KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Probabilistic Inference for Dynamic and Relational Models

Jonas Vlasselaer

Supervisor:
Prof. dr. Luc De Raedt

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2016

Probabilistic Inference for Dynamic and Relational Models

Jonas VLASSELAER

Examination committee:
Prof. dr. ir. Hugo Hens, chair
Prof. dr. Luc De Raedt, supervisor
Prof. dr. ir. Hendrik Blockeel
Dr. ir. Wannes Meert
Dr. Bart De Ketelaere

Prof. dr. ir. Guy Van den Broeck
(University of California, Los Angeles, USA)
Prof. dr. Oliver Schulte
(Simon Fraser University, Canada)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2016

Abstract

Within the field of *Artificial Intelligence*, there is a lot of interest in combining probability and expressive representations for dealing with complex relational and dynamic domains. A common approach to reason with these representations is to rely on existing techniques for propositional models and thus requires to first *ground* the underlying model into a propositional representation. This strategy comes at a cost, however, as capturing the semantics of the original representation might lead to a combinatorial explosion and quickly renders inference intractable. This dissertation investigates how *weighted model counting* and *knowledge compilation* can be used to directly perform inference on the original representation.

This thesis has three main contributions. First, we propose an exact probabilistic inference algorithm for propositional dynamic domains. Our approach allows to exploit different types of structures by compiling the transition model into an efficient circuit representation. Second, we propose an anytime probabilistic inference algorithm for relational domains. An efficient circuit representation is compiled in an incremental way and, at any time in the process, hard bounds on the inferred probabilities can be computed. Third, we deal with relational dynamic domains by combining principles from the first and second contribution. In addition, our approach exploits the given observations to further scale-up inference.

The techniques presented in this dissertation are evaluated empirically on various real-world domains and applications such as biological and social network analysis, web-page classification, electronic circuit diagnosis and game playing. They outperform state-of-the-art approaches on these problems with respect to time, space and quality of results.

Beknopte samenvatting

Binnen het onderzoeksgebied van *Kunstmatige Intelligentie* is er een grote interesse om probabilistische informatie en expressieve representaties te combineren om zo complexe relationele en dynamische domeinen te kunnen modelleren. Het redeneren met deze representaties wordt typisch gedaan door gebruik te maken van bestaande technieken voor propositionele modellen, en hiervoor is het noodzakelijk het onderliggende model eerst om te zetten naar een propositionele representatie. Deze aanpak brengt echter een zekere kost met zich mee aangezien de omzetting kan leiden tot een combinatorische ontploffing en zo inferentie onmogelijk wordt. Dit proefschrift onderzoekt hoe we inferentie rechtstreeks op de originele representatie kunnen uitvoeren door gebruik te maken van *kennis compilatie* en het *optellen van gewogen modellen*.

Dit proefschrift heeft drie belangrijke bijdrages. Ten eerste stellen we een exact probabilistisch inferentie algoritme voor propositionele dynamische domeinen voor. Onze aanpak benut verschillende structuren in het netwerk door het overgangsmodel te compileren naar een efficiënte circuit representatie. Ten tweede stellen we een probabilistisch inferentie algoritme voor relationele domeinen voor. Een efficiënte circuit representatie wordt gecompileerd op een incrementele manier en, op elk moment van het proces, kunnen gegarandeerde grenzen voor de probabiliteiten berekend worden. Ten derde beschouwen we inferentie voor relationele dynamische domeinen en combineren hiervoor principes van de eerste en tweede bijdrage. Daarbovenop laten we de inferentie nog beter schalen door de gegeven observaties te benutten.

De technieken beschreven in dit proefschrift worden empirisch geëvalueerd op verschillende reële domeinen en toepassingen zoals het analyseren van biologische en sociale netwerken, het classificeren van web pagina's, diagnose van elektronische circuits en het spelen van spellen. Ze overtreffen de bestaande technieken op deze problemen voor zowel tijd, geheugen als kwaliteit van de resultaten.

Acknowledgments

Pursuing a PhD has been an unpredictable and challenging experience but, at the same time, it was also very exciting and rewarding. Over the past years, I had the chance to discuss and collaborate with some of the most influential researchers from our field and I want to thank everyone who directly or indirectly influenced this dissertation.

First of all, I want to thank my supervisor Luc De Raedt for giving me the opportunity to start a PhD. Luc gave me the freedom to apply for a personal grant, to explore different research directions and to pursue the topics I thought were most interesting. Of course, we did not always and agree and I must admit I was often rather skeptical about his ideas and suggestions. Fortunately, Luc is known for being very patient and he simply kept repeating his point of view until I could prove he was wrong. In most cases, it turned out I could not. One thing I certainly learned from Luc is that nobody expects you to solve all problems with one algorithm or in one paper.

I want to thank all members of my jury for the interesting discussions and feedback on earlier versions of this text. Special thanks go to Hugo Hens for chairing the jury, to Oliver Schulte and Guy Van den Broeck for attending the preliminary defense (despite the early hour) and for traveling to Leuven to attend the public defense, to Wannes Meert for reading and commenting on the text while being on holiday and to Hendrik Blockeel and Bart De Ketelaere for being part of my supervisory committee and keeping me on track throughout my PhD.

Wannes and Guy are much more than only members of my jury. They continuously challenged me with interesting questions and research ideas and helped me wherever they could. I was also lucky to have them as a co-author for almost all of my papers and especially appreciated their devotion to convert every single character or symbol into the correct font or to work out all details of a figure. As an example, Figure 4.4 originates from a paper we submitted to

a workshop and was designed by Guy, early in the morning, while he was on a holiday. Furthermore, I especially want to thank Wannes for guiding me trough the first year(s) of my PhD. Without his help, I would certainly not have made it till the end.

A couple of years ago, after obtaining my Master of Industrial Sciences degree, I came across the Master of Artificial Intelligence offered at the KU Leuven and this is when I met Danny De Schreye. His enthusiasm convinced me to start this program and this indirectly led to this dissertations. I also want to thank the IWT (agency for Innovation by Science and Technology) for their financial support trough a personal doctoral scholarship as well as the POM2 project (Prognostics for Optimal Maintenance).

During the past years, I had the pleasure of sharing an office with Angelika, Anton, Dimitar, Dries, Irma, Joris, Mathias, Matthijs, Pedro and Theo and enjoyed many coffee breaks and discussions with them. Anton was always available, even after moving offices, to answer any of my Python or ProbLog related questions. I also regularly talked to Jan VH, although we never shared an office, and could always rely on him for administrative related questions.

Earlier this year, I had the unique opportunity to visit Guy and the Automated Reasoning Group, led by Adnan Darwiche, at the University of California, Los Angeles. For almost three months, I shared an office with Arthur, Jessa and Umut and had a very interesting and enjoyable time. I really want to thank all of them for their hospitality and hope we will meet each other again.

Last, but certainly not least, I would like to thank Jolina for her patience and understanding. She did not only have to deal with my ever changing mood, when paper deadlines got missed or experiments did not run as expected, but also had to accept my absence when I suddenly decided it would be a good idea to visit UCLA and move to LA for almost three months. Bedankt!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Artificial Intelligence

Not even that long ago, searching for a suitable route between two locations involved staring at a road map for several minutes. Nowadays, we simply turn on our GPS device or smart-phone and, after filling in the desired destination, it only takes seconds before we get prompted with the shortest route, the fastest route, a route without speed-ways, etc. The most critical component of these devices is, arguably, the search algorithm that underlies the system. Despite all recent developments in computational power, exhaustively exploring all possible routes between two given locations is generally infeasible. Instead, the algorithm should act *intelligently* in that it avoids considering routes which are certainly known to be too long. In a sense, the algorithm has to perform some *human like reasoning* and has to take into account that any reasonable route will be centered around a straight line connecting the two locations of interest.

We are gradually evolving to a world were it is common to rely on intelligent systems to assist us with every-day tasks. Recent developments in the automotive industry, e.g. the driver assistant offered by most brands, is only one of the applications that supports this claim. These systems should act intelligently in that they have to deal with an ever changing environment and reasoning about the past, present and future is needed to properly anticipate certain events. Imagine a situation where the sensors of an autonomous vehicle detect a ball-shaped object in front of a driving car and, at the same time, a kid is detected on the sidewalk. Human like reasoning would advice the car to slow down, as there is a non zero probability the kid will cross the street to pick-up the ball.

The development of intelligent algorithms, where human-like reasoning is combined with computational power, is one of the main incentives of *Artificial Intelligence* (AI). While there are many definitions of what exactly AI is, one of the most widely accepted interpretations is that *artificial intelligence is the study of agents that act intelligently.* Furthermore, an agent is said to *act intelligently* if it does the *right thing*, given what it knows (Poole and Mackworth 2010; Russell and Norvig 2009). The broad area of artificial intelligence accommodates different subfields and application domains, including; computer vision, speech recognition, medical diagnosis, planning, game playing, robot tracking, etc. Although it has drawn on many research methodologies, AI research arguably builds on two formal foundations: *probability* and *logic*. This dissertation is situated exactly on the intersection of these two fields.

1.2 Logic and Reasoning

The field of logic emerged thousands of years ago, way before AI or computers were born, and has been studied ever since. Nowadays, logic is the standard formalism for knowledge representation and is considered to be one of the cornerstones of AI. Furthermore, logic has played a prominent role in the development of *automated reasoning*, that is, the ability to infer new knowledge in an automatic way. In general, one distinguishes between *propositional logic* and *first-order logic*.

Propositional logic makes use of propositional variables to express knowledge about single properties of the world. For example, propositional logic allows us to express that; *If it rains, the grass must be wet.* We can represent this knowledge with only the two propositional variables *rain* and *wet*, and both of these variables can be either *True* or *False*. Then, an agent presented with this knowledge that observes it is raining only requires simple reasoning to infer that the grass will be wet.

First-order logic makes use of logical variables to express knowledge about objects in the world. For example, first-order logic allows us to express that; *If a person* Y *has a smart-phone, then* Y *must be popular.* In this case, Y is a logical variable and our knowledge (potentially) deals with all people in the world, i.e. we can replace Y by any person we want. Then, an agent presented with this knowledge that observes a person named *john* has a smart-phone only requires simple reasoning to infer that *john* will be popular.

Logical variables can be interpreted as placeholders for more specific entities, making first-order logic an expressive formalism to represent complex knowledge in a compact and structured way. Logic in general, however, is mostly limited to

deterministic knowledge where propositions are either true or false. It does not easily allow one to express non-deterministic dependencies where the amount of non-determinism is precisely quantified. For example, logic cannot be used to correctly express the knowledge that *Only 80 % of bird species can fly.*

1.3 Probability and Uncertainty

Where logic deals with deterministic dependencies, *probability theory* is concerned with non-deterministic events or random phenomena. Similar to logic, probability theory emerged hundreds of years ago and is by now embraced as one of the cornerstones of AI. Intuitively, probabilities indicate our degree of belief in the outcome of a non-deterministic event and reasoning with probabilities allows us to properly estimate the outcome of a sequence of "random" actions. In this dissertation, we adopt the notion of probability theory as laid out by Kolmogorov (1933).

Consider the following simple experiment. We request an agent to throw five different dice. For every die showing less than six pips, the agent should move one step to the left, otherwise it should move to the right. Now, simple (probabilistic) reasoning suffices to conclude the absolute displacement of the agent is most likely towards left. Computing the likelihood of its exact new position, however, requires some more advanced reasoning.

In general, probability theory allows one to capture the *uncertainty* that is inherently present in the environment. Uncertainty might be caused by noisy sensor measurements, hidden knowledge, unpredictable actions, etc. Therefore, an agent cannot simply act pretending that it knows what is true and, ideally, accommodates for this by modeling its uncertainty. Probability theory on its own, however, does not provide us a framework to reason about objects and relations amongst individuals. Therefore, it makes sense to combine logical representations and probabilities, leading to the field of *Statistical Relational Artificial Intelligence* (De Raedt, Kersting, et al. 2016).

1.4 Problem Statement and Motivation

It is an ever lasting dream of many AI researchers to build expressive models that allow us to enhance the quality of our every-day life. For example, combining the knowledge of all doctors in the world to build a model that can be queried to quickly and accurately diagnose the disease of a patient. Or autonomous agents that reflect human-like reasoning to accompany elderly and help them

where needed. The typical models required in this context, ideally include three different types of knowledge or information:
(1) probabilities to deal with uncertainty
(2) relations to deal with objects in the environment
(3) dynamics to deal with time related information

Within the last decade, researchers from the field of Statistical Relational AI realized the need for combining these different types of knowledge, leading to the realization of various formalisms (e.g. Manfredotti (2009), Nitti et al. (2014), and Thon et al. (2011)). As these representations allow one to express complex knowledge and dependencies, the task of probabilistic reasoning in the underlying model is computationally very challenging. Therefore, one often has to rely on approximate inference techniques that only provide estimates, or one puts a restriction on the models that can be expressed by the representation.

We now use some examples from the domain of medical diagnosis to further illustrate the models and concepts we will consider throughout this dissertation.

Probabilistic Models

One of the problems a doctor deals with on a daily basis is that of medical diagnosis, i.e. he has to determine which disease a patient might have, based on the observable symptoms. Additionally, the doctor might perform some medical tests to confirm the presence of a certain disease. The task of diagnosis is quite challenging as many of the symptoms are nonspecific and can have different causes. For example, symptoms such as a headache or fever might have been caused by dozens of diseases. Furthermore, medical tests are not always reliable and the results might lead to wrong conclusions. Hence, in the process of diagnosis, a doctor will not solely rely on the direct observations but additionally tries to incorporate the uncertainty.

Modeling the task of medical diagnosis as an AI problem requires one to capture the connection between diseases, symptoms and medical tests. For example, if a disease is known to cause a certain symptom or a positive test, the model should contain a dependency that describes this knowledge. Not all of the available information is deterministic and, ideally, our model should include this uncertainty. For example, to diagnose whether a patient has HIV, we should be able to include the following information:

Four weeks after infection, 5 out of 100 HIV tests return a false negative.
HIV causes a headache only in 33 out of 100 cases.

The first *rule* states that an HIV test is not completely trustworthy if it is taken four weeks after infection. The second rule states that only one out of three

patients with HIV suffer from a headache caused by this disease.

Where logical inference suffices to draw conclusions from a deterministic model, dealing with uncertainty and probabilities requires us to rely on *probabilistic reasoning* techniques. For the above example, probabilistic reasoning will not return a deterministic answer but, instead, compute the degree of belief that a patient has HIV based on the observed symptoms and test-results. In case our model includes several diseases, probabilistic reasoning allows one to compute which disease the patient most likely suffers from. Intuitively, probabilistic inference is harder than logical inference as it requires to take into account the probabilities.

Probabilistic graphical models is one of the formalisms that allows us to combine propositional knowledge and probabilities. Probabilistic reasoning in these models is well-studied and gave rise to many inference methods. One state-of-the-art approach, known as *weighted model counting*, has shown to outperform other techniques by exploiting structure in the model. Probabilistic inference techniques based on weighted model counting is the general theme of this dissertation. We will deal with more complex models, however, containing relational and/or dynamic (time-related) dependencies.

Relational Models

Medical diagnosis often benefits from taking into account information about the environment of the patient. For hereditary diseases, it is relevant to know whether one of the relatives is known to have the same disease. For contagious diseases, it might be helpful to additionally know whether one of the close friends of the patient has the disease. Hence, a doctor will combine this knowledge together with the observed symptoms and medical tests to better estimate the disease a patient might suffer from.

As a doctor takes into account knowledge about the relatives, friends and colleagues of the patient, also our AI model should contain this information. Therefore, we might want to include the following rules:

For all people X*, the probability that* X *has the flu is 0.001.*
For all people X *and* Y*, if* X *and* Y *are friends and* Y *has the flu, the probability that* X *has the flu is 0.1.*

The first rule states there is only a small chance that a patient has the flu. The probability of having the flu, which is known to be a contagious disease, increases once it is known that one of the friends of the patients has the flu, as stated by the second rule.

Where propositional models would require us to explicitly write down all our knowledge for each of the patients, *relational* models allow us to make abstraction of the specific entities. The above rules makes use of logical variables, denoted as X and Y, that act as a placeholder for all of the people we potentially care about. Furthermore, our second rule describes some complex knowledge in a compact way. To know whether the person Y has the flu, we can apply the same rule again and, as such, we do not only consider the direct friends of person X but also his indirect friends.

While relational representations allow us to express our knowledge in a rather compact way, the underlying model is often very complex. Firstly, we have to deal with large domains, e.g. all patients of a hospital. Secondly, our knowledge might introduce cyclic dependencies in the model. For example, a person X can be friends with a person Y who in turn is a friend of a person Z and Z is again friends with X. Effectively dealing with these cyclic dependencies has shown to be extremely challenging and is one of the issues we address in this dissertation.

Dynamic Models

In medical diagnosis it is common to take into account the medical history of a patient. This allows a doctor to, for example, use his previous experience with a patient to more quickly recognize certain symptoms or to specifically adapt a new treatment. Furthermore, a doctor might want to combine information from the past and the present to predict how a patient should evolve over time. This allows one to more quickly intervene if the patient would diverge from the expected behavior. Hence, a doctor might use information from each of its appointments with the patient in order to make better decisions.

To reason about the past, present and future we have to extend our AI model to cope with time-related information. For example, a model that describes potential organ failures after a surgery (Sandri et al. 2014) might contain the following information:

For any day T *following a surgery, the probability that the patient has a lung failure is 0.001.*
For any day T *following a surgery, if the patient has a heart failure at day* T*, the probability of having a lung failure at day* $\mathsf{T}+1$ *is 0.1.*

The first rule states there is only a small probability of having a lung failure after having surgery. But, as stated by the second rule, the probability of having a lung failure after having a surgery increased if the patient had a heart failure the day before.

Where static models typically only deal with information from one specific moment in time, *dynamic models* allow us to deal with sequences of information and time-related dependencies. In other words, dynamic models include information from the past to better estimate the present or to predict the future. Similar to relational models that make abstraction of specific entities, dynamic models make abstraction of the specific time. The rules we present above do not refer to any specific day and range over all days following a surgery. Hence, dynamic models typically act as a template for (potentially) infinite moments in time.

Dynamic models often range over a large time span, e.g. the complete medical history of a patient, and dealing with all this knowledge is computationally hard. Under certain assumptions, however, it is known that it suffices to maintain a *belief state* which summarizes all information from the past. Then, probabilistic reasoning involves updating this belief state in the presence of new observations. Efficiently representing and updating this belief state are two of the issues we address in this dissertation.

1.5 Contributions

Research on knowledge representation and automated reasoning has resulted in different inference tasks for propositional logic. One of these tasks, known as weighted model counting, serves as an assembly language for inference in probabilistic models. In other words, one can translate a probabilistic model into a weighted propositional logical formula, or knowledge base, after which inference in the original model corresponds to computing the weighted model count on the formula. Intuitively, weighted model counting enumerates the weights of all models that satisfy the weighted formula. One of the key-steps of the reduction approach is to actually encode the probabilistic model as a formula in propositional logic.

The encoding of a propositional model, e.g. probabilistic graphical models, into a propositional knowledge base is well-studied and different approaches have been proposed in the literature. On the other hand, the encoding of an *expressive probabilistic model*, including template models for dynamic domains and logical models for relational domains, typically requires to first *unfold* the model into a propositional representation. This unfolding can be with regard to time (dynamic models) or with respect to cyclic dependencies (relational models). Then, an existing encoding for propositional models can be applied. An overview of such a sequential pipeline is depicted in Figure 1.1.

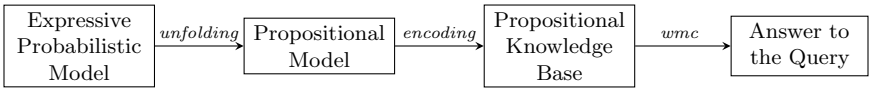


Figure 1.1: A sequential pipeline for probabilistic inference.

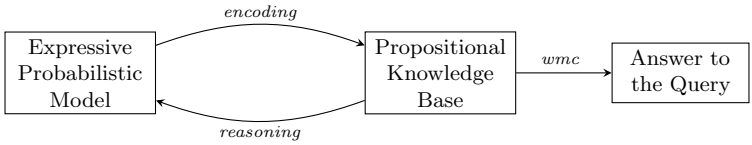


Figure 1.2: An incremental pipeline for probabilistic inference.

Unfolding of an expressive probabilistic model into a propositional model is generally not desired, as capturing the characteristics and semantics of the original model often leads to a blow-up of the corresponding knowledge base and quickly renders probabilistic inference intractable. Therefore, the main question we address in this dissertation is the following:

> *Can we exploit the characteristics of an expressive probabilistic model to further scale-up probabilistic inference by weighted model counting?*

Intuitively, we want to obtain an encoding that avoids the need to explicitly unfold the model. Instead, it should act on the original representation and, if possible, exploit its characteristics. In the ideal case, we would then obtain an incremental pipeline as depicted in Figure 1.2, where reasoning on the original model is interleaved with the encoding step.

The techniques we will present throughout this dissertation all use the incremental strategy, as depicted in Figure 1.2, in contrast to many of the existing techniques that rely on a sequential strategy. Important to note is that, at the end, we perform inference on a propositional knowledge base. Hence, our work is not situated in the field of lifted inference. We can identify three main contributions:

The **first contribution** is the **Structural Interface Algorithm**, an exact inference algorithm for propositional dynamic models. Reasoning in probabilistic models is generally known to be computationally hard. Exponential speed gains can be obtained, however, by exploiting independences between variables and *local structure* in the model. Local structure might arise in the form of deterministic dependencies between variables or by using equal probabilities

when modeling uncertainty. Furthermore, specific purpose algorithms for dynamic models are known to benefit from exploiting the *repeated structure* in the model that arises by duplicating the template model along the time dimension. In this dissertation, we show how inference techniques based on weighted model counting and knowledge compilation allow us to exploit local as well as repeated structure in the underlying model.

The **second contribution** is **$T_\mathcal{P}$-compilation**, an anytime inference algorithm for relational models. Many of the typical applications for relational methods, including social networks and linked web-pages, introduce cyclic dependencies in the underlying model. Inherently, many of the existing inference approaches do not support these cyclic dependencies and require to explicitly break down cycles by means of a preprocessing step. This might lead to an exponential blow-up of the model and probabilistic inference quickly becomes intractable. In this dissertation, we propose a novel inference technique with built-in support for cyclic dependencies, making exact inference for relational models much more efficient. Furthermore, our approach can be stopped anytime and provides a hard bound for the computed probabilities.

The **third contribution** is **dynamic $T_\mathcal{P}$-compilation**, an exact inference algorithm for relational dynamic models. Specific purpose inference algorithms for dynamic models maintain a belief state that represents the current belief about the possible states, given all observations and information up to that point. In the presence of new observations, probabilistic inference includes updating this belief state. These observations can lead to additional structure, in the form of deterministic dependencies, allowing us to represent the belief state in a more compact way. In this dissertation, we show how we can exploit the structure in the belief state to further boost inference in dynamic models.

1.6 Structure of the Thesis

The remainder of this text consists of 5 chapters.

Chapter 2 gives the necessary background on probability theory, propositional representations and relational representations. This includes logical reasoning by knowledge compilation and probabilistic reasoning by weighted model counting.

Chapter 3 introduces the *structural interface algorithm*. It unifies state-of-the-art techniques for inference in static and dynamic models to exploit the repeated nature of a dynamic model as well as the local structure. We experimentally

show how our technique can tackle models that are considerably more complex than what can currently be dealt with by exact inference techniques. This chapter is based on the following publication:

J. Vlasselaer, W. Meert, G. Van den Broeck, and L. De Raedt (2016a). "Exploiting local and repeated structure in dynamic Bayesian networks". In: *Artificial Intelligence* 232, pp. 43–53

Chapter 4 introduces the *$T_\mathcal{P}$-compilation algorithm.* It interleaves the knowledge compilation step for weighted model counting with forward reasoning on the logical representation and avoids the need to explicitly unfold cyclic dependencies. An experimental evaluation demonstrates that the new technique outperforms existing exact and approximate techniques on real-world applications such as biological and social networks and web-page classification. This chapter is based on the following two publications :

J. Vlasselaer, J. Renkens, G. Van den Broeck, and L. De Raedt (2014). "Compiling probabilistic logic programs into sentential decision diagrams". In: *Proceedings of the Workshop on Probabilistic Logic Programming (PLP)*

J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2015). "Anytime inference in probabilistic logic programs with Tp-compilation". In: *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*

Chapter 5 introduces the *dynamic $T_\mathcal{P}$-compilation algorithm.* It combines the structural interface algorithm and $T_\mathcal{P}$-compilation to deal with dynamic relational domains. Furthermore, we show how additional structure, introduced by observation, can be exploited to further scale-up inference. This chapter is based on the following publication:

J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2016b). "Tp-Compilation for inference in probabilistic logic programs". In: *International Journal of Approximate Reasoning* 78, pp. 15–32

Chapter 6 concludes this text with a discussion on the techniques presented throughout this dissertation. Furthermore, we propose some interesting directions for future work.

Chapter 2

Background

This chapter introduces the representations and methods on which we will build further in the remainder of this text. The different formalisms can be separated along two dimensions (see Table 2.1). Along the first dimension we have logical representations, which only deal with purely deterministic dependencies, versus probabilistic representations, which allow us to include probabilities to model non-deterministic events. Along the second dimension we have propositional representations, where we can only express knowledge about single entities, versus relational representations, which allow us to express knowledge about more abstract object in the world.

	Logical	**Probabilistic**
Propositional	Propositional logic	Probabilistic graphical models, *Bayesian* and *Markov networks*
Relational	First-order logic, logic programs	Probabilistic first-order logic, *probabilistic logic programs*

Table 2.1: An overview of the different formalisms used throughout this text.

We start by reviewing standard notions of probability theory in Section 2.1. Section 2.2 lays out the foundations on propositional representations and reasoning. This includes logical reasoning by knowledge compilation and probabilistic reasoning by weighted model counting. Section 2.3 introduces relational representations, including first-order logic and (probabilistic) logic programming.

2.1 Probability Theory

An agent that acts under uncertainty is essentially gambling on its outcome and probability theory is the calculus of gambling (Poole and Mackworth 2010). Probability theory allows us to precisely quantify uncertainty and is nowadays an inherent part of many research in AI.

2.1.1 Random Variables

Probability theory is the branch of mathematics that deals with *experiments* that are subject to random phenomena or chance. An experiment is said to be *non-deterministic* if it has more than one possible outcome, and *deterministic* if it has only one outcome. The *sample space* of an experiment is the set of all its possible outcomes. An *event* is any subset of the sample space, including the empty set and the sample space itself.

Probability theory is centered around *random variables*, that is, variables whose value is subject to randomness. The *domain* of a random variable is the set of possible different values it can take. A *binary* random variable has a domain of exactly two values. Typically, the possible values of a variable represent the possible outcomes of an experiment. Throughout this dissertation, we will only consider *discrete* random variables with a finite domain.

Example 2.1 Consider the task of rolling a fair die. We could use the random variable *die* to describe this experiment. The sample space of the experiment, as well as the domain of the variable, is $\{1, 2, 3, 4, 5, 6\}$. A possible event would be *"More than 2 pips"* for which we have $\{3, 4, 5, 6\}$ as subset of the sample space.

As convention, we use lower case letters (e.g. y) to denote random variables and upper-case letters (e.g. Y) to denote their instantiation or value assignment. Bold letters represent sets of variables (e.g. $\mathbf{y}$) and their instantiations (e.g. $\mathbf{Y}$).

2.1.2 Probability Distributions

Each of the values in the domain of a random variable comes with an associated *probability*. Intuitively, probabilities are a measure of our belief that a certain event will occur. By convention, probabilities are denoted by means of a number between 0 and 1 and, the higher the probability of an event, the more certain

we are that it will occur. A probability of 0 indicates impossibility, i.e. the event will never occur, and 1 indicates certainty, i.e. it will always occur.

A random variable induces a *probability distribution* on its domain. We use $\Pr(y)$ to denote the probability distribution of the random variable y on all values Y^i in its domain and $\Pr(y = Y^i)$ denotes the probability that variable y takes value Y^i. The latter is often shortened to $Pr(Y^i)$. It is common to denote a probability distribution by means of a table. Intuitively, each of the rows in the table correspond to a *possible world*, i.e. the world will be in one of these states after performing the experiment. By convention, the probability of all possible worlds sums op to 1.

die	$\Pr(die)$
1	1/6
2	1/6
3	1/6
4	1/6
5	1/6
6	1/6
	1

Table 2.2: The probability distribution for the experiment from example 2.2.

Example 2.2 Continuing with our die example, the probability that the random variable *die* takes one of the variables from the domain $\{1, 2, 3, 4, 5, 6\}$ is 1/6. The complete distribution is denoted by Table 2.2. Computing the probability for an event comes down to enumerating the probability of each of the values in the sample space that coincide with the event. For the event *"More than 2 pips"*, we have:

$$\begin{aligned}\Pr(die > 2) =& \Pr(die = 3) + \Pr(die = 4) + \Pr(die = 5) + \Pr(die = 6)\\ =& 1/6 + 1/6 + 1/6 + 1/6 = 4/6\end{aligned}$$

Joint Probability Distribution For more complex experiments it is often cumbersome to describe all possible outcomes by only one variable and, instead, one uses a set of different variables. A *joint probability distribution* $\Pr(y^1, \ldots, y^n)$ expresses the distribution of the random variables $y^1, \ldots, y^n$ on all values $Y^{1,1}, \ldots, Y^{n,j}$. Implicitly, a joint distribution combines random variables by means of a conjunction and denotes a probability to all possible outcomes of an experiment.

Example 2.3 Consider the task of tossing two fair coins where we use variable *coin1* to represent the first coin and *coin2* to represent the second coin. The joint probability distribution is given by Table 2.3, where we use H and T to denote heads and tail, respectively. Again, computing the probability for an event comes down to enumerating the probability of each of the possible worlds that coincide with the event. For the event *"At least one head"*, we would have:

$$\begin{aligned}\Pr(coin1 = H \vee coin2 = H) = &\Pr(coin1 = H, coin2 = H) \\ &+ \Pr(coin1 = H, coin2 = T) \\ &+ \Pr(coin1 = T, coin2 = H) \\ = &\ 1/4 + 1/4 + 1/4 = 3/4\end{aligned}$$

Marginal Probability Distribution Given a joint probability distribution $\Pr(y^1, \ldots, y^n)$, the probability distribution of any one of the random variables y^i can be obtained by summing over all values of the other variables. Concretely, the *marginal probability distribution* of the random variable y^i is obtained as:

$$\Pr(y^i) = \sum_{y^1,\ldots,y^{i-1},y^{i+1},\ldots,y^n} \Pr(y^1, \ldots, y^n)$$

Example 2.4 Continuing with our coin example, given the joint probability distribution for *coin1* and *coin2*, we can compute the marginal probability for *coin1* being heads (and similarly for *coin1* being tails) as:

$$\begin{aligned}\Pr(coin1 = H) = &\sum_{coin2} \Pr(coin1 = H, coin2) \\ = &\Pr(coin1 = H, coin2 = H) \\ &+ \Pr(coin1 = H, coin2 = T) \\ = &\ 1/4 + 1/4 = 1/2\end{aligned}$$

coin1	*coin2*	$\Pr(coin1, coin2)$
H	H	1/4
H	T	1/4
T	H	1/4
T	T	1/4
		1

Table 2.3: The probability distribution for the experiment from example 2.3.

Conditional Probability Given a joint probability distribution $\Pr(y^1, y^2)$, the *conditional probability distribution* of y^2 given y^1 is denoted as $\Pr(y^2|y^1)$ and is given by:

$$\Pr(y^2|y^1) = \frac{\Pr(y^1, y^2)}{\Pr(y^1)}$$

The conditional probability expresses the probability that the random variable y^2 takes a certain value given that the value of y^1 is known and requires that $\Pr(y^1) \neq 0$. If we know the value for a random variable, we say it is *observed* or it is *evidence*. Note that conditional probabilities can also be defined in terms of more than two random variables.

The notion of conditional probability is one of the most important concepts in probability theory. It allows us to express the belief that a certain event will occur, given that another event has occurred. Concretely, we can use conditional probabilities to express $\Pr(effect|cause)$, i.e. the belief that a *cause* has a certain *effect*. In health diagnosis, for example, this would become $\Pr(symptom|disease)$ or $\Pr(testResult|disease)$.

Example 2.5 The probability of a random person having fever is rather low. Knowing that a person has the flu, however, increases its probability of having fever. We can now simply express this knowledge as:

$$\Pr(fever) = 0.05$$

$$\Pr(fever|flu) = 0.4$$

Bayes' Rule Where conditional probabilities allow us to represent our belief in the effect of a cause, reasoning often goes in the opposite direction. Based on observed effects, e.g. symptoms, one is interested in computing the probability that a certain cause actually caused these effects. Concretely, we typically

want to compute $\Pr(cause|effect)$. This type of reasoning can be obtained by exploiting the symmetric behavior of conditional probabilities. We can write conditional probabilities, as defined above, in the following way:

$$\Pr(y^2|y^1)\Pr(y^1) = \Pr(y^2, y^1) = \Pr(y^1|y^2)\Pr(y^2)$$

from which follows that:

$$\Pr(y^2|y^1) = \frac{\Pr(y^1|y^2)\Pr(y^2)}{\Pr(y^1)}$$

The above equation is known as the *rule of Bayes* and defines a conditional distribution in terms of another conditional distribution, rather than a joint distribution. This has shown to be extremely useful, and underlies most modern AI systems for probabilistic inference.

Example 2.6 Continuing with our flu example, given additional knowledge that $\Pr(flu) = 0.0001$, we can now use Bayes' rule to compute the probability that a patient has the flu, after observing he has fever, in the following way :

$$\Pr(flu|fever) = \frac{\Pr(fever|flu)\Pr(flu)}{\Pr(fever)} = \frac{0.4 \cdot 0.0001}{0.05} = 0.0008$$

Hence, after observing the patient has fever, the posterior belief of having the flu is increased compared to the prior belief.

2.1.3 Independence

The notion of independence between random variables is, besides conditional probabilities, one of the most important concepts in probability theory. Firstly, independence allows one to reduce the amount of information necessary to specify the joint distribution. Secondly, one can exploit independences to perform probability computations in a more efficient way. In practice, we distinguish different types of independence. We now shortly introduce absolute and conditional independence and deal with context-specific independence later on (see Section 2.2.4).

