms which rely on decomposable scores such as the cutting planes cannot be used. Another problem resulting from having hidden outcomes is that splitting the initial outcomes from the iteration outcomes using the approach in Section 5.2.3 will not consider the atoms generating the hidden outcomes. This chapter presents a learning algorithm which alleviates the former problem by using a metaheuristics search and the latter problem by injecting more declarative bias. Section 6.1 discusses the problems arising from having hidden outcomes and introduces the learning problem. Section 6.2 introduces the simulated annealing search, the metaheuristics method that will be adopted. Section 6.3 explains the learning algorithm. Finally, Section 6.4 shows the experiments conducted to learn five programs.

6.1 Introduction

In Algorithm 8, the probabilistic atoms in a PRISM program are chosen so that they encode outcome dependencies with the optimal log marginal likelihood. This is achieved with the cutting planes search which finds the optimal score of the set of atoms encoding the dependencies given the scores of each individual atom in the bottom clause. This step thus requires the score to be decomposable. However, as discussed in Section 4.2.1, one of the three conditions necessary for the log marginal likelihood to be decomposable is that the data is fully observed. When there are some hidden outcomes, the decomposition of the log marginal likelihood into the

scores in (4.3) no longer holds. A solution to this problem is to alter Algorithm 8 so that it uses the greedy generalisation or the random search for finding the outcome dependencies instead of the cutting planes. However, it has been shown in Chapter 5 that greedy generalisation is problematic when there is a large number of atoms in the bottom clause. Random search is completely stochastic and does not have any guidance. An alternative direction is to apply a search strategy over complete recursive PRISM programs. This differs from Algorithm 8 which divides the learning into two search tasks, the first is searching for a PRISM clause which encodes outcome dependencies and the second is searching for arguments to pass as inter-iteration dependencies. The motivation behind performing a single search step is that the advantage of the division in Algorithm 8 which allows searching for outcome dependencies with the optimal log marginal likelihood score using family scores no longer applies. Applying any other search which does not guarantee finding the optimal set of atoms encoding the dependencies and then applying another search for finding inter-iteration dependencies has no salient advantage over combining these two steps into a single search task. By combining these two steps, the search starts from a random complete recursive PRISM program and moves in the search space by altering the program according to the BIC score. An obvious choice for this task is the structural-EM algorithm given in Algorithm 2. However, in this thesis the interaction with the PRISM system is at the modelling level. At this level, the intermediate values of the parameters when running the gEM cannot be accessed. Moreover, the parameters cannot be initialised to specific values[1] (Sato et al., 2010). In order to control these settings, interacting with PRISM at the system level is required; however, this is beyond the scope of this thesis. Metaheuristics methods allow searching a very large search space by exploring different areas in the space (Dréo et al., 2006). Simulating annealing is a metaheuristics search which avoids the local maximum resulting from hill-climbing. It allows moving downhill with a certain probability at each step of the search. Figure 6.1 shows a scenario of avoiding a local maximum by moving downhill from state $S(n)$ to state $S(n+1)$ and then to state $S(n+2)$. This chapter employs simulated annealing search in learning recursive PRISM programs with hidden outcomes.

The approach of building the definition of the recursive predicate explained in Section 5.2.3 is based upon the values in the observations that the atoms generate. Based upon these values, the initial atoms are split from the iteration atoms and the recursive predicate definition is formed. When hidden outcomes are present in

[1]The work in this thesis is based on PRISM version 2.0.

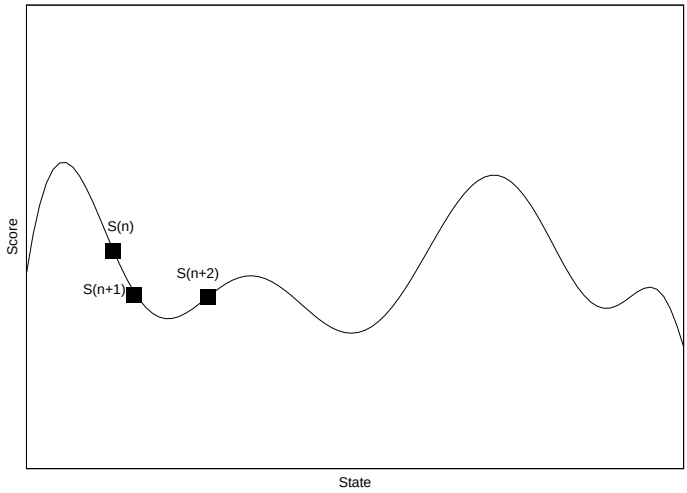


Figure 6.1: Allowing a move downhill to alleviate getting trapped in a local maximum. If the search moves downhill from state $S(n)$ to state $S(n+1)$, another area of the objective function can be explored by moving to $S(n+2)$.

the programs, their values do not exist in the observations and the approach cannot determine whether the atoms which generate these outcomes are initial atoms or iteration atoms. To overcome this problem, the algorithm assumes that a declarative bias is provided in the form of a theory. This theory defines the hidden outcomes that may be generated in the body of the target predicate definition (initial hidden outcomes) and the hidden outcomes that may be generated in the body of the recursive predicate definition (iteration hidden outcomes).

The learning problem is defined as follows

Given:

- A set D of observations in FOL.
- Background knowledge B in FOL consisting of
  - A set F of k functions where each function represents a set of switches.
  - A set of ground terms G representing the values that the switches may generate.
  - Sets $l_1, \ldots, l_k$ such that $\forall i : 1 \leq i \leq k$, $l_i \subseteq G$ and $\bigcap_{i=1}^{k} l_i = \emptyset$.
  - A set SD of atoms $\texttt{values}(f_i, l_i)$ where $f_i \in F$ and $1 \leq i \leq k$ declaring the switches.

  - A set of probabilistic atoms $\texttt{msw}(f, v)$ where $f \in F$ and $\exists l_i : v \in l_i$ such that $\texttt{values}(f, l_i) \in SD$.

- A declarative bias HT specifying the function f representing the set of switches that halt the generative process. Let these switches generate a value `t` to halt the process, then the bias is given as follows:
  ```
  stop:- msw(f,t).
  ```
  where $f \in F$ and $\exists l_i : t \in l_i$ such that $\texttt{values}(f, l_i) \in SD$.

- A declarative bias HID specifying the initial hidden outcomes and the iteration hidden outcomes as follows:

  - A clause c_1 such that $head(c_1)$ is an atom of any user defined predicate $\neq$ `rec_def/0` and $body(c_1) = \{\bigwedge_{i=1}^{j} \texttt{msw}(f_i, \_), \texttt{rec\_def}\}$ where $1 \leq j \leq k$, $f_i \in F$ such that f_i is a function representing switches that may generate initial hidden outcomes.
  - A clause c_2 such that $head(c_2) = \texttt{rec\_def}$ and $body(c_2) = \{\bigwedge_{i=1}^{j} \texttt{msw}(f_i, \_), \texttt{rec\_def}\}$ where $1 \leq j \leq k$, $f_i \in F$ such that f_i is a function representing switches that may generate iteration hidden outcomes.

Find: a PRISM program S represented by the predicates in $D \cup B \cup \{\texttt{rec\_def/n}, \texttt{stop/m}\}$ modelling a generative process which halts as specified by HT and may generate hidden outcomes as specified by HID such that $\forall d \in D : B \cup S \models d$ and

$$S = \arg\max_{S'} \ \log P(D|S', \hat{\boldsymbol{\theta}}) - \frac{dim}{2} \log N$$

In Figure 6.2, the biases HT and HID in `BK 1` state that halting the process is based upon the hidden outcome of `msw(cont(_),_)` in the body of the recursive predicate definition such that the process stops when this outcome is unified with `halt`. `BK 2` changes the outcome space of the term `start` to the set {`begin`,`new_paragraph`,`new_sentence`} and makes this outcome hidden. This outcome is specified by the bias HID to be in the body of the target predicate definition (initial hidden outcome). The bias defines the hidden outcomes in the body of the recursive predicate definition in the same way defined in `BK 1`.

The rest of this chapter is organised as follows: Section 6.2 introduces the simulated annealing search, Section 6.3 explains the learning algorithm and Section 6.4 shows experiments conducted to learn five programs.

BK 1:

```
values(start,['Hello!','Welcome!']).
values(noun(_),[person1,person2,
                person3]).
values(verb,[likes,goes,studies]).
values(adj(_),[playing,walking,
               shopping]).
values(punc(_),[comma,full_stop]).

% Hidden
values(cont(_),[halt,continue]).

% The bias specifying the halt (HT)
stop:- msw(cont(_),halt).

% The bias specifying hidden outcomes
% (HID)

% No initial hidden outcome
sentence:-
    rec_def.
% The iteration hidden outcome
rec_def:-
    msw(cont(_),_),
    rec_def.
```

BK 2:

```
values(noun(_),[person1,person2,
                person3]).
values(verb,[likes,goes,studies]).
values(adj(_),[playing,walking,
               shopping]).
values(punc(_),[comma,full_stop]).

% Hidden
values(start,[begin,new_paragraph,
              new_sentence]).
values(cont(_),[halt,continue]).

% The bias of the halting condition
stop:- msw(cont(_),halt).

% The bias specifying hidden outcomes
% (HID)

% The initial hidden outcome
sentence:-
    msw(start,_),
    rec_def.
% The iteration hidden outcome
rec_def:-
    msw(cont(_),_),
    rec_def.
```

Figure 6.2: On the left is a BK with a bias stating that there are no hidden outcomes in the body of the target predicate definition and there is one hidden outcome in the body of the recursive predicate definition. The bias in the BK on the right states that there is one hidden outcome in the bodies of the target and recursive predicate definitions.

6.2 Simulated Annealing

The *simulated annealing* (SA) algorithm simulates a process in physics applied to solid materials to increase the size of crystals in them. This is achieved by bringing the system to a state of a lower energy than the state it was. In this process, a material is heated above its melting temperature and brought to a state of high energy. Then, it is gradually cooled down with control. Cooling the material slowly allows the energy to decrease gradually and more states of energy are visited. This leads to a state with lower energy than the energy of the material before it has been heated. On the other hand, cooling the material quickly reduces the states that are visited during the

Algorithm 9 Simulated annealing

1: $CurrentState$ = initial state
2: $Score_{current} = Score(CurrentState)$
3: $Score_{best} = Score_{current}$
4: $BestState = CurrentState$
5: T = initial high temperature
6: **while** T is not low enough **do**
7: $T = cool\_according\_to\_schedule(T)$
8: $NewState = pick\_random\_neighbour(CurrentState)$
9: $Score_{new} = Score(NewState)$
10: $\Delta Score = Score_{new} - Score_{current}$
11: **if** $\Delta Score > 0$ **then**
12: $CurrentState = NewState$
13: $Score_{current} = Score_{new}$
14: **else**
15: $P = \exp(\frac{\Delta Score}{T})$
16: $CurrentState$ = move to $NewState$ with probability P. If the move has taken place apply $Score_{current} = Score_{new}$
17: **end if**
18: **if** $Score_{current} > Score_{best}$ **then**
19: $Score_{best} = Score_{current}$
20: $BestState = CurrentState$
21: **end if**
22: **end while**
23: **return** $BestState$

cooling process and makes the system reach a local minimum energy. This process is known in physics as *annealing* (Dréo et al., 2006).

Kirkpatrick et al. (1983) simulated the annealing process to solve problems in optimisation. The aim is to avoid local minima/maxima by exploring more areas in the search space. In applying the algorithm to finding a state with a high score, the energy of a state corresponds to its score. The algorithm starts from an initial state which can be chosen randomly. It then randomly chooses a neighbouring state and computes its score. If the score of the new state is higher than the current one, the algorithm moves to it. If the score is lower, the algorithm moves to the new state with probability $P = \exp(\frac{\Delta Score}{T})$ where

$$\Delta Score = Score(NewState) - Score(CurrentState)$$

and T is the current temperature. It can be noticed that the probability of moving to a state with a lower score depends on both the difference in the scores and the temperature. The higher the temperature, the greater the probability of moving to states with lower scores. Thus, the higher temperature, the higher the chance of moving downhill and exploring other areas in the search space. The temperature is reduced iteratively according to some schedule. When the temperature becomes low, the search behaves more like hill-climbing as the probability of moving downhill becomes very low. The state with the best score visited during the search is then returned[2]. SA is shown in Algorithm 9.

6.3 Learning

The learning starts from a randomly selected recursive PRISM program. This program is built by, first, computing the lgg of all the observations to obtain the head of the target predicate definition which includes the observed outcomes. A clause is then formed whose head is the result of the lgg and whose body consists of all the atoms generating the outcomes in the head and all the atoms defined to generate the hidden outcomes by the bias HID. For instance, From `BK 2` in Figure 6.2 and the observations in Figure 6.3, the following clause is built

```
sentence([Person,VB,Do,Pun]):-
    msw(subject,Person),
    msw(start,H1),
    msw(verb,VB),
    msw(object(_),Do),
    msw(punc(_),Pun),
    msw(cont(_),H2).
```

Note that the clause does not encode any outcome dependencies. Thus, this clause is not a valid PRISM clause as atoms with parameterised terms are not grounded when sampling from this clause. Before selecting a random setting of the outcome dependencies, the atoms in the body need to be split into the initial atoms and the iteration atoms. Applying this step before choosing a random setting of the outcome dependencies is needed because initial atoms must depend on initial outcomes. The splitting is performed using the approach in Section 5.2.3. Random settings of the outcome dependencies and the inter-iteration dependencies are then chosen and an initial program is built accordingly.

[2]Some references design the algorithm so that it returns the state at which the search terminates. The rationale behind this is to exactly simulate the annealing process. But as the aim is optimisation, returning the state with best score amongst all the visited states is more appropriate.

```
sentence([person1,studies,playing,',',
          person1,studies,playing,',',
          person3,goes,playing,'.',
          person2,goes,shopping,',',]).
sentence(['Hello!',
          person3,likes,playing,'.']).
sentence([person3,goes,walking,',']).
sentence([person1,likes,playing,',',
          person2,goes,walking,'.',
          person3,studies,shopping,',',
          person3,studies,playing,'.']).
.
.
.
```

Figure 6.3: Observations generated by a PRISM program which needs to be learned according to `BK 2` in Figure 6.2.

SA takes the randomly built program as an initial state to start with. SA was configured to start with the initial temperature 1000000. The cooling schedule is configured according to the approach given by Kirkpatrick et al. (1983) in which the temperature at step n is defined as follows

$$T_n = \alpha \, T_{n-1}$$

We follow Kirkpatrick et al. with the setting $\alpha = 0.95$.

We define five move operations which the SA algorithm applies to move to a neighbour in the search space. These operations are defined such that each move changes a dependency either between two outcomes in the same iteration or between two outcomes in two consecutive iterations. These operations are the following

Change a dependency in the body of the target predicate definition: when this move operation is performed, an atom in the body of the target predicate definition which has a parameterised term t is selected randomly. One of the variables $A \in vars(t)$ in this term is replaced by an initial outcome $A' \notin vars(t)$ such that when the operation is performed $vars(t) = vars(t) \setminus \{A\} \cup \{A'\}$. For example, consider the following definition of the target predicate

```
target_pred([A,B,C,D|Tail]):-
    msw(t1,A),
    msw(t2,B),
    msw(t3(B),H1),
    rec_def(H1,[C,D|Tail]).
```

By applying this operation, the argument `B` to the term `t3/1` is changed with `A` and the atom becomes `msw(t3(A),H1)`.

Change a dependency in the body of the recursive predicate definition: this move operation changes a dependency in the body of the recursive predicate definition `rec_def/n` in the same way as the previous operation does in the body of the target predicate definition.

Replace an argument to the recursive predicate: this operation replaces one of the initial outcomes passed from the body of the target predicate definition to the recursive predicate with another initial outcome. Given the definition of the target predicate shown in the first operation, a result which can be obtained is to replace `H1` with either `A` or `B`. If `A` is randomly selected, the result then becomes

```
target_pred([A,B,C,D|Tail]):-
    msw(t1,A),
    msw(t2,B),
    msw(t3(B),H1),
    rec_def(A,[C,D|Tail]).
```

Replace an argument to the `stop/m` predicate: this operation behaves like the previous one. However, it works on the body of the definition of the recursive predicate `rec_def/n` instead of the body of the target predicate definition. It replaces an argument to the predicate `stop/m` with another iteration outcome which is not in the current list of input arguments to `stop/m`. However, the outcome which determines halting the generative process is excluded from the list of arguments that can be replaced.

Add an argument to the recursive predicate: this operation adds an input argument to the recursive predicate which is not in the current list of input. The argument that is added is selected randomly from those which are not in the current list. For instance applying this operation to the definition of the target predicate given in the first operation by randomly selecting `B` leads to the following new definition

```
target_pred([A,B,C,D|Tail]):-
    msw(t1,A),
    msw(t2,B),
    msw(t3(B),H1),
    rec_def(H1,B,[C,D|Tail]).
```

Note that this operation is applied to all the clauses in the program. Increasing the arity of the recursive predicate `rec_def/n` requires changing the arity of the `stop/m` predicate. Therefore, an iteration outcome which does not belong to the input arguments to the `stop/m` predicate is randomly selected and added to the list.

Some of these operations are not applicable on certain states. For example, if the search reaches a state where the definition of the target predicate is the following

```
target_pred([A,B,C,D|Tail]):-
    msw(t1,A),
    msw(t2,B),
    msw(t3(B),H1),
    rec_def(H1,B,A,[C,D|Tail]).
```

the operation of adding an argument to the recursive predicate is not applicable as there are no further outcomes to add. In this case, this move operation is neglected and another operation is randomly chosen.

When the SA algorithm terminates and returns a program, this program might have some hidden outcomes which no other atoms depend upon. These hidden outcomes have no salient role in the program. Therefore, a final step is to eliminate these hidden outcomes from the program returned by the SA algorithm.

6.4 Experiments

The five programs in Section 5.3 were modified as follows: in the *small language* program, two hidden outcomes were added such that one is an initial outcome and the other is an iteration outcome. In the *Asia* program, two iteration hidden outcomes were added to the program. In the *cervical cancer diagnosis* and the *maintenance decision making* programs one initial hidden outcome and one iteration hidden outcome were added in both. Finally, in the *alarm* program one initial hidden outcome and two iteration hidden outcomes were added. After adding the hidden outcomes to the five programs, the outcome dependencies and the inter-iteration dependencies were changed.

Five sets of observations with each of the sizes 500, 1000, 1500 and 2000 were sampled from these programs. The algorithm was then run on these sets of observations each with its corresponding BK and biases. The scores of the learned programs and the original programs, the ratios of the learned programs to the original ones, the average ratios for each of the set sizes, the learning times and the scoring times are given in the tables 6.1, 6.2, 6.3, 6.4 and 6.5 for the four programs. It can be noticed that the learning time is considerably longer than the learning time of the algorithm designed to learn from fully observed outcomes. This is because, in this algorithm, the search uses BIC to score each candidate program (each state it moves to in the search). Therefore, most of the learning time is spent with the gEM used by the BIC score. For instance, in scoring some candidate *cervical cancer diagnosis* programs to which SA has moved in learning from 2000 observations, gEM took more than three minutes. It can also be noticed that there is a large difference in the learning times of programs from sets of observations with the same size. This is because different runs of the search move to different neighbourhoods. Some neighbourhoods contain programs which take more time to score than programs in other neighbourhoods. Figure 6.4 shows the average ratios of the five BIC scores of the learned programs to the corresponding original programs for the four set sizes for each program. Apart from the *cervical cancer diagnosis* program, the graphs show that the scores of the learned programs from the four sets of observations, on average, are very close to the scores of the original ones. For the *cervical cancer diagnosis* program, the scores for the four set sizes, on average, are higher than the scores of the original program. This is different from the behaviour of the curve of the learned programs from fully observed outcomes where the scores of the learned programs are less than the scores of the original one. This is because, after the addition of the hidden outcomes, the number of parameters increased; thus, a large number of observations is needed to better approximate the dependencies between the outcomes. This is noticed in the curve where the scores of the learned programs approach the scores of the original one as the number of observations increases.