Absolute or Marginal Independence Two random variables are said to be *absolute* or *marginally independent* if the occurrence of one does not effect the probability of the other. In other words, two random variables y^1 and y^2 are absolute independent if their joint probability equals the product of their probabilities, given as:

$$\Pr(y^1, y^2) = \Pr(y^1) \cdot \Pr(y^2)$$

or:

$$\Pr(y^2|y^1) = \Pr(y^2)$$

Example 2.7 Tossing two fair coins is an example of two absolute independent events. Observing whether the first coin results in heads or tails does not give any information about the second coin. Hence we can write:

$$\Pr(\mathit{coin1}, \mathit{coin2}) = \Pr(\mathit{coin1}) \cdot \Pr(\mathit{coin2})$$

Conditional Independence Two random variables are *conditionally independent* given a third variable if information of the latter variable makes the former two variables absolute independent. In other words, two random variables y^1 and y^2 are conditional independent given y^3, if and only if given the value of y^3, the occurrence of y^1 or y^2 does not influence the probability of y^2 or y^1, respectively. We can write this as:

$$\Pr(y^1, y^2|y^3) = \Pr(y^1|y^3) \cdot \Pr(y^2|y^3)$$

or:

$$\Pr(y^2|y^1, y^3) = \Pr(y^2|y^3)$$

Example 2.8 Lets continue with our flu example, but now with headache as a second symptom. In case we do not know whether our patient has the flu, fever and headache depend on each other. Indeed, observing one of the symptoms will increase the probability of having the flu, hence increasing the probability of also having the other symptom. Knowing that a patient has the flu, however, breaks this dependency and the two symptoms become independent, allowing us to write:

$$\Pr(fever, headache|flu) = \Pr(fever|flu) \cdot \Pr(headache|flu)$$

2.2 Propositional Foundations

We start this section by reviewing propositional logic to represent deterministic knowledge and reasoning by knowledge compilation. Next, we will discuss probabilistic graphical models to represent non-deterministic logic and probabilistic reasoning by weighted model counting.

2.2.1 Propositional Logic

Propositional or Boolean logic is the branch of logic that makes use of *propositional* or *Boolean variables* to express knowledge about single properties in the world. Propositional variables differ from random variables as their domain is restricted to the values *True* (T) and *False* (F). A *proposition* or *sentence* is a variable or a combination of variables and logical connectives. The three primitive connectives are *negation* (NOT,$\neg$), *disjunction* (OR,$\vee$) and *conjunction* (AND,$\wedge$). Other connectives such as *implication* ($\Rightarrow$) or *equivalence* ($\Leftrightarrow$) can be defined in terms of the three primitive connectives.

Example 2.9 Assume we want to model the behavior of a digital electronic circuit. Each of the gates can either be *healthy*, i.e. the output is a function of the input , or *broken*, i.e. the output cannot be determined based on the input. We can describe (part of) our knowledge about an inverter gate (`NOT-gate`) with the following English sentence: *"If a `NOT-gate` is healthy and the input is low (false), the output should be high (true)"*. We can express this knowledge by means of the following logical sentence:

$$healthy \wedge \neg in \Rightarrow out$$

where *healthy*, *in* and *out* are three propositional variables. We can rewrite the sentence by only using primitive connectives as follows:

$$\neg healthy \vee in \vee out$$

A *propositional theory* or *knowledge base* is a set of sentences that implicitly form a conjunction. A propositional literal is either a propositional variable x or its negation $\neg x$ and a *clause* is a disjunction of literals. A theory is said to be in conjunctive normal from (CNF) if it is a conjunction of clauses. An interpretation ω, also called *possible world*, is a truth value assignment to all variables. An interpretation that satisfies a sentence λ is denoted as $\omega \models \lambda$ and is called a *model* of that sentence. We say that a sentence is *satisfied* if, for a given truth value assignment to all variables, the sentence evaluates to *True*.

Example 2.10 The sentence $healthy \wedge \neg in \Rightarrow out$ contains three propositional variables which each can take two values. Hence, this sentence has $2^3 = 8$ interpretations or possible worlds. The only interpretation which is not a model of the sentence is $\{healthy, \neg in, \neg out\}$.

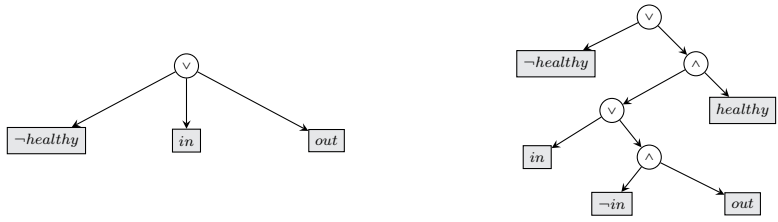
Given a propositional knowledge base, one can consider many logical inference tasks. Arguably the best-known problem is that of *satisfiability* (SAT) where the task is to find whether the knowledge base has at least one model. In case the theory is unsatisfiable, the *maximum satisfiability* (MAX-SAT) task allows us to find the interpretation that maximizes the number of satisfied clauses. The *partial maximum satisfiability* (PMAX-SAT) task combines the principles of SAT and MAX-SAT as it searches for an interpretation that certainly satisfies the *hard clauses*, and maximally satisfies the *soft clauses*.

2.2.2 Knowledge Compilation

Knowledge compilation is a key direction of research for dealing with the computational intractability of general propositional reasoning (Darwiche and Marquis 2002). Logical inference on a given propositional knowledge base is known to be computationally hard, unless the knowledge base comes with certain restrictions. The key idea of knowledge compilation is to compile a propositional theory into a specific target language or circuit representation that allows one to perform inference in polytime. The main advantage of knowledge compilation compared to other techniques, for example based on search, is circuit reuse. A theory only has to be compiled once, after which the compiled representation can be reused in order to answer different queries in polytime.

Research on knowledge representation and compilation has resulted in a wide range of target languages. Each of them comes with certain restrictions allowing them to support a set of inference tasks in polytime. We limit our discussion to the languages relevant for probabilistic inference and refer to the literature for a more elaborate overview (Darwiche 2011; Darwiche and Marquis 2002; Van den Broeck and Darwiche 2015).

Example 2.11 A Boolean circuit representation for the sentence $healthy \wedge \neg in \Rightarrow out$ is depicted in Figure 2.1a. More efficient representations, as they support more operations, for the same sentence are depicted in Figures 2.1b, 2.2a and 2.2b.



(a) Boolean Circuit (b) d-DNNF Circuit

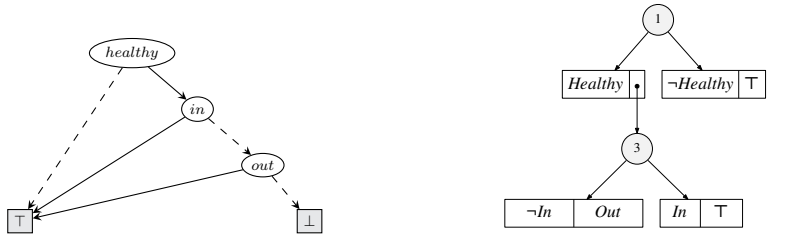
Figure 2.1: Circuit representation for the sentence $healthy \wedge \neg in \Rightarrow out$.

A circuit in **deterministic-Decomposable Negation Normal Form** (d-DNNF) (Darwiche 2004) for the sentence $healthy \wedge \neg in \Rightarrow out$ is depicted[1] in Figure 2.1b. A d-DNNF is a Boolean circuit with certain additional restrictions: negation can only appear in the leafs, the children of a conjunction range over disjoint sets of variables (decomposability), and the children of a disjunction are mutually exclusive (determinism). Some tasks require the d-DNNF to be smooth, that is, children of disjunctions should range over the same set of variables. The d-DNNF for our example is not smooth, as the left-child of the disjunction on the top of the circuit only mentions variable *healthy* while the right-child additionally mentions variables *in* and *out*. A smooth d-DNNF (sd-DNNF) for the same sentence is depicted in Figure 2.3a.

A **Binary Decision Diagram** (BDD) (Bryant 1992) for the sentence $healthy \wedge \neg in \Rightarrow out$ is depicted in Figure 2.2a. A circular node represent a decision, whether the variable in its label is true or false. Outgoing solid edges denote the variable being true, and dashed edges denote the variable being false. When a terminal is reached, the function is determined to either be true ($\top$) or false ($\bot$). In practice, the therm BDD almost always refers to Ordered BDD (OBDD), where all paths from the root to the terminals should mention the variables in the same order.

A **Sentential Decision Diagram** (SDD) (Darwiche 2011) for the sentence $healthy \wedge \neg in \Rightarrow out$ is depicted in Figure 2.2b. Circular nodes represent disjunctions and pairs of boxes represent a conjunction between their two children. More intuitively, circular nodes again represent decisions and the decisions are themselves represented as SDDs. While for BDDs decisions are only made over one single variable, SDDs can make decisions over mutually exclusive complex sentences. While the difference between BDD and SDD might not be immediately clear for our simple example sentence, a somewhat more complex SDD representation will be discussed in Figure 4.5c and Example 4.11.

[1] Circuit representations depicted in this chapter are inspired by Van den Broeck (2013)



(a) Binary Decision Diagram (b) Sentential Decision Diagram

Figure 2.2: Decision diagrams for the sentence $healthy \wedge \neg in \Rightarrow out$.

In general, one defines three key properties for each target language: its succinctness, the class of tractable queries it supports and the class of tractable transformations it admits. In general, a target representations is said to be tractable for a given operation if it supports the operation in time polynomial in its size. We limit our discussion to the properties relevant for this dissertation and refer to the literature for a more elaborate overview.

Succinctness

Succinctness refers to the size of the smallest compiled circuit for every Boolean formula. The succinctness ordering for the languages we consider in this text is

$$\text{d-DNNF} < \text{SDD} < \text{OBDD},$$

where d-DNNF $<$ SDD denotes that d-DNNF is strictly more succinct than SDD, and SDD $<$ OBDD denotes that SDD is is strictly more succinct than OBDD. Intuitively, there exists a Boolean formula whose smallest OBDD representation is exponentially larger than its smallest SDD representation (Bova 2016; Xue et al. 2012). The same holds for SDD representations compared to d-DNNF.

Tractable Queries

Typically, a propositional theory is queried in order to retrieve useful information from it. Each of the target compilation languages supports a set of different queries that can be answered in polytime. Table 2.4 summarizes the tractability of *model counting* and *equivalence checking* for the three target compilation languages introduced before.

Language	Equivalence Checking	Model Counting
d-DNNF	?	√
SDD	√	√
OBDD	√	√

Table 2.4: Efficient queries for the different languages. ? means "unknown" and √ means "satisfies".

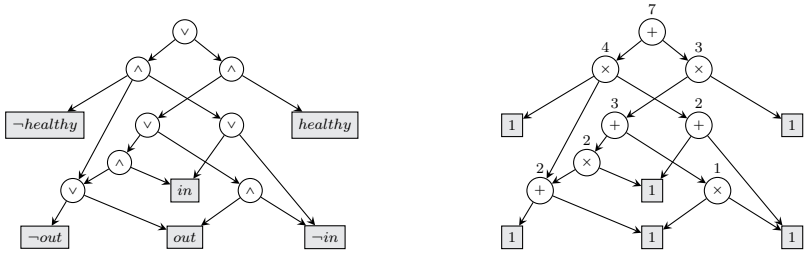
Language	Negation	Conjunction	Disjunction	Conditioning
d-DNNF	?	○	○	√
SDD	√	√	√	√
OBDD	√	√	√	√

Table 2.5: Efficient transformations for the different languages. ? means "unknown", √ means "satisfies", while ○ means "does not satisfy unless `P=NP`. Boolean operations assume a bounded number of operands, that are OBDDs with the same variable order, or SDDs with the same variable tree.

Equivalence checking is the task of testing whether two sentences are equivalent, that is, if for every value assignment of the boolean variables, the sentences evaluate to the same value. Both OBDD and SDD support this query in polytime, for d-DNNF this is unknown.

Example 2.12 The sentence $(a \vee b) \wedge c$ is logical equivalent to $(a \wedge c) \vee (b \wedge c)$ as all models of the first sentence are also models to the second sentence and vice versa.

Model counting returns the number of models of a theory, that is, the number of possible worlds that satisfy the theory. A smooth d-DNNF representation is known the be the most general to support model counting. To do so, one converts the sd-DNNF into an arithmetic circuit where each literal is replaced by the constant 1, conjunctions by multiplications and disjunctions by additions. Then, the model count of the sentence can be computed by evaluating the circuit in a bottom-up manner. An sd-DNNF and corresponding arithmetic circuit for the sentence $\mathit{healthy} \wedge \neg \mathit{in} \Rightarrow \mathit{out}$ is depicted in Figure 2.3. As OBDD and SDD are both a subset of the d-DNNF language, they also support model counting in polytime.



(a) sd-DNNF circuit (b) Model counting circuit

Figure 2.3: Circuit representation for the sentence $healthy \wedge \neg in \Rightarrow out$.

Example 2.13 The sentence $healthy \wedge \neg in \Rightarrow out$ has eight interpretations of which only one is not a model. Hence, the model count for this sentence is seven, as also computed by the arithmetic circuit in Figure 2.3b.

Polytime Transformations

Where a query returns information about a theory without changing it, a transformation is an operation that returns a modified theory, which is then operated on using queries. Table 2.5 summarizes the tractability of *Boolean operations* and *conditioning* for the three target compilation languages introduced before.

Boolean operations on sentences are supported for OBDDs and SDDs by means of an *apply operator* that is, two OBDDs or SDDs can be combined with a Boolean operator (disjunction or conjunction) only requiring linear[2] time and space. Support for Boolean operations is especially useful for incremental formula construction as we will further discuss in Chapter 4 and Chapter 5. Boolean operations are not supported by the d-DNNF language.

Conditioning replaces the variables $\mathbf{v}$ in sentence λ by their assignment in $\mathbf{V}$ and is denoted as $(\lambda|\mathbf{V})$. Then, these values are propagated while preserving the properties of the target representation. After this transformation, the sentence λ will not mention any of the variables in $\mathbf{v}$ anymore. Each of the considered target representations supports conditioning in polytime[2].

[2]Note that the apply operator, as well as conditioning, is not guaranteed to be polynomial for *reduced* SDDs. Fore more details, we refer to Van den Broeck and Darwiche (2015)

Example 2.14 Assume we want to condition the sentence $healthy \wedge \neg in \Rightarrow out$ on $\neg in$, this will simply return the sentence $healthy \wedge True \Rightarrow out$, which then simplifies to $healthy \Rightarrow out$. Conditioning the same original sentence on in would result in $healthy \wedge False \Rightarrow out$, which then simplifies to $True$.

Knowledge Compilation in Practice

The class of transformations admitted by a target representations defines the strategy we can employ to actually compile a theory. We distinguish *top-down* and *bottom-up* compilation techniques.

Top-Down Compilation Top-down knowledge compilers take as input a complete knowledge base and recursively divide the knowledge base into smaller fragments by means of conditioning. These smaller fragments are than compiled and combined in order to obtain the compilation of the complete theory. To obtain the smaller fragments in an efficient way, state-of-the-art compilers rely on techniques from the SAT literature. All existing top-down compilers, e.g. dsharp, c2d or minic2d, assume a knowledge base in CNF as input. Compilation into a d-DNNF representation is always done in a top-down manner.

Bottom-Up Compilation Target languages that efficiently support Boolean operations by means of an apply operator, such as OBDD and SDD, allow one to compile a theory in a bottom-up manner. The compiler performs a bottom-up pass through the knowledge base to first compile small pieces which are then combined using the Boolean operations. While top-down compilation into d-DNNF should result in more compact circuits (from a theoretical point of view), bottom-up compilation into SDD has practically shown to often outperform d-DNNF compilation (see, for example, Choi et al. (2013)). One of the reasons is that bottom-up compilation does not require the knowledge base to be in a CNF representation and this allows to more efficiently exploit structure in the model.

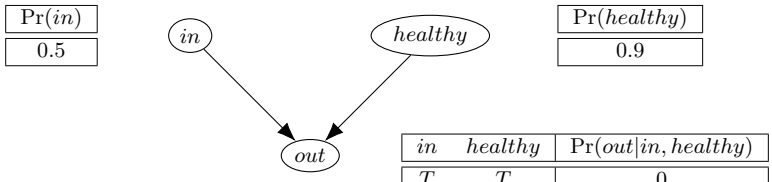
2.2.3 Probabilistic Graphical Models

Standard logic, as presented in the previous sections, does not easily allow one to express non-deterministic dependencies between propositions where the level of uncertainty is precisely quantified. This is a severe limitation when modeling problems with non-deterministic events or randomness. To accommodate for this, principles from knowledge representation and probability theory have been

combined, allowing one to effectively deal with uncertainty. The most important stream of research in this direction is, arguably, that of probabilistic graphical models.

A *probabilistic graphical model* (Koller and Friedman 2009) is a probabilistic model for which a graph expresses the dependencies between random variables. The graph encodes a joint distribution over all variables by means of a factorized representation where one exploits the (conditional) independences that hold in the distribution. The two main branches of graphical representations are *Bayesian networks* and *Markov networks*. One of the main differences between these two formalisms is the set of independencies they can encode. We will focus on Bayesian networks as they underlie the dynamic models we discuss in Chapter 3.

A *Bayesian network* (Pearl 1988) is a directed acyclic graph where each node represents a random variable and each edge indicates a direct influence among the variables. The network defines a conditional probability distribution $\Pr(x^i|\mathbf{Pa}(x^i)$ for every variable x^i, where $\mathbf{Pa}(x^i)$ are the parents of x^i in the graph. The conditional probabilities are given by a parameter $\theta_{x|\mathbf{Pa}(x)}$ and the conditional distributions are usually represented by means of a Conditional Probability Table (CPT).



in	*healthy*	Pr(*out*|*in*, *healthy*)
T	*T*	0
T	*F*	0.5
F	*T*	1
F	*F*	0.5

Figure 2.4: A Bayesian network modeling the behavior of a digital `NOT-gate`.

Example 2.15 Consider a Bayesian network that models the behavior of a digital inverter (`NOT-gate`), as depicted in Figure 2.4. The input, output and health state of the gate are each represented by a variable in the network. Each of the variables comes with a conditional probability table that expresses the probability of the variable given its parents. For example, the third row from the table for variable *out* tells us that if the gate is healthy and the input is low ($False$), the output should certainly be high ($True$).

The distribution represented by a Bayesian network induces that a variable depends on its parents and is independent of other variables (that are not its descendants) given its parents. In other words, two nodes in the network are conditionally independent, given the values of their parents. This allows us to factor the joint distribution into a product of conditional distributions:

$$\Pr(x^1, x^2, \ldots, x^n) = \prod_{i=1}^{n} \Pr(x^i | \mathbf{Pa}(x^i))$$

Conditional independence in a Bayesian network can also be defined in terms of sets of variables. A set of variables $\mathbf{z}$ is said to *d-separate* two sets of variables $\mathbf{x}$ and $\mathbf{y}$, if knowing the values for the variables in $\mathbf{z}$ makes the variables in $\mathbf{x}$ independent from the variables in $\mathbf{y}$. For more details on how to find these sets of variables, we refer to the literature (e.g. Pearl (1988)).

The main advantage of a Bayesian network is that its factorized representation allows one to represent a joint distribution in a compact way. The distribution induced by a network with n Boolean variables can be represented by a table with 2^n rows. Each of the rows corresponds to an unique assignment of truth values to the variables and is called a *possible world*. In general, the total number of rows required to represent each of the CPTs is much smaller than 2^n.

Example 2.16 For the network depicted in Figure 2.4, the factored representation is given by the following expression:

$$\Pr(in, healthy, out) = \Pr(out | in, healthy) \Pr(in) \Pr(healthy)$$

The network has three (Boolean) variables and, as such, 2^3 possible worlds. The joint probability distribution induced by the network is shown in Table 2.6.

The most common inference task in Bayesian networks is, arguably, to ask for marginal posterior probabilities, that is, the probability distribution of a single variable given evidence. More formally, for a query variable q, a set of observed variables $\mathbf{e}$ and a vector $\mathbf{E}$ with their observed truth values, we want to compute $\Pr(q | \mathbf{e} = \mathbf{E})$. Given a joint probability distribution, computing marginal probabilities can simply be done by means of Bayes' rule and involves enumerating the probabilities of the possible worlds that coincide with the query and evidence.

Example 2.17 A typical query for our example network would be to compute the probability that a gate is healthy, given observed signals at the input and output. For example:

$$\Pr(h | \neg in, out) = \frac{\Pr(h, \neg in, out)}{\Pr(\neg in, out)} = \frac{0.45}{0.45 + 0.225} = 0.67$$

in	$healthy$	out	$\Pr(in, healthy, out)$
T	T	T	$0.5 \cdot 0.9 \cdot 0$
T	T	F	$0.5 \cdot 0.9 \cdot (1\text{-}0)$
T	F	T	$0.5 \cdot (1\text{-}0.9) \cdot 0.5$
T	F	F	$0.5 \cdot (1\text{-}0.9) \cdot (1\text{-}0.5)$
F	T	T	$(1\text{-}0.5) \cdot 0.9 \cdot 1$
F	T	F	$(1\text{-}0.5) \cdot 0.9 \cdot (1\text{-}1)$
F	F	T	$(1\text{-}0.5) \cdot (1\text{-}0.9) \cdot 0.5$
F	F	F	$(1\text{-}0.5) \cdot (1\text{-}0.9) \cdot (1\text{-}0.5)$

Table 2.6: The joint probability distribution encoded by the Bayesian network depicted in Figure 2.4.

While the joint probability table defines the semantics of a Bayesian network, exhaustively generating all possible world is computationally infeasible for all but the smallest problems. For larger networks, when modeling real-world problems, we have to rely on more efficient inference algorithms.

2.2.4 Probabilistic Inference by Weighted Model Counting

Probabilistic inference in graphical models, and more specifically Bayesian networks, is a well studied problem and has resulted in many algorithms, e.g. variable elimination, recursive conditioning and junction trees (Darwiche 2009; Koller and Friedman 2009). Throughout this dissertation we are mainly interested in approaches based on weighted model counting, and this for two reasons. Firstly, weighted model counting is a well-studied task that serves as an assembly language for probabilistic inference, i.e. inference in different types of probabilistic models can be *reduced* to the task of weighted model counting. Secondly, techniques based on weighted model counting allow us to exploit *local structure* in the distribution induced by the network.

Weighted Model Counting

Given a propositional sentence λ, model counting returns the number of interpretations that satisfy the sentence. For weighted model counting (WMC), a weight is associated with every model, and the task is to enumerate the weight of all models. The weight of one model is the product of the weights associated with the literals.

The WMC of a sentence λ can then be computed as:

$$WMC(\lambda) = \sum_{\omega \models \lambda} \prod_{l \in \omega} w(l)$$

where l are the literals in the model ω. The weight function $w(\cdot)$ assigns a weight to each literal. Note that, in general, the weight of a literal is not required to be between 0 and 1. We often use a short-hand notation where the weight function for a variable is denoted by a tuple. In this case, the first element in this tuple denotes the weight for the positive literal and the second element denotes the weight for the negative literal.

Example 2.18 Assume the propositional sentence $healthy \wedge \neg in \Rightarrow out$ with the following weight function: $w(healthy) = 0.4$, $w(\neg\ healthy) = 3$, $w(input) = 2$, $w(\neg\ input) = 27$, $w(output) = 1$, $w(\neg\ output) = 8$. The weighted model count of the sentence is 801, as depicted by Table 2.7. The fourth interpretation is not a model of the sentence and its weight does not contribute to the weighted model count.

Reduction to Weighted Model Counting

Weighted model counting is defined for a propositional sentence or knowledge base and probabilistic inference by weighted model counting thus requires one to encode the probabilistic model by means of a propositional knowledge base. Intuitively, the reduction scheme for Bayesian networks constructs a knowledge base such that each of the models of the knowledge base are in one-to-one correspondence with the rows of the joint distribution induced by the network. Additionally, the weights to the literals are assigned in such a way that the weight of each model is equal to the probability of the corresponding row in the joint distribution.

The literature proposes different reduction schemes or encodings for Bayesian networks (Chavira and Darwiche 2008). Throughout this dissertation, we will use the encoding proposed by Fierens et al. (2015) which is based on the encoding presented by Sang et al. (2005b). We choose this encoding as it not only applies to Bayesian networks, but also to probabilistic logic programs as we will discuss in Section 4.2.3.

healthy	*in*	*out*	Weight
T	T	T	$0.4 \cdot 2 \cdot 1$
T	T	F	$0.4 \cdot 2 \cdot 8$
T	F	T	$0.4 \cdot 27 \cdot 1$
T	F	F	-
F	T	T	$3 \cdot 2 \cdot 1$
F	T	F	$3 \cdot 2 \cdot 8$
F	F	T	$3 \cdot 27 \cdot 1$
F	F	F	$3 \cdot 27 \cdot 8$
		total	801

Table 2.7: Weighted model counting for the sentence $healthy \wedge \neg in \Rightarrow out$.

For Bayesian networks, the encoding introduces an *indicator variable* for each random variable x and a *parameter variable* for each CPT parameter $\theta_{x|\mathbf{Pa}(x)}$. Then, each row of a CPT is encoded as a conjunction of the literal variables and a CPT is encoded as a disjunction of these conjunctions. The weight functions sets a weight of (1,1) to each of the indicator variables and ($\theta_{x|\mathbf{Pa}(x)}$, 1 - $\theta_{x|\mathbf{Pa}(x)}$) to each of the parameter variable.

Once the Bayesian network is encoded as a propositional knowledge base Σ, we can perform probabilistic inference in the network by means of WMC on Σ. Concretely, we compute the marginal probability for a query q as follows:

$$\Pr(q) = \frac{WMC(\Sigma \wedge q, w)}{WMC(\Sigma, w)}$$

where w is the weight function.

The conditional probability for a query q given evidence $\mathbf{E}$ can be computed as:

$$\Pr(q|E) = \frac{WMC(\Sigma \wedge q \wedge \mathbf{E}, w)}{WMC(\Sigma \wedge \mathbf{E}, w)}$$

Example 2.19 We can encode the Bayesian network depicted in Figure 2.4 into to following knowledge base:

$$
\begin{aligned}
in &\Leftrightarrow p\_in \\
healthy &\Leftrightarrow p\_healthy \\
out &\Leftrightarrow (in \wedge healthy \wedge p\_row1) \\
&\quad \vee (in \wedge \neg healthy \wedge p\_row2) \\
&\quad \vee (\neg in \wedge healthy \wedge p\_row3) \\
&\quad \vee (\neg in \wedge \neg healthy \wedge p\_row4)
\end{aligned}
$$

where each of the p_* variables correspond to a CPT parameter $\theta_{x|\mathbf{Pa}(x)}$. We have the following weight function for the indicator variables: $w(in) = (1, 1)$, $w(out) = (1, 1)$, $w(healthy) = (1, 1)$, and for the parameter variables: $w(p\_in) = (0.5, 0.5)$, $w(p\_healthy) = (0.9, 0.1)$, $w(p\_row1) = (0, 1)$, $w(p\_row2) = (0.5, 0.5)$, $w(p\_row3) = (1, 0)$, $w(p\_row4) = (0.5, 0.5)$.

Local Structure

One of the benefits of the reduction to WMC is that it allows one to exploit local structure in the model. Bayesian networks often exhibit abundant local structure in the form of *determinism* and *context-specific independence* (CSI) (Boutilier et al. 1996). Determinism is introduced by 0 and 1 *parameters* in the network while CSI is often the result of *equal parameters*. Exploiting local structure can lead to exponential speed gains and allows one to perform inference in networks of high treewidth, where this is otherwise impossible (Chavira and Darwiche 2005).

In general, all entries in a CPT with a 0 parameter can be dropped as they give rise to models with a weight of 0. Furthermore, our encoding allows to safely omit parameter variables with a probability of 1 as they will not change the weighted model count. Finally, entries in the CPT with equal parameters often lead to context-specific independence and allows one to obtain a more compact encoding by combining rows from the table.

Example 2.20 When exploiting local structure, we can encode the Bayesian network depicted in Figure 2.4 into to following knowledge base:

$$in \Leftrightarrow p\_in$$

$$healthy \Leftrightarrow p\_healthy$$

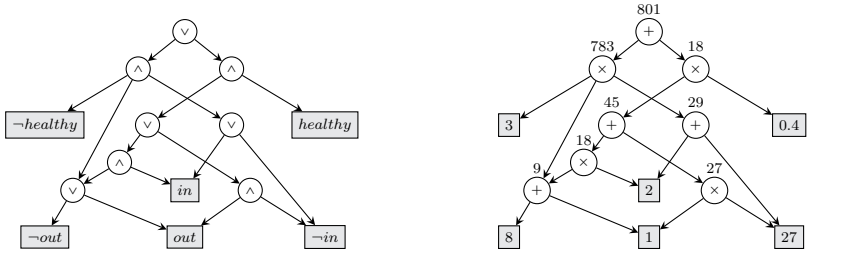
$$out \Leftrightarrow (\neg healthy \wedge p\_faulty)$$

$$\vee\ (\neg in \wedge healthy)$$

where we have $w(p\_faulty) = (0.5, 0.5)$. Intuitively, computing the weighted model count of this formula can be done more efficiently, compared to the formula from Example 2.19, as we were able to drop three (parameter) variables.

Weighted Model Counting and Knowledge Compilation

A given propositional knowledge base, as shown in Example 2.19 and 2.20, does not allow one the efficiently perform weighted model counting. One way to deal with this is to compile the knowledge base into a more tractable target representation which supports WMC in polytime. The language often used as target representation is sd-DNNF, as it is the best known language to support weighted model counting in polynomial time. Similar to model counting, we convert the sd-DNNF into an arithmetic circuit but now replace the literals by the constant given by the weight function. Then, the weighted model count can be computed by evaluating the circuit in a bottom-up manner.



(a) sd-DNNF circuit (b) Weighted model counting circuit

Figure 2.5: Circuit representation for the sentence $healthy \wedge \neg in \Rightarrow out$.

Example 2.21 For our example sentence $healthy \wedge \neg in \Rightarrow out$ and the weight function given in Example 2.18, the sd-DNNF and arithmetic circuit are depicted in Figure 2.5. The WMC computed by evaluating the circuit in a bottom

up manner is 801 and is the same as the WMC computed by exhaustively enumerating all interpretations that satisfy the sentence, as was done in Table 2.7.

The advantage of knowledge compilation, compared to other techniques for WMC, is circuit reuse. In the presence of new observations or evidence, we only have to adapt the weight function, in order to incorporate the evidence, and there is no need to recompile the network. Hence, for a given knowledge base we can summarize probabilistic inference by knowledge compilation and weighted model counting as a three step procedure:

(1) Compile the knowledge base Σ into a sd-DNNF Δ (or any of its sub-languages).

$$\Delta = \textsc{Compile}(\Sigma)$$

(2) Incorporate evidence $\mathbf{E}$ by setting to zero the weight of any literal that is not compatible with the evidence.

$$w'(z) = \begin{cases} w(z) & \neg z \notin \mathbf{E} \\ 0 & \neg z \in \mathbf{E} \end{cases}$$

(3) Traverse the compiled representation Δ to either:
(a) compute the weighted model count with an upward pass only:

$$\Pr(\mathbf{E}) = \textsc{Eval}_{\uparrow}(\Delta, w')$$

(b) compute the marginal probability $\Pr(z|\mathbf{E})$, for all variables Z in parallel, with one upward and downward pass (Darwiche 2009, Algorithm 34):

$$\Pr(z|\mathbf{E}) = \textsc{Eval}_{\uparrow\downarrow}(\Delta, w')$$

In the presence of new evidence we do not have to recompile the theory, i.e. we can omit step (1) of the above algorithm, and only need to redo step (2) and (3). As a final remark, we would like to note that the conversion to an arithmetic circuit, as shown in Figure 2.3 and 2.5, is not strictly required and one can compute the (weighted) model count directly on an sd-DNNF representation.

2.3 Relational Foundations

We will now introduce relational representations where we mainly focus on (probabilistic) logic programming. Reasoning techniques for these representations will be dealt with in Chapter 4.

2.3.1 First-Order Logic

First-order logic, also called predicate logic, extends propositional logic to formulas involving quantified logical variables and predicate symbols. This allows one to express general knowledge about multiple objects in the world where, in propositional logic, we would need a specific proposition for each of these objects. Hence, first-order logic offers us a more expressive formalism to deal with structured knowledge in a compact way.

A *term* is a logical variable, a constant, or a functor applied on terms. A logical variable is denoted with an upper-case letter (e.g. X). An *atom* is of the form $p(t_1, \ldots t_n)$ where p is a predicate of arity n and the t_i are terms. A *literal* is an atom or its negation. A first order formula is recursively constructed from atoms using logical connectives (as for propositional sentences) and quantifiers ($\forall$ and $\exists$). A theory is a set of formulas that implicitly form a conjunction. An expression is called *ground* if it does not contain variables.

Example 2.22 Consider the propositional sentence $healthy \wedge \neg in \Rightarrow out$ that we introduced in Example 2.9. We can now express more general knowledge by means of first-order logic in the following way:

$$\forall \mathsf{X}, inverterGate(\mathsf{X}) \Rightarrow (healthy(\mathsf{X}) \wedge \neg input(\mathsf{X}) \Rightarrow output(\mathsf{X}))$$

The logical variable X serves as a placeholder for, potentially, each digital inverter gate (`NOT gate`) in the world. In case of propositional logic, dealing with multiple gates would require to add a specific proposition and corresponding knowledge for each of the gates.

The *Herbrand Base* of a FOL theory is the set if all ground atoms constructed using the predicates, functors and constants in the theory. A *Herbrand interpretation*, also called *possible world*, is a truth value assignment to all atoms in the Herbrand base. It is common to write it as the subset of *True* atoms (with all others being *False*), or as a conjunction of atoms. An interpretation satisfying all formulas in the theory, i.e. if all formulas resolve to true, is a *model* of the theory.

2.3.2 Logic Programming

Logic programming (Lloyd 1989) uses a subset of first-order logic where only *Horn clauses*, are allowed. A Horn clause is a universally quantified clause that has at most one positive literal. A *definite clause* is a Horn clause with exactly one positive literal and, following the Prolog tradition, it is written

as $\mathtt{h} :- \mathtt{b_1}, ..., \mathtt{b_n}$ where $\mathtt{h}$ and the $\mathtt{b_i}$ are atoms and the comma denotes a conjunction. The atom $\mathtt{h}$ is called the *head* of the clause and $\mathtt{b_1}, ..., \mathtt{b_n}$ the *body*. The meaning of such a clause is that whenever the body is true, the head has to be true as well. A *fact* is a clause that has `true` as its body, i.e. it is a definite clause with no negative literals, and is written more compactly as $\mathtt{h}$.

A logic program or *definite clause program* is a finite set of definite clauses, also called *rules*. Let $\mathcal{A}$ be the set of all ground atoms that can be constructed from the constants, functors and predicates in a definite clause program $\mathcal{P}$. A Herbrand interpretation, as defined before, satisfying all rules in the program $\mathcal{P}$ is a *Herbrand model*. The model-theoretic semantics of a definite clause program is given by its unique *least Herbrand model*, that is, the set of all ground atoms $a \in \mathcal{A}$ that are entailed by the logic program, written $\mathcal{P} \models a$. The task of logical inference is to determine whether a program $\mathcal{P}$ entails a given atom, called *query*. We will discuss techniques for logical inference in Section 4.2.

A *normal logic program* extends a definite clause program and allows for *negation*, i.e., it is a finite set of *normal clauses* of the form $\mathtt{h} :- \mathtt{b_1}, ..., \mathtt{b_n}$ where $\mathtt{h}$ are atoms and the $\mathtt{b_i}$ are *literals*. A literal is an atom (positive literal) or its negation (negative literal). For normal logic programs, the least Herbrand Model extends to a canonical model[3].

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

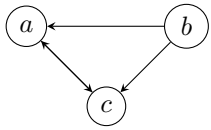


Figure 2.6: A logic program modeling a cyclic graph.

Example 2.23 Consider the definite clause program depicted in Figure 2.6. The facts represent the edges between two nodes in a graph[4] and the rules define whether there is a path between two nodes. The least Herbrand model is given by $\{\mathtt{edge}(b,a), \mathtt{edge}(b,c), \mathtt{edge}(a,c), \mathtt{edge}(c,a), \mathtt{path}(b,a), \mathtt{path}(b,c), \mathtt{path}(a,c), \mathtt{path}(c,a), \mathtt{path}(a,a), \mathtt{path}(c,c)\}$.

[3]For a full discussion of the semantics of general logic programs, we refer to Van Gelder et al. (1991).

[4]Note the difference in arrows to differentiate between Bayesian networks and graphs.

Logic Programs vs. First-Order Logic A crucial difference between the semantics in logic programming and first-order logic is that the former makes use of the *closed world assumption*. Under this assumption, everything that is not certainly true is assumed to be false. Hence, in logic programming an atom a is defined to be true if and only if at least one of the rule bodies, for which a is the head, is true. As we will see later on, the difference in semantics between first-order logic and logic programming is crucial in the conversion of a ground logic program into a propositional formula.

Example 2.24 Consider the logical theory $\{a \Leftarrow b\}$ which has three models, namely $\{\neg a, \neg b\}, \{a, \neg b\}$ and $\{a, b\}$. The syntactically equivalent logic program {`a :- b.`} has only one model, namely the least Herbrand model $\{\neg a, \neg b\}$.

2.3.3 Probabilistic Logic Programming

Where probabilistic graphical models combine propositional knowledge with probability theory, probabilistic logics combine relational or first-order knowledge with probability theory. This allows for complex, yet compact, models that express probabilistic relations between objects in the world. One stream of research in this direction are *probabilistic logic programs* which enrich logic programs, typically Prolog, with probabilities to deal with uncertainty.

Many probabilistic programming languages, including PRISM (Sato and Kameya 2001), ICL (Poole 1993), ProbLog (De Raedt, Kimmig, and Toivonen 2007), and LPADs (Vennekens, Verbaeten, et al. 2004) are based on the same semantics, known as Sato's distribution semantics (Sato 1995). Throughout this dissertation we will use ProbLog as it has the simplest syntax and comes with the least restrictions. PRISM and ICL, for example, require rules to be acyclic and cannot be used to model cyclic programs. For a general overview of probabilistic logic programming, and more details on the relation between these languages, we refer to De Raedt and Kimmig (2015).

A ProbLog program $\mathcal{P}$ consists of a set $\mathcal{R}$ of *rules* and a set $\mathcal{F}$ of *probabilistic facts*. Without sacrificing generality, we assume that no probabilistic fact unifies with a rule head. Every grounding $f\theta$ of a probabilistic fact $p :: f$ independently takes the value `true` (with probability p) or `false` (with probability $1-p$). For ease of notation, we assume that $\mathcal{F}$ is ground.

Example 2.25 The logic program shown in Figure 2.6 can be extended with probabilities as shown in Figure 2.7. The probabilistic facts represent that edges between two nodes are only true with a certain probability. As a consequence, the rules now express a probabilistic path.

$0.4 :: \texttt{edge}(b, a).$ $\quad$ $0.3 :: \texttt{edge}(b, c).$

$0.8 :: \texttt{edge}(a, c).$ $\quad$ $0.9 :: \texttt{edge}(c, a).$

$\texttt{path}(\mathsf{X}, \mathsf{Y})$:- $\texttt{edge}(\mathsf{X}, \mathsf{Y}).$

$\texttt{path}(\mathsf{X}, \mathsf{Y})$:- $\texttt{edge}(\mathsf{X}, \mathsf{Z}), \texttt{path}(\mathsf{Z}, \mathsf{Y}).$

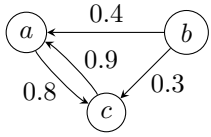


Figure 2.7: A probabilistic logic program modeling a cyclic probabilistic graph.

A ProbLog program specifies a probability distribution over its Herbrand interpretations, also called possible worlds. A *total choice* $C \subseteq \mathcal{F}$ assigns a truth value to every ground probabilistic fact, and the corresponding logic program $C \cup \mathcal{R}$ has a canonical model (Fierens et al. 2015); the probability of this model is that of C. Interpretations that do not correspond to any total choice have probability zero. For a probabilistic logic program with n ground probabilistic facts, the number of distinct total choices is 2^n.

Example 2.26 Consider the ProbLog program depicted in Figure 2.8. Each of the total choices together with their corresponding model and weight is depicted in Table 2.8. For example, $\{\texttt{edge}(a, b), \texttt{edge}(a, c), \texttt{path}(c, b)\}$ is an interpretation that is not a model of the program because, as the rules in the program state, if an edge is true its corresponding path should be true. Hence, this interpretation has a probability of 0.

Where inference in logic programs considers the task of computing whether a *query* atom q is entailed by the program, inference in probabilistic logic programs considers the task of computing the probability that a query is entailed. Concretely, the probability of a query is the sum over all total choices whose program entails q and can be computed as follows:

$$\Pr(q) := \sum_{C \subseteq \mathcal{F} : C \cup \mathcal{R} \models q} \prod_{f_i \in C} p_i \cdot \prod_{f_i \in \mathcal{F} \setminus C} (1 - p_i) \,. \tag{2.1}$$

Exhaustively generating all total choices and computing whether their corresponding program entails the query is computational infeasible for all but the smallest programs. Therefore, we have to rely on more efficient algorithms which we will discuss in Chapter 4.

$0.8 :: \texttt{edge}(a, c).$ $\quad$ $0.3 :: \texttt{edge}(a, b).$

$0.4 :: \texttt{edge}(c, b).$

$\texttt{path}(X, Y) \texttt{ :- } \texttt{edge}(X, Y).$

$\texttt{path}(X, Y) \texttt{ :- } \texttt{edge}(X, Z), \texttt{path}(Z, Y).$

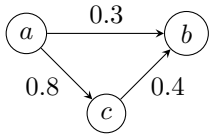


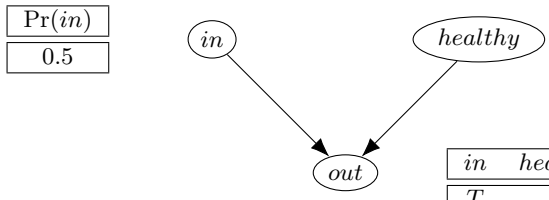
Figure 2.8: A probabilistic logic program modeling an acyclic probabilistic graph.

$\texttt{e}(a,b)$	$\texttt{e}(a,c)$	$\texttt{e}(c,b)$	$\texttt{p}(a,b)$	$\texttt{p}(a,c)$	$\texttt{p}(c,b)$	weight
T	T	T	T	T	T	$0.3 \cdot 0.8 \cdot 0.4$
T	T	F	T	T	F	$0.3 \cdot 0.8 \cdot (1\text{-}0.4)$
T	F	T	T	F	T	$0.3 \cdot (1\text{-}0.8) \cdot 0.4$
T	F	F	T	F	F	$0.3 \cdot (1\text{-}0.8) \cdot (1\text{-}0.4)$
F	T	T	T	T	T	$(1\text{-}0.3) \cdot 0.8 \cdot 0.4$
F	T	F	F	T	F	$(1\text{-}0.3) \cdot 0.8 \cdot (1\text{-}0.4)$
F	F	T	F	F	T	$(1\text{-}0.3) \cdot (1\text{-}0.8) \cdot 0.4$
F	F	F	F	F	F	$(1\text{-}0.3) \cdot (1\text{-}0.8) \cdot (1\text{-}0.4)$
					total	1

Table 2.8: All total choices for the ProbLog program depicted in Figure 2.8.

Example 2.27 Consider the ProbLog program depicted in Figure 2.8. With Table 2.8 listing all total choices (abbreviating predicate names by initials), we can compute the probability for a query atom q by simply enumerating the weight of all rows (models) in which q is true. For an example query $\texttt{path}(a, b)$, we have to enumerate the weight of the first five rows from Table 2.8 and this results in $\Pr(\texttt{path}(a, b)) = 0.524$.

Example 2.28 Where probabilistic logic programs are especially useful to deal with relational knowledge, they also allow us to represent propositional models. Figure 2.9 depicts the Bayesian network introduced before and the probabilistic logic program that describes this network. Without exploiting local structure, we need a corresponding probabilistic fact for each of the parameters in the network, and for each row in the CPTs we have a corresponding rule that describes the dependencies.



in	*healthy*	Pr(*out*|*in*, *healthy*)
T	*T*	0
T	*F*	0.5
F	*T*	1
F	*F*	0.5

0.5::p_input. 0.9::p_healthy.

0.5::p_row1. 1.0::p_row2. 0.5::p_row3. 0.0::p_row4.

```
input :- p_input.

healthy :- p_healthy.

output :- input, healthy, p_row1.

output :- input, ¬healthy, p_row2.

output :- ¬input, healthy, p_row3.

output :- ¬input, ¬healthy, p_row4.
```

Which we can rewrite by exploiting local structure as:

0.5::p_input. 0.9::p_healthy. 0.5::p_faulty.

```
input :- p_input.

healthy :- p_healthy.

output :- ¬healthy, p_faulty.

output :- ¬input, healthy.
```

Figure 2.9: A probabilistic logic program modeling the behavior of a digital NOT-gate.

Chapter 3

Dynamic Bayesian Networks

3.1 Introduction

Bayesian Networks (BNs) have shown to be powerful and popular tools for reasoning about uncertainty (Pearl 1988). While BNs were originally developed for static domains, they have been extended towards dynamic domains to cope with time-related or sequential data. These Dynamic Bayesian Networks (DBNs) (Dean and Kanazawa 1989; Murphy 2002) allow us to reason about the past, the present and the future and are widely used in applications such as computer vision, speech recognition, robot localization, health monitoring, bio-sequence analysis, machine monitoring, forecasting, games, etc (Boyen and Koller 1998; Forbes et al. 1995; Huang et al. 1994; Kjaerulff 1995; Sandri et al. 2014; Theocharous et al. 2004; Zweig and Russell 1998).

Dynamic Bayesian networks allow us to compactly model a stochastic process by making abstraction of time. In other words, we can fully model a dynamic process by only defining a prior distribution and a transition model. The former expresses our belief at the start of the process and the latter acts as a template for (potentially) an infinite number of time instances. More specific models such as the popular Hidden Markov Models (HMMs) (Rabiner 1989) and Kalman filters (Kalman 1960) are all generalized by DBNs.

As a special kind of Bayesian networks, inference in dynamic networks can be performed by applying any of the existing algorithms for static networks. These inference methods, including junction trees and variable elimination, typically exploit conditional independencies (CI) by using a factorized representation of the probability distribution (Darwiche 2009; Koller and Friedman 2009).

Approach	CI	LS	RS
1. Traditional BN algorithm on the unrolled network	✓		
2. Knowledge compilation on the unrolled network	✓	✓	
3. Interface algorithm	✓		✓
4. Structural interface algorithm	✓	✓	✓

Table 3.1: Properties exploited by DBN inference algorithms.

More recent techniques, including knowledge compilation and weighted model counting, additionally exploit local structure (LS) in the network. It is well known that local structure, e.g. deterministic dependencies or equal parameters, can induce additional independencies in the network and exponential speed gains can be obtained by exploiting this structure, see for example Chavira and Darwiche (2005).

Special purpose inference algorithms for DBNs, such as the *interface algorithm* (Murphy 2002), exploit CI in the network as well as the repeated structure (RS) obtained from duplicating the template model along the time dimension. As a result, inference time is guaranteed to scale linearly with the number of required time steps and memory resources remain nearly constant. These specific inference algorithms for DBNs, however, are based on more traditional inference methods and do not exploit any of the local structure in the network.

In this chapter, we show how to use weighted model counting and knowledge compilation techniques for efficient exact inference in DBNs. Where current techniques based on weighted model counting require to first unfold the network, our approach directly acts on the transition model to exploit the repeated structure. Our main contribution is the *structural interface algorithm*, an exact inference algorithm for dynamic Bayesian networks that speeds up inference by exploiting CI, RS and LS (see Table 3.1). As a result, we can tackle dynamic models that are considerably more complex than what is currently possible with exact solvers.

The structural interface algorithm and experimental results were previously published in

J. Vlasselaer, W. Meert, G. Van den Broeck, and L. De Raedt (2016a). "Exploiting local and repeated structure in dynamic Bayesian networks". In: *Artificial Intelligence* 232, pp. 43–53

Structure of this Chapter

In Section 3.2 we introduce dynamic Bayesian networks and shortly discuss its properties. Section 3.3 deals with existing inference methods for these networks, including the *interface algorithm*. We then introduce our *structural interface algorithm* in Section 3.4 where we investigate the trade-offs of compiling the transition model of a DBN into a circuit representation. Section 3.5 proposes some additional optimization to further speed up inference in DBNs. We evaluate our algorithm on two classes of benchmark DBNs in Section 3.6. Related work is discussed in Section 3.6 and we conclude this chapter with a discussion in Section 3.7.

3.2 Preliminaries

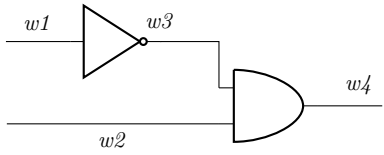
We now formally introduce dynamic Bayesian networks as well as the relevant inference tasks considered on these networks.

3.2.1 Representation

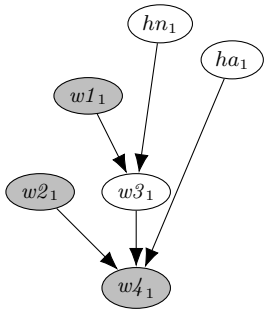
A Dynamic Bayesian Network (DBN) (Dean and Kanazawa 1989; Murphy 2002) is a directed acyclic graphical model that represents a stochastic process. It models a probability distribution over a semi-infinite collection of random variables $\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \ldots$, where $\mathbf{z}_t$ are the variables at time t and $\mathbf{z}_{1:T}$ denotes all variables up until time T. In other words, each of the variables in a DBN is associated with a *time slice* t, and the number of time slices required to model a particular problem is the *time span* T. One often distinguishes the set of unobserved or hidden variables $\mathbf{x}_t$, which are the state variables, and $\mathbf{e}_t$, which are the observed variables. The probability distribution over the state variables $\mathbf{x}_t$, i.e. all unobserved variables for a certain time slice t, is referred to as the *belief state*.

A dynamic Bayesian network is defined by two networks: B_1, which specifies the prior or initial state distribution $\Pr(\mathbf{z}_1)$, and $B_{\rightarrow}$, a two-slice temporal BN (2TBN) that specifies the transition model $\Pr(\mathbf{z}_t|\mathbf{z}_{t-1})$. Together, they represent the distribution

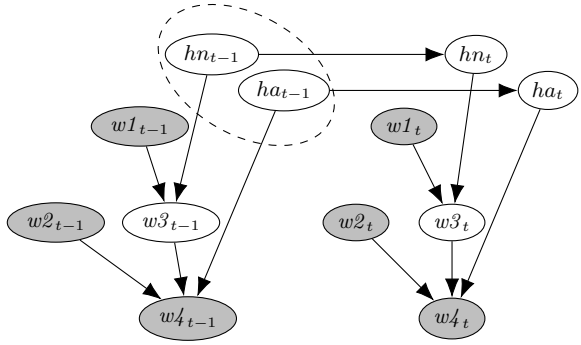
$$\Pr(\mathbf{z}_{1:T}) = \Pr(\mathbf{z}_1) \prod_{t=2}^{T} \Pr(\mathbf{z}_t|\mathbf{z}_{t-1}) \tag{3.1}$$



(a) An electronic digital circuit.



(b) Prior distribution.



(c) Transition model.

Figure 3.1: A digital circuit and corresponding dynamic Bayesian network. The dashed oval indicates the variables in the *interface*.

The initial network B_1 is a regular Bayesian network, which factorizes the distribution over its N variables as $\Pr(\mathbf{z}_1) = \prod_{i=1}^{N} \Pr(z_1^i | \mathbf{Pa}(z_1^i))$, where z_t^i is the ith variable at time t and $\mathbf{Pa}(z_t^i)$ are the parents of z_t^i in the network.

The transition model $B_{\rightarrow}$ is not a regular Bayesian network as only the nodes in the second slice (for time t) of the 2TBN have an associated conditional probability distribution. Thus, the transition model factorizes as $\Pr(\mathbf{z}_t | \mathbf{z}_{t-1}) = \prod_{i=1}^{N} \Pr(z_t^i | \mathbf{Pa}(z_t^i))$, where $\mathbf{Pa}(z_t^i)$ can contain variables from either $\mathbf{z}_t$ or $\mathbf{z}_{t-1}$.

Example 3.1 Consider as an example the task of finding failing components in the digital circuit depicted in Figure 3.1a. The circuit contains a logical `NOT-gate` with wire *w1* as input and wire *w3* as output and a logical `AND-gate` with wires *w2* and *w3* as input and wire *w4* as output. The prior distribution and transition model for the corresponding DBN is depicted in Figure 3.1b and 3.1c, respectively. We use *hn* to represent the state of the `NOT-gate` and *ha* for the `AND-gate`. Hence, each of the variables in $\mathbf{z}_t$ either represents the state of a wire (e.g. *high* or *low*) or the state of a component (e.g *healthy* or *faulty*). The transition model defines the dynamics of the components' state over time. The shaded nodes are observed.

3.2.2 Semantics

A DBN is completely defined by the prior distribution and transition model and implicitly represents an infinite number of time slices. The semantics of a DBN, however, is defined by *unrolling* the transition model (2TBN) for a finite number of time slices T. Intuitively, this comes down to take copies of the transition model and "glue" them together. In practice, the number of required time slices is determined by the available observations or the inference task.

Example 3.2 Figure 3.2 depicts the network from Figure 3.1b and 3.1c, but now unrolled for three time slices. This implies we have observations for the third time slice or we want to compute the posterior probability for some of the variables of the third time slice.

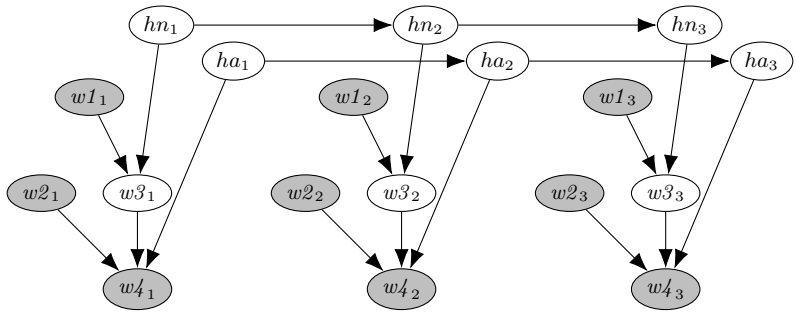


Figure 3.2: An unrolled network for three time slices.

It is important to note that a DBN has the same structure at any time slice t and the word "dynamic" implies that the network models a dynamic process, not that the structure dynamically changes over time. This is easy to verify as the unrolled network is obtained by duplicating the transition model along the time dimension.

3.2.3 Markov Assumption

It is common to assume that a DBN models a first-order Markov process where the current state only depends on the previous state and not on any earlier states. In other words, cross-slice edges are only allowed from time slice t to slice $t + 1$. Hence, the belief state provides enough information to make the future conditionally independent of the past, that is, the hidden state variables

$\mathbf{x}_t$ d-separate the past from the future and we have

$$\Pr(\mathbf{x}_t|\mathbf{x}_{1:t-1}) = \Pr(\mathbf{x}_t|\mathbf{x}_{t-1}) \tag{3.2}$$

One can only define the transition model by means of a 2TBN in case the first-order Markov assumption holds. In general, a DBN modeling a $(k-1)^{\text{th}}$-order Markov process can be defined by means of a kTBN.

Example 3.3 The unrolled network depicted in Figure 3.2 clearly satisfies the first-order Markov assumption as variables in time slice t only depend on variables from within the same time slice or from time slice $t-1$.

While the first-order Markov property is typically assumed, most inference algorithms can be easily generalized towards higher-order Markov processes by operating on the kTBN instead of the 2TBN. Another way to deal with higher-order Markov processes is to decouple cross-slice edges by means of additional variables with the *identity function* (Murphy 2002).

3.2.4 Inference Tasks

The goal of (marginal) inference in temporal models is to compute $\Pr(x_t^i|\mathbf{E}_{1:\tau})$, that is, the probability of a hidden variable x^i at time t, given a sequence of observations $\mathbf{E}_{1:\tau}$ up until time τ. In general, one considers the following three cases:

1. With $t = \tau$ we have *filtering* or *monitoring*: Computing the probability of some present events given evidence about the past and present, for example $\Pr(x_3^i|\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3)$. This task is typically performed by an agent as it allows him to keep track of the current state and to make decisions in a more informed way.

2. With $t > \tau$ we have *prediction* or *forecasting*: Computing the probability of some future events given evidence about the past, for example $\Pr(x_4^i|\mathbf{E}_1, \mathbf{E}_2)$. This task is useful to estimate how a certain state will evolve over time.

3. With $t < \tau$ we have *smoothing* or *diagnosis*: Computing the probability of some past events given evidence about the future, for example $\Pr(x_2^i|\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3, \mathbf{E}_4)$. This task provides a better estimate of the state than was available at the time as it incorporates more evidence.

From an inference point of view, filtering and prediction only need to incorporate information from the past and the present and only requires a *forward pass* trough the network. Smoothing also deals with observations from the future and additionally requires a *backward pass*.

Example 3.4 For our example on finding failing components in digital circuits (see Example 3.1), one is typically interested in the health states of the components at time t, given a sequence of observed electrical inputs and outputs up to and including t, i.e. the task of filtering. This corresponds to computing $\Pr(ha_t|\mathbf{W1}_{1:t}, \mathbf{W2}_{1:t}, \mathbf{W4}_{1:t})$ for the `AND-gate` and $\Pr(hn_t|\mathbf{W1}_{1:t}, \mathbf{W2}_{1:t}, \mathbf{W4}_{1:t})$ for the `NOT-gate`.

3.3 Inference in Dynamic Bayesian Networks

We now shortly discuss some different approaches to perform inference in dynamic Bayesian networks, for a more elaborate overview we refer to the literature (e.g. Koller and Friedman (2009) and Murphy (2002)).

3.3.1 Unrolled Network

Unrolling the transition model of a DBN for a finite number of time slices T results in a network that is equivalent to a static Bayesian network. This allows one to perform inference with any standard exact or approximate algorithm for BNs (e.g. Darwiche (2009) and Koller and Friedman (2009)). Despite the wide range of existing algorithms, naively unrolling the network for T time slices has multiple drawbacks:

- The time complexity of inference depends on heuristics used by the algorithms and is not guaranteed to scale linearly with T.
- The space complexity is $O(T)$, i.e. the required memory depends on the time span T.
- The time span T is often unknown upfront and adding an extra time slice, to cope with new observations, requires to redo inference in the complete network.

The overcome one or more of these drawbacks, one has proposed a range of more specific approaches which we will discuss next.

3.3.2 Constant Space Algorithms

Inference algorithms for static Bayesian networks typically rely on general purpose heuristics, e.g. min-fill, to find a good variable ordering. By unrolling the network, however, these heuristics are not guaranteed to find a good ordering, leading to a non-linear time complexity when T increases. One way to overcome this behavior is by using a *constrained* or so called "slice-by-slice" variable ordering. This ordering forces all variables from slice t to appear before variables from slice $t+1$ and guarantees inference to scale linearly for increasing T.

Example 3.5 For a general DBN, the constrained or "slice-by-slice" ordering will be $\mathbf{z}_0 < \mathbf{z}_1 < \cdots < \mathbf{z}_T$. For our running example, in any possible constrained ordering, variable $w3_t$ will, for examples, appear before variable $w1_{t+1}$.

Besides a linear time complexity when T increases, constrained variable orderings also allow one to perform inference in DBNs with constant space, i.e. memory resources do not depend on T. This is obtained by a forward and backward pass trough the network where the conditional probability tables are dynamically generated and removed (Darwiche 2001a).

3.3.3 Conversion to Hidden Markov Models

It is known that one can represent any discrete-variable DBN by means of an hidden Markov model (HMM) (Russell and Norvig 2009). This is done by combining all state variables in the DBN into one single state variable whose values are all the possible tuples of values of the individual variables. After this conversion, one can rely on the well-known *forward-backward* algorithm for inference in HMMs (Rabiner 1989). This algorithm makes use of dynamic programming principles to efficiently compute the marginal probabilities by means of a forward and backward pass trough the network.

The conversion of a DBN into an HMM comes with a cost, however. A DBN allows one to decompose the state of the underlying process into a set of variables that can take advantage of sparseness in the transition model. This is not possible with an HMM and the number of values of the single state variable, as well as the corresponding transition matrix, typically explodes. Hence, the conversion into an HMM quickly becomes intractable for all but the smallest DBNs.

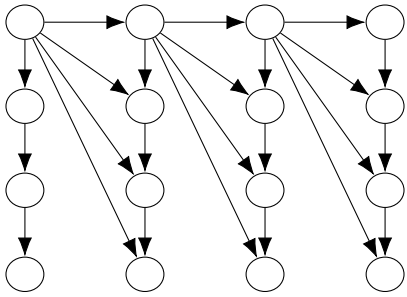


Figure 3.3: The "cascade" DBN.

Example 3.6 Consider the "cascade" DBN depicted in Figure 3.3 and adopted from Darwiche (2001a). If we assume all state variables are Boolean, the transition model has $4 + 8 + 8 + 8 = 28$ parameters. The corresponding HMM, however, would have 2^4 states and therefore $2^8 = 256$ parameters in the transition matrix.

3.3.4 Interface Algorithm

A key property of DBNs is that the hidden variables $\mathbf{x}_t$ d-separate the past from the future, that is, the belief state of the present makes the future independent of the past. Often, a subset $\mathbf{i}_t$ of $\mathbf{x}_t$ also suffices to d-separate the past from the future. This set $\mathbf{i}_t$, referred to as the *interface*, consists of the nodes from time slice $t-1$ that have an outgoing arc to nodes in time slice t (Murphy 2002).

Example 3.7 For the transition model depicted in Figure 3.1c, the interface is denoted with the dashed oval. It is easy to verify that knowing the values for these two variables suffices to d-separate the past from the future as they block all possible paths between variables from time slice t and $t-1$.

The forward-backward algorithm for HMMs is generalized towards DBNs to exploit the Markov property and d-separation in the network. The resulting approach, to which one refers as the *interface algorithm* (ibid.), additionally exploits the notion of the interface by first reducing the transition model from a 2TBN to an 1.5TBN. The latter is obtained by removing all non-interface variables and all arcs in the first time slice of the 2TBN. Then, the 1.5TBN is used to perform a forward and backward pass trough the network without the need to explicitly unroll the network.

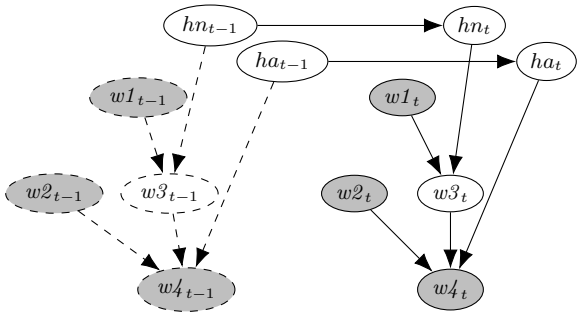


Figure 3.4: Reducing the transition model to an 1.5TBN.

Example 3.8 The transition model from our running example is reduced to a 1.5TBN by removing all dashed nodes and arrows as shown in Figure 3.4.

The forward pass involves computing the joint probability distributions $\Pr(\mathbf{i}_t|\mathbf{E}_{1:t})$ for every time step t. These distributions, referred to as the *forward messages*, can be computed recursively as follows:

$$\Pr(\mathbf{i}_t|\mathbf{E}_{1:t}) = \sum_{\mathbf{i}_{t-1}} \Pr(\mathbf{i}_t|\mathbf{i}_{t-1}, \mathbf{E}_t) \Pr(\mathbf{i}_{t-1}|\mathbf{E}_{1:t-1}) \tag{3.3}$$

and

$$\Pr(\mathbf{i}_t|\mathbf{i}_{t-1}, \mathbf{E}_t) = \sum_{\mathbf{z}_t \setminus \mathbf{i}_t} \Pr(\mathbf{z}_t|\mathbf{i}_{t-1}, \mathbf{E}_t) \tag{3.4}$$

The factor $\Pr(\mathbf{i}_t|\mathbf{i}_{t-1}, \mathbf{E}_t)$ can be computed on the 1.5TBN without the need to unroll the network. Then, for the task of filtering or prediction, marginal probabilities can be computed based on the forward messages as follows:

$$\Pr(z_T^i|\mathbf{E}_{1:\tau}) = \sum_{\mathbf{i}_{T-1}} \Pr(z_T^i|\mathbf{i}_{T-1}, \mathbf{E}_\tau) \Pr(\mathbf{i}_{T-1}|\mathbf{E}_{1:\tau-1})$$

It is important to note that the forward pass requires to compute the joint probability distribution over all variables in the interface. Computing a joint distribution is not a standard inference task in BNs, however, and typically requires some additional steps. Furthermore, computing the forward message for a certain time slice requires to incorporate the forward message from the previous time slice. The standard implementation of the interface algorithm[1] makes use of the *junction tree* algorithm to compute the messages (Murphy

[1] Available in the Bayes Net Toolbox for Matlab (https://github.com/bayesnet/bnt)

2002). With this approach, it is enforced that all variables in $\mathbf{i}_{t-1}$ each form a clique allowing one to condition the variables of the incoming interface on the incoming message $\Pr(\mathbf{i}_{t-1})$. Also the variables in $\mathbf{i}_t$ need to from a clique, allowing one to compute the outgoing message $\Pr(\mathbf{i}_t)$.

Example 3.9 The advantage of computing the forward and backward pass on the DBN, rather than on the corresponding HMM, can be easily seen for the "cascade" DBN depicted in Figure 3.3. This network only contains one interface variable and it suffices to recursively compute the probability distribution for this single (binary) variable. Remember that the single state variable in the corresponding HMM would have 2^4 values.

While performing a forward pass suffices for filtering and prediction, the task of smoothing also requires a backward pass. Computing the backward message $\Pr(\mathbf{i}_t|\mathbf{E}_{1:T})$ can be done on the 1.5TBN in an analogues way as computing the forward message, as shown by Murphy (ibid.). While the literature often distinguishes a forward and backward interface, to perform the forward and backward pass, the interface algorithm does not require to make this difference.

3.4 The Structural Interface Algorithm

We now introduce the *structural interface algorithm* for exact probabilistic inference in dynamic Bayesian networks. It unifies state-of-the-art techniques for inference in static and dynamic networks, by combining principles of *knowledge compilation* and *weighted model counting* with the *interface algorithm*. The resulting algorithm not only exploits the repeated structure, but also the local structure in the distribution induced by the network.

We will first show how the encode the transition model into a propositional knowledge base for weighted model counting. Next, we explore several approaches to encode the interface and to actually perform inference. To demonstrate our approach, we will focus on the transition model as this is the one that repeats over time. Computing messages for the first time slice, i.e. the prior distribution, can be done in a similar way.

3.4.1 Encoding the Transition Model

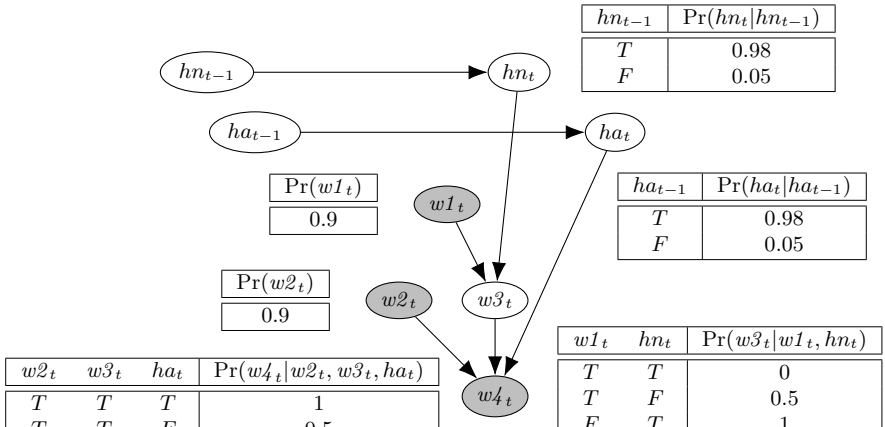
Following the interface algorithm, we start by reducing the 2TBN transition model to an 1.5TBN. As noted earlier, the 2TBN and 1.5TBN are not regular Bayesian networks as nodes from the first time slice do not have an associated conditional probability distribution. For the 1.5TBN, nodes without a CPT belong, by definition, to the incoming interface $\mathbf{i}_{t-1}$.

To represent the 1.5TBN by means of a propositional knowledge base $\Sigma_{1.5}$, we use the encoding as proposed in Section 2.2.4. Each of the CPTs is encoded into a formula and $\Sigma_{1.5}$ consists of the conjunction of these formulas. Variables from the incoming interface $\mathbf{i}_{t-1}$ are not explicitly encoded into a formula as they do not have a corresponding CPT. These variables will have a corresponding indicator variable in $\Sigma_{1.5}$, however, as they appear as a parent for at least one of the nodes of the second time slice of the 1.5TBN.