Figure 6.5 shows a DBN modelling the *maintenance decision making* with hidden variables. Figure 6.7 shows the original *maintenance decision making* program which generalises the DBN in Figure 6.5, and a program learned from a set of 1000 observations and the BK and biases in Figure 6.6.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Small Language	500	1	**Learned**	−5524.18	0.995	52 sec	< 1 sec
			Original	−5549.2			< 1 sec
		2	**Learned**	−5356.73	0.995	40 sec	< 1 sec
			Original	−5382.84			< 1 sec
		3	**Learned**	−5203.02	0.995	38 sec	< 1 sec
			Original	−5227.94			< 1 sec
		4	**Learned**	−5657.52	0.995	38 sec	< 1 sec
			Original	−5684.13			< 1 sec
		5	**Learned**	−5545.21	0.995	40 sec	< 1 sec
			Original	−5570.4			< 1 sec
		Average ratio			0.995		
		Standard deviation			0		
	1000	1	**Learned**	−10618.4	0.997	1 min & 35 sec	< 1 sec
			Original	−10651.4			< 1 sec
		2	**Learned**	−10722.1	0.997	1 min & 7 sec	< 1 sec
			Original	−10751.2			< 1 sec
		3	**Learned**	−10845.6	0.997	1 min & 2 sec	< 1 sec
			Original	−10873.4			< 1 sec
		4	**Learned**	−9965.89	0.997	1 min & 15 sec	< 1 sec
			Original	−9994.4			< 1 sec
		5	**Learned**	−11417.9	0.997	59 sec	< 1 sec
			Original	−11446.7			< 1 sec
		Average ratio			0.997		
		Standard deviation			0		
	1500	1	**Learned**	−15862.6	0.998	2 min & 30 sec	< 1 sec
			Original	−15892.2			< 1 sec
		2	**Learned**	−15561.3	0.998	1 min & 45 sec	< 1 sec
			Original	−15591.8			< 1 sec
		3	**Learned**	−15976.2	0.998	1 min & 42 sec	< 1 sec
			Original	−16006.4			< 1 sec
		4	**Learned**	−16512.4	0.998	1 min & 27 sec	< 1 sec
			Original	−16542.3			< 1 sec
		5	**Learned**	−16150.2	0.998	1 min & 45 sec	< 1 sec
			Original	−16180.4			< 1 sec
		Average ratio			0.998		
		Standard deviation			0		
	2000	1	**Learned**	−21415.7	0.998	4 min & 41 sec	< 1 sec
			Original	−21448.1			< 1 sec
		2	**Learned**	−21790.6	0.998	2 min & 41 sec	< 1 sec
			Original	−21822.8			< 1 sec
		3	**Learned**	−20684.1	0.998	3 min & 32 sec	< 1 sec
			Original	−20715.1			< 1 sec
		4	**Learned**	−21496.9	0.998	3 min & 52 sec	< 1 sec
			Original	−21527.8			< 1 sec
		5	**Learned**	−21212.3	0.998	3 min & 20 sec	< 1 sec
			Original	−21244.5			< 1 sec
		Average ratio			0.998		
		Standard deviation			0		

Table 6.1: The result of the experiments on learning the *small language* program with hidden outcomes.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Asia	500	1	**Learned**	−2986.6	0.996	43 sec	< 1 sec
			Original	−2997.2			< 1 sec
		2	**Learned**	−3438.74	1.005	35 sec	< 1 sec
			Original	−3418.81			< 1 sec
		3	**Learned**	−3390.2	1.002	35 sec	< 1 sec
			Original	−3380.35			< 1 sec
		4	**Learned**	−3473.11	1.003	36 sec	< 1 sec
			Original	−3460.09			< 1 sec
		5	**Learned**	−3563.33	1.001	30 sec	< 1 sec
			Original	−3557.13			< 1 sec
		Average ratio			1.002		
		Standard deviation			0.003		
	1000	1	**Learned**	−5944.21	1.001	1 min & 8 sec	< 1 sec
			Original	−5934.5			< 1 sec
		2	**Learned**	−6470.16	1.006	1 min & 7 sec	< 1 sec
			Original	−6426.6			< 1 sec
		3	**Learned**	−6521.41	1.007	55 sec	< 1 sec
			Original	−6475.36			< 1 sec
		4	**Learned**	−6538.74	0.980	1 min & 8 sec	< 1 sec
			Original	−6667.21			< 1 sec
		5	**Learned**	−6512.47	1	56 sec	< 1 sec
			Original	−6507.44			< 1 sec
		Average ratio			0.999		
		Standard deviation			0.009		
	1500	1	**Learned**	−8759.48	1.004	1 min & 15 sec	< 1 sec
			Original	−8718.95			< 1 sec
		2	**Learned**	−9744.95	1.002	1 min & 14 sec	< 1 sec
			Original	−9723.34			< 1 sec
		3	**Learned**	−9859.62	1.001	1 min & 38 sec	< 1 sec
			Original	−9843.84			< 1 sec
		4	**Learned**	−9450.13	1	1 min & 8 sec	< 1 sec
			Original	−9445.44			< 1 sec
		5	**Learned**	−9554.81	1.007	1 min & 2 sec	< 1 sec
			Original	−9484.6			< 1 sec
		Average ratio			1.003		
		Standard deviation			0.002		
	2000	1	**Learned**	−11818.8	0.999	2 min & 8 sec	< 1 sec
			Original	−11820.5			< 1 sec
		2	**Learned**	−13460.5	1.002	1 min & 15 sec	< 1 sec
			Original	−13424.6			< 1 sec
		3	**Learned**	−13942.8	1.004	1 min & 28 sec	< 1 sec
			Original	−13877.9			< 1 sec
		4	**Learned**	−13738.0	1	1 min & 10 sec	< 1 sec
			Original	−13737.2			< 1 sec
		5	**Learned**	−13735.2	1.003	1 min & 36 sec	< 1 sec
			Original	−13691.0			< 1 sec
		Average ratio			1.002		
		Standard deviation			0.001		

Table 6.2: The result of the experiments on learning the *Asia* program with hidden outcomes.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Cervical	500	1	**Learned**	−18367.8	0.623	140 min & 34 sec	20 sec
			Original	29481.6			33 sec
		2	**Learned**	−17097.9	0.574	8 min & 55 sec	1 sec
			Original	−29736.9			38 sec
		3	**Learned**	−17226.1	0.586	61 min & 36 sec	14 sec
			Original	−29367.9			35 sec
		4	**Learned**	−18837.8	0.637	111 min & 41 sec	15 sec
			Original	−29545.7			50 sec
		5	**Learned**	−18289.9	0.611	159 min & 49 sec	20 sec
			Original	−29903.7			38 sec
		Average ratio			0.606		
		Standard deviation			0.023		
	1000	1	**Learned**	−29447.6	0.730	256 min & 48 sec	30 sec
			Original	−40311.8			1 min & 26 sec
		2	**Learned**	−28876.2	0.709	305 min & 14 sec	44 sec
			Original	−40682.9			1 min & 59
		3	**Learned**	−28719.8	0.719	156 min & 42 sec	15 sec
			Original	−39889.4			1 min & 39
		4	**Learned**	−28373.1	0.692	240 min & 46 sec	1 min & 6 sec
			Original	−40957.8			1 min & 36
		5	**Learned**	−29538.7	0.735	407 min & 44 sec	56 sec
			Original	−40160.3			1 min & 27
		Average ratio			0.717		
		Standard deviation			0.015		
	1500	1	**Learned**	−39597.3	0.792	456 min & 5 sec	57 sec
			Original	−49986.8			2 min & 50 sec
		2	**Learned**	−42355.8	0.840	294 min & 2 sec	17 sec
			Original	−50373.0			3 min & 51 sec
		3	**Learned**	−38192.4	0.763	252 min & 20 sec	19 sec
			Original	−50046.2			1 min & 53 sec
		4	**Learned**	−39484.1	0.783	232 min & 29 sec	21 sec
			Original	−50396.1			6 min & 41 sec
		5	**Learned**	−40170.2	0.783	690 min & 30 sec	1 min & 8 sec
			Original	−51298.6			2 min & 53 sec
		Average ratio			0.792		
		Standard deviation			0.025		
	2000	1	**Learned**	−49565.1	0.836	1199 min & 27 sec	1 min & 20 sec
			Original	−59270.0			6 min & 16 sec
		2	**Learned**	−45278.1	0.760	927 min & 20 sec	1 min & 59
			Original	−59566.3			4 min & 29 sec
		3	**Learned**	−52459.2	0.875	1749 min & 55 sec	1 min & 24
			Original	−59943.3			6 min & 24 sec
		4	**Learned**	−53083.2	0.891	1682 min & 11 sec	17 sec
			Original	−59542.9			5 min & 47
		5	**Learned**	−47922.5	0.797	1980 min & 12 sec	2 min & 18
			Original	−60111.6			7 min & 27
		Average ratio			0.832		
		Standard deviation			0.048		

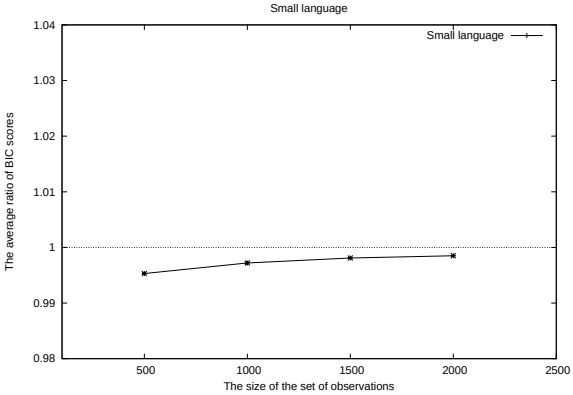
Table 6.3: The result of the experiments on learning the *cervical cancer diagnosis* program with hidden outcomes.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Maintenance	500	1	**Learned**	−11093.5	0.967	2 min & 49 sec	< 1 sec
			Original	−11471.6			< 1 sec
		2	**Learned**	−11113.2	0.980	1 min & 43 sec	< 1 sec
			Original	−11332.8			< 1 sec
		3	**Learned**	−11623.0	1.002	1 min & 51 sec	< 1 sec
			Original	−11595.9			< 1 sec
		4	**Learned**	−10828.4	0.923	1 min & 46 sec	< 1 sec
			Original	−11724.7			< 1 sec
		5	**Learned**	−10302.8	0.946	1 min & 12 sec	< 1 sec
			Original	−10882.2			< 1 sec
		Average ratio			0.964		
		Standard deviation			0.027		
	1000	1	**Learned**	−21177.8	1.005	6 min & 44 sec	< 1 sec
			Original	−21059.7			1 sec
		2	**Learned**	−21840.0	1.018	24 min & 21 sec	< 1 sec
			Original	−21446.3			1 sec
		3	**Learned**	−21047.3	0.987	10 min & 18 sec	< 1 sec
			Original	−21312.5			1 sec
		4	**Learned**	−21866.3	1.014	5 min & 12 sec	< 1 sec
			Original	−21559.0			1 sec
		5	**Learned**	−20876.3	0.979	5 min & 31 sec	< 1 sec
			Original	−21302.5			1 sec
		Average ratio			1.001		
		Standard deviation			0.014		
	1500	1	**Learned**	−30136.5	0.996	13 min & 43 sec	1 sec
			Original	−30237.6			3 sec
		2	**Learned**	−30382.8	1.008	10 min & 6 sec	2 sec
			Original	−30130.3			3 sec
		3	**Learned**	−31140.5	1.031	10 min	2 sec
			Original	−30175.1			3 sec
		4	**Learned**	−30297.3	0.999	8 min & 13 sec	2 sec
			Original	−30320.7			3 sec
		5	**Learned**	−30992.7	1.030	14 min & 4 sec	2 sec
			Original	−30061.8			3 sec
		Average ratio			1.013		
		Standard deviation			0.015		
	2000	1	**Learned**	−42208.7	1.027	27 min & 43 sec	16 sec
			Original	−41070.7			1 sec
		2	**Learned**	−42654.8	1.043	29 min & 12 sec	10 sec
			Original	−40875.7			1 sec
		3	**Learned**	−41543.0	1.018	24 min & 39 sec	6 sec
			Original	−40778.9			1 sec
		4	**Learned**	−39295.6	0.993	17 min & 20 sec	2 sec
			Original	−39563.7			1 sec
		5	**Learned**	−40951.7	1.002	28 min & 11 sec	5 sec
			Original	−40849.1			1 sec
		Average ratio			1.017		
		Standard deviation			0.017		