Example 3.10 Let us consider the 1.5TBN for our running example as depicted in Figure 3.5. The encoding of the transition model as a propositional formula gives us the following knowledge base (where we exploit local structure):

$$
\begin{aligned}
&w1_t \Leftrightarrow p\_w1_t \\
&w2_t \Leftrightarrow p\_w2_t \\
&w3_t \Leftrightarrow (\neg w1_t \wedge hn_t) \\
&\qquad\qquad \vee (\neg hn_t \wedge p\_faulty\_not_t) \\
&w4_t \Leftrightarrow (w2_t \wedge w3_t \wedge ha_t) \\
&\qquad\qquad \vee (\neg ha_t \wedge p\_faulty\_and_t) \\
&hn_t \Leftrightarrow (hn_{t-1} \wedge p\_hn\_row1_t) \\
&\qquad\qquad \vee (\neg hn_{t-1} \wedge p\_hn\_row2_t) \\
&ha_t \Leftrightarrow (ha_{t-1} \wedge p\_ha\_row1_t) \\
&\qquad\qquad \vee (\neg ha_{t-1} \wedge p\_ha\_row2_t)
\end{aligned}
$$

The first two equivalences define the distribution of the input wires of the circuit, the third and fourth equivalence define the behavior of the `NOT-gate` and `AND-gate`, respectively, and the final two equivalences define the dynamics of the components' health state.



hn_{t-1}	$\Pr(hn_t \mid hn_{t-1})$
T	0.98
F	0.05

$\Pr(w1_t)$
0.9

ha_{t-1}	$\Pr(ha_t \mid ha_{t-1})$
T	0.98
F	0.05

$\Pr(w2_t)$
0.9

$w1_t$	hn_t	$\Pr(w3_t \mid w1_t, hn_t)$
T	T	0
T	F	0.5
F	T	1
F	F	0.5

$w2_t$	$w3_t$	ha_t	$\Pr(w4_t \mid w2_t, w3_t, ha_t)$
T	T	T	1
T	T	F	0.5
T	F	T	0
T	F	F	0.5
F	T	T	0
F	T	F	0.5
F	F	T	0
F	F	F	0.5

Figure 3.5: The 1.5TBN for our running example and corresponding CPTs.

3.4.2 Computing the Forward Message

Our approach performs inference in a DBN by recursively computing the forward message $\Pr(\mathbf{i}_t|\mathbf{i}_{t-1}, \mathbf{E}_t)$ on the 1.5TBN by means of weighted model counting on the propositional knowledge base $\Sigma_{1.5}$. This does not only involve encoding, then compiling, the 1.5TBN, but also requires to represent the joint distributions $\Pr(\mathbf{i}_{t-1})$, to condition the transition model on the incoming message, and $\Pr(\mathbf{i}_t)$, to compute the outgoing message.

We explore several approaches to represent the required distributions, i.e. to integrate the interface algorithm with knowledge compilation and weighted model counting. They have different memory requirements and trade-offs between putting the burden on the compiler, a post-compilation (conditioning) step or the inference step. Table 3.2 summarizes the complexity of the different steps for each of the different interface encodings we present below.

For notational convenience, and because of analogy between the forward and backward pass, we focus on computing the forward message. Furthermore, we omit the observations $\mathbf{E}_{1:t}$ in the remainder of this chapter and refer to the forward message as $\Pr(\mathbf{i}_t)$. Its different entries (possible variable instantiations)

	ENC1	ENC2	ENC3	ENC4
Compilation	$\mathcal{O}(2^{\omega_{1.5\text{TBN}}+2\cdot\lvert\mathbf{i}\rvert})$	$\mathcal{O}(2^{\omega_{1.5\text{TBN}}})$	$\mathcal{O}(2^{\omega_{1.5\text{TBN}}})$	$\mathcal{O}(2^{\omega_{1.5\text{TBN}}})$
Conditioning	n\a	$2\cdot 2^{\lvert\mathbf{i}\rvert}\cdot\mathcal{O}(\lvert\Delta\rvert)$	n\a	$2^{\lvert\mathbf{i}\rvert}\cdot\mathcal{O}(\lvert\Delta\rvert)$
Evaluation	$2\cdot\mathcal{O}(\lvert\Delta\rvert)$	$2\cdot\mathcal{O}(\lvert\Delta\rvert)$	$2^{2\cdot\lvert\mathbf{i}\rvert}\cdot\mathcal{O}(\lvert\Delta\rvert)$	$2^{\lvert\mathbf{i}\rvert}\cdot\mathcal{O}(\lvert\Delta\rvert)$

Table 3.2: Complexity of each step for the different interface encodings. Parameter ω_N represents the treewidth of network N. Circuit Δ refers to the one constructed in the previous step. For Conditioning and Evaluation, we report the asymptotic complexity of one call ($\mathcal{O}(|\Delta|)$), multiplied by the number of required calls (e.g. 2).

are denoted by $(\mathbf{I}_t^1, \mathbf{I}_t^2, \ldots, \mathbf{I}_t^M)$. In case all variables are binary, we have $M = 2^{|\mathbf{i}|}$. We assume the target language for compilation is d-DNNF, but the methods also apply to any of its sub-languages.

Compiling the Interface into the Circuit (ENC1)

A joint distribution $\Pr(\mathbf{i})$ can be naturally encoded into a knowledge base $\Sigma_{\mathbf{i}}$ in a similar ways as discussed in Section 2.2.4. This requires $2^{|\mathbf{i}|}$ formulas and indicator variables to be added, all in one-to-one correspondence to the rows of $\Pr(\mathbf{i})$. Following this method, we end-up with three knowledge bases being $\Sigma_{\mathbf{i}_{t-1}}$ to encode $\Pr(\mathbf{i}_{t-1})$, $\Sigma_{\mathbf{i}_t}$ to encode $\Pr(\mathbf{i}_t)$ and $\Sigma_{1.5}$ to encode the 1.5TBN.

Example 3.11 For our running example, with variables hn_{t-1} and ha_{t-1} in the incoming interface, $\Sigma_{\mathbf{i}_{t-1}}$ is given by the following 4 formulas (and similar for the outgoing interface):

$$state^1_{t-1} \Leftrightarrow hn_{t-1} \wedge ha_{t-1} \qquad \text{(for } \mathbf{I}^1_{t-1}\text{)}$$

$$state^2_{t-1} \Leftrightarrow hn_{t-1} \wedge \neg ha_{t-1} \qquad \text{(for } \mathbf{I}^2_{t-1}\text{)}$$

$$state^3_{t-1} \Leftrightarrow \neg hn_{t-1} \wedge ha_{t-1} \qquad \text{(for } \mathbf{I}^3_{t-1}\text{)}$$

$$state^4_{t-1} \Leftrightarrow \neg hn_{t-1} \wedge \neg ha_{t-1} \qquad \text{(for } \mathbf{I}^4_{t-1}\text{)}$$

Once we have obtained the three knowledge bases, we can conjoin them and compute the forward message as follows:

$$(\Pr(\mathbf{I}_t^1), \ldots, \Pr(\mathbf{I}_t^n)) = \textsc{Eval}_{\uparrow\downarrow}(\textsc{Compile}(\Sigma_{\mathbf{i}_{t-1}} \wedge \Sigma_{1.5} \wedge \Sigma_{\mathbf{i}_t}), w)$$

where the weight function w is updated to incorporate the incoming message. For our example, this would be with $w(state_{t-1}^j) = (\Pr(\mathbf{i}_{t-1}^j), 1)$.

The **advantage** of this encoding is that, for each time step, only two passes trough the circuit are needed to compute the forward message (i.e., one call to $\textsc{Eval}_{\uparrow\downarrow}$). The **disadvantage** is that the number of required formulas to encode $\Pr(\mathbf{i})$ scales exponentially in $|\mathbf{i}|$ (i.e. the number of interface variables).

Conditioning the Interface into the Circuit (ENC2)

The exponential aspect of $\Sigma_{\mathbf{i}}$ and $\Sigma_{\mathbf{i}_{t-1}}$ has an adverse effect on the heuristics used by general-purpose compilation tools as it not only dwarfs $\Sigma_{1.5}$ in size, but also represents a joint distribution without any local structure. A d-DNNF that is logically equivalent with the one obtained by $\textsc{Compile}(\Sigma_{1.5} \wedge \Sigma_{\mathbf{i}} \wedge \Sigma_{\mathbf{i}_{t-1}})$ can be obtained by only compiling $\Sigma_{1.5}$ with a general-purpose tool and adding $\Pr(\mathbf{i})$ and $\Pr(\mathbf{i}_{t-1})$ to the resulting circuit by means of conditioning (cf. Section 2.2.2). Concretely, a joint distribution over all variables in $\mathbf{i}$ can be added to a compiled circuit Δ in the following way:

$$\textsc{Addi}(\Delta, \mathbf{i}) = \bigvee_{\mathbf{I}^j \in \mathbf{i}} (\Delta | \mathbf{I}^j) \wedge state^j \wedge \mathbf{I}^j \tag{3.5}$$

Example 3.12 For our running example, with variables hn_{t-1} and ha_{t-1} in the incoming interface, calling $\textsc{Addi}(\Delta, \mathbf{i}_{t-1})$ results in the following formula (and similar for the outgoing interface):

$$\begin{aligned}
&(\Delta |\ \ hn_{t-1} \wedge \ \ ha_{t-1}) \wedge state^1 \wedge \ \ hn_{t-1} \wedge \ \ ha_{t-1} \\
\vee &(\Delta |\ \ hn_{t-1} \wedge \neg ha_{t-1}) \wedge state^2 \wedge \ \ hn_{t-1} \wedge \neg ha_{t-1} \\
\vee &(\Delta | \neg hn_{t-1} \wedge \ \ ha_{t-1}) \wedge state^3 \wedge \neg hn_{t-1} \wedge \ \ ha_{t-1} \\
\vee &(\Delta | \neg hn_{t-1} \wedge \neg ha_{t-1}) \wedge state^4 \wedge \neg hn_{t-1} \wedge \neg ha_{t-1}
\end{aligned}$$

The result of $\textsc{Addi}(\Delta, \mathbf{i})$ is a d-DNNF which allows us to compute the forward message as follows:

$$(\Pr(\mathbf{I}_t^1), \ldots \Pr(\mathbf{I}_t^n)) = \textsc{Eval}_{\uparrow\downarrow}(\textsc{Addi}(\textsc{Addi}(\textsc{Compile}(\Sigma_{1.5}), \mathbf{i}_{t-1}), \mathbf{i}_t), w)$$

where the weight function w is updated with $w(state_{t-1}^j) = (\Pr(\mathbf{I}_{t-1}^j), 1)$.

The **advantage** of incorporating $\Pr(\mathbf{i})$ directly into the d-DNNF is that the heuristic of the compiler does not have to deal with $\Sigma_{\mathbf{i}}$ and can focus on better compiling the much smaller and more structured sentence $\Sigma_{1.5}$. Furthermore, this approach allows one to share identical subcircuits, leading to an efficient computation of the forward message with only two passes trough the obtained circuit. The **disadvantage** is that the number of conditioning operations scales exponentially with $|\mathbf{i}|$.

Introducing the Interface as Evidence (ENC3)

We can compute the forward message using only $\Delta_{1.5}$, i.e. the circuit obtained by $\textsc{Compile}(\Sigma_{1.5})$, without the need to explicitly encode $\Pr(\mathbf{i}_{t-1})$ and $\Pr(\mathbf{i}_t)$. This is done by repeatedly updating the weight function to incorporate each of the combinations of instantiations of $\Pr(\mathbf{i}_{t-1})$ and $\Pr(\mathbf{i}_t)$ as evidence (see Step (2), section 2.2.4). Concretely, the probability of the j-th instantiation in the forward message can be computed in the following way:

$$\Pr(\mathbf{I}_t^j) = \sum_{k=0}^{M} \textsc{Eval}_{\uparrow}(\textsc{Compile}(\Sigma_{1.5}), w_{k \to j}) \cdot \Pr(\mathbf{I}_{t-1}^k)$$

where $w_{k \to j}$ incorporates the instantiations $\mathbf{I}_{t-1}^k$ and $\mathbf{I}_t^j$ and $M = 2^{|\mathbf{i}|}$ in case all interface variables are binary.

Example 3.13 For our running example, assume we have $\mathbf{I}_t^j = hn_t \wedge \neg ha_t$ and $\mathbf{I}_{t-1}^k = \neg hn_{t-1} \wedge ha_{t-1}$, the weight function $w_{k \to j}$ would set the weight to the variables in the interface as follows:

$$w(hn_t) = (1, 0)$$

$$w(ha_t) = (0, 1)$$

$$w(hn_{t-1}) = (0, 1)$$

$$w(ha_{t-1}) = (1, 0)$$

The **advantage** of bypassing an explicit encoding of the interfaces is that it lowers the memory requirements as the forward message is directly computed

on the circuit $\Delta_{1.5}$. The **disadvantage** is that computing the forward message requires $2^{2\cdot|\mathbf{i}|}$ passes trough the circuit. Moreover, $2^{2\cdot|\mathbf{i}|} \cdot |\Delta_{1.5}|$ will be larger than $2\cdot|\Delta|$ (the evaluation step of the previous two encodings) because identical subcircuits are note shared.

Encoding for the Structural Interface Algorithm (ENC4)

The approach of compiling $\Sigma_{\mathbf{i}_{t-1}} \wedge \Sigma_{1.5} \wedge \Sigma_{\mathbf{i}_t}$ (ENC1) is similar to the interface algorithm where one adds edges to the moral graph between all nodes in $\mathbf{i}_{t-1}$ and $\mathbf{i}_t$ (Murphy 2002). Since the compilation step is the most complex step in the weighted model counting pipeline, and this approach potentially has to deal with a more complex knowledge base, we do not prefer this encoding.

For the structural interface algorithm, we propose a hybrid encoding that employs ENC2 as well as ENC3. Concretely, we explicitly introduce $\Pr(\mathbf{i}_{t-1})$ by conditioning while $\Pr(\mathbf{i}_t)$ is implicitly introduced as evidence. This allows us to compute the probability of the j-th instantiation in the forward message as follows:

$$\Pr(\mathbf{I}_t^j) = \textsc{Eval}_{\uparrow}(\textsc{Addi}(\textsc{Compile}(\Sigma_{1.5}), \mathbf{i}_{t-1}), w_{\rightarrow j}) \tag{3.6}$$

where $w_{\rightarrow j}$ is updated with $w(state_{t-1}^k) = (\Pr(\mathbf{I}_{t-1}^k), 1)$ and incorporates the instantiation $\mathbf{I}_t^j$. For each time slice, $2^{|\mathbf{i}|}$ passes trough the circuit are required to compute the forward message.

The **advantage** of this encoding is that it combines the advantages of ENC2 and ENC3. More precisely, the benefit of evaluating the circuit multiple times (ENC3) is that the cost of compilation is amortized over all queries. The benefit of conditioning (ENC2) is that subcircuits and computations are shared. By using the hybrid approach, we get some of both advantages, which we will empirically show to be a good trade-off.

3.5 Optimizations

The use of knowledge compilation to compute the forward message does not only allow us to exploit local structure, but with some additional optimization we also fully exploit the repeated structure in the network.

3.5.1 Static Structure

By definition, the structure of the transition model is time-invariant and there is no need to repeat the process of encoding and compiling the 1.5TBN and to introduce $\Pr(\mathbf{i}_{t-1})$. This allows us to split Equation 3.6 in two parts:

$$\Delta_R = \textsc{Addi}(\textsc{Compile}(\Sigma_{1.5}), \mathbf{i}_{t-1}), \tag{3.7}$$

which is performed only once, and

$$\Pr(\mathbf{I}_t^j) = \textsc{Eval}_{\uparrow}(\Delta_R, w_{\rightarrow j}) \tag{3.8}$$

which is performed for each $\mathbf{I}_t^j \in \mathbf{i}_t$ and for each $t < T$. Hence, the one-time cost of Equation 3.7 is amortized over $2^{|\mathbf{i}|} \cdot T$ queries.

In general, one assumes that not only the structure of the transition model but also the parameters are time-invariant. This is not a strict requirement for our approach, however. In case the parameters would change over time, we only have to update the weight function without the need to recompile the model.

3.5.2 Repeated Counting

Computing the forward message by means of Equation 3.8 requires an update of the weight-function w before any new evaluation pass trough Δ_R. Some variables in the d-DNNF, however, are mapped to time-invariant weights that never change. We have variables with time-invariant weights in case the following two conditions are met: (1) the variable is not observed and, (2) not queried. In general, all of the parameter variables and a subset of the indicator variables meet these two conditions.

In case a subformula only contains variables with time-invariant weights, its weighted model count will remain the same for each evaluation pass trough the compiled circuit. Hence, it suffices to compute the WMC of this subformula only once and cache the value for subsequent computations. Another way to deal with this is to combine variables with time-invariant weights and replace them by a smaller set of variables.

Example 3.14 For our running example (see Figure 3.1), variable $w3_t$ models the state of *wire 3* and is a purely internal variable that is never queried or observed. Assume we have a d-DNNF which contains the following subformula

and weight function:

$$w4_t \wedge (w3_t \wedge p\_faulty\_and_t) \qquad \text{with} \qquad \begin{cases} w(w3_t) & = a \\ w(p\_faulty\_and_t) & = b \end{cases}$$

This can be replaced by:

$$w4_t \wedge p\_new_t \qquad \text{with} \qquad w(p\_new_t) = a \cdot b$$

The effect of this transformation is that it reduces the number of unnecessary computations in each pass trough the circuit. If we would not employ this transformation, the multiplication $a \cdot b$ will be performed $T \cdot 2^{|\mathbf{i}|}$ times although the result will always be the same. This transformation can be performed in a deterministic manner by means of one bottom-up pass trough the d-DNNF. As it only needs to be computed once, i.e. before the evaluation step, the cost is amortized over $2^{|\mathbf{i}|} \cdot T$ queries

3.6 Experiments

Our experiments address the following four questions:

Q1 How do different algorithms scale with an increasing number of time steps?

Q2 How do both of the interface algorithms scale in the presence of local structure in the transition model?

Q3 How does the structural interface algorithm scale in case local structure is not fully exploited?

Q4 How do the different interface encodings compare?

We implemented our algorithm in ProbLog[2]. For compilation, we use both the c2d[3] and dsharp[4] compilers, and retain the smallest circuit. Experiments are run with a working memory of 8 GB and a timeout of 1 hour.

[2]Available at http://dtai.cs.kuleuven.be/problog/
[3]Available at http://reasoning.cs.ucla.edu/c2d/
[4]Available at https://bitbucket.org/haz/dsharp

3.6.1 Models

We generate networks for the following three domains:

Digital Circuit 1 These networks model electronic digital circuits similar to the one used as running example in this text (and adopted from Darwiche (2009)). A circuit contains logical `AND-gates` and `OR-gates` which all are randomly connected to each other (without forming loops). For a subset of logical gates, the input or output is observed and not connected to another gate. The interface contains all variables that model the health state of the component. Gates can share a health variable when, for example, they share a power line. We refer to the networks as `DC1-G-H`, with `G` the number of gates and `H` the number of health (interface) variables. The number of gates for which the input or output is observed is $2 \cdot \frac{G}{H}$. Observations are generated randomly. For each domain size, we randomly generate 3 networks and report average results.

Digital Circuit 2 These networks are a variant of the networks in `DC1` but now we have a separate health variable for each of the gates and the interface consists of one multi-valued variable. This variable aggregates all health variables and encodes, in an ordered way, which gate is most likely to be part of the failing gates. The introduction of the multi-valued variable facilitates the encoding of the interface, as compared to `DC1`, but offers an additional challenge for inference as it directly depends on each of the health variables. We refer to the networks as `DC2-G` with `G` the number of gates. For each domain size, we randomly generate 3 networks and report average results.

Mastermind We model the mastermind game, similar to the BNs used in Chavira, Darwiche, and Jaeger (2006). Instead of modeling the game for a fixed number of rounds, however, we represent the game as a DBN with one time slice per round. The interface contains a variable for each of the pegs the game is played with. The interface thus models the belief of the colors set by the opponent for each of the pegs. We refer to the networks as `MM-C-P`, with `C` the number colors and `P` the number of pegs (interface variables).

3.6.2 Algorithms

We make use of the following four algorithms:

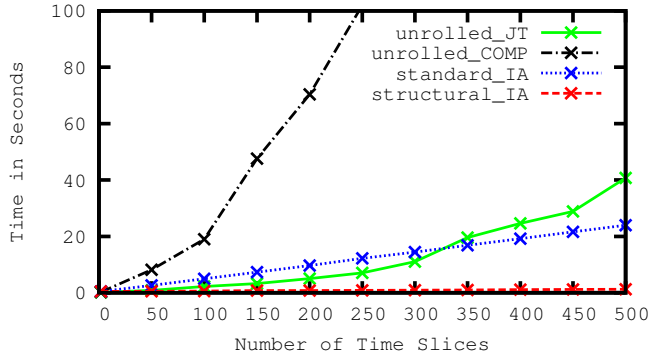
- `unrolled_JT` : The *junction tree algorithm* on the unrolled network for which we used SMILE[5].
- `unrolled_COMP` : Compiling the unrolled network, using the encoding introduced in Section 2.2.4.
- `standard_IA` : The standard *interface algorithm*[6] where we experimented with the `jtree_dbn_inf_engine` as well as with the `smoother_engine` but did not observe any significant difference.
- `structural_IA` : The *structural interface algorithm* where the interface is encoded using ENC4.

3.6.3 Results

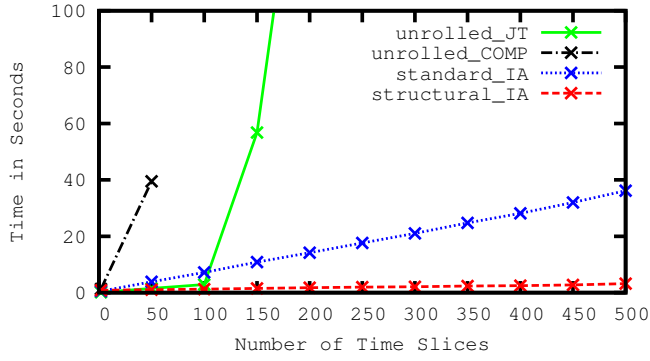
We compare the four algorithms introduced above for an increasing number of time slices. The results are depicted in Figure 3.6 and allow us to answer **(Q1)**. On each of the three domains, both of the interface algorithms scale linear with the number of time steps while this is not the case for inference in the unrolled network. This shows that, especially for a large number of time slices, the general-purpose heuristics fail to find a good variable ordering. We do observe, however, that `unrolled_JT` is more efficient, compared to `standard_IA`, when the number of time slices is rather small. The reason for this is that `standard_IA` has to deal with an extra constraint, being that all variables in the interface have to be in the same clique, which initially causes some overhead. Furthermore, `unrolled_JT` outperforms `unrolled_COMP` on each of the three domains despite the local structure present in the networks. Hence, no guarantees can be provided when a general-purpose implementation is used to perform inference in the unrolled network.

We compare `standard_IA` and `structural_IA` for the task of computing the forward message for 10 time slices (note that for both of the interface algorithms, the complexity of inference is not influenced by the number of time slices). The results are depicted in Table 3.3 and serve as an answer to **Q2**. The structural interface algorithm, which exploits local structure, successfully performs inference on all of the networks while this is not the case

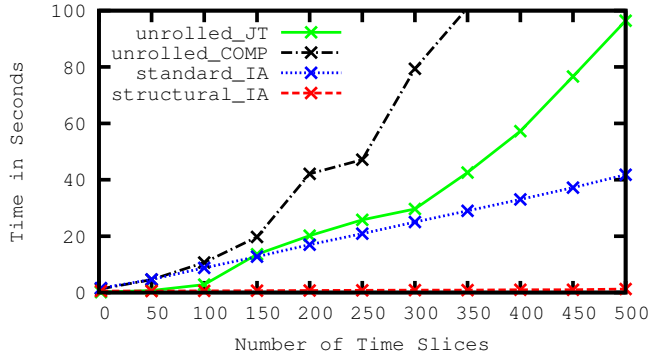
[5] Available at `http://genie.sis.pitt.edu/`
[6] Available at `https://github.com/bayesnet/bnt`



(a) DC1-20-5



(b) DC2-12



(c) MM-3-3

Figure 3.6: Total inference time for an increasing number of time slices.

for the standard interface algorithm. Furthermore, this table indicates that `structural_IA` works well in case the transition model is complex while the number of variables in the interface is rather limited. For example, `DC1-90-9` requires more compilation and evaluation time than `DC1-112-7`, although the latter contains more variables. This is explained by the exponential behavior of the interface.

We explore the effect of exploiting local structure by the CNF encoding when compiling the network. The results are depicted in Table 3.5 and serve as an answer to **Q3**. Concretely, we consider a CNF encoding that does not exploit any local structure, a CNF encoding that only exploits determinism and a CNF encoding that exploits determinism as well as equal parameters. We observe that, in case no local structure is exploited, the transition model is much harder to compile and results in very large circuits. Moreover, `standard_IA` clearly outperforms `structural_IA` in case the latter does not exploit local structure. Only exploiting determinism significantly simplifies the compilation process but, for most networks, we can still benefit from also exploiting equal parameters. For each of the three problem domains, only the results for the two smallest networks is depicted as inference in the more complex networks requires to exploit all local structure.

We compare the four different interface encodings proposed in Section 3.4.2. The results are shown in Table 3.4 and let us answer **Q4**. We first observe that ENC4, i.e. the encoding we propose for the structural interface algorithm, is the only encoding that successfully performs inference in each of the networks. Second, the mastermind experiment illustrates that compiling the knowledge base is harder when using ENC1, as was suggested by the complexity indicated in Table 3.2. Third, the compilation step for ENC3 is the most efficient one, as it does not compile the interface. Computing the forward message, however, is in general much slower compared to the other encodings, as also indicated in Table 3.2. Fourth, although the d-DNNF for ENC3 does not encode the interface, its size is in general not smaller compared to the other encodings. The reason for this is that by explicitly encoding the interface we actually do not increase the number of models in the d-DNNF but rather add extra constraints on the models already present. Hence, explicitly encoding the interface might increase the total compilation time but significantly reduces the evaluation time.

As a final remark, we would like to note that each of the experiments finishes rather fast compared to the given timeout (1 hour). Hence, the choice of the timeout has only a minimal impact on our conclusions.

Model	1.5TBN Vars	Max Clust	Card	Avg Card	$\Sigma_{1.5}$ Vars	clauses	d-DNNF Edges $\Delta_{1.5}$ #edges ×1000	Δ_R #edges ×1000	d-DNNF Time comp (s)	Rinf (s)	standard_IA Tinf (s)
DC1-G-H											
20 - 5	34	16	2-2	2.0	146	373	12	3	0.1	0.002	1
30 - 5	46	21	2-2	2.0	216	597	31	6	0.2	0.003	4
40 - 5	58	⩾27	2-2	2.0	282	801	61	11	0.3	0.005	-
60 - 6	82	⩾29	2-2	2.0	416	1,214	372	57	2.0	0.03	-
70 - 7	94	⩾28	2-2	2.0	482	1,413	1,235	145	6.8	0.1	-
112 - 7	142	⩾29	2-2	2.0	770	2,343	6,870	959	42.3	0.7	-
80 - 8	106	⩾32	2-2	2.0	548	1,612	3,059	333	18.0	0.3	-
104 - 8	133	⩾29	2-2	2.0	704	2,105	9,886	1,090	61.7	1.2	-
90 - 9	118	⩾29	2-2	2.0	614	1,811	9,051	851	56.0	1.5	-
DC2-G											
12	30	16	2-13	2.7	157	615	26	7	0.3	0.003	1
16	38	20	2-17	2.8	209	963	95	16	0.7	0.007	9
20	46	⩾22	2-21	2.8	261	1,375	310	38	2.2	0.02	-
24	54	⩾26	2-25	2.9	313	1,851	643	106	5.0	0.05	-
28	62	⩾30	2-29	2.9	365	2,391	3,050	376	23.1	0.2	-
32	70	⩾34	2-33	2.9	417	2,995	7,300	668	57.1	0.4	-
MM-C-P											
3 - 3	59	11	2-3	2.2	147	447	62	2	0.2	0.001	1
6 - 3	59	11	2-6	2.6	210	699	519	24	1.3	0.02	2
9 - 3	59	11	2-9	3.1	273	1,032	1,944	88	4.9	0.1	3
4 - 4	99	⩾20	2-4	2.2	293	1,058	4,590	55	8.7	0.05	-
6 - 4	99	⩾20	2-6	2.5	357	1,326	27,656	361	55.2	1.2	-
8 - 4	99	⩾20	2-8	2.7	421	1,642	98,120	1,350	220.7	13.6	-
3 - 5	149	⩾25	2-3	2.1	417	1,769	13,234	75	23.6	0.07	-
4 - 5	149	⩾25	2-4	2.2	462	1,934	58,467	519	128.6	1.7	-

Table 3.3: Results for computing the forward message for 10 time slices with `structural_IA` and `standard_IA`. *Max Clust* denotes the biggest cluster in the junction tree and *(Avg) Card* denotes the (average) cardinality of the variables in the transition model. *comp* includes the compilation time, conditioning time and the time to simplify the circuit (cf. Section 3.5.2). *Rinf* denotes the time needed with `structural_IA` to compute the forward message for one time slice. *Tinf* denotes the total inference needed by `standard_IA`.

Model	ENC1 Δ_R #edges	ENC1 comp (s)	ENC1 Rinf (s)	ENC2 Δ_R #edges	ENC2 comp (s)	ENC2 Rinf (s)	ENC3 Δ_R #edges	ENC3 comp (s)	ENC3 Rinf (s)	ENC4 Δ_R #edges	ENC4 comp (s)	ENC4 Rinf (s)
DC1-G-H	×1000			×1000			×1000			×1000		
20 - 5	5	0.2	0.004	15	0.2	0.01	1	0.09	0.02	3	0.1	0.002
30 - 5	7	0.2	0.006	18	0.3	0.02	4	0.2	0.02	6	0.2	0.003
40 - 5	13	0.4	0.01	24	0.5	0.02	10	0.3	0.05	11	0.3	0.005
60 - 6	63	2.3	0.05	116	2.5	0.1	54	1.9	0.8	57	2.0	0.03
70 - 7	171	8.4	0.1	389	9.3	0.4	136	6.7	6.6	145	6.8	0.1
112 - 7	975	45.6	0.8	1,222	44.9	1.0	950	42.2	48.7	959	42.3	0.7
80 - 8	416	24.1	0.4	1,386	28.2	1.4	314	17.9	56.4	333	18.0	0.3
104 - 8	1,172	73.7	1.0	2,160	71.8	2.0	1,070	62.0	212.8	1,090	61.7	1.2
90 - 9	1,140	86.3	1.0	5,317	99.6	5.7	807	55.7	-	851	56.0	1.5
DC2-G												
12	9	0.2	0.007	10	0.3	0.009	8	0.2	0.01	7	0.3	0.003
16	18	0.6	0.02	22	0.9	0.02	17	0.6	0.04	16	0.7	0.007
20	42	1.9	0.03	48	2.4	0.04	40	1.8	0.1	38	2.2	0.02
24	113	4.6	0.09	124	5.5	0.1	110	4.4	0.4	106	5.0	0.05
28	387	22.5	0.3	403	23.9	0.3	382	22.2	1.7	376	23.1	0.2
32	684	56.2	0.5	707	58.1	0.6	677	55.5	3.7	668	57.1	0.4
MM-C-P												
3 - 3	1	0.2	0.001	2	0.2	0.001	2	0.2	0.02	2	0.2	0.001
6 - 3	21	4.9	0.01	24	5.8	0.01	20	1.1	3.0	24	1.3	0.02
9 - 3	75	164.7	0.04	88	92.17	0.05	56	3.7	68.8	88	4.9	0.1
4 - 4	50	11.6	0.03	54	14.0	0.03	77	8.5	13.6	55	8.7	0.05
6 - 4	495	1024.5	0.4	360	275.3	0.2	396	53.3	-	361	55.2	1.2
8 - 4	-	-	-	-	-	-	1,312	198.5	-	1,350	220.7	13.6
3 - 5	6	11.5	0.03	74	27.0	0.04	171	22.7	27.9	75	23.6	0.07
4 - 5	1,401	929.4	4.7	515	228.0	0.3	879	125.0	-	519	128.6	1.7

Table 3.4: A comparison of the different encodings for the interface. *comp* includes the compilation time, conditioning time (if applicable) and the time to simplify the circuit (cf. Section 3.5.2). *Rinf* denotes the time needed to compute the forward message for one time slice.

Model	No Local Structure Δ_R #edges	No Local Structure comp (s)	Only Det Δ_R #edges	Only Det comp (s)	Det & Equal Par Δ_R #edges	Det & Equal Par comp (s)	standard_IA Tinf (s)
DC1-G - H	×1000		×1000		×1000		
20 - 5	4,262	10.4	906	2.6	17	0.1	1
30 - 5	193,594	588.6	19,789	48.6	36	0.2	4
DC2-G							
12	-	-	1,289	2.5	26	0.1	1
16	-	-	2,039	4.1	95	0.4	9
MM-C-P							
3 - 3	1,096	3.2	15	0.3	38	0.1	1
6 - 3	-	-	-	-	441	4.5	2

Table 3.5: A comparison of different levels of exploiting local structure in the transition model. We use interface encoding ENC1 and do not simplify the circuit. Hence, *comp* only includes compilation time. *Tinf* denotes the total inference needed by `standard_IA` to compute the forward message for 10 time slices.

3.7 Related Work

Standard algorithms for exact inference on Bayesian networks (Darwiche 2001c; Dechter 1996; Jensen, Lauritzen, et al. 1990; Lauritzen and Spiegelhalter 1988; Zhang and Poole 1996) only exploit topological structure in the form of conditional independencies and their complexity is known to be exponential in the treewidth of the network. Networks exhibiting high treewidth are not necessarily difficult for exact inference, however, in case they also exhibit a certain amount of local structure in the form of determinism (Jensen and Anderson 1990) and context-specific independence (Boutilier et al. 1996). Exponential speed gains can be obtained by exploiting this local structure and the complexity of inference is only in worst-case exponential in the treewidth, i.e. when no local structure exists.

The literature proposes a wide range of approaches for exploiting local structure, e.g. Chavira and Darwiche (2007), Darwiche (2002), Larkin and Dechter (2003), Poole and Zhang (2011), and Sang et al. (2005b). In this chapter, we used a subset of these algorithms which reduce the problem of inference into one of weighted model counting. We can distinguish two approaches, performing WMC on the knowledge base by means of search (Sang, Bacchus, et al. 2004; Sang et al. 2005a,b) or by means of knowledge compilation (Chavira and Darwiche 2005; Darwiche 2001b). The latter can be done by compiling the knowledge base into a OBDD representation (Brace et al. 1990), d-DNNF representation (Darwiche 2004; Muise et al. 2010) or SDD representation (Darwiche 2011; Oztok and Darwiche 2015).