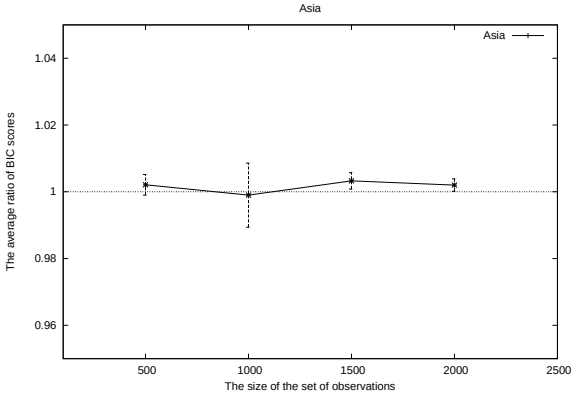
Table 6.4: The result of the experiments on learning the *maintenance decision making* program with hidden outcomes.

Program	Sample size	Sample	Program	Score	Ratio	Learning Time	Scoring Time
Alarm	500	1	**Learned**	−10288.8	1.031	6 min & 52 sec	< 1
			Original	−9977.9			< 1
		2	**Learned**	−10935.8	1.008	4 min & 26 sec	< 1
			Original	−10845.5			< 1
		3	**Learned**	−10992.1	1.016	2 min & 49 sec	< 1
			Original	−10809.6			< 1
		4	**Learned**	−11316.9	1.008	5 min & 43 sec	< 1
			Original	−11226.1			< 1
		5	**Learned**	−10750.5	1.010	2 min & 44 sec	< 1
			Original	−10643.8			< 1
		Average ratio			1.014		
		Standard deviation			0.008		
	1000	1	**Learned**	−20923.7	1.031	19 min & 12 sec	< 1 sec
			Original	−20277.5			1 sec
		2	**Learned**	−20505.3	1.013	22 min & 51 sec	16 sec
			Original	−20230.6			3 sec
		3	**Learned**	−20168.7	1.012	8 min & 30 sec	< 1 sec
			Original	−19927.4			1 sec
		4	**Learned**	−19891.5	1.006	5 min & 46 sec	< 1 sec
			Original	−19759.8			1 sec
		5	**Learned**	−20373.4	1.029	11 min & 20 sec	< 1 sec
			Original	−19783.3			1 sec
		Average ratio			1.018		
		Standard deviation			0.010		
	1500	1	**Learned**	−30851.0	1.009	52 min & 22 sec	16 sec
			Original	−30547.1			3 sec
		2	**Learned**	−27951.0	1.008	30 min & 8 sec	1 sec
			Original	−27714.5			4 sec
		3	**Learned**	−26918.4	0.999	14 min & 23 sec	6 sec
			Original	−26926.5			1 sec
		4	**Learned**	−27641.3	1.017	14 min & 31 sec	< 1 sec
			Original	−27166.8			3 sec
		5	**Learned**	−28812.0	1.026	31 min & 57 sec	26 sec
			Original	−28079.4			1 sec
		Average ratio			1.012		
		Standard deviation			0.008		
	2000	1	**Learned**	−41564.2	1.035	81 min & 11 sec	46 sec
			Original	−40158.3			3 sec
		2	**Learned**	−37269.7	1.002	37 min & 4 sec	11 sec
			Original	−37178.6			3 sec
		3	**Learned**	−37222.6	1.016	88 min & 37 sec	14 sec
			Original	−36610.5			6 sec
		4	**Learned**	−37737.0	0.997	19 min & 45 sec	< 1 sec
			Original	−37846.6			3 sec
		5	**Learned**	−37433.0	1.013	33 min & 36 sec	7 sec
			Original	−36945.4			8 sec
		Average ratio			1.012		
		Standard deviation			0.013		

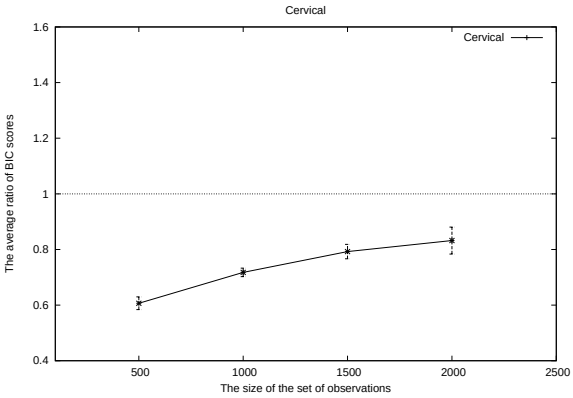
Table 6.5: The result of the experiments on learning the *alarm* program with hidden outcomes.



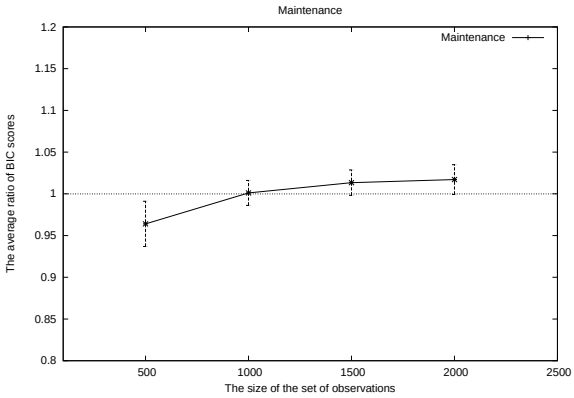
(a) The *small language* program.



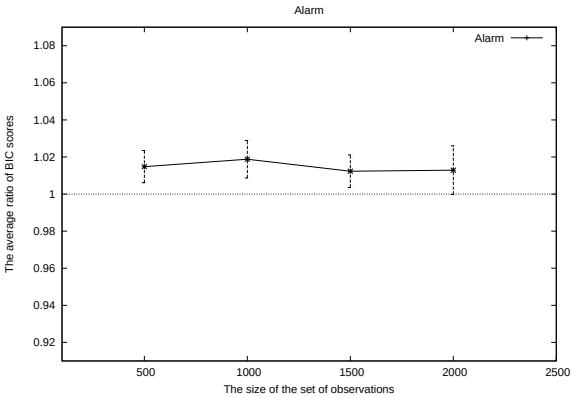
(b) The *Asia* program.



(c) The *cervical cancer diagnosis* program.



(d) The *maintenance decision making* program.



(e) The *alarm* program.

Figure 6.4: The ratios of the BIC scores of the learned programs with hidden outcomes to the original programs. Each point represents an average of 5 ratios of BIC scores of programs learned from 5 different and independent samples to the BIC scores of the original programs from which these samples were generated. The bars represent the standard deviations.

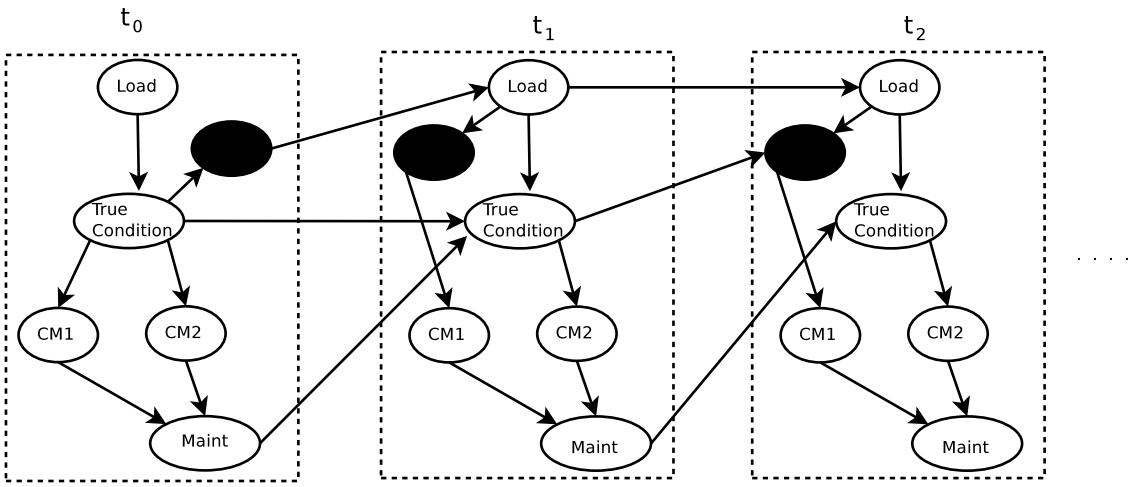


Figure 6.5: The *maintenance decision making* DBN with hidden outcomes.

```
values(load,[normal,abnormal]).
values(true_condition(_),[good,wear_1,wear_2,wear_3,failure_mode1,failure_mode2]).
values(cm1(_),[low1,medium1,high1]).
values(cm2(_),[low2,medium2,high2]).
values(maintenance(_,_),[none,reset,replace]).
values(hidden(_),[hidden1,hidden2,hidden3]).
values(load_t(_),[t_normal,t_abnormal,t_halt]).
values(true_condition_t(_,_,_),[t_good,t_wear_1,t_wear_2,t_wear_3,t_failure_mode1,
                                t_failure_mode2]).
values(cm1_t(_),[t_low1,t_medium1,t_high1]).
values(cm2_t(_),[t_low2,t_medium2,t_high2]).
values(maintenance_t(_,_),[t_none,t_reset,t_replace]).
% The outcomes are hidden
values(hidden1(_,_),[hidden_t1,hidden_t2]).

% The bias specifying the halt (HT)
stop:- msw(load_t(_),t_halt).

% The bias specifying hidden outcomes
% (HID)

% The initial hidden outcome
decision:-
    msw(hidden(_),H1),
    rec_def.

% The iteration hidden outcome
rec_def:-
    msw(hidden1(_,_),H2),
    rec_def.
```

Figure 6.6: The BK and biases used to learn the *maintenance decision making* PRISM programs with hidden outcomes.