Exact inference for hidden Markov models is based on principles of dynamic

programming and exploits the fact that the belief state of the present makes the past independent of the future (Rabiner 1989). This approach was first generalized towards dynamic Bayesian networks by Zweig (1996) and is referred to as the frontier algorithm. The notion of a frontier was also used by Kjaerulff (1995) and later specified to the interface by Murphy (2002) and Darwiche (2001a).

Exact inference in DBNs requires one to fully compute and represent the belief state over all variables in the interface. This quickly becomes intractable when dealing with real-life applications and one often has to resort to approximate inference methods. A popular approach is to use a factored representation of the belief state which can be computed in a more efficient way (Boyen and Koller 1998; Murphy and Weiss 2001). Another approach is to rely on sampling such as sequential Monte Carlo methods (Doucet and Freitas 2001) or Rao-Blackwellized particle filters (Doucet, Freitas, et al. 2000).

3.8 Discussion

In this chapter, we proposed the structural interface algorithm for exact probabilistic inference in dynamic Bayesian networks. It relies on knowledge compilation and weighted model counting to exploit the Markov property in the network as well as local structure in the transition model. We have shown that our approach can tackle dynamic models that are considerably more complex than what can currently be dealt with by exact inference techniques. We have experimentally shown this on two classes of problems, namely finding failures in an electronic circuit and performing filtering in the mastermind game.

The use of weighted model counting allows us to encode and exploit local structure in dynamic networks in a similar way as for static networks. As demonstrated by our experiments, significant speed gains can be obtained in case the transition model exhibits a certain amount of local structure. On the other hand, our approach requires a full representation of the belief state, i.e. the distribution over the variables in the interface, and this is, similar as for other specific-purpose algorithms, exponential in the number of variables in the interface.

Example 3.15 Assume, for our running example depicted in Figure 3.1, we would have $\{w1_t, \neg w2_t, w4_t\}$ as observations. In this case, we certainly know the `AND-gate` is faulty as its output is *high* while at least one of its inputs is *low*. This implies that $\Pr(ha_t, hn_t) = 0$ and $\Pr(ha_t, \neg hn_t) = 0$, allowing us to more compactly write $\Pr(ha_t) = 0$.

Local structure in the belief state, in the form of determinism or equal probabilities, potentially allows us to encode and represent the belief state in a more compact way. This requires a more flexible approach, however, as it is not guaranteed this local structure will persist over time and new observations might introduce additional local structure. In Chapter 5 we will discuss how the structural interface algorithm can be extended to additionally exploit local structure in the belief state.

Chapter 4

Probabilistic Logic Programs

4.1 Introduction

Many real world reasoning tasks, such as gene interaction networks, social networks and web-page classification, involve both relational structure and uncertainty. This caused a significant interest in statistical relational learning (Getoor and Taskar 2007) and probabilistic logic programming (De Raedt, Frasconi, et al. 2008). Probabilistic logic programming (PLP) languages, e.g. ProbLog (De Raedt, Kimmig, and Toivonen 2007), extend the logic programming language Prolog with probabilistic choices on which facts are `true` or `false`. The use of a logic programming language allows one to declaratively express relational structure and, at the same time, the use of probabilistic choices allows one to deal with uncertainty.

Probabilistic logic programming languages offer an expressive framework to compactly describe (structural) knowledge. Logical variables are used to represent abstract objects, rather than specific entities, and range over a (potentially) infinite domain. As a result, probabilistic reasoning is extremely challenging as the underlying model might be very complex. Especially programs with cyclic dependencies which, for example, arise when modeling probabilistic graphs, are known to be hard to deal with.

Inference in probabilistic logic programs, similar to graphical models, can be reduced to the task of weighted model counting (WMC) (Fierens et al. 2015). This reduction involves the conversion of the grounded logic program into an equivalent (weighted) propositional formula in a CNF representations, after which any state-of-the-art WMC solver can be called. The conversion step

does not come without a cost, however, as capturing the semantics of the logic program might lead to an explosion of the corresponding knowledge base. This puts a burden on the WMC solver and quickly renders inference intractable.

Recent developments in knowledge representation and knowledge compilation have created new opportunities for inference in probabilistic models. The Sentential Decision Diagram (SDD) (Darwiche 2011) is a newly introduced target representation that combines the advantages of Ordered Binary Decision Diagrams (OBDD) and deterministic-Decomposable Negation Normal Form (d-DNNF). On the one hand, SDD allows for incremental formula construction in the same way as OBDD and, on the other hand, they come with the same size upper bounds as for d-DNNF compilation. Incremental formula construction is especially useful as it allows for a more flexible compilation strategy, i.e. it is not strictly required to represent to knowledge base as a formula in CNF.

In this chapter, we first show how probabilistic logic programs can be compiled into a Sentential Decision Diagram in a bottom-up manner. Next, and our main contribution, is $T_{\mathcal{P}}$-compilation for inference in probabilistic logic programs. $T_{\mathcal{P}}$-compilation is different from any of the existing methods in that it interleaves knowledge compilation with forward reasoning on the logic program and, as such, does not require to encode the logic program into an intermediate knowledge base representation. Furthermore, $T_{\mathcal{P}}$-compilation is an anytime algorithm that, in case the available resources do not allow to compute the exact solution, provides hard bounds on the inferred probabilities.

Bottom-up compilation of probabilistic logic programs into Sentential Decision Diagrams was previously published in

J. Vlasselaer, J. Renkens, G. Van den Broeck, and L. De Raedt (2014). "Compiling probabilistic logic programs into sentential decision diagrams". In: *Proceedings of the Workshop on Probabilistic Logic Programming (PLP)*

Incremental formula construction with $T_{\mathcal{P}}$-compilation and experimental results were previously published in

J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2015). "Anytime inference in probabilistic logic programs with Tp-compilation". In: *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*

Structure of this Chapter

In section 4.2 we review logical inference in logic programs and probabilistic inference by weighted model counting for probabilistic logic programs. Section

4.3 shows how we can represent a ground logic program as a Boolean circuit which then can be compiled into an SDD representation. Section 4.4 introduces $T_{\mathcal{P}}$-Compilation, a novel anytime inference algorithm for probabilistic logic programs. Section 4.5 shows the experimental results. Related work is discussed in Section 4.6 and we conclude this chapter with a discussion in Section 4.7.

4.2 Preliminaries

We will first review the necessary background on logical inference for definite clause programs and logic programs. For a more detailed overview regarding this topic, we refer to (Nilsson and Maluszynski 1995). Next, we review exact inference for probabilistic logic programs by means of weighted model counting and various techniques for approximate inference.

4.2.1 Logical Inference for Definite Clause Programs

A *definite clause program* is a finite set of definite clauses, also called *rules*. Let $\mathcal{A}$ be the set of all ground atoms that can be constructed from the constants, functors and predicates in a definite clause program $\mathcal{P}$. The model-theoretic semantics of a definite clause program is given by its unique *least Herbrand model*, that is, the set of all ground atoms $a \in \mathcal{A}$ that are entailed by the logic program, written $\mathcal{P} \models a$. The task of logical inference is to determine whether a program $\mathcal{P}$ entails a given atom, called *query*. This allows one to compute, for example, whether two nodes in a graph are connected, i.e. whether there is a path between two nodes.

Example 4.1 Consider the definite clause program depicted in Figure 4.1. The least Herbrand model is given by $\{\texttt{edge}(b,a), \texttt{edge}(b,c), \texttt{edge}(a,c), \texttt{edge}(c,a), \texttt{path}(b,a), \texttt{path}(b,c), \texttt{path}(a,c), \texttt{path}(c,a), \texttt{path}(a,a), \texttt{path}(c,c)\}$. For a query $\texttt{path}(b,c)$, logical inference will return *yes* as there indeed is a path going from node b to node c and $\texttt{path}(b,c)$ is in the least Herbrand model. For a query $\texttt{path}(b,a)$, logical inference will return *no* as there is no path going from node b to node a and $\texttt{path}(b,a)$ is not in the least Herbrand model.

One of the most common techniques to inference is known as *backward reasoning* or *SLD-resolution*. This approach starts from the query and reasons back towards the facts. Intuitively, backward reasoning starts by trying to *unify* the query atom with the heads of the rules in the program. If it succeeds, each of the atoms in the body of this rule is treated as a query and the procedure

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

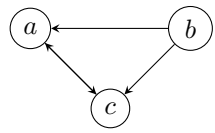


Figure 4.1: A logic program modeling a cyclic graph.

repeats. It is common to represent this process by means of an SLD-tree (see Figure 4.2). The process continues till all branches in the tree either succeed or fail. All facts that appear in a branch that succeeds is often referred to as a *proof* or an *explanation* of the query. An advantage of backward reasoning is that it starts from the query and, as such, it is goal oriented. A disadvantage is that special care is needed to avoid unnecessary computations, especially in the presence of cyclic dependencies (Rocha et al. 2000).

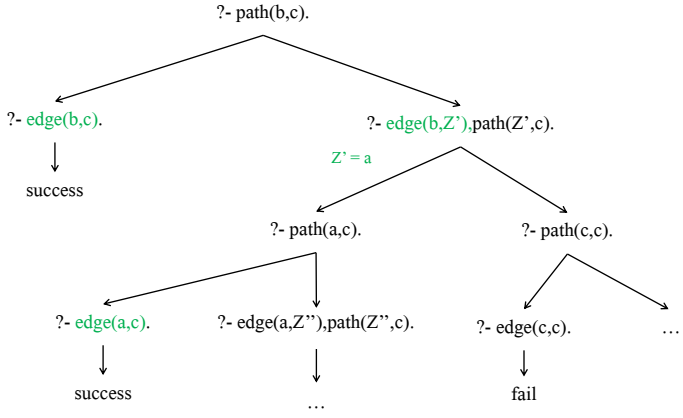


Figure 4.2: (Part of) SLD-tree for the program shown in Figure 4.1 and the query $\texttt{path}(b, c)$.

Example 4.2 For the program depicted in Figure 4.1 and the query $\texttt{path}(b, c)$, part of the SLD-tree is shown in Figure 4.2. One of the explanations for the query is $\{\texttt{edge}(b, a), \texttt{edge}(a, c)\}$ as these two edges suffice to form a path between node b and c.

A second technique to inference is known as *forward reasoning*. This approach starts from the facts and reasons forward to derive new knowledge. The advantage of this technique is that it naturally deals with cyclic dependencies and avoids unnecessary computations. The drawback is that it blindly generates new knowledge without taking into account the query, i.e. it is not goal-oriented. Forward reasoning is done by means of the immediate consequence operator $T_\mathcal{P}$ (Van Emden and Kowalski 1976).

Definition 1 ($T_\mathcal{P}$ operator) *Let $\mathcal{P}$ be a ground definite clause program. For a Herbrand interpretation I, the $T_\mathcal{P}$ operator returns*

$$T_\mathcal{P}(I) = \{\mathtt{h} \mid \mathtt{h} \texttt{ :- } \mathtt{b_1}, \ldots, \mathtt{b_n} \in \mathcal{P} \textit{ and } \{\mathtt{b_1}, \ldots, \mathtt{b_n}\} \subseteq I\}$$

The *least fixpoint* of this operator is the least Herbrand model of $\mathcal{P}$ and is the least set of atoms I such that $T_\mathcal{P}(I) \equiv I$. Let $T_\mathcal{P}^k(\varnothing)$ denote the result of k consecutive calls of the $T_\mathcal{P}$ operator, ΔI^i be the difference between $T_\mathcal{P}^{i-1}(\varnothing)$ and $T_\mathcal{P}^i(\varnothing)$. Then $T_\mathcal{P}^\infty(\varnothing)$ is the least fixpoint interpretation of $T_\mathcal{P}$.

Example 4.3 The least fixpoint can be computed efficiently using a semi-naive evaluation algorithm where only new knowledge is added to an interpretation. For the program given in Example 4.1, this results in:

$$I^0 = \varnothing$$

$$\Delta I^1 = \{\mathtt{e}(b,a), \mathtt{e}(b,c), \mathtt{e}(a,c), \mathtt{e}(c,a)\}$$

$$\Delta I^2 = \{\mathtt{p}(b,a), \mathtt{p}(b,c), \mathtt{p}(a,c), \mathtt{p}(c,a)\}$$

$$\Delta I^3 = \{\mathtt{p}(a,a), \mathtt{p}(c,c)\}$$

$$\Delta I^4 = \varnothing$$

and $T_\mathcal{P}^\infty(\varnothing) = \bigcup_i \Delta I^i$ is the least Herbrand model as given above.

Property 1 *Employing the $T_\mathcal{P}$ operator on a subset of the least fixpoint leads again to the same least fixpoint:*

$$\forall I \subseteq T_\mathcal{P}^\infty(\varnothing) : T_\mathcal{P}^\infty(I) = T_\mathcal{P}^\infty(\varnothing)$$

Property 2 *Adding a set of definite clauses $\mathcal{P}'$ to program $\mathcal{P}$ leads to a superset of the least fixpoint for $\mathcal{P}$:*

$$T_\mathcal{P}^\infty(\varnothing) \subseteq T_{\mathcal{P}\cup\mathcal{P}'}^\infty(\varnothing)$$

4.2.2 Logical Inference for Normal Logic Programs

A *normal logic program* extends a definite clause program and allows for *negation*, i.e., it is a finite set of *normal clauses*. The $T_{\mathcal{P}}$ operator for definite clause programs can be generalized towards *stratified* normal logic programs, where the rules in the program are partitioned according to their strata. Let $\mathcal{P}^{\mathtt{h}}$ be the subset of clauses in $\mathcal{P}$ where $\mathtt{h}$ is the head, then a stratified program is defined as follows:

Definition 2 (Stratified Program) *A normal logic program $\mathcal{P}$ is said to be stratified if there exists a partitioning $\mathcal{P}_1 \cup \cdots \cup \mathcal{P}_m$ of $\mathcal{P}$ such that:*

$$\textit{if } \mathtt{h} \texttt{ :- } \ldots, \mathtt{b}, \ldots \in \mathcal{P}_i \qquad \textit{then } \mathcal{P}^{\mathtt{b}} \subseteq \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_i$$

$$\textit{if } \mathtt{h} \texttt{ :- } \ldots, \neg\mathtt{b}, \ldots \in \mathcal{P}_i \qquad \textit{then } \mathcal{P}^{\mathtt{b}} \subseteq \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_{i-1}$$

The $T_{\mathcal{P}}$ operator for normal logic programs is defined as:

$$T_{\mathcal{P}}(I) = \{\mathtt{h} \mid \mathtt{h} \texttt{ :- } \mathtt{b_1}, \ldots, \mathtt{b_n} \in \mathcal{P} \text{ and } I \models \mathtt{b_1}, \ldots, \mathtt{b_n}\}$$

where $I \models \mathtt{b_i}$ if $\mathtt{b_i} \in I$ and $I \models \neg\mathtt{b_i}$ if $\mathtt{b_i} \notin I$. Then, the *canonical* model can be obtained by iteratively computing the fixpoint for each stratum. Let I_i be the Herbrand interpretation for stratum i, the canonical model for a stratified program with m strata is computed as:

$$I_1 = T^{\infty}_{\mathcal{P}_1}(\varnothing)$$

$$I_2 = T^{\infty}_{\mathcal{P}_2}(I_1) \cup I_1$$

$$\vdots$$

$$I_m = T^{\infty}_{\mathcal{P}_m}(I_{m-1}) \cup I_{m-1}$$

and $T^{\infty}_{\mathcal{P}}(\varnothing) = I_m$.

Example 4.4 Consider the following normal logic program with two strata:

$$\mathcal{P}_1 \begin{cases} \texttt{sprinklerOn}. \\ \texttt{rain :- cloudy}. \\ \texttt{wetGrass :- rain}. \end{cases}$$
$$\mathcal{P}_2 \begin{cases} \texttt{sprinkler :- } \neg\texttt{cloudy}, \texttt{sprinklerOn}. \\ \texttt{wetGrass :- sprinkler}. \end{cases}$$

for which computing the canonical model results in:

$$I_1 = \{\texttt{sprinklerOn}\}$$

$$I_2 = \{\texttt{sprinkler}, \texttt{wetGrass}\} \cup I_1$$

and $T_{\mathcal{P}}^{\infty}(\varnothing) = \{\texttt{sprinklerOn}, \texttt{sprinkler}, \texttt{wetGrass}\}$.

4.2.3 Probabilistic Inference by Weighted Model Counting

Many of the existing inference techniques for probabilistic logic programs rely on weighted model counting and knowledge compilation. Historically, a logic program is encoded by means of an hand-tailored intermediate representation which then is compiled into an (ordered) binary decision diagram (Mantadelis and Janssens 2010). Most recent techniques follow a more general approach where the program is first encoded into a CNF representation which then can be fed to any off-the-shelf WMC solver. We briefly discuss the key steps of this approach and refer to Fierens et al. (2015) for full details.

Grounding

The first step is to ground the program with respect to the queries and evidence. Concretely, one tries to find the part of the grounding that is relevant to the queries $\mathbf{q}$ and evidence $\mathbf{e}$=$\mathbf{E}$. Intuitively, this is done by applying SLD-resolution on all atoms in $\mathbf{q} \cup \mathbf{e}$, that is all query and evidence atoms. The set of ground rules encountered during this process is referred to as the *relevant ground program* with respect to $\mathbf{q}$ and $\mathbf{E} = \mathbf{e}$. It is safe to restrict the grounding to these rules only and the relevant ground program contains all the necessary information to compute $\Pr(\mathbf{q}|\mathbf{e} = \mathbf{E})$.

$0.4 :: \texttt{edge}(b, a).$ $\quad$ $0.3 :: \texttt{edge}(b, c).$

$0.8 :: \texttt{edge}(a, c).$ $\quad$ $0.9 :: \texttt{edge}(c, a).$

$\texttt{path}(\texttt{X}, \texttt{Y})$:- $\texttt{edge}(\texttt{X}, \texttt{Y}).$

$\texttt{path}(\texttt{X}, \texttt{Y})$:- $\texttt{edge}(\texttt{X}, \texttt{Z}), \texttt{path}(\texttt{Z}, \texttt{Y}).$

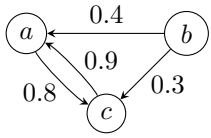


Figure 4.3: A probabilistic logic program modeling a cyclic probabilistic graph.

Example 4.5 For the probabilistic logic program shown in Figure 4.3 and a query $\texttt{path}(\texttt{a}, \texttt{c})$, we would obtain the following relevant ground program:

$$\texttt{path}(a, c) \text{ :- } \texttt{edge}(a, c).$$

$$\texttt{path}(a, c) \text{ :- } \texttt{edge}(a, c), \texttt{path}(c, c).$$

$$\texttt{path}(c, c) \text{ :- } \texttt{edge}(c, a), \texttt{path}(a, c).$$

Note that the probabilistic facts $\texttt{edge}(b, a)$ and $\texttt{edge}(b, c)$ do not appear in the relevant ground program as they are not required to answer the query.

Conversion

The second step is to convert the rules in the relevant ground program into an equivalent propositional formula, typically in CNF. This step includes taking into account the difference in semantics between logic programming (with closed world assumption) and first-order logic (without closed world assumption). Hence, conversion into a propositional knowledge base is, at least for cyclic programs, not simply a matter of rewriting.

The conversion of a ground logic program $\mathcal{P}$ into an equivalent propositional knowledge base $\Sigma_\mathcal{P}$ requires that $MOD(\mathcal{P}) = SAT(\Sigma_\mathcal{P})$, where $MOD(\mathcal{P})$ denotes the set of models of $\mathcal{P}$ and $SAT(\Sigma_\mathcal{P})$ denotes the set of models of $\Sigma_\mathcal{P}$. For acyclic rules, the conversion step is straightforward and only requires to take *Clark's completion* (Clark 1978). That is, for each head atom $\texttt{h}$ with k corresponding rules, $\texttt{h}$:- $\texttt{b}_{1,1}, \ldots, \texttt{b}_{1,\texttt{m}}$, $\ldots$, $\texttt{h}$:- $\texttt{b}_{\texttt{k},1}, \ldots, \texttt{b}_{\texttt{k},\texttt{n}}$ we add to the knowledge base the following formula: $\texttt{h} \Leftrightarrow (\texttt{b}_{1,1} \wedge \cdots \wedge \texttt{b}_{1,\texttt{m}}) \vee \cdots \vee (\texttt{b}_{\texttt{k},1} \wedge \cdots \wedge \texttt{b}_{\texttt{k},\texttt{n}})$.

For programs with cyclic rules, the conversion step is more complicated and requires additional variables and clauses to correctly capture the semantics of every cycle. One way to do this, is by rewriting the ground program in order to "break" the cycles (Janhunen 2004). Intuitively, rewriting can be seen as duplicating ground atoms where different copies are used in different contexts. Once a cycle free ground program is obtained, Clark's completion can be used to obtain the knowledge base as described above.

Example 4.6 Assume we have the following ground logic program:

$$\texttt{path}(b, c) \texttt{ :- } \texttt{edge}(b, c).$$

$$\texttt{path}(b, c) \texttt{ :- } \texttt{edge}(b, a), \texttt{path}(a, c).$$

$$\texttt{path}(a, c) \texttt{ :- } \texttt{edge}(a, c).$$

$$\texttt{path}(a, c) \texttt{ :- } \texttt{edge}(a, b), \texttt{path}(b, c).$$

The completion of this set of rules can be satisfied by setting $\texttt{path}(a, c)$, $\texttt{path}(b, c)$, $\texttt{edge}(b, a)$, $\texttt{edge}(a, b)$ to true and everything else to false. This is not a valid possible world under logic programming semantics, however, as there is no base-case to make either $\texttt{path}(a, c)$ or $\texttt{path}(b, c)$ true. One way to correctly deal with this program is to rewrite it as follows:

$$\texttt{path}(b, c) \texttt{ :- } \texttt{edge}(b, c).$$

$$\texttt{path}(b, c) \texttt{ :- } \texttt{edge}(b, a), \texttt{aux\_path}(a, c).$$

$$\texttt{path}(a, c) \texttt{ :- } \texttt{edge}(a, c).$$

$$\texttt{path}(a, c) \texttt{ :- } \texttt{edge}(a, b), \texttt{aux\_path}(b, c).$$

$$\texttt{aux\_path}(b, c) \texttt{ :- } \texttt{edge}(b, c).$$

$$\texttt{aux\_path}(a, c) \texttt{ :- } \texttt{edge}(a, c).$$

Where the original ground program contains cyclic dependencies, such as $\texttt{path}(b, c) - \texttt{path}(a, c) - \texttt{path}(b, c)$, these do not appear in the rewritten program. The new program is obtained by keeping a trace for each of the atoms and duplicate the rules which do not form a loop, i.e. the rules for which none of the atoms in the body is already in the trace.

Then, Clark's completion would result in the following knowledge base (and obtaining a CNF out of this knowledge base is simply a matter of rewriting):

$$\texttt{path}(b,c) \Leftrightarrow \texttt{edge}(b,c) \vee (\texttt{edge}(b,a) \wedge \texttt{aux\_path}(a,c))$$

$$\texttt{path}(a,c) \Leftrightarrow \texttt{edge}(a,c) \vee (\texttt{edge}(a,b) \wedge \texttt{aux\_path}(b,c))$$

$$\texttt{aux\_path}(b,c) \Leftrightarrow \texttt{edge}(b,c).$$

$$\texttt{aux\_path}(a,c) \Leftrightarrow \texttt{edge}(a,c).$$

Weighted Model Counting

The third step is to define a weight function such that the weight of a model in the probabilistic logic program is the same as the weight of the corresponding model in the propositional knowledge base. To do so, we distinguish two types of literals:
(1) for each of the literals f and $\neg f$ in the knowledge base that correspond to a probabilistic fact $\texttt{p}::\texttt{f}$ we set $w(f) = \texttt{p}$ and $w(\neg f) = 1 - \texttt{p}$
(2) for all other literals l and $\neg l$ we set $w(l) = 1$ and $w(\neg l) = 1$

Once a weighted propositional knowledge base is obtained, we can perform probabilistic inference by means of weighted model counting. For probabilistic logic programs, it is common to rely on knowledge compilation.

4.2.4 Approximate Inference

Despite a lot of progress in weighted model counting and knowledge compilation, exact inference is often infeasible when dealing with real-life problems and one quickly has to resort to approximate methods. We can distinguish different methodologies and shortly discuss some of the relevant techniques.

Simplified Problem

A straightforward, yet effective, way to deal with problems where exact inference is infeasible is to simplify the problem. A first approach is to only consider a subset of the probabilistic facts as this will typically reduce the size of the relevant ground program. A second approach is to rephrase the query of interest. A typical example for the latter are the probabilistic graphs, as used in our examples, where one rewrites the query to only allow paths up to a certain length rather than "infinite" length. While this is certainly effective for most

graphs, as longer paths typically come with smaller probabilities, it is not always easy to generalize towards other programs.

The advantage of simplifying the problem is that it allows us to reuse any of the existing (exact) inference techniques. One only has to sufficiently reduce the size of the relevant ground program in order to make inference tractable for the available recourses, i.e. available time and memory. The disadvantage is that the computed probabilities are rather meaningless as it is impossible to verify whether they are an underestimate or an overestimate of the actual probability. Hence, simplifying the problem is only useful for a small subset of problems and should be avoided in general.

Example 4.7 Consider the following probabilistic logic program:

$$0.9 :: \texttt{a}. \qquad 0.9 :: \texttt{b}.$$

$$\texttt{c} \texttt{ :- } \neg \texttt{a}.$$

$$\texttt{c} \texttt{ :- b}.$$

for which we have $\Pr(\texttt{c}) = 0.91$. Simplifying the program by removing probabilistic fact `a` gives $\Pr(\texttt{c}) = 1$ and the computed probability is an overestimate of the actual probability. Simplifying the program by removing probabilistic fact `b` gives $\Pr(\texttt{c}) = 0.1$ and the computed probability is an underestimate of the actual probability.

Sampling

Many approximate inference techniques for probabilistic models rely on some sort of sampling and avoid the need to simplify the problem. *Forward sampling* acts directly on the probabilistic logic program and generates different samples of the program in order to estimate the probability of the query. Intuitively, forward sampling picks a truth-value for each of the probabilistic facts, according to their probability, and uses logical inference to infer whether the program entails the query. By doing this multiple times, one can get an estimate of the true probability of the query. An alternative sampling approach is to first convert the relevant ground program into a weighted propositional knowledge base, in the same way as done for exact inference. Next, one can use a sampling approach that acts on propositional knowledge bases, e.g. MC-SAT (Poon and Domingos 2006).

The advantage of sampling is that it is quite straightforward and easy to implement. A disadvantage is that it does not give strong guarantees (in the

form of bounds) on the computed probabilities. Especially for highly skewed distributions, i.e. in the presence of probabilistic facts with extremely high or low probabilities, it is known that sampling techniques often perform poorly.

Selected Explanations

All of the approximate techniques discussed above only provide an estimate and it is unknown whether this is an underestimate or overestimate of the actual probability. More informative results are obtained in case the approximate algorithms return hard bounds with respect to the actual probability.

A hard *lower bound* can be obtained by only performing inference on a subset of the explanations or proofs obtained after (or during) the grounding step. Possible selection criteria include all explanations up to a certain length, all explanations for which the probability is above a certain threshold, or the k-best explanations (for a given number of k) for which this probability is the highest.

Example 4.8 Consider the following probabilistic logic program:

$$0.4 :: \texttt{edge}(b, a). \qquad 0.3 :: \texttt{edge}(b, c).$$

$$0.8 :: \texttt{edge}(a, c).$$

$$\texttt{path}(\mathsf{X}, \mathsf{Y}) \texttt{ :- } \texttt{edge}(\mathsf{X}, \mathsf{Y}).$$

$$\texttt{path}(\mathsf{X}, \mathsf{Y}) \texttt{ :- } \texttt{edge}(\mathsf{X}, \mathsf{Z}), \texttt{path}(\mathsf{Z}, \mathsf{Y}).$$

For the query $\texttt{path}(b, c)$ we obtain the following two explanations; $\texttt{edge}(b, c)$ or $\texttt{edge}(b, a) \wedge \texttt{edge}(a, c)$. The probabilities of the explanations are 0.3 (for $\texttt{edge}(b, c)$) and $0.8 \cdot 0.4 = 0.32$ (for $\texttt{edge}(b, a) \wedge \texttt{edge}(a, c)$), and with $\mathrm{k} = 1$ we would thus select the longer explanation.

A hard *upper bound* for the actual probability can be obtained by only performing inference on partial explanations, i.e. by only considering a subset of the probabilistic facts in an explanation. Partial explanations can be obtained by stopping SLD-resolution before it is known whether the branch succeeds or fails, or by cutting the SLD-tree at a certain depth (Poole 1993). In practice, this approach is only valid for definite clause programs and often computes probabilities close to 1, i.e. the trivial upper bound.

Recently, Renkens, Kimmig, et al. (2014) proposed to first encode the ground relevant program into a propositional knowledge, in the same way as for exact inference, and formulates the explanations search as weighted PMAX-SAT

problem on this knowledge base. This approach is encoded such that solutions can iteratively be obtained from a standard (weighted) PMAX-SAT solver. These explanations then provide a lower bound for the probability of the query. An upper bound can be obtained based on explanations for the negation of the query.

4.3 Bottom-Up Compilation for Probabilistic Logic Programs

Exact inference for probabilistic logic programs first encodes the relevant ground program into a propositional knowledge base which then is fed to an off-the-shelf solver for weighted model counting. Many of these solvers, for example d-DNNF knowledge compilers, require the knowledge base to be in a CNF representation, i.e. a conjunction of clauses. For probabilistic logic programs, however, the conversion into CNF requires the introduction of auxiliary variables. This increases the search space the solver has to deal with, e.g. to find a good variable ordering, and significantly decreases its performance.

Target representations and knowledge compilers that efficiently support incremental formula construction do not require to encode the knowledge base into a CNF representation as they can compile a formula in a bottom-up manner. Recently, bottom-up compilation of graphical models into sentential decision diagrams has shown to outperform other compilation approaches (Choi et al. 2013). As we will show in the remainder of this section, bottom-up compilation is even more relevant for probabilistic logic programs as it allows us to omit the auxiliary variables.

4.3.1 Auxiliary Variables by Conversion to CNF

Applying Clark's completing on a ground logic program results in a set of equivalences where a variable is equivalent with a disjunction of conjunctions. One way to convert this knowledge base into a CNF representation is to decompose this disjunction by introducing an auxiliary variable for each of the conjunctions. While this avoids an exponential blow-up of the CNF, it obviously requires a significant amount of additional variables.

Example 4.9 Assume we have the following simple relevant ground program:

$$\texttt{a :- b, c.}$$

$$\texttt{a :- d, e.}$$

Clark's completion will result in the following formula:

$$\texttt{a} \Leftrightarrow (\texttt{b} \wedge \texttt{c}) \vee (\texttt{d} \wedge \texttt{e})$$

Conversion into CNF will first introduce two auxiliary variables and rewrites the formula as follow:

$$\texttt{aux1} \Leftrightarrow \texttt{b} \wedge \texttt{c}$$

$$\texttt{aux2} \Leftrightarrow \texttt{d} \wedge \texttt{e}$$

$$\texttt{a} \Leftrightarrow \texttt{aux1} \vee \texttt{aux2}$$

After which the CNF can be obtained by simply rewriting.

4.3.2 Auxiliary Variables by Cycle Breaking

As illustrated in Section 4.2.3, auxiliary variables are required to correctly capture the semantics of cyclic rules in a logic program. We now show a more elaborate example to illustrate how cycling breaking might lead to a blow-up of the relevant ground program.

Example 4.10 Consider a *social network* with a domain of three persons which all possibly smoke. The goal of the program is to compute the probability for each person that they actually smoke, based on their stress-level and friends.

$0.4 :: \texttt{friends}(a, b).$ $0.8 :: \texttt{friends}(a, c).$ $0.2 :: \texttt{friends}(c, b).$

$0.5 :: \texttt{friends}(b, a).$ $0.9 :: \texttt{friends}(c, a).$ $0.1 :: \texttt{friends}(b, c).$

$0.1 :: \texttt{stress}(a).$ $0.5 :: \texttt{stress}(b).$ $0.9 :: \texttt{stress}(c).$

$\texttt{smokes}(X)$:- $\texttt{stress}(X).$

$\texttt{smokes}(X)$:- $\texttt{friends}(X, Y), \texttt{smokes}(Y).$

The relevant ground program is shown below (we dropped the probabilistic facts as they are the same as in the non-grounded program)

```
smokes(a) :- stress(a).      smokes(b) :- stress(b).      smokes(c) :- stress(c).

smokes(a) :- friends(a,b), smokes(b).      smokes(a) :- friends(a,c), smokes(c).

smokes(b) :- friends(b,a), smokes(a).      smokes(b) :- friends(b,c), smokes(c).

smokes(c) :- friends(c,a), smokes(a).      smokes(c) :- friends(c,b), smokes(b).
```

The cycle-free ground program looks as follow, all `smokes-*` atoms are auxiliary variables necessary to break the loops.

```
smokes(a) :- stress(a).      smokes(a) :- friends(a,b), smokes-a(b).

smokes(b) :- stress(b).      smokes(a) :- friends(a,c), smokes-a(c).

smokes(c) :- stress(c).      smokes(b) :- friends(b,a), smokes-b(a).

                             smokes(b) :- friends(b,c), smokes-b(c).

smokes-a(b) :- stress(b).    smokes(c) :- friends(c,a), smokes-c(a).

smokes-a(c) :- stress(c).    smokes(c) :- friends(c,b), smokes-c(b).


smokes-b(a) :- stress(a).    smokes-a(b) :- friends(b,c), smokes-ab(c).

smokes-b(c) :- stress(c).    smokes-a(c) :- friends(c,b), smokes-ac(b).


smokes-c(a) :- stress(c).    smokes-b(a) :- friends(a,c), smokes-ab(c).

smokes-c(b) :- stress(b).    smokes-b(c) :- friends(c,a), smokes-bc(a).


smokes-bc(a) :- stress(a).   smokes-c(a) :- friends(a,b), smokes-ac(b).

smokes-ac(b) :- stress(b).   smokes-c(b) :- friends(b,a), smokes-bc(a).

smokes-ab(c) :- stress(c).
```

4.3.3 Compilation Without Auxiliary Variables

We can avoid the need to explicitly introduce the auxiliary variables by representing the loop-free ground program as a Boolean circuit. Figure 4.4 depicts the Boolean circuit for the ground program of Example 4.10. Each of the leafs is a variable that corresponds to a probabilistic fact, and each of the root nodes corresponds to a ground atom we want to compute the probability of. Intuitively, each of the other variables is directly replaced by its corresponding formula. For example, the left-most child of $\texttt{smokes}(a)$ (the node annotated with (ii)) encodes the Boolean function for the clause $\texttt{smokes}(a)$ `:-` $\texttt{friends}(a, b), \texttt{smokes-a}(b)$. and its subgoals where, $\texttt{smokes-a}(b)$ is encoded by the node annotated with (i). Once this Boolean circuit is obtained, it can be compiled into an SDD representation by means of a single bottom-up pass by repeated application of the *apply-function*

Example 4.11 Figure 4.5 depicts how we can compile the left-most child of $\texttt{smokes}(a)$, that is, the clause $\texttt{smokes}(a)$ `:-` $\texttt{friends}(a, b), \texttt{smokes-a}(b)$. and it subgoals, into an SDD representation. Important to note is that none of the auxiliary variables explicitly appears in the compiled representation. Instead, each of these variables correspond to a formula and, in each of the steps, we build further upon the formulas compiled in the previous steps. For example, we first compile an SDD for $\texttt{smokes-a}(b)$ (the second step) which then is used to compile and SDD for $\texttt{smokes}(a)$ (the third step).