Original:

```
values(load,[normal,abnormal]).
values(true_condition(_),
      [good,wear_1,wear_2,wear_3,
      failure_mode1,failure_mode2]).
values(cm1(_),[low1,medium1,high1]).
values(cm2(_),[low2,medium2,high2]).
values(maintenance(_,_),
       [none,reset,replace]).
values(hidden(_),[hidden1,hidden2,
                  hidden3]).
values(load_t(_),
       [t_normal,t_abnormal,t_halt]).
values(true_condition_t(_,_,_),
       [t_good,t_wear_1,t_wear_2,
        t_wear_3,t_failure_mode1,
        t_failure_mode2]).
values(cm1_t(_),
       [t_low1,t_medium1,t_high1]).
values(cm2_t(_),
       [t_low2,t_medium2,t_high2]).
values(maintenance_t(_,_),
       [t_none,t_reset,t_replace]).
values(hidden1(_,_),[hidden_t1,
                     hidden_t2]).
decision([L,TC,CM1,CM2,MAINT|Tail]):-
   msw(load,L),
   msw(true_condition(L),TC),
   msw(cm1(TC),CM1),
   msw(hidden(CM1),H1),
   msw(cm2(TC),CM2),
   msw(maintenance(CM1,CM2),MAINT),
   decision1(H1,TC,MAINT,Tail).
decision1(L,TC,MAINT,[L_t,TC_t,CM1_t,
               CM2_t,MAINT_t|Tail]):-
   msw(load_t(L),L_t),
   msw(true_condition_t(L_t,TC,MAINT),
                    TC_t),
   msw(hidden1(L_t,TC),H2),
   msw(cm1_t(H2),CM1_t),
   msw(cm2_t(TC_t),CM2_t),
   msw(maintenance_t(CM1_t,CM2_t),
                  MAINT_t),
   stop(L_t,TC_t,MAINT_t,Tail).
stop(t_halt,_,_,[]):- !.
stop(L_t,TC_t,MAINT_t,Tail):-
   decision1(L_t,TC_t,MAINT_t,Tail).
```

Learned:

```
values(load,[normal,abnormal]).
values(true_condition(_),
      [good,wear_1,wear_2,wear_3,
      failure_mode1,failure_mode2]).
values(cm1(_),[low1,medium1,high1]).
values(cm2(_),[low2,medium2,high2]).
values(maintenance(_,_),
       [none,reset,replace]).
values(hidden(_),[hidden1,hidden2,
                  hidden3]).
values(load_t(_),
       [t_normal,t_abnormal,t_halt]).
values(true_condition_t(_,_,_),
       [t_good,t_wear_1,t_wear_2,
        t_wear_3,t_failure_mode1,
        t_failure_mode2]).
values(cm1_t(_),
       [t_low1,t_medium1,t_high1]).
values(cm2_t(_),
       [t_low2,t_medium2,t_high2]).
values(maintenance_t(_,_),
       [t_none,t_reset,t_replace]).
values(hidden1(_,_),[hidden_t1,
                     hidden_t2]).
decision([A,B,C,D,E,F,G,H,I,J|K]):-
   msw(load,A),
   msw(hidden(A),L),
   msw(true_condition(L),B),
   msw(cm1(B),C),
   msw(cm2(L),D),
   msw(maintenance(C,D),E),
   rec_def(C,L,B,A,D,E,[F,G,H,I,J|K]).

rec_def(C,L,B,A,D,E,[F,G,H,I,J|K]):-
   msw(load_t(E),F),
   msw(hidden1(F,A),M),
   msw(true_condition_t(D,C,A),G),
   msw(cm1_t(B),H),
   msw(cm2_t(G),I),
   msw(maintenance_t(I,L),J),
   stop(F,I,G,M,J,H,K).

stop(t_halt,C,A,E,D,B,[]):-
   !.
stop(A,D,B,G,E,C,F):-
   rec_def(A,D,B,G,E,C,F).
```

Figure 6.7: The original *maintenance decision making* program with hidden outcomes from which 1000 observations were sampled, and the program which was learned from these observations and the BK given in Figure 6.6.

Chapter 7

Summary, Conclusion and Recommendations for Further Work

This chapter summarises the content of this thesis and highlights the main contributions. It also points out to different directions that can be followed to extend the main contributions.

7.1 Summary and Conclusion

This thesis proposed algorithms to learn recursive PRISM programs. The algorithms combine ideas from inductive logic programming and statistical inference for learning Bayesian networks. The programs learned generalise dynamic Bayesian networks by defining a halting distribution. By defining a halting distribution, these programs represent self-terminating functions. They provide generative processes from which a sequence of values can be sampled until a halting condition defined by some distribution is met. Meanwhile, dynamic Bayesian networks can be sampled by only determining a fixed length of sequences. Modelling self-terminating functions is useful. For example, machines such as stochastic context-free grammars use self-terminating functions to generate some symbols and halt.

The thesis started by providing the necessary background. It provided the syntax and semantics of logic programming. Bayesian network and dynamic Bayesian network were then explained. Hidden Markov models, an important subclass of dynamic Bayesian networks, were also described. The PRISM formalism was then introduced. First, the distribution semantics underlying the formalism was introduced. We then showed how probabilistic modelling with PRISM is performed. PRISM relies on four

conditions which need to be satisfied in modelling. These conditions were listed. We then explained the difference between failure and failure-free PRISM programs. We showed how recursive failure-free PRISM programs generalise dynamic Bayesian networks. The chapter closes by going through some learning utilities in the PRISM system. These utilities make it possible to learn the parameters of given programs and score candidate programs with different scoring metrics.

A review on the area of inductive logic programming was given. Learning from positive and negative examples and learning from only positive examples were discussed. We defined the three learning settings, learning from entailment, learning from interpretations and learning from satisfiability. The review concentrated on the setting of learning from entailment. The design of the search operations was discussed. We highlighted a number of learning approaches: top-down learning, bottom-up learning, random search, theory revision and predicate invention. Finally, we reviewed the work on learning recursive clauses. We described the system MERLIN 2.0 which uses theory revision to learn recursive logic programs. MERLIN 2.0 builds a deterministic finite state automaton of the clauses in the initial, overly generalised, theory used in the SLD-resolutions of the examples. It then generates the final program from this automaton.

The 'search and score' approach of learning Bayesian networks was reviewed. Scoring Bayesian networks was discussed in the case where the variables are fully observed and in the case where there are some hidden variables. We explained the decomposition of the log marginal likelihood score in the case where the variables are fully observed. Different approximations of the log marginal likelihood score were discussed. Amongst these approximations is the Bayesian information criterion (BIC). Four search strategies were highlighted. These are local search, K3 search, structural-EM search and the cutting planes method. The cutting planes method finds the Bayesian network with the optimal decomposed score given the scores of the families. Finally, a review of the work on learning dynamic Bayesian network was provided.

The thesis then provides the main contributions as follows

Learning recursive PRISM programs with fully observed outcomes: we proposed an algorithm to learn recursive PRISM programs when all the outcomes of the probabilistic atoms are observed. The algorithm is given background knowledge consisting of switch declarations and a bias defining the atom responsible for controlling the halt of the generative process. It then searches for

a PRISM program with the highest possible BIC score. The algorithm builds a bottom clause consisting of all possible dependencies amongst the probabilistic atoms. It then uses the cutting planes algorithm to find the set of atoms with the optimal log marginal likelihood. It splits these atoms into initial atoms and iteration atoms to define a recursive predicate. The idea used in MERLIN 2.0 of building a deterministic finite state automaton is adapted such that the automaton is built by the terms representing the switches that have generated the values in the observations. The recursive predicate definition is then learned based upon this automaton. Dependencies between the different iterations are learned based upon the statistics of the first and second iterations. The learning algorithm invents a predicate `stop/m` which defines how the process halts. Learning the definition of `stop/m` is based upon the given bias. To test the learning algorithm, five programs were built from which five samples of each of the sizes 500, 1000, 1500 and 2000 were generated. The learning algorithm was run on these samples and the corresponding background knowledge and bias, and the average of the ratios of the learned programs to the original ones was reported for each sample size. The results show that the scores of the learned programs, on average, approach the scores of the original programs as the number of observations increases.

Learning recursive PRISM programs with hidden outcomes: we proposed an algorithm to learn recursive PRISM programs when some of the outcomes are hidden. Search algorithms based on decomposable scores are not useful in this case. Splitting the atoms belonging to the body of the target predicate definition from those belonging to the body of the recursive predicate definition is based upon the values in the observations. This splitting approach cannot handle hidden outcomes. The non-decomposability problem was solved by applying the simulated annealing search over complete PRISM programs. The splitting problem was solved by providing a declarative bias specifying the atoms generating the hidden outcomes that may exist in the body of the target predicate definition and the atoms generating the hidden outcomes that may exist in the body of the recursive predicate definition. To test the learning algorithm, atoms generating hidden outcomes were added to the five programs used in the experiments of learning programs with fully observed outcomes. In the same way, from each program, five samples of each of the sizes 500, 1000, 1500 and 2000 were generated. The learning algorithm was then given the background knowledge of each program along with the biases and these samples to learn from.

The results show that the scores of four learned programs with the different sample sizes, on average, are close to the scores of the corresponding original programs. For one program, the scores of the learned programs for the four sample sizes were, on average, higher than the scores of the original program. However, the scores of the learned programs approach the scores of the original program as the size of the sample increases.

7.2 Recommendations for Further Work

Further directions that can be followed to build upon this work are as follows

Using more background knowledge: Algorithm 8 learns programs with fully observed outcomes by building bottom clauses. In large problems, the bottom clauses can be massive and require extensive computational resources. By providing more background knowledge and more declarative bias, the bottom clause can be considerably reduced and larger problems can be learned more efficiently. The proposed algorithm for learning programs with hidden outcomes does not build bottom clauses. However, when learning large programs, the search space of complete PRISM programs can become huge so that the number of iterations in the simulated annealing algorithm cover just a small proportion of it. Injecting more background knowledge and declarative bias is needed in learning large problems in both cases.

Learning PRISM programs with failure: PRISM programs with failure encode very complex probability distributions. Learning PRISM programs with failure requires considering more atoms than those considered by the proposed algorithms. Atoms which need to be considered are those which generate outcomes that unify with the outcomes generated by other atoms. Moreover, non-probabilistic atoms need also to be considered. By considering these atoms, the sought program is not bound as it may include an infinite number of atoms. Therefore, learning PRISM programs with failure can be carried out in conjunction with using more background knowledge and declarative bias.

Learning general recursive PRISM programs: the proposed algorithms learn tail recursive PRISM programs. A further direction can be taken to learn unrestricted recursive PRISM programs. Our preliminary notion for this task is in the light of the simulated annealing search. In simulated annealing, move

operations can be added to those explained in Section 6.3 which add recursive calls in different positions in the body. However, further investigation needs to be carried out.