Some more intuition on how to "read" the obtained SDD representation depicted in Figure 4.5c. The decision in node 4 is whether $\texttt{friends}(b, c) \wedge \texttt{stress}(c)$ is true (represented by node 3) or false (represented by node 1). In the first case, one proceeds with checking whether $\texttt{friends}(a, b)$ is true. In the second case, one proceeds with checking node 2, which is itself an SDD.

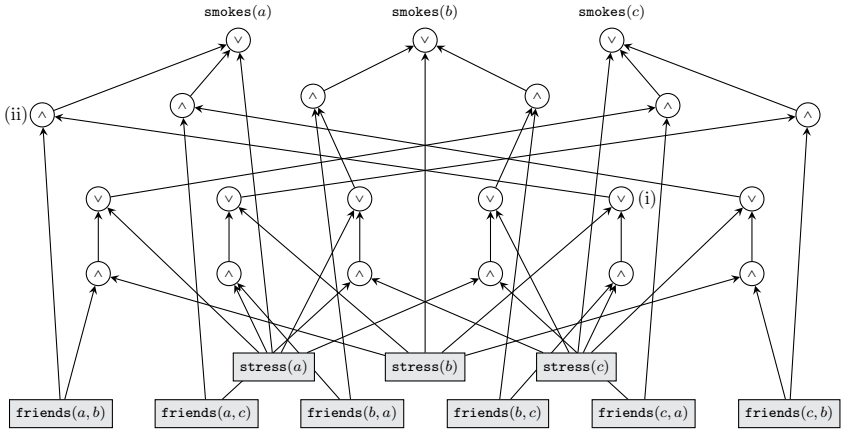
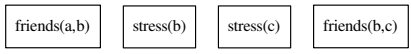
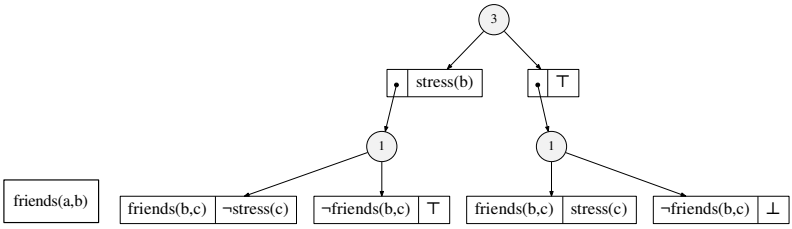


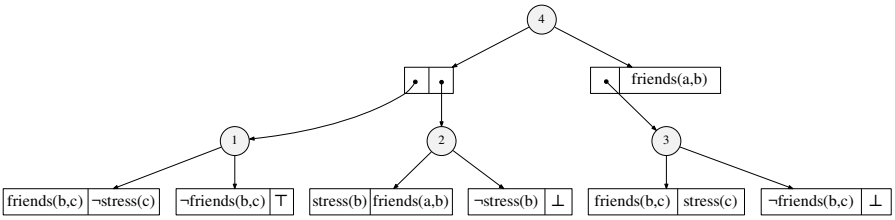
Figure 4.4: Boolean circuit for the program from Example 4.10.



(a) Step 1; Instantiate the required variables into an SDD representation.



(b) Step 2; Compiling $\texttt{smokes-a}(b) \Leftrightarrow (\texttt{friends}(b, c) \wedge \texttt{stress}(c)) \vee \texttt{stress}(b)$.



(c) Step 3; Compiling $\texttt{smokes}(a) \Leftrightarrow \texttt{friends}(a, b), \texttt{smokes-a}(b)$.

Figure 4.5: The three required steps to compile, in a bottom-up manner, the clause $\texttt{smokes}(a)$ `:-` $\texttt{friends}(a, b), \texttt{smokes-a}(b)$. into an SDD representation.

4.4 Tp-Compilation

Exact inference techniques for probabilistic logic programs, perform a sequence of isolated steps; (1) grounding of the program, (2) conversion of the ground program into an equivalent propositional knowledge base, and (3) perform inference by means of knowledge compilation and weighted model counting. Even the bottom-up compilation approach, as presented in the previous section, requires to completely unfold the underlying model. This includes breaking the cycles, by rewriting the program, and encoding of the ground program as an intermediate Boolean circuit representation.

We now propose $T_{\mathcal{P}}$*-compilation* for anytime inference in probabilistic logic programs. The incentive of our inference algorithm is to interleave formula construction and compilation by means of forward reasoning. The two advantages are that (a) the conversion to propositional logic happens during rather than after reasoning within the logic programming semantics, avoiding the expensive introduction of additional variables and propositions to deal with cycles, and (b) at any time in the process, the current formulas provide hard bounds on the probabilities.

4.4.1 $Tc_{\mathcal{P}}$ Operator

The formal basics of our approach is the $Tc_{\mathcal{P}}$ operator which generalizes the $T_{\mathcal{P}}$ operator for logic program towards the probabilistic setting. Although the $T_{\mathcal{P}}$ operator, and forward reasoning in general, naturally considers all consequences of a program, using the relevant ground program allows us to restrict the approach to the queries of interest. As common in probabilistic logic programming, we assume the *finite support condition*, i.e., the queries depend on a finite number of ground probabilistic facts. For ease of notation, we first only consider definite clause programs and treat normal logic programs towards the end of this section.

Definite Clause Programs

We use forward reasoning to build a formula λ_a for every atom $a \in \mathcal{A}$ such that λ_a exactly describes the total choices $C \subseteq \mathcal{F}$ for which $C \cup \mathcal{R} \models a$. Such λ_a can be used to compute the probability of a via WMC.

Definition 3 (Parameterized interpretation) *A parameterized interpretation $\mathcal{I}$ of a ground probabilistic logic program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$ and atoms $\mathcal{A}$ is a set of tuples (a, λ_a) with $a \in \mathcal{A}$ and λ_a a propositional formula over $\mathcal{F}$ expressing in which interpretations a is true.*

Example 4.12 For the program shown in Figure 4.3, abbreviating predicate names by initials, the parameterized interpretation is:

$$\{\big(e(b,a), \lambda_{e(b,a)}\big), \big(e(b,c), \lambda_{e(b,c)}\big), \big(e(a,c), \lambda_{e(a,c)}\big), \big(e(c,a), \lambda_{e(c,a)}\big),$$
$$\big(p(b,a), \lambda_{p(b,a)}\big), \big(p(b,c), \lambda_{p(b,c)}\big), \big(p(a,c), \lambda_{p(a,c)}\big), \big(p(c,a), \lambda_{p(c,a)}\big),$$
$$\big(p(a,a), \lambda_{p(a,a)}\big), \big(p(c,c), \lambda_{p(c,c)}\big)\}.$$

and, for instance, $\lambda_{e(b,a)} = e(b,a)$ since $e(b,a)$ is `true` in exactly those worlds where the total choice includes this edge, and $\lambda_{p(b,c)} = e(b,c) \vee [e(b,a) \wedge e(a,c)]$ since $p(b,c)$ is `true` in exactly those worlds where the total choice includes the direct edge or the two-edge path over a.

A naive approach to construct the λ_a would be to compute $I_i = T^{\infty}_{\mathcal{R} \cup C_i}(\varnothing)$ for every total choice $C_i \subseteq \mathcal{F}$ and to set $\lambda_a = \bigvee_{i:a \in I_i} \bigwedge_{f \in C_i} f$, that is, the disjunction explicitly listing all total choices contributing to the probability of a. Clearly, this requires a number of fixpoint computations exponential in $|\mathcal{F}|$, and furthermore, doing these computations independently does not exploit the potentially large structural overlap between them.

Therefore, we introduce the $Tc_{\mathcal{P}}$ operator. It generalizes the $T_{\mathcal{P}}$ operator to work on the parameterized interpretation and builds, for all atoms in parallel on demand, formulas that are logically equivalent to the λ_a introduced above. For ease of notation, we assume that every parameterized interpretation implicitly contains a tuple $(\texttt{true}, \top)$, and, just as in regular interpretations, we do not list atoms with $\lambda_a \equiv \bot$. Thus, the empty set implicitly represents the parameterized interpretation $\{(\texttt{true}, \top)\} \cup \{(a, \bot) | a \in \mathcal{A}\}$ for a set of atoms $\mathcal{A}$.

Definition 4 ($Tc_{\mathcal{P}}$ operator) *Let $\mathcal{P}$ be a ground probabilistic logic program with probabilistic facts $\mathcal{F}$ and atoms $\mathcal{A}$. Let $\mathcal{I}$ be a parameterized interpretation with pairs (a, λ_a). Then, the $Tc_{\mathcal{P}}$ operator is $Tc_{\mathcal{P}}(\mathcal{I}) = \{(a, \lambda'_a) \mid a \in \mathcal{A}\}$ where*

$$\lambda'_a = \begin{cases} a & \text{if } a \in \mathcal{F} \\ \bigvee_{(\mathtt{a} \text{ :- } \mathtt{b_1},\ldots,\mathtt{b_n}) \in \mathcal{P}} (\lambda_{b_1} \wedge \cdots \wedge \lambda_{b_n}) & \text{if } a \in \mathcal{A} \backslash \mathcal{F}. \end{cases}$$

Intuitively, where the $T_{\mathcal{P}}$ operator (repeatedly) adds an atom a to the interpretation whenever the body of a rule defining a is `true`, the $Tc_{\mathcal{P}}$ operator

adds to the formula for a the description of the total choices for which the rule body is `true`. In contrast to the $T_{\mathcal{P}}$ operator, where a *syntactic* check suffices to detect that the fixpoint is reached, the $Tc_{\mathcal{P}}$ operator requires a *semantic* fixpoint check for each formula λ_a, which we write as $\mathcal{I}^i \equiv Tc_{\mathcal{P}}(\mathcal{I}^{i-1})$.

Definition 5 (Fixpoint of $Tc_{\mathcal{P}}$) *A parameterized interpretation $\mathcal{I}$ is a fixpoint of the $Tc_{\mathcal{P}}$ operator if and only if for all $a \in \mathcal{A}$, $\lambda_a \equiv \lambda'_a$, where λ_a and λ'_a are the formulas for a in $\mathcal{I}$ and $Tc_{\mathcal{P}}(\mathcal{I})$, respectively.*

It is easy to verify that for $\mathcal{F} = \varnothing$, i.e., a ground logic program $\mathcal{P}$, the iterative execution of the $Tc_{\mathcal{P}}$ operator directly mirrors that of the $T_{\mathcal{P}}$ operator, representing atoms as $(a, \top)$. We use λ^i_a to denote the formula associated with atom a after i iterations of $Tc_{\mathcal{P}}$ starting from $\varnothing$. We use SDDs to efficiently represent the formulas λ_a as will be discussed in Section 4.4.2.

Example 4.13 Applying $Tc_{\mathcal{P}}$ to the program given in Figure 4.3 results in the following sequence:

The first application of $Tc_{\mathcal{P}}$ sets:

$$\lambda^1_{e(b,a)} = e(b,a), \quad \lambda^1_{e(a,c)} = e(a,c), \quad \lambda^1_{e(b,c)} = e(b,c), \quad \lambda^1_{e(c,a)} = e(c,a)$$

These remain the same in all subsequent iterations.

The second application of $Tc_{\mathcal{P}}$ starts adding formulas for path atoms, which we illustrate for just two atoms:

$$\begin{aligned}\lambda^2_{p(b,c)} &= \lambda^1_{e(b,c)} \vee (\lambda^1_{e(b,a)} \wedge \lambda^1_{p(a,c)}) \vee (\lambda^1_{e(b,c)} \wedge \lambda^1_{p(c,c)}) \\ &= e(b,c) \vee (e(b,a) \wedge \bot) \vee (e(a,c) \wedge \bot) \equiv e(b,c)\end{aligned}$$

$$\lambda^2_{p(c,c)} = (\lambda^1_{e(c,a)} \wedge \lambda^1_{p(a,c)}) = (e(c,a) \wedge \bot) \equiv \bot$$

That is, the second step considers paths of length at most 1 and adds $(p(b,c), e(b,c))$ to the parameterized interpretation, but does not add a formula for $p(c,c)$, as no total choices making this atom `true` have been found yet.

The third iteration, adds information on paths of length at most 2:

$$\begin{aligned}\lambda^3_{p(b,c)} &= \lambda^2_{e(b,c)} \vee (\lambda^2_{e(b,a)} \wedge \lambda^2_{p(a,c)}) \vee (\lambda^2_{e(b,c)} \wedge \lambda^2_{p(c,c)}) \\ &\equiv e(b,c) \vee (e(b,a) \wedge e(a,c))\end{aligned}$$

$$\lambda^3_{p(c,c)} = (\lambda^2_{e(c,a)} \wedge \lambda^2_{p(a,c)}) \equiv (e(c,a) \wedge e(a,c))$$

Intuitively, $Tc_{\mathcal{P}}$ keeps adding longer sequences of edges connecting the corresponding nodes to the path formulas, reaching a fixpoint once all acyclic sequences have been added.

Correctness We show that for increasing i, $Tc^i_{\mathcal{P}}(\varnothing)$ reaches a least fixpoint $Tc^\infty_{\mathcal{P}}(\varnothing)$ where the λ_a are exactly the formulas needed to compute the probability for each atom by WMC.

Theorem 1 *For a ground probabilistic definite clause program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$, rules $\mathcal{R}$ and atoms $\mathcal{A}$, let λ^i_a be the formula associated with atom a in $Tc^i_{\mathcal{P}}(\varnothing)$. For every atom a, total choice $C \subseteq \mathcal{F}$ and iteration i, we have:*

$$C \models \lambda^i_a \quad \rightarrow \quad C \cup \mathcal{R} \models a$$

Proof by induction: $i = 1$: easily verified. $i \rightarrow i+1$: easily verified for $a \in \mathcal{F}$; for $a \in \mathcal{A} \backslash \mathcal{F}$, let $C \models \lambda^{i+1}_a$, that is, $C \models \bigvee_{(\texttt{a :- b}_1,\ldots,\texttt{b}_n)\in\mathcal{P}} (\lambda^i_{b_1} \wedge \cdots \wedge \lambda^i_{b_n})$. Thus, there is a $\texttt{a :- b}_1,\ldots,\texttt{b}_n \in \mathcal{P}$ with $C \models \lambda^i_{b_j}$ for all $1 \leqslant j \leqslant n$. By assumption, $C \cup R \models b_j$ for all such j and thus $C \cup \mathcal{R} \models a$. □

Corollary 1 *After each iteration i, we have* $\mathrm{WMC}(\lambda^i_a) \leqslant \Pr(a)$.

Theorem 2 *For a ground probabilistic definite clause program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$, rules $\mathcal{R}$ and atoms $\mathcal{A}$, let λ^i_a be the formula associated with atom a in $Tc^i_{\mathcal{P}}(\varnothing)$. For every atom a and total choice $C \subseteq \mathcal{F}$, there is an i_0 such that for every iteration $i \geqslant i_0$, we have*

$$C \cup \mathcal{R} \models a \quad \Leftrightarrow \quad C \models \lambda^i_a$$

Proof: $\leftarrow$: Theorem 1. $\rightarrow$: $C \cup \mathcal{R} \models a$ implies $\exists i_0 \forall i \geqslant i_0 : a \in T^i_{C\cup\mathcal{R}}(\varnothing)$. We further show $\forall j : a \in T^j_{C\cup\mathcal{R}}(\varnothing) \rightarrow C \models \lambda^j_a$ by induction. $j = 1$: easily verified. $j \rightarrow j+1$: easily verified for $a \in \mathcal{F}$; for other atoms, $a \in T^{j+1}_{C\cup\mathcal{R}}(\varnothing)$ implies there is a rule $\texttt{a :- b}_1,\ldots,\texttt{b}_n \in \mathcal{R}$ such that $\forall k : b_k \in T^j_{C\cup\mathcal{R}}(\varnothing)$. By assumption, $\forall k : C \models \lambda^j_{b_k}$, and by definition, $C \models \lambda^{j+1}_a$. □

Thus, for every atom a, the λ^i_a reach a fixpoint λ^∞_a exactly describing the possible worlds entailing a, and the $Tc_{\mathcal{P}}$ operator therefore reaches a fixpoint where for all atoms $\Pr(a) = \mathrm{WMC}(\lambda^\infty_a)$.[1]

[1]The finite support condition ensures this happens in finite time.

Normal Logic Programs

The correspondence with the $T_{\mathcal{P}}$ operator allows us to extend the $Tc_{\mathcal{P}}$ operator towards stratified normal logic programs (Sec. 4.2.2). To do so, we apply the $Tc_{\mathcal{P}}$ operator stratum by stratum, that is, we first have to reach a fixpoint for $\mathcal{P}_i$ before considering the rules in $\mathcal{P}_{i+1}$. Let $\mathcal{I}_i$ be the parametrized interpretation for stratum i, the set of formulas for a stratified program with m strata is obtained by:

$$\begin{aligned}
\mathcal{I}_1 &= Tc^{\infty}_{\mathcal{P}_1}(\varnothing) \\
\mathcal{I}_2 &= Tc^{\infty}_{\mathcal{P}_2}(\mathcal{I}_1) \\
&\vdots \\
\mathcal{I}_m &= Tc^{\infty}_{\mathcal{P}_m}(\mathcal{I}_{m-1})
\end{aligned}$$

Following the definition of stratified programs, the formula for a negative literal $\neg a$ is only required once a fixpoint for the positive literal a has been reached. Hence, $\lambda_{\neg a}$ can be obtained as $\neg\lambda_a$.

4.4.2 Exact Inference

Probabilistic inference iteratively calls the $Tc_{\mathcal{P}}$ operator until the fixpoint is reached. This involves incremental formula construction (cf. Definition 4) and equivalence checking (cf. Definition 5). Then, for each query q, the probability is computed as $\mathrm{WMC}(\lambda_q)$. An arbitrary propositional sentence does not efficiently support these operations and we need to represent our sentences λ_a by means of a tractable target language.

An efficient realization of our evaluation algorithm is obtained by representing the formulas in the interpretation $\mathcal{I}$ by means of a Sentential Decision Diagram (SDD), which can handle the required operations efficiently (Darwiche 2011). Hence, we can replace each λ_a in Definition 4 by its equivalent SDD representation Λ_a and each of the *Boolean operations* by the *apply-operator* for SDDs which, given $\circ \in \{\vee, \wedge\}$ and two SDDs Λ_a and Λ_b, returns an SDD equivalent with $(\Lambda_a \circ \Lambda_b)$.

The $Tc_{\mathcal{P}}$ operator is, by definition, called on $\mathcal{I}$. To allow for different evaluation strategies, however, we propose a more fine-grained algorithm where, in each iteration, the operator is only called on one specific atom a, i.e., only the rules for which a is the head are evaluated, denoted by $Tc_{\mathcal{P}}(a, \mathcal{I}^{i-1})$. Note that, in case of normal logic programs, we only consider the rules within one stratum.

Each iteration i of $T_{\mathcal{P}}$-compilation consists of two steps;

1. Select an atom $a \in \mathcal{A}$.
2. Compute $\mathcal{I}^i = Tc_{\mathcal{P}}(a, \mathcal{I}^{i-1})$

The result of Step 2 is that only the formula for atom a is updated and, for each of the other atoms, the formula in $\mathcal{I}^i$ is the same as in $\mathcal{I}^{i-1}$. It is easy to verify that $T_{\mathcal{P}}$-compilation reaches the fixpoint $Tc^{\infty}_{\mathcal{P}}(\varnothing)$ in case the selection procedure frequently returns each of the atoms in $\mathcal{P}$.

Conditional Probabilities Once a fixpoint is reached, conditional probabilities can be computed using Bayes' rule. The probability of a query q, given a set $\mathbf{e}$ of *observed* (or *evidence*) atoms, and a vector $\mathbf{E}$ of observed truth values, is computed as:

$$\Pr(q|\mathbf{e} = \mathbf{E}) = \frac{\mathrm{WMC}(\lambda^{\infty}_{q} \wedge \lambda^{\infty}_{\mathbf{E}})}{\mathrm{WMC}(\lambda^{\infty}_{\mathbf{E}})} \qquad \text{with} \quad \lambda^{\infty}_{\mathbf{E}} = \bigwedge_{e \in \mathbf{E}} \lambda^{\infty}_{e}$$

Hence, computing conditional probabilities additionally requires the construction of a formula $\lambda^{\infty}_{\mathbf{E}}$ that represents the evidence $\mathbf{E}$.

4.4.3 Anytime Inference

Computing the exact probability of a query involves iteratively calling the $Tc_{\mathcal{P}}$ operator until the fixpoint is reached. Our algorithm can be stopped at any time, however, and provides a hard lower bound on the actual probability. Simultaneously, we can compile a second SDD representation for each of the query atoms to also provide a hard upper bound. As computation of the lower and upper bound operates on different formulas, an anytime algorithm should alternate between compiling these formulas.

Lower Bound

Following Theorem 1, we know that, after each iteration i, $\mathrm{WMC}(\lambda^{i}_{a})$ is a lower bound on the probability of atom a, i.e., $\mathrm{WMC}(\lambda^{i}_{a}) \leqslant \Pr(a) = \mathrm{WMC}(\lambda^{\infty}_{a})$, which holds for definite clause programs as well as for stratified normal logic programs. To quickly increase $\mathrm{WMC}(\lambda^{i}_{a})$ and, at the same time, avoid a blow-up of the formulas in $\mathcal{I}$, the selection procedure we employ picks the atom which

maximizes the following heuristic value:

$$\frac{\text{WMC}(\Lambda_a^i) - \text{WMC}(\Lambda_a^{i-1})}{\phi_a \cdot (\textsc{Size}(\Lambda_a^i) - \textsc{Size}(\Lambda_a^{i-1}))/\textsc{Size}(\Lambda_a^{i-1})}$$

where $\textsc{Size}(\Lambda)$ denotes the number of edges in SDD Λ and ϕ_a adjusts for the importance of a in proving queries.

Concretely, Step 1 of $T_\mathcal{P}$-compilation calls $Tc_\mathcal{P}(a, \mathcal{I}^{i-1})$ for each $a \in \mathcal{A}$, computes the heuristic value and returns the atom a' for which this value is the highest. Then, Step 2 performs $\mathcal{I}^i = Tc_\mathcal{P}(a', \mathcal{I}^{i-1})$. Although there is overhead involved in computing the heuristic value, as many formulas are compiled without storing them, this strategy works well in practice. We take as value for ϕ_a the minimal depth of the atom a in the SLD-tree for each of the queries of interest. This value is a measure for the influence of the atom on the probability of the queries.

Example 4.14 For our example program depicted in Figure 4.3, and the query $\mathtt{p}(b, c)$, the use of ϕ_a would give priority to compile $\mathtt{p}(b, c)$ as it is on top of the SLD-tree (see Figure 4.2). Without ϕ_a, the heuristic would give priority to compile $\mathtt{p}(a, c)$ as it has the highest probability.

Upper bound

To compute an upper bound for definite clause programs, we select $\mathcal{F}' \subset \mathcal{F}$ and treat each $f \in \mathcal{F}'$ as a logical fact rather than a probabilistic fact, that is, we conjoin each λ_a with $\bigwedge_{f \in \mathcal{F}'} \lambda_f$. In doing so, we simplify the compilation step of our algorithm, because the number of possible total choices decreases. Furthermore, at a fixpoint, we have an upper bound on the probability of the atoms, i.e., $\text{WMC}(\lambda_a^\infty | \lambda_{\mathcal{F}'}) \geqslant \Pr(a)$, because we overestimate the probability of each fact in $\mathcal{F}'$.

Randomly selecting $\mathcal{F}' \subset \mathcal{F}$ does not yield informative upper bounds (they are close to 1). As a heuristic, we compute for each of the facts the minimal depth in the SLD-trees of the queries of interest and select for $\mathcal{F}'$ all facts whose depth is smaller than some constant d. Hence, we avoid the query to be deterministically `true` as for each of the proofs, i.e., traces in the SLD-tree, we consider at least one probabilistic fact. This yields tighter upper bounds.

Example 4.15 For our example program depicted in Figure 4.3, and the query $\mathtt{p}(b, c)$, both of the edges starting in node b are at a depth of 1 in the SLD-tree (see Figure 4.2). Hence, it suffices to compile only them, and treat both other edges as logical facts, to obtain an upper bound smaller than 1.

4.5 Experiments

Our experiments address the following questions:

Q1 How does $T_{\mathcal{P}}$-compilation compare to exact sequential WMC approaches?

Q2 How does $T_{\mathcal{P}}$-compilation compare to anytime sequential approaches?

Q3 What is the impact of approximating the model?

We compute relevant ground programs as well as CNFs (where applicable) following Fierens et al. (2015) and use the SDD package developed at UCLA[2]. Experiments are run on machines with 16 GB of memory.

4.5.1 Exact Inference

We use datasets of increasing size from two domains:

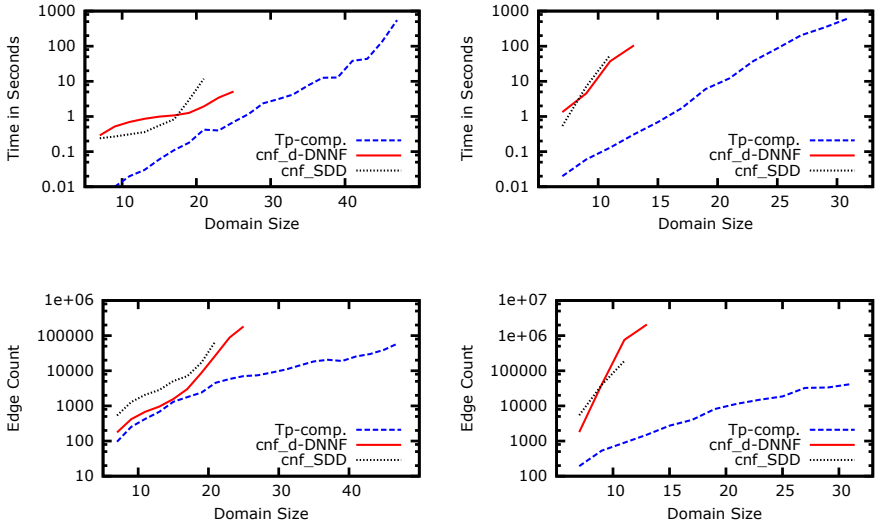
Smokers. Following Fierens et al. (ibid.), we generate random power law graphs for the standard 'Smokers' social network domain. Cycles in the program are introduced by the following rule:

$$\texttt{smokes}(\mathsf{X})\ \texttt{:-}\ \texttt{friends}(\mathsf{X},\mathsf{Y}),\texttt{smokes}(\mathsf{Y}).$$

Alzheimer. We use series of connected subgraphs of the Alzheimer network of De Raedt, Kimmig, and Toivonen (2007), starting from a subsample connecting the two genes of interest 'HGNC 582' and 'HGNC 983', and adding nodes that are connected with at least two other nodes in the graph. The logic program (i.e., the set of rules) is similar to the one we used in our example (see Figure 4.3) and cycles are introduced by the second rule for `path`.

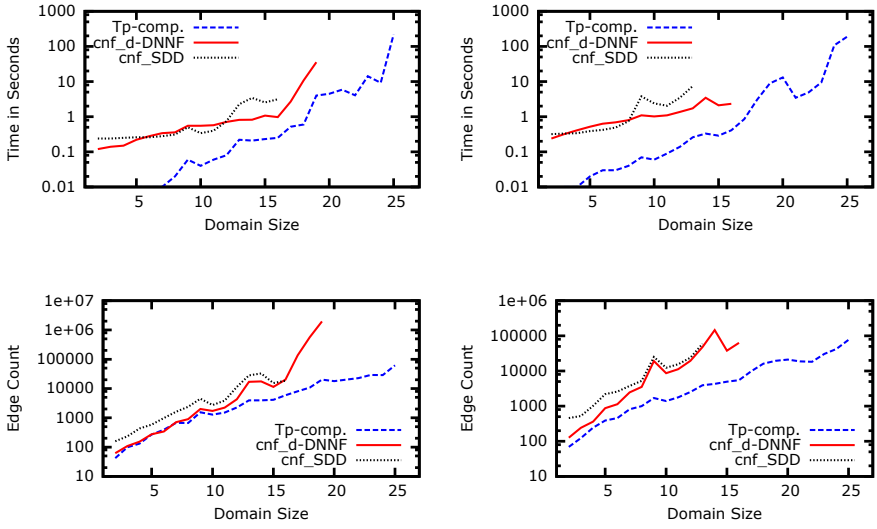
The relevant ground program is computed for one specific query as well as for multiple queries. For the *Smokers* domain, this is `cancer(p)` for a specific person `p` versus for each person. For the *Alzheimer* domain, this is the connection between the two genes of interest versus all genes.

For the sequential approach, we perform WMC using either SDDs, or d-DNNFs compiled with c2d[3] (Darwiche 2004). For each domain size (#persons or #nodes) we consider nine instances with a timeout of one hour per setting. We report median running times and target representation sizes, using the standard measure of #edges for the d-DNNFs and $3 \cdot$ #edges for the SDDs. The results are depicted in Figure 4.6 and 4.7 and provide an answer for **Q1**.



One query. Multiple queries.

Figure 4.6: Exact inference on *Alzheimer*.



One query. Multiple queries.

Figure 4.7: Exact inference on *Smokers*.

In all cases, the $T_\mathcal{P}$-compilation (`Tp-comp`) scales to larger domains than the sequential approach with both SDD (`cnf_SDD`) and d-DNNF (`cnf_d-DNNF`) and produces smaller compiled structures, which makes subsequent WMC computations more efficient. The smaller structures are mainly obtained because our approach does not require auxiliary variables to correctly handle the cycles in the program. For *Smokers*, all queries depend on almost the full network structure, and the relevant ground programs – and thus the performance of $T_\mathcal{P}$-compilation – for one or all queries are almost identical. The difference between the settings for the sequential approaches is due to CNF conversion introducing more variables in case of multiple queries.

4.5.2 Anytime Inference

We consider an approximated ($\mathcal{P}_{apr}$) as well as the original ($\mathcal{P}_{org}$) model of two domains:

Genes. Following Renkens, Kimmig, et al. (2014) and Renkens, Van den Broeck, et al. (2012), we use the biological network of Ourfali et al. (2007) and its 500 connection queries on gene pairs. The logic program is similar to the one we used in our example (see Figure 4.3). The original $\mathcal{P}_{org}$ considers connections of arbitrary length, whereas $\mathcal{P}_{apr}$ restricts connections to a maximum of five edges.

WebKB. We use the WebKB[4] dataset restricted to the 100 most frequent words (Davis and Domingos 2009) and with random probabilities from [0.01, 0.1]. Cycles in the program are introduced by the following rule:

$$\texttt{hasClass}(\mathsf{P},\mathsf{C})\ \texttt{:-}\ \texttt{linksTo}(\mathsf{P},\mathsf{P2}),\texttt{hasClass}(\mathsf{P2},\mathsf{C2}).$$

Following Renkens, Kimmig, et al. (2014), $\mathcal{P}_{apr}$ is a random subsample of 150 pages. $\mathcal{P}_{org}$ uses all pages from the Cornell database. This results in a dataset with 63 queries for the class of a page.

We employ the anytime algorithm as discussed in Sections 4.4.3 and 4.4.3 and alternate between computations for lower and upper bound at fixed intervals. We compare against two sequential approaches. The first compiles subformulas of the CNF selected by weighted PMAX-SAT (WPMS) (ibid.), the second approximates the WMC of the formula by sampling using the MC-SAT algorithm implemented in the Alchemy package[5].

[2] `http://reasoning.cs.ucla.edu/sdd/`
[3] `http://reasoning.cs.ucla.edu/c2d/`
[4] `http://www.cs.cmu.edu/webkb/`
[5] `http://alchemy.cs.washington.edu/`

		$\mathcal{P}_{apr}$		$\mathcal{P}_{org}$	
		WPMS	$T_{\mathcal{P}}$-comp	WPMS	$T_{\mathcal{P}}$-comp
Genes	`Almost Exact`	308	419	0	30
	`Tight Bound`	135	81	0	207
	`Loose Bound`	54	0	0	263
	`No Answer`	3	0	500	0
WebKB	`Almost Exact`	1	7	0	0
	`Tight Bound`	2	34	0	19
	`Loose Bound`	2	22	0	44
	`No Answer`	58	0	63	0

Table 4.1: Anytime inference: Number of queries with difference between bounds < 0.01 (`Almost Exact`), in $[0.01, 0.25)$ (`Tight Bound`), in $[0.25, 1.0)$ (`Loose Bound`), and 1.0 (`No Answer`).

		$\mathcal{P}_{apr}$		$\mathcal{P}_{org}$
		MCsat_{5000}	MCsat_{10000}	MCsat
Genes	`In Bounds`	150	151	0
	`Out Bounds`	350	349	0
	`N/A`	0	0	500

Table 4.2: Anytime inference with MC-SAT: numbers of results within and outside the bounds obtained by $T_{\mathcal{P}}$-compilation on $\mathcal{P}_{apr}$, using 5000 or 10000 samples per CNF variable.

Following Renkens, Kimmig, et al. (2014), we run inference for each query separately. The time budget is 5 minutes for $\mathcal{P}_{apr}$ and 15 minutes for $\mathcal{P}_{org}$ (excluding the time to construct the relevant ground program). For MC-SAT, we sample either 5,000 or 10,000 times per variable in the CNF, which yields approximately the same runtime as our approach. Results are depicted in Tables 4.1, 4.2 and Figure 4.8 and allow us to answer **Q2** and **Q3**.