Appendix A

Violating the Exclusiveness Condition in PRISM

Problems modelled in PRISM need to comply with the conditions given in Section 2.3.3. ProbLog drops the exclusiveness condition by representing the explanations of any goal as a *binary decision diagram* (BDD) (De Raedt et al., 2009; Akers, 1978). This makes ProbLog an ideal choice in modelling problems where the exclusiveness condition is not met. However, another choice to model these problems is to use PRISM with more steps in the modelling part. In this appendix, we define these steps and show a general procedure of transforming a PRISM program which violates the exclusiveness condition, and thus the correct value of the success probability cannot be computed, to a PRISM program from which the correct value of the success probability can be obtained. We first introduce a motivating example, the *path* problem, which is studied in the literature of ProbLog.

A.1 The Path Problem

In the *path* problem provided by De Raedt et al. (2009), which is shown in Figure A.1, the target predicate that we want to query is `path/2`. As shown in the graph, events of moving from one point to two different points are not mutually exclusive. For instance, $p(edge(a,b)) + p(edge(a,c)) > 1$. This leads to the explanations of any goal to be non-mutually exclusive as well. Figure A.2 shows the ProbLog program which defines the distribution over the possible paths between any two points in the graph (De Raedt et al., 2009). Figure A.3 shows the success probabilities of the two queries `path(a,c)` and `path(a,e)`. Figure A.4 shows the PRISM declarative version of this program. Querying the PRISM version about the success probabilities of the two queries `path(a,c)` and `path(a,e)` will lead to computing incorrect values. For

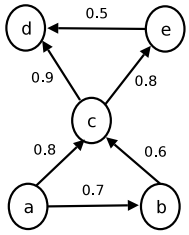


Figure A.1: The *path* problem.

```
0.8:: edge(a, c).       0.7:: edge(a, b).
0.8:: edge(c, e).       0.6:: edge(b, c).
0.9:: edge(c, d).       0.5:: edge(e, d).

path(X,Y):-
     edge(X,Y).
path(X,Y):-
     edge(X,Z),
     path(Z,Y).
```

Figure A.2: A ProbLog program which defines a distribution over possible paths between two points in a graph.

instance, PRISM will answer the query `prob(path(a,c))` with `1.22` which is not a probability value. This is because the two explanations $edge(a, c)$ and $edge(a, b) \wedge edge(b, c)$ are not mutually exclusive. Figure A.5 shows the output of PRISM for the probability computation queries of the two targets `path(a,c)` and `path(a,e)`. In the next section we explain a transformation procedure which takes a PRISM declarative version of a ProbLog program defining a distribution over non-mutually exclusive events and transforms it to another PRISM program which defines the distribution properly with some procedural predicates.

A.2 Transformation Procedure

Let $\mathcal{X}$ be the set of all random variables in a PRISM program. Let $\mathbf{X} \subseteq \mathcal{X}$ represent a set of random variables in a particular explanation of a successful goal and $\mathbf{x}$ be a the values that $\mathbf{X}$ is instantiated to in this explanation. Let $\mathbf{X}' = \mathcal{X} \setminus \mathbf{X}$, then the probability of this explanation is the marginal shown in (A.1).

$$P(\mathbf{X} = \mathbf{x}) = \sum_{\mathbf{x}'} P(\mathbf{X} = \mathbf{x}, \mathbf{X}' = \mathbf{x}') \tag{A.1}$$

```
  ?- problog_exact(path(a,c),Prob,S).
2 proofs
50 ms BDD processing
Prob = 0.884,
S = ok ?
yes
  ?- problog_exact(path(a,e),Prob,S).
2 proofs
17 ms BDD processing
Prob = 0.7072,
S = ok ?
yes
```

Figure A.3: The success probabilities of the two queries `path(a,c)` and `path(a,e)` in ProbLog.

Given a PRISM program which contains non-mutually exclusive explanations, the value of the above marginal is computed for each explanation and the sum of these marginals is given as the success probability of the query[1]. To show that this is not the right value, let $\mathbf{X}_1, \ldots, \mathbf{X}_n \subseteq \mathcal{X}$ be the sets of random variables in all the explanations of a certain query and let $\mathbf{X'}_1 = \mathcal{X} \setminus \mathbf{X}_1, \ldots, \mathbf{X'}_n = \mathcal{X} \setminus \mathbf{X}_n$. The value of the success probability that will be given by PRISM will be equivalent to the value computed in (A.2). It can be noticed that the probability of some joint instantiations are considered more than once leading to an improper answer to the `prob/1` query.

$$\begin{aligned} &\sum_{\mathbf{x'}_1} P(\mathbf{X}_1 = \mathbf{x}_1, \mathbf{X'}_1 = \mathbf{x'}_1) + \sum_{\mathbf{x'}_2} P(\mathbf{X}_2 = \mathbf{x}_2, \mathbf{X'}_2 = \mathbf{x'}_2) + \ldots \\ &+ \sum_{\mathbf{x'}_n} P(\mathbf{X}_n = \mathbf{x}_n, \mathbf{X'}_n = \mathbf{x'}_n) \end{aligned} \tag{A.2}$$

First, let $E_1 = \{X_{1,1} = x_{1,1}, X_{1,2} = x_{1,2}, \ldots, X_{1,k} = x_{1,k}\}, \ldots, E_n = \{X_{n,1} = x_{n,1}, X_{n,2} = x_{n,2}, \ldots, X_{n,m} = x_{n,m}\}$ be n explanations for a certain query. If we construct a union of the random variables in these explanations and consider all possible joint instantiations of them, each joint instantiation of the random variables

[1]PRISM does not compute marginals in the way represented in (A.1), it rather uses more efficient computational methods. A detailed discussion of the probability computation in PRISM can be found in Sato and Kameya (2001) and Kameya et al. (2004).

```
values(edge(_,_),[t,f]).

edge(a,b).      edge(b,c).
edge(a,c).      edge(c,e).
edge(c,d).      edge(e,d).

set_params:-
     set_sw(edge(a,b), [0.7,0.3]),    set_sw(edge(b,c), [0.6,0.4]),
     set_sw(edge(a,c), [0.8,0.2]),    set_sw(edge(c,e), [0.8,0.2]),
     set_sw(edge(c,d), [0.9,0.1]),    set_sw(edge(e,d), [0.5,0.5]).

path(X,Y):-
     edge(X,Y),  msw(edge(X,Y), t).

path(X,Y):-
     edge(X,Z),   msw(edge(X,Z), t), path(Z,Y).
```

Figure A.4: The PRISM version of the ProbLog program shown in Figure A.2.

will be considered once, and the double-counting will be resolved. However, some joint instantiations might not represent any explanation; therefore, we will consider only those joint instantiations of which at least one explanation E_i is a subset. In this case, PRISM will compute the correct probability of the success query whose value is equivalent to the value computed by (A.3).

$$P(goal) = \sum_{\forall x_{1,1},\ldots,x_{n,m}:\exists E \subseteq \{x_{1,1},\ldots,x_{n,m}\}} P(X_{1,1} = x_{1,1}, \ldots, X_{n,m} = x_{n,m}) \qquad \text{(A.3)}$$

We end up with the following transformation procedure:

i Define a predicate `findAllExplanations/`$\mathtt{n}_1$ such that it finds all explanations for any goal.

ii Define a predicate `constructUnion/`$\mathtt{n}_2$ such that it constructs the union of the random variables in the explanations found by `findAllExplanations/`$\mathtt{n}_1$.

iii Define a predicate `JointInstantiations/`$\mathtt{n}_3$ such that it finds all possible joint instantiations of the random variables in the set produced by `constructUnion/`$\mathtt{n}_2$. Each joint instantiation must represent at least one explanation.

```
| ?- prob(path(a,c)).

Probability of path(a,c) is: 1.220000000000000

yes
| ?- prob(path(a,e)).

Probability of path(a,e) is: 0.976000000000000

yes
```

Figure A.5: The incorrect values computed by the queries `prob(path(a,c))` and `prob(path(a,e))` in PRISM.

iv Define a predicate `findProb/`$\mathtt{n}_4$ such that it finds the sum of the probabilities of the joint instantiations found by `JointInstantiations/`$\mathtt{n}_3$. This gives the value represented by equation (A.3).

v Define a predicate `trans_target/`$\mathtt{n}_5$ using the predicates above in their respected order. It replaces the original target predicate to find the correct probability.

Figure A.6 shows the output of the transformation procedure applied to the PRISM program shown in Figure A.4. Figure A.7 shows the success probabilities of two queries `trans_path(a,c)` and `trans_path(a,e)` obtained from the transformed program. These probabilities are the same values computed from the ProbLog program in Figure A.3.

The probability of the success query is computed in the transformed program in the way given in (A.3) exactly. It can be noticed that this is inefficient. The number of joint instantiations of the random variables in the explanations of the success query grows exponentially in the number of the random variables. The sum of the joint instantiations is a sum of products and can be computed more efficiently than the way given in (A.3). The *variable elimination* algorithm (Zhang and Poole, 1994) utilises the factorisation of the joint distribution and reuses the sum of the products of the factors. The variable elimination algorithm can thus be added to the transformed program to improve its efficiency. Another way of computing the probability of the success query more efficiently is to follow the approach of ProbLog and represent the explanations found in the first step of the procedure above in a BDD. The probabilities can then be computed from this BDD. However, we keep the

transformation procedure as shown above, and we leave the problem of modifying it to improve the efficiency of the transformed program open.

```
in_graph(edge(a, b)).  in_graph(edge(b, c)).
in_graph(edge(a, c)).  in_graph(edge(c, e)).
in_graph(edge(c, d)).  in_graph(edge(e, d)).

set_params:-
        set_sw(edge(a, b), [0.7, 0.3]),
        set_sw(edge(b, c), [0.6, 0.4]),
        set_sw(edge(a, c), [0.8, 0.2]),
        set_sw(edge(c, e), [0.8, 0.2]),
        set_sw(edge(c, d), [0.9, 0.1]),
        set_sw(edge(e, d), [0.5, 0.5]).

trans_path(X,Y):-
        findAllExplanations(X,Y,[],All),
        constructUnion(All,Union), get_values(Union,Vals),
        jointInstant(Union,Vals,All,[],[],Joint), findProb(Joint), !.

findAllExplanations(X,Y, PartialExps, AllExps):-
        path(X,Y,PartialExps,[],Exp),
        findAllExplanations(X,Y,[Exp|PartialExps],AllExps).

findAllExplanations(_,_, AllExps, AllExps).

path(X,Y, KnownExps, PartialExp, Exp):-
        edge(X,Y),  Exp=[msw(edge(X,Y),t)|PartialExp],
        \+ isExplanation(Exp,KnownExps).

path(X,Y,KnownExps,PartialExp,Exp):-
        edge(X,Z),  Temp=[msw(edge(X,Z),t)|PartialExp],
        path(Z,Y,KnownExps,Temp,Exp).

constructUnion([H], H).
constructUnion([H, H1|Tail],U):-
        union(H, H1,Temp),
        constructUnion([Temp|Tail],U).

obtain_values([],[]).
obtain_values([msw(Head,_)|Tail],Values):-
        obtain_values(Tail,Temp),
        get_values1(Head,V),     % Built-in in PRISM to get the
        Values = [V|Temp].       % outcomes of the switch
```

```
jointIns([],_,Exps,Expl,PartialAll,[Expl|PartialAll]):-
        isExplanation(Expl,Exps), !.
jointIns([],_,_,_,PartialAll,PartialAll).
jointIns([_|_],[[]|_],_,_,PartialAll,PartialAll).
jointIns([msw(S, V)|Rest],[[Value|OtherV]|Tail],Exps,PExp,PAll,All):-
        TempExpl = [msw(S, Value)| PExp],
        jointIns(Rest,Tail,Exps,TempExpl,PAll,TempAll),
        jointIns([msw(S,V)|Rest],[OtherV|Tail],Exps,PExp,TempAll,All).

findProb([Model]):-
        explanationProb(Model),!.
findProb([Model|Rest]):-
        explanationProb(Model);  findProb(Rest).

explanationProb([]).
explanationProb([msw(S,V)|Tail]):-
        msw(S,V),
        explanationProb(Tail).

isExplanation(World,[Path]):-
        subset(Path,World),!.
isExplanation(World,[Path1,Path2|Rest]):-
        (subset(Path1,World),!);  isExplanation(World,[Path2|Rest]).
```

Figure A.6: The output of the transformation procedure applied to the PRISM program shown in Figure A.4.

```
 | ?- prob(trans_path(a,c)).

Probability of trans_path(a,c) is: 0.884000000000000

yes
| ?- prob(trans_path(a,e)).

Probability of trans_path(a,e) is: 0.707200000000000
```

Figure A.7: Probabilities computed from the program transformed from the one in Figure A.4. The probabilities are computed correctly.

Bibliography

Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, TU Berlin, 2007.

Hilde Adé, Bart Malfait, and Luc De Raedt. RUTH: an ILP theory revision system Methodologies for Intelligent Systems. *Methodologies for Intelligent Systems*, 869: 336–345, 1994.

Agnieszka, Marek J. Druzdzel, and R. Marshall Austin. Application of dynamic Bayesian networks to cervical cancer screening. In *Proceedings on XI International Conference on Artificial Intelligence: (Al-24'2009)*, pages 5–14, Siedlce, Poland, 2009. Publishing House of the University of Podlasie.

David W. Aha, Stephane Lapointe, Charles X. Ling, and Stan Matwin. Learning Recursive Relations with Randomly Selected Small Training Sets. In William W. Cohen and Haym Hirsh, editors, *Proceedings of the Eleventh International Conference of Machine Learning*, pages 12–18. Morgan Kaufmann, 1994.

Sheldon B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27 (6):509–516, 1978.

Hagai Attias. Inferring Parameters and Structure of Latent Variable Models by Variational Bayes. In *Proceedings of the Proceedings of the Fifteenth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 21–30, San Francisco, CA, 1999. Morgan Kaufmann.

Hagai Attias. A Variational Bayesian Framework for Graphical Models. In *Advances in Neural Information Processing Systems*, volume 12, pages 209–215. MIT Press, 2000.

Kwok-Chung Au and Kwok-Wai Cheung. Learning Hidden Markov Model Topology Based on KL Divergence for Information Extraction. In *Advances in Knowledge Discovery and Data Mining: 8th Pacific-Asia Conference*, Lecture Notes in Computer Science, pages 590–594. Springer, 2004.

Matthew J. Beal and Zoubin Ghahramani. The variational Bayesian EM algorithm for incomplete data: with application to scoring graphical model structures. In *Bayesian Statistics*, volume 7, pages 453–464. Oxford University Press, 2003.

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, New York, 2006.

Henrik Boström. Predicate invention and learning from positive examples only. In Claire Nédellec and Céline Rouveirol, editors, *Proceedings of the 10th European Conference on Machine Learning (ECML-98)*, volume 1398 of *Lecture Notes in Computer Science*, chapter 29, pages 226–237. Springer-Verlag, Berlin/Heidelberg, 1998.

Remco R. Bouckaert. Probabilistic network construction using the minimum description length principle. In Michael Clarke, Rudolf Kruse, and Serafín Moral, editors, *Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, volume 747 of *Lecture Notes in Computer Science*, chapter 6, pages 41–48. Springer Berlin, Berlin/Heidelberg, 1993.

Wray Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, September 1988.

Irene Cantone, Lucia Marucci, Francesco Iorio, Maria A. Ricci, Vincenzo Belcastro, Mukesh Bansal, Stefania Santini, Mario di Bernardo, Diego di Bernardo, and Maria P. Cosma. A Yeast Synthetic Network for In Vivo Assessment of Reverse-Engineering and Modeling Approaches. *Cell*, 137(1):172–181, April 2009.

Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics.

David M. Chickering. Learning Bayesian Networks is NP-Complete. In *Learning from Data: Artificial Intelligence and Statistics V*. Springer-Verlag, 1996.

David M. Chickering and David Heckerman. Efficient Approximations for the Marginal Likelihood of Incomplete Data Given a Bayesian Network. In *Proceedings of the Twelfth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-96)*, pages 158–168, San Francisco, CA, 1996. Morgan Kaufmann.

David M. Chickering, David Heckerman, and Christopher Meek. Large-Sample Learning of Bayesian Networks is NP-Hard. *Journal of Machine Learning Research*, 5: 1287–1330, December 2004.

William W. Cohen. Pac-Learning a Restricted Class of Recursive Logic Programs. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 86–92. Morgan Kaufman, 1993.

Robert G. Cowell. Efficient maximum likelihood pedigree reconstruction. *Theoretical Population Biology*, 76(4):285–291, December 2009.

Robert G. Cowell, A. Philip Dawid, Steffen L. Lauritzen, and David J. Spiegelhalter. *Probabilistic Networks and Expert Systems: Exact Computational Methods for Bayesian Networks (Information Science and Statistics)*. Springer, New York, 2007.

James Cussens. Parameter Estimation in Stochastic Logic Programs. *Machine Learning*, 44(3):245–271, September 2001.

James Cussens. Bayesian network learning with cutting planes. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI-11)*, pages 153–160, Corvallis, Oregon, 2011. AUAI Press.

James Cussens. Online Bayesian inference for the parameters of PRISM programs. *Machine Learning*, pages 1–19, July 2012.

James Cussens and Sašo Džeroski, editors. *Learning language in logic*. Springer-Verlag, New York, USA, 2000.

James Cussens, David Page, Stephen Muggleton, and Ashwin Srinivasan. Using Inductive Logic Programming for Natural Language Processing. In W. Daelemans, A. Van den Bosch, and A. Weijters, editors, *Workshop Notes of the ECML / MLnet Workshop on Empirical Learning of Natural Language Processing Tasks*, pages 25–34, April 1997.

Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 1 edition, April 2009.

Luc De Raedt. Logical settings for concept-learning. *Artificial Intelligence*, 95:187–201, August 1997.

Luc De Raedt. *Logical and Relational Learning*. Springer, 1 edition, October 2008.

Luc De Raedt and Luc Dehaspe. Learning from satisfiability. In *Proceedings of the Ninth Dutch Conference on Artificial Intelligence (NAIC'97)*, pages 303–312, November 1997a.

Luc De Raedt and Luc Dehaspe. Clausal Discovery. *Machine Learning*, 26(2-3): 99–146, March 1997b.

Luc De Raedt and Wim Van Laer. Inductive Constraint Logic. In *Proceedings of the 6th International Workshop on Algorithmic Learning Theory*, Lecture Notes in Computer Science, pages 80–94, London, UK, UK, 1995. Springer-Verlag.

Luc De Raedt, Angelika Kimmig, Bernd Gutmann, Kristian Kersting, Vitor Santos Costa, and Hannu Toivonen. Probabilistic inductive querying using ProbLog. Technical Report CW 552, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, June 2009.

Arthur P. Dempster, Nan M. Laird, and Donald B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

DLMF. NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/, Release 1.0.5 of 2012-10-01. Online companion to (Olver et al., 2010).

Pedro Domingos and Matthew Richardson. Markov Logic: A Unifying Framework for Statistical Relationl Learning. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*, pages 339–371. The MIT press, 2007.

David Draper. Assessment and Propagation of Model Uncertainty. *Journal of the Royal Statistical Society*, 57(1):45–97, 1995.

Johann Dréo, Patrick Siarry, Alain Pétrowski, and Eric Taillard. *Metaheuristics for Hard Optimization*. Springer-Verlag, Berlin, 2006.

Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, April 1998.

Sašo Džeroski. Inductive Logic Programming in a Nutshell. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*, pages 57–92. The MIT press, 2007.

Peter Flach. *Simply Logical: Intelligent Reasoning by Example*. Wiley, 1 edition, January 1994.

Peter Flach and Nada Lavrač. Learning in Clausal Logic: A Perspective on Inductive Logic Programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond (Essays in Honour of Robert A. Kowalski)*, volume LNAI 2407, pages 437–471. Springer-Verlag, June 2002.

Daniel C. Fredouille, Christopher H. Bryant, Channa K. Jayawickre, Steven Jupe, and Simon Topp. An ILP Refinement Operator for Biological Grammar Learning. In Stephen Muggleton, Ramon Otero, and Alireza T. Nezhad, editors, *Inductive Logic Programming*, pages 214–228. Springer-Verlag, Berlin, 2007.

Nir Friedman. Learning Belief Networks in the Presence of Missing Values and Hidden Variables. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML)*, pages 125–133. Morgan Kaufmann, 1997.

Nir Friedman. The Bayesian Structural EM Algorithm. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, pages 129–138, San Francisco, CA, USA, 1998. Morgan Kaufmann.

Nir Friedman, Kevin Murphy, and Stuart Russell. Learning the Structure of Dynamic Probabilistic Networks. In *Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 139–147. Morgan Kaufmann, 1998.

Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning Probabilistic Relational Models. In *International Joint Conference on Artificial Intelligence*. Springer-Verlag, 1999.

Mitsue Furusawa, Nobuhiro Inuzuka, Hirohisa Seki, and Hidenori Itoh. Bottom-Up Induction of Logic Programs With More Than One Recursive Clause. In *Proceedings of the IJCAI-97 Workshop on Frontiers of ILP*, 1997.