Table 4.1 shows that $T_{\mathcal{P}}$-compilation returns bounds for all queries in all settings, whereas WPMS did not produce any answer for $\mathcal{P}_{org}$. The latter is due to reaching the time limit before conversion to CNF was completed. For the approximate model $\mathcal{P}_{apr}$ on the Genes domain, both approaches solve a majority of queries (almost) exactly. Figure 4.8 plots the number of queries that reached a bound difference below different thresholds against the running time, showing that $T_{\mathcal{P}}$-compilation converges faster than WPMS. Furthermore, this figure shows that the choice of our timeout has only a limited impact on the results. Finally, for the Genes domain, Table 4.2 shows the number of queries where the result of MC-SAT (using different numbers of samples per variable in the CNF) lies within or outside the bounds computed by $T_{\mathcal{P}}$-compilation.

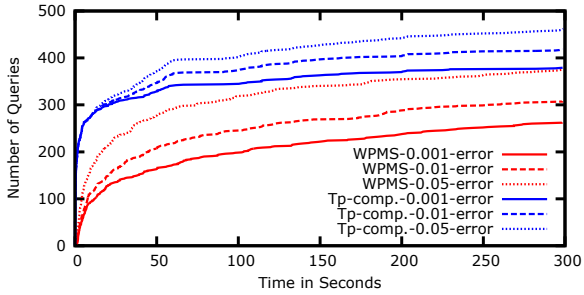


Figure 4.8: Anytime inference on Genes with $\mathcal{P}_{apr}$: number of queries with bound difference below threshold at any time.

For the original model, no complete CNF is available within the time budget; for the approximate model, more than two thirds of the results are outside the guaranteed bounds obtained by our approach.

We further observed that for 53 queries on the Genes domain, the lower bound returned by our approach using the original model is higher than the upper bound returned by WPMS with the approximated model. This illustrates that computing upper bounds on an approximate model does not provide any guarantees with respect to the full model. On the other hand, for 423 queries in the Genes domain, $T_{\mathcal{P}}$-compilation obtained higher lower bounds with $\mathcal{P}_{apr}$ than with $\mathcal{P}_{org}$, and lower bounds are guaranteed in both cases.

In summary, we conclude that approximating the model can result in misleading upper bounds, but reaches better lower bounds (**Q3**), and that $T_{\mathcal{P}}$-compilation outperforms the sequential approaches for time, space and quality of result in all experiments (**Q2**).

4.6 Related Work

During the last few years there has been a significant interest in combining relational structure with uncertainty. This has resulted in the fields of statistical relational learning (De Raedt, Frasconi, et al. 2008; Getoor and Taskar 2007), probabilistic programming (Pfeffer 2014) and probabilistic databases (Suciu et al. 2011), which all address this combination. Probabilistic logic programming (PLP) languages such as PRISM (Sato 1995), ICL (Poole 1993), ProbLog (De Raedt, Kimmig, and Toivonen 2007), LPADs (Vennekens, Verbaeten, et al. 2004) and CP-logic (Vennekens, Denecker, et al. 2009) form one stream of work in these fields where logic programming languages are extended with

probabilities. Statistical relational learning (SRL) techniques such as Markov Logic (Poon and Domingos 2006) and relational Bayesian networks (Jaeger 1997) form a different stream where graphical models are extended with relational representations.

Knowledge compilation and weighted model counting has shown to be very effective for inference in probabilistic logic programs (De Raedt, Kimmig, and Toivonen 2007; Fierens et al. 2015; Riguzzi 2007; Riguzzi and Swift 2011) as well as relational Bayesian networks (Chavira, Darwiche, and Jaeger 2006). Furthermore, compiling a formula in a bottom-up manner has shown to compare favorable against compiling in a top-down manner for probabilistic graphical models (Choi et al. 2013).

Exact compilation of a propositional formula is computational expensive and one often has to resort to approximate techniques. One way to do so is to first convert the program into a propositional formula, as done for exact compilation, but then only compile selected subformulas (Renkens, Kimmig, et al. 2014) or feed the formula to a sampling algorithm, e.g. MC-SAT (Poon and Domingos 2006). In case also construction of the complete propositional formula becomes infeasible, one can transform the original program to an approximate, simplified program that represents, ideally, a similar probability distribution (Renkens, Van den Broeck, et al. 2012). Other approaches employ forward reasoning to directly sample on the logic program, e.g., Goodman et al. (2008), Gutmann et al. (2011), Milch et al. (2005), and Nitti et al. (2014), but these do not provide guaranteed lower or upper bounds on the probability of the queries. Anytime PLP algorithms based on backward reasoning have been proposed in the past but they do not allow to answer multiple queries in parallel (De Raedt, Kimmig, and Toivonen 2007; Poole 1993). The problem of highly cyclic domains has recently also been addressed using lazy clause generation (Aziz et al. 2015), but only for exact inference.

Probabilistic logic programs under the distribution semantics define a distribution over possible worlds, which randomly fixes the truth values of probabilistic facts and then permits any type of logical reasoning within a possible world. While our approach focusses on stratified programs with finite support, the fixpoint operator is more recently also extended towards general normal programs with function symbols (Bogaerts and Van den Broeck 2015; Riguzzi 2016). A second class of probabilistic Prologs, including Stochastic Logic Programs (SLPs) (Muggleton 1996), uses a different approach, where a distribution over the groundings of a query is defined based on a distribution over the derivations in the query's SLD tree, making an independent decision on which branch to take at every node. The semantics is thus closely tied to backward reasoning, and our forward reasoning based approach does not easily apply in this setting.

4.7 Conclusions

We have introduced $T_\mathcal{P}$-compilation, a novel anytime inference approach for probabilistic logic programs that combines the advantages of forward reasoning with state-of-the-art techniques for weighted model counting and knowledge compilation. Concretely, $T_\mathcal{P}$-compilation acts directly on the probabilistic logic program and does not require to completely unfold and encode the model as an intermediate propositional knowledge base. Our extensive experimental evaluation demonstrates that the new technique outperforms existing exact and approximate techniques on real-world applications such as biological and social networks and web-page classification.

The advantage of forward reasoning, as used by $T_\mathcal{P}$-compilation, is that it naturally deals with cyclic dependencies and avoids the expensive introduction of additional variables and propositions. A drawback of forward reasoning, compared to backward reasoning, is that it blindly generates new knowledge without taking into account the queries of interest. To accommodate for this, we proposed to first use backward reasoning to compute the relevant ground program and defined $T_\mathcal{P}$-compilation on this ground program. This is not a strict requirement, however. We can perform $T_\mathcal{P}$-compilation directly on the non-ground probabilistic logic program and use a magic set transformation (Bancilhon et al. 1986; Gutmann et al. 2011), forcing forward reasoning to only generate relevant knowledge for the queries of interest.

The overhead on first unfolding and encoding the complete model becomes especially clear in the context of anytime inference. Where $T_\mathcal{P}$-compilation could compute informative bounds for any of the problem instances considered in our experimental evaluation, techniques relying on a successful construction of a formula in CNF could not return an answer for any of the original problems. On the other hand, our anytime algorithm also comes with some limitations. Computation of the upper bounds only holds for definite clause programs as threating a subset of probabilistic facts as logical facts in only an overestimate for programs without negation. Computation of the lower bounds is valid for programs with negation but rules need to be compiled according to their stratification. Furthermore, better lower bounds are obtained when approximating the model, implying that our heuristic does not yet optimally deals with the large search space. Hence, especially in the context of anytime inference, there is still room for improvement.

Chapter 5

Probabilistic Logic Programs with Time

5.1 Introduction

Probabilistic logic programs are generally assumed to represent a static model where the domain for each of the objects is fixed and fully known. When modeling real-world problems and applications such as robot tracking, online multi-player games and medical diagnosis, this assumption does not always reflect the real situation. Firstly, the program might represent a dynamic model where dependencies between objects range over different moments in time. Secondly, frequent updates to the program might be needed to cope with an ever changing environment. While most relational representations allow us to deal with time-related information, the general-purpose inference techniques developed for these representations are known to scale poorly in this context. Significant speed-ups can be obtained by treating time as a first-class citizen.

We differentiate two types of probabilistic logic programs with time; programs where time is present in an *implicit* way and programs that *explicitly* model time. The probabilistic facts in a probabilistic logic program typically represent the knowledge available in some sort of database. Changes to this database require to update the facts in the program and implicitly introduce a dependency over time. Explicitly modeling time in the program allows one to represent a relational stochastic process where the current state depends on the past and influences the future. These dynamic relational models are, for example, useful in the context of robot tracking as it allows to take into account the maximal

displacement a robot can make in between two consecutive time steps.

In this Chapter, we extend our $T_\mathcal{P}$-compilation inference algorithm towards probabilistic logic programs with time. In the presence of program updates, we avoid the need to completely recompile the model by reusing past compilation results. This can cause significant savings compared to restarting inference from scratch. For dynamic relational models, we unify $T_\mathcal{P}$-compilation with the structural interface algorithm to combine the benefits of both approaches. The resulting algorithm does not only exploit the repeated structure in the underlying model, but additionally exploits local structure in the belief state maintained during inference. This allows for a more compact representation of the belief state and further speeds-up inference.

A extension of $T_\mathcal{P}$-compilation towards probabilistic logic programs with time and experimental results were previously published in

J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2016b). "Tp-Compilation for inference in probabilistic logic programs". In: *International Journal of Approximate Reasoning* 78, pp. 15–32

Structure of this Chapter

This chapter is organized as follows. Section 5.2 introduces probabilistic logic programs where time is present in an implicit or explicit way. Section 5.3 extends $T_\mathcal{P}$-compilation to efficiently deal with program updates. Dynamic $T_\mathcal{P}$-compilation deals with dynamic relational models and is introduced in Section 5.4. Section 5.5 shows experimental results. We discuss related work in Section 5.6 and conclude this Chapter in Section 5.7.

5.2 Preliminaries

We now shortly introduce two types of probabilistic logic programs with time.

5.2.1 Programs with Implicit Time

A probabilistic logic program typically represents a stationary model where the set of rules $\mathcal{R}$ as well as the set of facts $\mathcal{F}$ is fixed. As the world is constantly evolving, a static program only reflects our knowledge at one specific moment in time while the underlying process is subject to an ever changing environment.

In a more general setting, one would pass on these changes to the program and inference is required to update our beliefs in the presence of the new information.

In general, new knowledge is reflected by an update of the facts $\mathcal{F}$ while the non-ground rules $\mathcal{R}$ in the program remains unchanged. In case the updates are relevant to the queries of interest, the set of ground rules in the relevant ground program will change. For definite clause programs, adding facts leads to new clauses in the ground program while removing facts removes clauses from the grounded program. For programs with negation, one cannot easily make this distinction as adding facts might remove clauses from the grounded program and removing facts might add clauses to the program.

Example 5.1 Consider the three probabilistic graphs as shown in Figure 5.1. For the graph on the left, we have following probabilistic facts:

$$0.4 :: \texttt{edge}(b, a). \qquad 0.3 :: \texttt{edge}(b, c). \qquad 0.8 :: \texttt{edge}(a, c).$$

For the graph in the middle, we have:

$$0.4 :: \texttt{edge}(b, a). \qquad 0.3 :: \texttt{edge}(b, c).$$

$$0.8 :: \texttt{edge}(a, c). \qquad 0.9 :: \texttt{edge}(c, a).$$

And, for the graph on the right, we have:

$$0.4 :: \texttt{edge}(b, a). \qquad 0.8 :: \texttt{edge}(a, c). \qquad 0.9 :: \texttt{edge}(c, a).$$

$$0.3 :: \texttt{edge}(b, c). \qquad 0.6 :: \texttt{edge}(b, d).$$

Each of the corresponding programs will have the same set of non-ground rules:

$$\texttt{path}(\mathsf{X}, \mathsf{Y}) \texttt{ :- } \texttt{edge}(\mathsf{X}, \mathsf{Y})).$$

$$\texttt{path}(\mathsf{X}, \mathsf{Y})) \texttt{ :- } \texttt{edge}(\mathsf{X}, \mathsf{Z})), \texttt{path}(\mathsf{Z}), \mathsf{Y})).$$

The relevant ground program will be different for each of the graphs. For example, the relevant ground program for the graphs in the middle and on the right will contain the ground rule

$$\texttt{path}(c, c) \texttt{ :- } \texttt{edge}(c, a), \texttt{path}(a, c).$$

which is not part of the relevant ground program for the graph on the left as this graph does not contain an edge going from node c to node a.

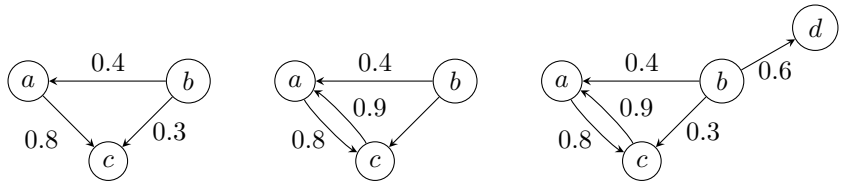


Figure 5.1: A sequence of probabilistic graphs.

5.2.2 Programs with Explicit Time

Probabilistic logic programs have the ability to compactly represent a stochastic process allowing us to reason about the past, the present and the future. This is done by means of a dynamic model (Nitti et al. 2013), where one inserts time in an explicit manner. Each rule is of the form $\mathtt{h_t} :- \mathtt{b_{1,t1}}, ..., \mathtt{b_{n,tn}}$, where $\mathtt{t}, \mathtt{t1}, \ldots, \mathtt{tn}$ denotes the time step to which the corresponding atom belongs. As for dynamic Bayesian networks (cf. Chapter 3), we assume that the first-order Markov property holds, and the rules can be rewritten as $\mathtt{h_t} :- \mathtt{b_{1,t}}, \ldots, \mathtt{b_{m,t}}, \mathtt{b_{m+1,t-1}}, \ldots, \mathtt{b_{n,t-1}}$, i.e., each atom can only depend on atoms from the same or previous time step. This also implies that cycles cannot range over different time steps.

Example 5.2 Consider the following probabilistic logic program which, intuitively, models a a dynamic social network where people are more likely to be sick in case they have friends who are sick, or if they were sick on a previous time step:

$0.2 :: \mathtt{coldOutside}(\mathsf{T}).$

$0.4 :: \mathtt{staySick}(\mathsf{X}, \mathsf{T}).$

$\mathtt{sick}(\mathsf{X}, \mathsf{T})$:- $\mathtt{coldOutside}(\mathsf{T}).$

$\mathtt{sick}(\mathsf{X}, \mathsf{T})$:- $\mathtt{friends}(\mathsf{X}, \mathsf{Y}), \mathtt{sick}(\mathsf{Y}, \mathsf{T}).$

$\mathtt{sick}(\mathsf{X}, \mathsf{T})$:- Tprev is $\mathsf{T} - 1, \mathtt{sick}(\mathsf{X}, \mathsf{Tprev}), \mathtt{staySick}(\mathsf{X}, \mathsf{T}).$

The variable T is used to denote a specific time step. We assume the `friends` relation remains constant over time and therefore omit the time variable.

To simplify notation, we will denote the time parameter by means of a subscript:

$$0.2 :: \texttt{coldOutside}_t.$$

$$0.4 :: \texttt{staySick}(\mathsf{X})_t.$$

$$\texttt{sick}(\mathsf{X})_t \texttt{ :- coldOutside}_t.$$

$$\texttt{sick}(\mathsf{X})_t \texttt{ :- friends}(\mathsf{X}, \mathsf{Y}), \texttt{sick}(\mathsf{Y})_t.$$

$$\texttt{sick}(\mathsf{X})_t \texttt{ :- sick}(\mathsf{X})_{t-1}, \texttt{staySick}(\mathsf{X})_t.$$

In the context of dynamic models, one distinguishes the initial model and the transition model (cf. Section 3.2.1). The former expresses the *prior distribution*, i.e., the distribution for the first time step, while the latter latter serves as a template for all subsequent time steps. We will focus on the transition model as this is the one that repeats over time.

5.3 Inference with Program Updates

Updates to a probabilistic logic program, by means of adding or removing probabilistic facts, will be reflected in the relevant ground program. One way to deal with these program changes is to restart inference from scratch after each update of $\mathcal{F}$. In the context of our $T_\mathcal{P}$-compilation inference algorithm, as introduced in the previous Chapter, this involves recompiling the new program without using any of the past compilation results. In case $\mathcal{F}$ changes only marginally, however, a more efficient approach is to restart from the previously compiled sentences. This can be achieved using the $Tc_\mathcal{P}$ operator (cf. Section 4.4.1), which allows for adding clauses to and removing clauses from the program.

In the remainder of this Section, we only consider program updates for definite clause programs. For normal logic programs, we would need to restart compilation from the lowest strata affected by the update and this is only advantageous in a limited number of cases.

Adding Clauses

For definite clause programs, we know that employing the $T_\mathcal{P}$ operator on a subset of the fixpoint reaches the fixpoint (see Property 1, Chapter 4). Moreover, adding definite clauses leads to a superset of the fixpoint (see Property 2,

Chapter 4). Hence, it is safe to restart the $T_{\mathcal{P}}$ operator from a previous fixpoint after adding clauses. Assume the set of facts $\mathcal{F}$ is extended to $\mathcal{F}'$, then we have:

$$T^{\infty}_{\mathcal{R}\cup\mathcal{F}'}(\varnothing) = T^{\infty}_{\mathcal{R}\cup\mathcal{F}'}(T^{\infty}_{\mathcal{R}\cup\mathcal{F}}(\varnothing))$$

Due to the correspondence established in Theorem 2, this also applies to $Tc_{\mathcal{P}}$. Intuitively, the $Tc_{\mathcal{P}}$ operator aims to construct a parametrized interpretation that reflects all the knowledge available in the relevant ground program. When clauses are added to the program, we simply need some additional iterations of the $Tc_{\mathcal{P}}$ operator in order to add this new information to the parametrized interpretation.

Example 5.3 Consider the sequence of graphs depicted in Figure 5.1 where we move from left to right. Adding $0.9 :: \texttt{edge}(c, a)$ to the program leads to new paths, such as $\texttt{path}(a, a)$, and increases the probability of existing paths, such as $\texttt{path}(b, a)$. Using past compilation results is especially advantageous when adding $0.6 :: \texttt{edge}(b, d)$ to the program as it does not affect any of the existing paths but only adds one new path, being $\texttt{path}(b, d)$.

Removing Clauses

When removing clauses from a definite clause program, atoms in the fixpoint may become invalid. We therefore reset the computed fixpoints for all total choices where the removed clause could have been applied. This is done by conjoining each of the formulas in the parametrized interpretation with the negation of the formula for the head of the removed clause. Let $\mathcal{I}$ denote the parametrized interpretation and $\mathcal{H}$ the set of atoms in the head of a removed clause, the new parameterized interpretation $\mathcal{I}'$ is obtained as:

$$\mathcal{I}' = \{(a, \lambda'_a) \mid (a, \lambda_a) \in \mathcal{I} \text{ with } \lambda'_a = \lambda_a \wedge \bigwedge_{h\in\mathcal{H}} \neg\lambda_h\}$$

Then, we restart the $Tc_{\mathcal{P}}$ operator from the adjusted parametrized interpretation $\mathcal{I}'$, to recompute the fixpoint for the total choices that were removed. Intuitively, removing clauses from the program requires to also remove this information from the parametrized interpretation. Conjoining each of the formulas with the negation of the removed information is a drastic, yet safe, option to do so.

Example 5.4 Consider the sequence of graphs depicted in Figure 5.1 where we now move from right to left. Removing $0.6 :: \texttt{edge}(b, d)$ from the program only removes $\texttt{path}(b, d)$ and none of the other compiled sentences will be affected by this update. Removing $0.9 :: \texttt{edge}(c, a)$ from the program will completely

remove some of the paths, such as $\texttt{path}(a, a)$, and decreases the probability of other paths, such as $\texttt{path}(b, a)$. Hence, we have to adapt the compiled formula for $\texttt{path}(b, a)$ to adopt for this new situation.

5.4 Inference in Dynamic Relational Models

A stochastic process can be compactly represented by means of a probabilistic logic program where a variable is used to make abstraction of time. Inference in these dynamic relational models can be done by first *unrolling* the complete model for a finite number of time steps, after which any standard inference algorithm applies. But, as in the propositional case, these general-purpose techniques are known to scale poorly for an increasing number of time steps.

We now extend our $T_{\mathcal{P}}$-compilation approach to treat time as a first class citizen and, in a similar way as done by the structural interface algorithm, this avoids the need to explicitly unroll the model. Compared to the structural interface algorithm, however, $T_{\mathcal{P}}$-compilation allows for a more flexible approach where local structure in the forward message can be exploited to further speed-up inference.

5.4.1 The Forward Message

Filtering in dynamic relational models, similar as for dynamic Bayesian networks, boils down to performing a forward pass trough the underlying model. This can be done by iteratively computing the forward message, that is, the joint probability distribution over all atoms in the interface. For dynamic relational models, the interface $\mathbf{i}_t$ for the grounded program $\mathcal{P}$ is given by:

$$\mathbf{i}_t = \{\mathtt{b}_{\mathtt{m}+1,\mathtt{t}-1}, \ldots, \mathtt{b}_{\mathtt{n},\mathtt{t}-1} \mid \mathtt{h}_{\mathtt{t}} :\!-\, \mathtt{b}_{1,\mathtt{t}}, \ldots, \mathtt{b}_{\mathtt{m},\mathtt{t}}, \mathtt{b}_{\mathtt{m}+1,\mathtt{t}-1}, \ldots, \mathtt{b}_{\mathtt{n},\mathtt{t}-1} \in \mathcal{P}\}$$

The forward message can be computed recursively as follows:

$$\Pr(\mathbf{i}_t|\mathbf{E}_{1:t}) = \sum_{\mathbf{i}_{t-1}} \Pr(\mathbf{i}_t|\mathbf{i}_{t-1}, \mathbf{E}_t) \Pr(\mathbf{i}_{t-1}|\mathbf{E}_{1:t-1}) \tag{5.1}$$

with $\mathbf{E}_t$ the truth values of the observed atoms at time. Then, $\Pr(\mathbf{i}_{t-1}|\mathbf{E}_{1:t-1})$ allows us to compute $\Pr(q_t|\mathbf{E}_{1:t})$ for each of the query atoms q_t.

Example 5.5 For the dynamic model presented in Example 5.2 and a domain of three persons, `ann`, `bob` and `cin`, the interface consists of three atoms:

$$\mathbf{i}_t = \{\texttt{sick(ann)}_t, \texttt{sick(bob)}_t, \texttt{sick(cin)}_t\}.$$

The forward pass involves computing the probability of each possible value assignment to the variables in the interface, given the forward message from the previous time step and evidence:

$$\Pr\left(\texttt{sick(ann)}_t \wedge \texttt{sick(bob)}_t \wedge \texttt{sick(cin)}_t \mid \mathbf{i}_{t-1}, \mathbf{E}_t\right),$$

$$\Pr\left(\texttt{sick(ann)}_t \wedge \texttt{sick(bob)}_t \wedge \neg\texttt{sick(cin)}_t \mid \mathbf{i}_{t-1}, \mathbf{E}_t\right),$$

$$\ldots$$

$$\Pr\left(\neg\texttt{sick(ann)}_t \wedge \neg\texttt{sick(bob)}_t \wedge \neg\texttt{sick(cin)}_t \mid \mathbf{i}_{t-1}, \mathbf{E}_t\right)$$

5.4.2 Dynamic $T_{\mathcal{P}}$-compilation

To efficiently compute the forward message, our $T_{\mathcal{P}}$-compilation approach is extended to operate in two phases:

- *Offline phase*: Treat the transition model as a static model and run $T_{\mathcal{P}}$-compilation on this model until a fixpoint is reached. The resulting parametrized interpretation $\mathcal{I}$ contains a tuple (a, Λ_a) for each atom a in the transition model. The compiled formulas Λ_a will then serve as a template for each time step that inference is required. The offline phase only has to be performed once.
- *Online phase*: Iterate over time steps t and compute the forward message. This includes extending the template formulas Λ_a, by means of an additional compilation step, to adjust for the situation at time t. The online phase has to be done for each $t < T$, with T the time span.

Computation of the forward message at time t requires extending the template formulas Λ_a to take into account the evidence for time step t, denoted with $\mathbf{E}_t$, as well as the forward message computed for $t-1$. We use Λ_a^t to denote the adjusted formula of atom a for time step t. Let $\mathbf{I}_t$ be one truth-value assignment to atoms in $\mathbf{i}_t$, its conditional probability is computed as:

$$\Pr(\mathbf{I}_t|\mathbf{E}_{1:t}) = \frac{\mathrm{WMC}\left(\bigwedge_{i\in\mathbf{I}_t} \Lambda_i^t\right)}{\mathrm{WMC}(\Lambda_{\mathbf{E}_t}^t)} \tag{5.2}$$

with

$$\Lambda_i^t = \bigvee_{\mathbf{I}^j \in \mathbf{i}} \left((\Lambda_i \wedge \Lambda_{\mathbf{E}_t}) | \mathbf{I}^j \right) \wedge state_{t-1}^j \wedge \mathbf{I}^j \tag{5.3}$$

where $\Lambda_{\mathbf{E}_t}$ is the formula representing the evidence (cf. Section 4.4.2) and $\Lambda|\mathbf{I}$ denotes that the formula Λ is conditioned on the values in $\mathbf{I}$. The auxiliary variable $state_{t-1}^j$ is required to correctly include the distribution from $t-1$ (cf. Equation 5.1) and the weight function sets $w(state_{t-1}^j) = \Pr(\mathbf{I}_{t-1}^j | \mathbf{E}_{1:t-1})$. The forward message (the complete distribution) is obtained by repeating Equation 5.2 for each $\mathbf{I}_t \in \mathbf{i}_t$.

To push more of the computational effort towards the offline phase, we could rewrite Equation 5.3 as:

$$\Lambda_i^t = \bigvee_{\mathbf{I}^j \in \mathbf{i}} \left(\Lambda_i | \mathbf{I}^j \right) \wedge \left(\Lambda_{\mathbf{E}_t} | \mathbf{I}^j \right) \wedge state_{t-1}^j \wedge \mathbf{I}^j \tag{5.4}$$

where, now, $\Lambda_i|\mathbf{I}^j$ is independent of the time step t and can be computed in the offline phase, i.e. it needs to be computed only once.

5.4.3 Dealing with Evidence

The key difference of dynamic $T_{\mathcal{P}}$-compilation, as presented in the previous section, compared to the structural interface algorithm, as presented in Section 3.4, is how evidence is treated.

The incentive of the structural interface algorithm is to push as much of the computational overhead, i.e., all compilation steps, into the offline phase. This requires the compilation of one large formula that contains a variable for each of the atoms in the transition model, including the ones that are (potentially) observed. Then, the forward message is computed by accordingly setting weights to incorporate the observations and requires an exponential number of WMC calls.

Dynamic $T_{\mathcal{P}}$-compilation, on the other hand, does not incorporate evidence by setting weights but adjusts the template formulas Λ_a to explicitly represent the evidence. This requires, for each time step t, to first compile the formula representing the evidence, which we denote as $\Lambda_{\mathbf{E}_t}$. Then $\Lambda_{\mathbf{E}_t}$ is conjoined with each of the required template formulas (cf. Equation 5.3).

While $T_{\mathcal{P}}$-compilation requires an additional compilation step within the *online phase*, which might be computationally expensive, an explicit representation of the evidence allows us to exploit local structure in the forward message. Indeed, if $w(state_{t-1}^j) = 0$ or if $(\Lambda_i \wedge \Lambda_{\mathbf{E}_t}) \equiv \bot$, Equation 5.3 simplifies significantly.

Hence, we avoid an exponential representation of the forward message in case $\Pr(\mathbf{i}_t|\mathbf{E}_{1:t})$ exhibits deterministic dependencies.

Example 5.6 Assume that, for our running example, we have evidence that `ann` is sick at time t. In this case, each of the truth-value assignments $\mathbf{I}_t$ to atoms in $\mathbf{i}_t$ that contains $\neg\texttt{sick}(\texttt{ann})_t$ (see Example 5.5) has a probability of zero as it does not coincide with the evidence. Hence, we can use a more compact representation for the forward message where we have $\Pr(\neg\texttt{sick}(\texttt{ann})_t \mid \mathbf{i}_{t-1}, \mathbf{E}_t) = 0$.

Where Equation 5.4 allows to push more of the computational effort to the offline phase, it cannot fully exploit deterministic dependencies in the forward message. As $\Lambda_i|\mathbf{I}^j$ is only computed once, we have to iterate over each $\mathbf{I}^j \in \mathbf{i}$ and need to maintain an exponential representation of the forward message. Hence, Equation 5.4 is only beneficial in case there is not enough determinism to compensate for the computational overhead in Equation 5.3. In our experimental evaluation we will refer to Equation 5.3 as *flexible* and to Equation 5.4 as *fixed* dynamic $T_\mathcal{P}$ compilation.

5.5 Experiments

Our experiments address the following questions:

Q1 Can we efficiently deal with program updates?

Q2 Can we efficiently exploit evidence in dynamic domains?

Experiments are run on machines with 16 GB of memory and we use a timeout of 1 hour.

5.5.1 Program Updates

We use datasets of increasing size from two domains:

Alzheimer. The Alzheimer domain as introduced in Section 4.5.1.

Smokers. A variant on the smokers domain as introduced in Section 4.5.1. Its domain consists of 150 persons and is the union of ten different random power law graphs with 15 nodes each.

We compare our standard $T_\mathcal{P}$-compilation algorithm (`Tp-comp`), which compiles the networks for each of the domain sizes from scratch, with the online algorithm

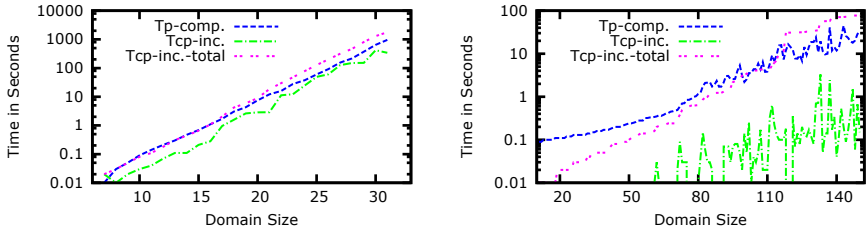


Figure 5.2: Program updates for *Alzheimer* (left) and *Smokers* (right).

(`Tcp-inc`) discussed in Section 5.3. We consider the multiple queries setting only, as it is the one that considers a "complete" grounding of the program, and report median results for nine runs. The results are depicted in Figure 5.2 and provide an answer to **Q1**.

For the Alzheimer domain, which is highly connected, incrementally adding the nodes (`Tcp-inc`) has no real benefit compared to recompiling the network from scratch (`Tp-comp`) and, consequently, the cumulative time of the incremental approach (`Tcp-inc-total`) is higher. For the smokers domain, on the other hand, the incremental approach is more efficient compared to recompiling the network, as it only updates the subnetwork to which the most recent person has been added.

5.5.2 Dynamic Inference

We evaluate dynamic $T_{\mathcal{P}}$-compilation on two different domains:

Mastermind. We represent the *mastermind game* (Chavira, Darwiche, and Jaeger 2006) as a dynamic model as done in Section 3.6.

Sickness. We use the dynamic *sickness* domain as depicted in Example 5.2 with random power law graphs to represent the networks. For each time step, evidence is randomly generated for $x\%$ of the `sick` atoms. The goal is to compute the belief state of the persons being sick after 10 time steps.

We consider nine instances and report median results. For the *mastermind* domain, we compare *fixed* as well as *flexible* dynamic $T_{\mathcal{P}}$ compilation (cf. Section 5.4.3) with the structural interface algorithm (SIA) where the target representation for compilation is d-DNNF. For the *sickness domain*, we compare *flexible* dynamic $T_{\mathcal{P}}$-compilation for three different levels of evidence, being on 0%, 33% and 66% of the `sick` atoms. The results are depicted in Table 5.1 and 5.2 and allow us to answer **Q2**.

Model	SIA size #edges	SIA $T_{offline}$ (s)	SIA T_{inf} (s)	fixed $T_{\mathcal{P}}$ size #edges	fixed $T_{\mathcal{P}}$ $T_{offline}$ (s)	fixed $T_{\mathcal{P}}$ T_{inf} (s)	flexible $T_{\mathcal{P}}$ size #edges	flexible $T_{\mathcal{P}}$ $T_{offline}$ (s)	flexible $T_{\mathcal{P}}$ T_{inf} (s)
C-P	×1000			×1000			×1000		
6 - 3	24	1.3	0.02	84	0.9	0.06	48	0.03	0.06
9 - 3	88	4.9	0.1	345	7.1	0.5	171	0.1	0.6
6 - 4	361	55.2	1.2	741	10.8	0.7	504	0.8	0.6
8 - 4	1,350	220.7	13.6	2,604	60.1	4.1	1,374	3.2	3.2
9 - 4	-	-	-	4,506	130.2	9.8	2,826	5.8	10.1
10 - 4	-	-	-	6,939	281.0	19.2	4,368	10.2	21.6
11 - 4	-	-	-	-	-	-	6,459	17.7	33.5
4 - 5	519	128.6	1.7	657	6.6	0.4	525	1.1	0.2
5 - 5	-	-	-	2,289	30.6	2.0	1,833	4.6	1.5
6 - 5	-	-	-	6,414	111.4	9.6	5,016	15.3	6.8
7 - 5	-	-	-	14,811	376.8	55.04	11,643	41.8	51.8

Table 5.1: Results for the *mastermind* game. We use *size* to denote the representation size (averaged over all time steps), $T_{offline}$ for runtimes of the offline phase and T_{inf} for the runtime to compute the forward message for one time step.

Domain	$T_{offline}$ (s)	0% evidence size #edges	0% evidence T_{inf} (s)	33% evidence size #edges	33% evidence T_{inf} (s)	66% evidence size #edges	66% evidence T_{inf} (s)
people		×1000		×1000		×1000	
3	0.01	0.9	0.01	0.5	0.01	0.4	0.01
4	0.01	39	0.01	14	0.01	3	0.01
5	0.01	45	0.01	15	0.01	4	0.01
6	0.02	330	0.07	27	0.04	13	0.02
7	0.2	2,541	0.6	210	0.5	40	0.15
8	0.8	21,889	40.1	1,281	5.9	84	0.73
9	12.1	-	-	4,170	75.6	423	16.4
10	11.0	-	-	-	-	654	19.1

Table 5.2: Results for the *sickness* network.

For the mastermind game we observe that dynamic $T_{\mathcal{P}}$-compilation offers significant speed-ups and scales to more complex domains compared to SIA. Furthermore, the flexible approach compares favourable to the fixed approach as compile times are lower and sizes of the obtained representations are smaller. Results for the sickness network show that inference (done with our flexible approach) benefits from exploiting evidence. With the fixed approach we would

only get as far as 0% evidence and, although not defined for cyclic programs, SIA would be comparable to 0% evidence for compilation. Hence, we conclude that dynamic $T_{\mathcal{P}}$-compilation allows us to efficiently exploit evidence to further push the boundaries of exact inference in dynamic relational domains.