Zoubin Ghahramani and Matthew J. Beal. Variational Inference for Bayesian Mixtures of Factor Analysers. In *Advances in Neural Information Processing Systems*, volume 12, pages 449–455. MIT Press, 2000.

Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. Parameter Learning in Probabilistic Databases: A Least Squares Approach. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, Lecture Notes in Computer Science, pages 473–488. Springer, 2008.

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

David Heckerman, Christopher Meek, and Daphne Koller. Probabilistic Models for Relational Data. Technical Report MSR-TR-2004-30, Microsoft Research, March 2004.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2 edition, November 2000.

Peter Idestam-Almquist. Efficient Induction of Recursive Definitions by Structural Analysis of Saturations. In Luc De Raedt, editor, *Advances in Inductive Logic Programming*, pages 192–205. IOS Press, 1996.

Tommi Jaakkola, David Sontag, Amir Globerson, and Marina Meila. Learning Bayesian network structure using lp relaxations. *Journal of Machine Learning Research - Proceedings Track*, 9:358–365, 2010.

Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Information Science and Statistics. Springer Verlag, 2nd edition, June 2007.

Yoshitaka Kameya, Taisuke Sato, and Neng-Fa Zhou. Yet More Efficient EM Learning for Parameterized Logic Programs by Inter-Goal Sharing. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, pages 490–494, 2004.

Kristian Kersting. *An Inductive Logic Programming Approach to Statistical Relational Learning*, volume 148 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, October 2006.

Kristian Kersting and Luc De Raedt. Bayesian Logic Programming: Theory and Tool. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*, pages 291–321. The MIT press, 2007.

Ross D. King, Kenneth E. Whelan, Ffion M. Jones, Philip G. K. Reiser, Christopher H. Bryant, Stephen H. Muggleton, Douglas B. Kell, and Stephen G. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427(6971):247–252, January 2004.

Scott Kirkpatrick, C. Daniel Gelatt, and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

Karl-Rudolf Koch. *Introduction to Bayesian Statistics*. Springer, 2nd, updated and enlarged ed. edition, September 2007.

Mikko Koivisto and Kismat Sood. Exact Bayesian Structure Discovery in Bayesian Networks. *Journal of Machine Learning Research*, 5:549–573, December 2004.

Daphne Koller, Nir Friedman, Lise Getoor, and Ben Taskar. Graphical Models in a Nutshell. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*, pages 13–55. The MIT press, 2007.

Robert Kowalski. Predicate Logic as Programming Language. In *Proceedings of the IFIP Congress*, pages 569–574. North Holland Publishing Company, 1974.

Robert Kowalski and Donald Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2(3-4):227–260, December 1971.

Kenichi Kurihara and Taisuke Sato. An application of the variational Bayesian approach to probabilistic context-free grammars. In *International Joint Conference on Natural Language Processing Workshop Beyond Shallow Analyses*, 2004.

Kenichi Kurihara and Taisuke Sato. Variational Bayesian Grammar Induction for Natural Language. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, *Proceedings of the 8th International Colloquium on Grammatical Inference*, volume 4201, chapter 8, pages 84–96. Springer, Berlin, Heidelberg, 2006.

Nada Lavrač and Sašo Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, 1994.

Christopher D. Manning and Hinrich Schüetze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1 edition, June 1999.

David Maxwell Chickering and David Heckerman. Efficient Approximations for the MarginalLikelihood of Bayesian Networks with Hidden Variables. *Mach. Learn.*, 29(2-3):181–212, November 1997.

John McCarthy. Programs with Common Sense. In *Teddington Conference on the Mechanization of Thought Processes*, December 1959.

John McCarthy. Epistemological Problems of Artificial Intelligence. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

John McCarthy. Circumscription -A form of non-monotonic reasoning. *Artificial Intelligence*, 13(1-2):27–39, April 1980.

K.R. McNaught and A. Zagorecki. Using dynamic Bayesian networks for prognostic modelling to inform maintenance decision making. In *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM 2009)*, pages 1155 –1159, dec. 2009. doi: 10.1109/IEEM.2009.5372973.

Raymond J. Mooney. Inductive Logic Programming for Natural Language Processing. In Stephen Muggleton, editor, *Inductive Logic Programming: Selected papers from the Sixth International Workshop*, pages 3–22. Springer Verlag, Berlin, 1996.

Raymond J. Mooney, Prem Melville, Lappoon R. Tang, Jude Shavlik, Inês de Castro Dutra, David Page, and Vítor S. Costa. Relational Data Mining with Inductive Logic Programming for Link Discovery. In *Proceedings of the National Science Foundation Workshop on Next Generation Data Mining*, Baltimore, MD, November 2002.

Steve Moyle. Using theory completion to learn a robot navigation control program. In *Proceedings of the 12th international conference on Inductive Logic Programming*, pages 182–197, Berlin, 2003. Springer-Verlag.

Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8: 295–318, February 1991.

Stephen Muggleton. Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

Stephen Muggleton. Learning from positive data. In Stephen Muggleton, editor, *Inductive Logic Programming*, volume 1314, chapter 21, pages 358–376. Springer, Berlin, Heidelberg, 1997.

Stephen Muggleton. Learning stochastic logic programs. In Lise Getoor and David Jensen, editors, *Proceedings of the AAAI 2000, workshop on Learning Statistical Models from Relational Data*, 2000.

Stephen Muggleton and Luc De Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19(20):629–679, 1994.

Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pages 368–381, 1990.

Stephen Muggleton and Alireza Tamaddoni-Nezhad. QG/GA: a stochastic search for Progol. *Machine Learning*, 70(2):121–133, March 2008.

Kevin Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, July 2002.

Kevin Murphy and Saira Mian. Modelling Gene Expression Data using Dynamic Bayesian Networks. Technical report, Computer Science Division, University of California, Berkeley, CA, 1999.

Raymond Ng and V. S. Subrahmanian. Probabilistic Logic Programming. *Information and Computation*, 101:150–201, December 1992.

Nils J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28:71–88, February 1986.

Ulf Nilsson and Jan Małuszyński. *Logic, Programming and Prolog*. John Wiley & Sons Inc, 2 sub edition, 1995.

F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, editors. *NIST Handbook of Mathematical Functions*. Cambridge University Press, New York, NY, 2010. Print companion to (DLMF).

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1 edition, September 1988.

Judea Pearl, Dan Geiger, and Tom Verma. Conditional independence and its representations. *Kybernetika*, 25(7):33–44, 1989.

Gordon D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

David Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.

David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2, Special issue on economic principles of multi-agent systems):7–56, July 1997.

Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 77, pages 257–286, 1989.

Lawrence R. Rabiner and Biing-Hwang Juang. An introduction to hidden Markov models. *IEEE ASSP Magazine*, 3(1):4–16, 1986.

John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

Joshua W. Robinson and Alexander J. Hartemink. Learning Non-Stationary Dynamic Bayesian Networks. *Journal of Machine Learning Research*, 11:3647–3680, December 2010.

R. W. Robinson. Counting labeled acyclic digraphs. In F. Harary, editor, *New Directions in Graph Theory*. New York: Academic Press, 1973.

Taisuke Sato. A Statistical Learning Method for Logic Programs with Distribution Semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP'95)*, pages 715–729. MIT Press, 1995.

Taisuke Sato. Generative Modeling by PRISM. In Patricia Hill and David Warren, editors, *Logic Programming*, volume 5649 of *Lecture Notes in Computer Science*, chapter 4, pages 24–35. Springer, Berlin, Heidelberg, 2009.

Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1330–1339, 1997.

Taisuke Sato and Yoshitaka Kameya. Parameter Learning of Logic Programs for Symbolic-Statistical Modeling. *Journal of Artificial Intelligence Research (JAIR)*, 15:391–454, 2001.

Taisuke Sato and Yoshitaka Kameya. New advances in logic-based probabilistic modeling by PRISM. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen H. Muggleton, editors, *Probabilistic Inductive Logic Programming*, volume 4911 of *Lecture Notes in Computer Science*. Springer, New York, 2008.

Taisuke Sato, Yoshitaka Kameya, and Kenichi Kurihara. Variational Bayes via propositionalized probability computation in PRISM. *Annals of Mathematics and Artificial Intelligence*, 54(1-3):135–158, 2008.

Taisuke Sato, Neng-Fa Zhou, Yoshitaka Kameya, and Yusuke Izumi. *PRISM User's Manual (Version 2.0)*, 2010.

Gideon Schwarz. Estimating the Dimension of a Model. *The Annals of Statistics*, 6 (2):461–464, 1978.

Eran Segal, Michael Shapira, Aviv Regev, Dana Pe'er, David Botstein, Daphne Koller, and Nir Friedman. Module networks: identifying regulatory modules and their condition-specific regulators from gene expression data. *Nature Genetics*, 34 (2):166–176, May 2003.

Mathieu Serrurier and Henri Prade. Improving inductive logic programming by using simulated annealing. *Information Sciences*, 178(6):1423–1441, March 2008.

Kristie Seymore, Andrew McCallum, and Ronald Rosenfeld. Learning Hidden Markov Model Structure for Information Extraction. In *AAAI 99 Workshop on Machine Learning for Information Extraction*, pages 37–42, 1999.

Tomi Silander and Petri Myllymäki. A simple approach for finding the globally optimal Bayesian network structure. In *Proceedings of the 22nd Annual Conference on Uncertainty in Artificial Intelligence*. AUAI Press, 2006.

Ashwin Srinivasan. A study of two probabilistic methods for searching large spaces with ILP. Technical Report PRG-TR-16-00, Oxford University Computing Laboratory, 2000.

Andreas Stolcke and Stephen Omohundro. Best-first model merging for hidden Markov model induction. Technical Report TR-94-003, International Computer Science Institute, Berkeley, CA, January 1994.

Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. International Series in Operations Research & Management Science. Springer, 3rd edition, November 2007.

Nguyen X. Vinh, Madhu Chetty, Ross Coppel, and Pramod P. Wangikar. GlobalMIT: learning globally optimal dynamic Bayesian network with the mutual information test criterion. *Bioinformatics*, 27(19):2765–2766, August 2011.

Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer Texts in Statistics. Springer, December 2003.

Kyoung J. Won, Thomas Hamelryck, Adam P. Bennett, and Anders Krogh. An evolutionary method for learning HMM structure: prediction of protein secondary structure. *BMC Bioinformatics*, 8(1):357+, September 2007.

Stefan Wrobel. Inductive Logic Programming for Knowledge Discovery in Databases. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*. Springer, Berlin, 2001.

Nevin Zhang and David Poole. A simple approach to Bayesian network computations. In *Proceedings of the Tenth Canadian Conference on Artificial Intelligence*, 1994.