5.6 Related Work

We can differentiate two streams of research that aim to efficiently cope with dynamic relational models. A first approach is to start from propositional dynamic models and extend them to the relational setting. This has, for example, resulted in logical hidden Markov models (Kersting, De Raedt, et al. 2006) and relational dynamic Bayesian networks (Manfredotti 2009). A second approach is to start from a logical or relational representation and extend the existing inference algorithms to more efficiently deal with time-related dependencies. Inference for dynamic relational domains in these representation relies on approximate techniques (de Salvo Braz et al. 2008; Nitti et al. 2013, 2016), only allow for acyclic dependencies (Sato 1995) or requires that each rule in the program considers a transition over time steps (Thon et al. 2011). Throughout this chapter, we proposed an exact technique that does not come with these requirements.

Online or dynamic inference has also been considered in the context of Markov logic networks, but this only with approximate inference, e.g. Geier and Biundo (2011) and Kersting, Ahmadi, et al. (2009). The advantage of exploiting evidence in the general context of probabilistic inference and knowledge compilation was previously investigated by Chavira, Allen, et al. (2005).

5.7 Conclusions

We have extended $T_{\mathcal{P}}$-compilation towards probabilistic logic programs with time. Program updates are often required to adopt for an ever changing environment and, as such, implicitly introduce a dependency between a sequence of programs. $T_{\mathcal{P}}$-compilation allows to restart from past compilation results and avoids the need to completely recompile the program from scratch in the presence of new information. Time can be explicitly introduced in a probabilistic logic program to compactly model a stochastic process. $T_{\mathcal{P}}$-compilation is unified with the structural interface algorithm to not only exploit the repeated structure in dynamic relational models but also deterministic dependencies introduced by the observations.

Chapter 6

Conclusions

We conclude this dissertation by summarizing the presented work and discuss some interesting directions for future research.

6.1 Thesis Summary

Expressive probabilistic models allow one to compactly represent and combine different types of structural knowledge and uncertainty. Throughout this dissertation, we have considered three, more specific, types of such models; *dynamic Bayesian networks* use a template notation to compactly represent a stochastic process, *probabilistic logic programs* make use of logical variables and relations to compactly represent knowledge about objects in the world and *dynamic relational models* combine a dynamic and logical representation to represent relational stochastic processes. For each of the above models, uncertainty can be precisely quantified by means of probabilities.

Our work mainly focuses on the task of probabilistic inference or reasoning, that is computing the posterior belief for query propositions given our knowledge and observed evidence. A typical example is to compute the probability of a patient having a certain disease, given a set of observable symptoms and the results of medical tests. While the compact representation might suggest otherwise, the underlying model for many real-world applications is typically very complex as it ranges over a large number of specific entities or exhibits cyclic dependencies. Therefore, probabilistic inference still remains a challenge, despite the progress we made with our techniques.

Contributions

We now shortly review the three main contributions of the thesis.

The first contribution is the **Structural Interface Algorithm**, an exact inference algorithm for dynamic Bayesian networks. It unifies state-of-the-art techniques for inference in static and dynamic networks, by combining principles of knowledge compilation with the interface algorithm. The resulting algorithm not only exploits the repeated nature of the model, as it avoids the need to unroll the network, but also the local structure, including determinism and context-specific independence. Empirically, we showed that the structural interface algorithm speeds up inference in the presence of local structure, and scales to larger and more complex networks.
From a more technical perspective, the structural interface algorithm first encodes, then compiles the template model (transition model) into an efficient circuit representation. Then, the forward message is recursively computed for each of the required time steps by means of weighted model counting on the obtained circuit. We explored several approaches to efficiently encode and compute the forward message. Each of these had different memory requirements and trade-offs between putting the burden on the compiler, a post-compilation step or the inference steps.

The second contribution is **$T_\mathcal{P}$-compilation**, an anytime inference algorithm for probabilistic logic programs. $T_\mathcal{P}$-compilation proceeds incrementally in that it interleaves the knowledge compilation step for weighted model counting with forward reasoning on the logic program. It directly acts on the logic program and avoids the need to completely unfold the model. This leads to a novel algorithm that, at any time in the process, provides hard bounds on the inferred probabilities. An empirical evaluation demonstrates that $T_\mathcal{P}$-compilation outperforms existing exact and approximate techniques on several real-world applications with respect to time, space and quality of results.
From a more technical perspective, most existing techniques for inference in probabilistic logic programs require to encode the ground program into an intermediate propositional formula representation. This step includes a conversion from logic programming semantics to first-order logic semantics and requires the introduction of additional propositions to correctly capture cyclic dependencies. For $T_\mathcal{P}$-compilation, on the other hand, the use of forward reasoning on the logic program allows for a natural way to handle cyclic dependencies. In other words, the conversion happens during rather than after reasoning within the logic programming semantics, avoiding the expensive introduction of additional propositions.

The third contribution is **Dynamic $T_{\mathcal{P}}$-compilation**, an exact inference algorithm for relational dynamic models. Dynamic $T_{\mathcal{P}}$-compilation unifies the structural interface algorithm for propositional dynamic models with $T_{\mathcal{P}}$-compilation for static relational models and combines the advantages of both approaches. In addition, it allows to exploit local structure introduced by the given observations to further scale-up inference. An empirical evaluation shows the promise of the technique.
From a more technical perspective, one of the main differences of $T_{\mathcal{P}}$-compilation, compared to other techniques, is that it results in a set of compiled sentences rather than one big formula. This offers a more flexible approach and allows us to extend the structural interface algorithm to incrementally encode and compute the forward message for each of the required time steps. In case of deterministic dependencies in the forward message, we can employ a more compact representation to further speed-up inference.

Each of the three contributions provides an affirmative answer to the general research question raised in the introductory chapter:

> *Can we exploit the characteristics of an expressive probabilistic model to further scale-up probabilistic inference by weighted model counting?*

As a general conclusion, we state that the methods presented in this dissertation push the boundaries for exact as well as approximate inference in expressive probabilistic models.

6.2 Future Work

We now discuss some concrete topics of future work. They are not only interesting from a research point of view but also for practical applications.

Applications

In the introductory chapter of this text, we mainly used medical diagnosis to demonstrate the need and usefulness of expressive probabilistic models. The applicability of the methods and models discussed throughout this text goes far beyond medical diagnosis, however. Models that combine a relational representation and dynamics have been successfully applied in many real-world settings, including video recognition (Manfredotti 2009), robotics (Nitti 2016), multi-player games (Thon 2011), etc. Also some of the most ambitious projects such as the self-driving car, where techniques from video recognition and robotics are combined, belong to the potential applications.

In the next few sections, we will consider some useful extensions to the techniques proposed throughout this dissertation. This should make them more generally applicable for real-world domains and applications.

Dynamic Models and Local Structure

Probabilistic inference in static models, e.g. Bayesian networks, benefits significantly from exploiting local structure in the form of determinism and context-specific independencies. In Chapter 3, we showed how this generalizes towards dynamic models where the same forms of local structure in the transition model can be exploited to obtain exponential speed gains. Dynamic $T_{\mathcal{P}}$-compilation, as introduced in Chapter 5, additionally exploits deterministic dependencies in the forward message allowing for a more compact representation and more efficient computations. An obvious next step is to investigate whether the forward message exhibits local structure in the form of context-specific independencies and whether this structure can be exploited to further scale-up inference.

Approximate Inference for Dynamic Models

Despite a lot of progress in knowledge compilation and other general reasoning techniques, exact inference is often infeasible for many real-world applications. In the context of dynamic models, representing and computing the forward message is exponential in the number of variables that d-separate the past from the future. Despite the occurrence of local structure in the forward message, it is unrealistic to believe that exact inference techniques would scale-up to dynamic models representing social networks with thousands of people or digital circuits with thousands of gates.

In Chapter 4, we proposed $T_{\mathcal{P}}$-compilation as an anytime inference algorithm for static models. For dynamic models, however, we only considered exact techniques and did not propose any approximate method. Interesting directions of future research within this context is to use a factorized representation of the forward message (Boyen and Koller 1998; Murphy and Weiss 2001) or to combine our knowledge compilation techniques with Rao-Blackwellized particle filters for dynamic domains (Doucet, Freitas, et al. 2000; Nitti et al. 2016).

Dynamic Models with Dynamic Structure

The dynamic models we considered throughout this dissertation were all assumed to have a static structure, i.e. the structure is exactly the same for each of the time steps. While this is the common assumption for dynamic Bayesian networks, more expressive representations are generally not limited to a fixed structure. We did extend $T_{\mathcal{P}}$-compilation to efficiently cope with program updates, however, but this only for probabilistic logic programs where time was not explicitly part of the model.

Applications such as video tracking, for example in the context of robotics, require dynamic relational models where the number of objects and relations can change over time. Indeed, the environment captured by the video constantly evolves as new objects can enter the scene while others disappear. To deal with these models, we have to extend our dynamic $T_{\mathcal{P}}$-compilation algorithm in a similar way as was done to deal with program updates. This will involve recompiling the template formula's for each of the time steps.

Continuous Variables

In the general context of probability theory, one distinguishes between two types of random variables; *discrete variables*, where to domain is a finite and countable set, and *continuous variables*, where the domain is given by an interval. The techniques discussed throughout this dissertation are all based on weighted model counting and only deal with discrete variables. Continuous variables offer more expressiveness, however, and are often desired in practical applications.

Over the last couple of years, existing representations and techniques for discrete variables have been generalized towards continuous variables. Probabilistic logic programs were extended with distributional clauses (Gutmann et al. 2011; Nitti 2016), weighted model counting was generalized towards weighted model integration (Belle et al. 2015) and target representations for knowledge compilation were developed to deal with hybrid domains, e.g. extended algebraic decision diagrams (Sanner and Abbasnejad 2012). An exciting direction of future work is to investigate how these methods unify with the techniques proposed in this dissertation as it would allow us to deal with many of the typical models used within the field of robotics.

Learning and Decisions

Probabilistic reasoning is, arguably, the most fundamental task within the area of probabilistic modeling. While its main purpose is to reason over a model and to compute the probability that a query holds in the presence of observations, inference is also a crucial sub-routine in many other algorithms used to solve different tasks. Two of such tasks are *parameter learning* and *decision making*.

The goal of parameter learning is to find a good estimate for the parameters in a model, given the structure of the model and a (partial) data set. It is common to rely on the well-known Expectation Maximization (EM) algorithm to achieve this task (Fierens et al. 2015). Within the E-step of this algorithm, inference is required to compute the probabilities of the parameters.

Decision making concerns the task of finding the best decision that one can make, given a model and a cost/reward function. Inference is required to compute the expected utility (reward) after making certain decisions. Decision making in the context of relational representation has previously been investigated, and offers a suitable formalism to deal with applications such as viral marketing (Van den Broeck, Thon, et al. 2010).

The task of parameter learning is generally required for many real-world problems in case the level of non-determinism is not known beforehand but has to be learned from the available data. Support for decision making would allow us consider application domains such as machine monitoring, where shutting down machines and replacing components comes with a certain cost. Or also for general game playing, where rewards can be obtained by performing a successful action or by winning the game. Therefore, we certainly have to investigate whether the techniques presented in this dissertation could provide new insights to further scale-up the methods used for parameter learning and decision making.

Bibliography

Aziz, R. A., G. Chu, C. Muise, and P. J. Stuckey (2015). "Stable Model Counting and Its Application in Probabilistic Logic Programming". In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on page 96).

Bancilhon, F., D. Maier, Y. Sagiv, and Jeffrey D. Ullman (1986). "Magic sets and other strange ways to implement logic programs (extended abstract)". In: *Proceedings of the 5th ACM SIGACT-SIGMOD symposium on Principles of database systems (PODS)* (Cited on page 97).

Belle, V., A. Passerini, and G. Van den Broeck (2015). "Probabilistic inference in hybrid domains by weighted model integration". In: *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)* (Cited on page 117).

Bogaerts, B. and G. Van den Broeck (2015). "Knowledge compilation of logic programs using approximation fixpoint theory". In: *Theory and Practice of Logic Programming* 15.4-5, pp. 464–480 (Cited on page 96).

Boutilier, C., N. Friedman, M. Goldszmidt, and D. Koller (1996). "Context-specific Independence in Bayesian Networks". In: *Proceedings of the 12th International Conference on Uncertainty in Artificial Intelligence (UAI)* (Cited on pages 30, 64).

Bova, S. (2016). "SDDs Are Exponentially More Succinct than OBDDs". In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on page 21).

Boyen, X. and D. Koller (1998). "Tractable Inference for Complex Stochastic Processes". In: *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI)* (Cited on pages 39, 65, 116).

Brace, K. S., R. L. Rudell, and R. E. Bryant (1990). "Efficient Implementation of a BDD Package." In: *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC)* (Cited on page 64).

Bryant, R. E. (1992). "Symbolic Boolean manipulation with ordered binary-decision diagrams". In: *ACM Computing Surveys (CSUR)* 24.3, pp. 293–318 (Cited on page 20).

Chavira, M., D. Allen, and A. Darwiche (2005). "Exploiting Evidence in Probabilistic Inference." In: *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI)* (Cited on page 111).

Chavira, M. and A. Darwiche (2005). "Compiling Bayesian Networks with Local Structure". In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)* (Cited on pages 30, 40, 64).

Chavira, M. and A. Darwiche (2007). "Compiling Bayesian Networks Using Variable Elimination." In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)* (Cited on page 64).

Chavira, M. and A. Darwiche (2008). "On probabilistic inference by weighted model counting". In: *Artificial Intelligence* 172.6-7, pp. 772–799 (Cited on page 28).

Chavira, M., A. Darwiche, and M. Jaeger (2006). "Compiling relational Bayesian networks for exact inference". In: *International Journal of Approximate Reasoning* 42.1, pp. 4–20 (Cited on pages 58, 96, 109).

Choi, A., D. Kisa, and A. Darwiche (2013). "Compiling Probabilistic Graphical Models using Sentential Decision Diagrams". In: *Proceedings of the 12th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU)* (Cited on pages 24, 79, 96).

Clark, K. L. (1978). "Negation as failure". In: *Logic and databases*, pp. 293–322 (Cited on page 74).

Darwiche, A. (2001a). "Constant-space reasoning in dynamic Bayesian networks". In: *International Journal of Approximate Reasoning* 26.3, pp. 161–178 (Cited on pages 46, 47, 65).

Darwiche, A. (2001b). "On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision." In: *Journal of Applied Non-Classical Logics* 11.1-2, pp. 11–34 (Cited on page 64).

Darwiche, A. (2001c). "Recursive conditioning". In: *Artificial Intelligence* 126.1-2, pp. 5–41 (Cited on page 64).

Darwiche, A. (2002). "A Logical Approach to Factoring Belief Networks." In: *Proceedings of the Eighth International Conference on Principles of knowledge representation and reasoning (KR)* (Cited on page 64).

Darwiche, A. (2004). "New Advances in Compiling CNF into Decomposable Negation Normal Form." In: *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)* (Cited on pages 20, 64, 91).

Darwiche, A. (2009). *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press (Cited on pages 27, 32, 39, 45, 58).

Darwiche, A. (2011). "SDD: A New Canonical Representation of Propositional Knowledge Bases". In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)* (Cited on pages 19, 20, 64, 68, 88).

Darwiche, A. and P. Marquis (2002). "A knowledge compilation map". In: *Journal of Artificial Intelligence Research* 17, pp. 229–264 (Cited on page 19).

Davis, J. and P. Domingos (2009). "Deep Transfer via Second-Order Markov Logic". In: *Proceedings of the 26th International Conference on Machine Learning (ICML)* (Cited on page 93).

De Raedt, L., P. Frasconi, K. Kersting, and S. Muggleton, eds. (2008). *Probabilistic Inductive Logic Programming — Theory and Applications*. Vol. 4911. Lecture Notes in Artificial Intelligence. Springer (Cited on pages 67, 95).

De Raedt, L., K. Kersting, S. Natarajan, and D. Poole (2016). *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Morgan & Claypool Publishers (Cited on page 3).

De Raedt, L. and A. Kimmig (2015). "Probabilistic (logic) programming concepts". In: *Machine Learning* 100.1, pp. 5–47 (Cited on page 35).

De Raedt, L., A. Kimmig, and H. Toivonen (2007). "ProbLog: A Probabilistic Prolog and Its Application in Link Discovery." In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)* (Cited on pages 35, 67, 91, 95, 96).

de Salvo Braz, R., N. Arora, E. Sudderth, and S. Russell (2008). "Open-universe state estimation with dblog". In: *NIPS 2008 Workshop* (Cited on page 111).

Dean, T. and K. Kanazawa (1989). "A model for reasoning about persistence and causation". In: *Computational Intelligence.* 5.3, pp. 142–150 (Cited on pages 39, 41).

Dechter, R. (1996). "Bucket Elimination: A Unifying Framework for Probabilistic Inference". In: *Proceedings of the 12th international conference on Uncertainty in artificial intelligence (UAI)* (Cited on page 64).

Doucet, A. and N. de Freitas (2001). *Sequential Monte Carlo methods in practice.* Springer (Cited on page 65).

Doucet, A., N. de Freitas, K. P. Murphy, and S. J. Russell (2000). "Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks". In: *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI)* (Cited on pages 65, 116).

Dries, A., A. Kimmig, W. Meert, J. Renkens, G. Van den Broeck, J. Vlasselaer, and L. De Raedt (2015). "ProbLog2: Probabilistic logic programming". In: *Lecture Notes in Computer Science, ECML PKDD*, pp. 312–315 (Cited on page 132).

Fierens, D., G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt (2015). "Inference and learning in probabilistic logic programs using weighted Boolean formulas". In: *Theory and Practice of Logic Programming* 15.03, pp. 358–401 (Cited on pages 28, 36, 67, 73, 91, 96, 118).

Forbes, J., T. Huang, K. Kanazawa, and S. Russell (1995). "The BATmobile: Towards a Bayesian Automated Taxi". In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)* (Cited on page 39).

Geier, T. and S. Biundo (2011). "Approximate Online Inference for Dynamic Markov Logic Networks". In: *Proceedings of the 23rd International Conference on Tools with Artificial Intelligence (ICTAI)* (Cited on page 111).

Getoor, L. and B. Taskar, eds. (2007). *An Introduction to Statistical Relational Learning*. MIT Press (Cited on pages 67, 95).

Goodman, N. D., V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum (2008). "Church: a language for generative models". In: *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence (UAI)* (Cited on page 96).

Gutmann, B., I. Thon, A. Kimmig, M. Bruynooghe, and L. De Raedt (2011). "The magic of logical inference in probabilistic programming". In: *Theory and Practice of Logic Programming* 11, pp. 663–680 (Cited on pages 96, 97, 117).

Huang, T., D. Koller, J. Malik, G. Ogasawara, B. Rao, S. Russell, and J. Weber (1994). "Automatic Symbolic Traffic Scene Analysis Using Belief Networks". In: *Proceedings of the 12th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on page 39).

Jaeger, M. (1997). "Relational Bayesian Networks". In: *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence (UAI)* (Cited on page 96).

Janhunen, T. (2004). "Representing normal programs with clauses". In: *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)* (Cited on page 75).

Jensen, F., S. Lauritzen, and K. Olsen (1990). "Bayesian updating in recursive graphical models by local computation". In: *Computational Statistics Quarterly* 4, pp. 269–282 (Cited on page 64).

Jensen, F. and S. K. Anderson (1990). "Approximations in Bayesian Belief Universe for Knowledge Based Systems". In: *Proceedings of the 6th Conference on Uncertainty in Artificial Intelligence (UAI)* (Cited on page 64).

Kalman, R. E. (1960). "A New Approach to Linear Filtering And Prediction Problems". In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D, pp. 35–45 (Cited on page 39).

Kersting, K., B. Ahmadi, and S. Natarajan (2009). "Counting Belief Propagation". In: *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI)* (Cited on page 111).

Kersting, K., L. De Raedt, and T. Raiko (2006). "Logical Hidden Markov Models." In: *Journal of Artificial Intelligence Research* 25, pp. 425–456 (Cited on page 111).

Kjaerulff, U. (1995). "dHugin: A computational system for dynamic time-sliced Bayesian networks". In: *International Journal of Forecasting* 11, pp. 89–111 (Cited on pages 39, 65).

Koller, D. and N. Friedman (2009). *Probabilistic graphical models: principles and techniques*. MIT press (Cited on pages 25, 27, 39, 45).

Kolmogorov, A. N. (1933). *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer, Berlin (Cited on page 3).

Langone, R., C. Alzate Perez, B. De Ketelaere, J. Vlasselaer, W. Meert, and J. Suykens (2015). "LS-SVM based spectral clustering and regression for predicting maintenance of industrial machines". In: *Engineering Applications of Artificial Intelligence* 37, pp. 268–278 (Cited on page 131).

Larkin, D. and R. Dechter (2003). "Bayesian Inference in the Presence of Determinism." In: *Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics (AISTATS)* (Cited on page 64).

Lauritzen, S. and D. J. Spiegelhalter (1988). "Local computations with probabilities on graphical structures and their application to expert systems". In: *Journal of the Royal Statistical Society series B* 50, pp. 157–224 (Cited on page 64).

Lloyd, J. W. (1989). *Foundations of Logic Programming*. 2nd. Springer (Cited on page 33).

Manfredotti, C. (2009). "Modeling and inference with relational dynamic Bayesian networks". In: *Proceedings of the 22nd Canadian Conference on Artificial Intelligence* (Cited on pages 4, 111, 115).

Mantadelis, T. and G. Janssens (2010). "Dedicated tabling for a probabilistic setting". In: *Technical Communications of the 26th International Conference on Logic Programming (ICLP)* (Cited on page 73).

Meert, W., J. Vlasselaer, and G. Van den Broeck (2016). "A relaxed Tseitin transformation for weighted model counting". In: *Proceedings of the 6th International Workshop on Statistical Relational AI (StarAI)* (Cited on page 132).

Milch, B., B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov (2005). "BLOG: Probabilistic Models with Unknown Objects". In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)* (Cited on page 96).

Muggleton, S. (1996). "Stochastic logic programs". In: *Advances in Inductive Logic Programming*. IOS Press, pp. 254–264 (Cited on page 96).

Muise, C. J., S. A. McIlraith, J. C. Beck, and E. I. Hsu (2010). "Fast d-DNNF Compilation with sharpSAT." In: *Workshop on Abstraction, Reformulation, and Approximation (AAAI Workshop)* (Cited on page 64).

Murphy, K. (2002). "Dynamic Bayesian Networks: Representation, Inference and Learning". PhD thesis. UC Berkeley, Computer Science Division (Cited on pages 39–41, 44, 45, 47–49, 55, 65).

Murphy, K. and Y. Weiss (2001). "The Factored Frontier Algorithm for Approximate Inference in DBNs". In: *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI)* (Cited on pages 65, 116).

Nilsson, U. and J. Maluszynski (1995). *Logic, Programming, and PROLOG*. 2nd. John Wiley & Sons, Inc. (Cited on page 69).

Nitti, D. (2016). "Hybrid Probabilistic Logic Programming". PhD thesis. KU Leuven, Department of Computer Science (Cited on pages 115, 117).

Nitti, D., T. De Laet, and L. De Raedt (2013). "A particle filter for hybrid relational domains". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Cited on pages 102, 111).

Nitti, D., T. De Laet, and L. De Raedt (2014). "Relational object tracking and learning". In: *IEEE International Conference on Robotics and Automation (ICRA)* (Cited on pages 4, 96).

Nitti, D., T. De Laet, and L. De Raedt (2016). "Probabilistic logic programming for hybrid relational domains". In: *Machine Learning*. Accepted (Cited on pages 111, 116).

Ourfali, O., T. Shlomi, T. Ideker, E. Ruppin, and R. Sharan (2007). "SPINE: a framework for signaling-regulatory pathway inference from cause-effect experiments". In: *Bioinformatics* 23.13, pp. 359–366 (Cited on page 93).

Oztok, U. and A. Darwiche (2015). "A Top-Down Compiler for Sentential Decision Diagrams". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)* (Cited on page 64).

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann (Cited on pages 25, 26, 39).

Pfeffer, A. (2014). *Practical Probabilistic Programming*. Manning Publications (Cited on page 95).

Poole, D. (1993). "Logic programming, abduction and probability". In: *New Generation Computing* 11, pp. 377–400 (Cited on pages 35, 78, 95, 96).

Poole, D. L. and N. L. Zhang (2011). "Exploiting Contextual Independence In Probabilistic Inference". In: *Journal of Artificial Intelligence Research* 18, pp. 263–313 (Cited on page 64).

Poole, D. and A. K. Mackworth (2010). *Artificial Intelligence - Foundations of Computational Agents*. Cambridge University Press (Cited on pages 2, 12).

Poon, H. and P. Domingos (2006). "Sound and Efficient Inference with Probabilistic and Deterministic Dependencies". In: *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)* (Cited on pages 77, 96).

Rabiner, L. R. (1989). "A tutorial on hidden Markov models and selected applications in speech recognition". In: *Proceedings of the IEEE*. Vol. 77, pp. 257–286 (Cited on pages 39, 46, 65).

Renkens, J., A. Kimmig, G. Van den Broeck, and L. De Raedt (2014). "Explanation-based approximate weighted model counting for probabilistic logics". In: *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on pages 78, 93, 94, 96).

Renkens, J., D. Shterionov, G. Van den Broeck, J. Vlasselaer, D. Fierens, W. Meert, G. Janssens, and L. De Raedt (2012). "ProbLog2: From probabilistic programming to statistical relational learning". In: *Proceedings of the NIPS Probabilistic Programming Workshop* (Cited on page 132).

Renkens, J., G. Van den Broeck, and S. Nijssen (2012). "k-optimal: A novel approximate inference algorithm for ProbLog". In: *Machine Learning* 89.3, pp. 215–231 (Cited on pages 93, 96).

Riguzzi, F. (2007). "A Top Down Interpreter for LPAD and CP-Logic". In: *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence (AI*IA)* (Cited on page 96).

Riguzzi, F. (2016). "The Distribution Semantics for Normal Programs with Function Symbols". In: *International Journal of Approximate Reasoning (Special Issue on Probabilistic Logic Programming)* 77, pp. 1–19 (Cited on page 96).

Riguzzi, F. and T. Swift (2011). "The PITA System: Tabling and Answer Subsumption for Reasoning under Uncertainty". In: *Theory and Practice of Logic Programming* 11.4–5, pp. 433–449 (Cited on page 96).

Rocha, R., F. Silva, and V. S. Costa (2000). "A Tabling Engine for the Yap Prolog System". In: *Proceedings of the APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP)* (Cited on page 70).

Russell, S. and P. Norvig (2009). *Artificial Intelligence: A Modern Approach*. 3rd. Prentice Hall Press (Cited on pages 2, 46).

Sandri, M., P. Berchialla, I. Baldi, D. Gregori, and R. A. D. Blasi (2014). "Dynamic Bayesian Networks to predict sequences of organ failures in patients admitted to ICU". In: *Journal of Biomedical Informatics* 48, pp. 106–113 (Cited on pages 6, 39).

Sang, T., F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi (2004). "Combining Component Caching and Clause Learning for Effective Model Counting." In: *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)* (Cited on page 64).

Sang, T., P. Beame, and H. A. Kautz (2005a). "Heuristics for Fast Exact Model Counting". In: *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT)* (Cited on page 64).

Sang, T., P. Beame, and H. A. Kautz (2005b). "Performing Bayesian Inference by Weighted Model Counting." In: *Proceedings of the 20th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on pages 28, 64).

Sanner, S. and E. Abbasnejad (2012). "Symbolic Variable Elimination for Discrete and Continuous Graphical Models". In: *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on page 117).

Sato, T. and Y. Kameya (2001). "Parameter Learning of Logic Programs for Symbolic-Statistical Modeling". In: *Journal of Artificial Intelligence Research* 15, pp. 391–454 (Cited on page 35).

Sato, T. (1995). "A statistical learning method for logic programs with distribution semantics". In: *Proceedings of the 12th International Conference on Logic Programming (ICLP)* (Cited on pages 35, 95, 111).

Shterionov, D., J. Renkens, J. Vlasselaer, A. Kimmig, W. Meert, and G. Janssens (2015). "The most probable explanation for probabilistic logic programs with annotated disjunctions". In: *Proceedings of the 25th International Conference on Inductive Logic Programming (ILP)* (Cited on page 131).

Suciu, D., D. Olteanu, R. Christopher, and C. Koch (2011). *Probabilistic Databases*. 1st. Morgan & Claypool Publishers (Cited on page 95).

Theocharous, G., K. Murphy, and L. P. Kaelbling (2004). "Representing hierarchical POMDPs as DBNs for multi-scale robot localization". In: *Proceedings of the International Conference on Robotics and Automation (ICRA)* (Cited on page 39).

Thon, I. (2011). "Stochastic Relational Processes and Models: Learning and Reasoning". PhD thesis. KU Leuven, Department of Computer Science (Cited on page 115).

Thon, I., L. Niels, and L. De Raedt (2011). "Stochastic relational processes: Efficient inference and applications". In: *Machine Learning* 82.2, pp. 239–272 (Cited on pages 4, 111).

Van den Broeck, G. (2013). "Lifted Inference and Learning in Statistical Relational Models". PhD thesis. KU Leuven, Department of Computer Science (Cited on page 20).

Van den Broeck, G. and A. Darwiche (2015). "On the Role of Canonicity in Knowledge Compilation". In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on pages 19, 23).

Van den Broeck, G., I. Thon, M. van Otterlo, and L. De Raedt (2010). "DTProbLog: A decision-theoretic probabilistic Prolog". In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on page 118).

Van Emden, M. H. and R. A. Kowalski (1976). "The Semantics of Predicate Logic as a Programming Language". In: *Journal of the ACM* 23, pp. 569–574 (Cited on page 71).

Van Gelder, A., K. A. Ross, and J. S. Schlipf (1991). "The Well-founded Semantics for General Logic Programs". In: *Journal of the ACM* 38.3, pp. 619–649 (Cited on page 34).

Vennekens, J., M. Denecker, and M. Bruynooghe (2009). "CP-logic: A language of causal probabilistic events and its relation to logic programming." In: *Theory and Practice of Logic Programming* 9.3, pp. 245–308 (Cited on page 95).

Vennekens, J., S. Verbaeten, and M. Bruynooghe (2004). "Logic programs with annotated disjunctions". In: *Proceedings of the 20th International Conference on Logic Programming (ICLP)* (Cited on pages 35, 95).

Vlasselaer, J., A. Kimmig, A. Dries, W. Meert, and L. De Raedt (2016). "Knowledge compilation and weighted model counting for inference in probabilistic logic programs". In: *Proceedings of the 1st Workshop of Beyond NP* (Cited on page 132).

Vlasselaer, J. and W. Meert (2012). "Statistical relational learning for prognostics". In: *Proceedings of the 21st Belgian-Dutch Conference on Machine Learning (BENELEARN)* (Cited on page 132).

Vlasselaer, J., W. Meert, R. Langone, and L. De Raedt (2014). "Condition monitoring with incomplete observations". In: *Proceedings of the Conference on Prestigious Applications of Intelligent Systems (PAIS)* (Cited on page 132).

Vlasselaer, J., W. Meert, G. Van den Broeck, and L. De Raedt (2014). "Efficient probabilistic inference for dynamic relational models". In: *Proceedings of the 4th International Workshop on Statistical Relational AI (StarAI)* (Cited on page 132).

Vlasselaer, J., W. Meert, G. Van den Broeck, and L. De Raedt (2016a). "Exploiting local and repeated structure in dynamic Bayesian networks". In: *Artificial Intelligence* 232, pp. 43–53 (Cited on pages 10, 40, 131).

Vlasselaer, J., J. Renkens, G. Van den Broeck, and L. De Raedt (2014). "Compiling probabilistic logic programs into sentential decision diagrams". In: *Proceedings of the Workshop on Probabilistic Logic Programming (PLP)* (Cited on pages 10, 68, 132).

Vlasselaer, J., G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2015). "Anytime inference in probabilistic logic programs with Tp-compilation". In: *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)* (Cited on pages 10, 68, 131).

Vlasselaer, J., G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2016b). "Tp-Compilation for inference in probabilistic logic programs". In: *International Journal of Approximate Reasoning* 78, pp. 15–32 (Cited on pages 10, 100, 131).

Xue, Y., A. Choi, and A. Darwiche (2012). "Basing Decisions on Sentences in Decision Diagrams". In: *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on page 21).

Zhang, N. L. and D. Poole (1996). "Exploiting Causal Independence in Bayesian Network Inference". In: *Journal of Artificial Intelligence Research* 5, pp. 301–328 (Cited on page 64).

Zweig, G. (1996). *A Forward-backward Algorithm for Inference in Bayesian Networks and an Empirical Comparison with HMMs: Research Project* (Cited on page 65).

Zweig, G. and S. Russell (1998). "Speech Recognition with Dynamic Bayesian Networks." In: *Proceedings of the 15th AAAI Conference on Artificial Intelligence (AAAI)* (Cited on page 39).

List of Publications

Journal Articles

J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2016b). "Tp-Compilation for inference in probabilistic logic programs". In: *International Journal of Approximate Reasoning* 78, pp. 15–32

J. Vlasselaer, W. Meert, G. Van den Broeck, and L. De Raedt (2016a). "Exploiting local and repeated structure in dynamic Bayesian networks". In: *Artificial Intelligence* 232, pp. 43–53

R. Langone, C. Alzate Perez, B. De Ketelaere, J. Vlasselaer, W. Meert, and J. Suykens (2015). "LS-SVM based spectral clustering and regression for predicting maintenance of industrial machines". In: *Engineering Applications of Artificial Intelligence* 37, pp. 268–278

Conference Papers

J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2015). "Anytime inference in probabilistic logic programs with Tp-compilation". In: *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*

D. Shterionov, J. Renkens, J. Vlasselaer, A. Kimmig, W. Meert, and G. Janssens (2015). "The most probable explanation for probabilistic logic programs with annotated disjunctions". In: *Proceedings of the 25th International Conference on Inductive Logic Programming (ILP)*

J. Vlasselaer, W. Meert, R. Langone, and L. De Raedt (2014). "Condition monitoring with incomplete observations". In: *Proceedings of the Conference on Prestigious Applications of Intelligent Systems (PAIS)*

J. Vlasselaer and W. Meert (2012). "Statistical relational learning for prognostics". In: *Proceedings of the 21st Belgian-Dutch Conference on Machine Learning (BENELEARN)*

Workshop Papers

W. Meert, J. Vlasselaer, and G. Van den Broeck (2016). "A relaxed Tseitin transformation for weighted model counting". In: *Proceedings of the 6th International Workshop on Statistical Relational AI (StarAI)*

J. Vlasselaer, A. Kimmig, A. Dries, W. Meert, and L. De Raedt (2016). "Knowledge compilation and weighted model counting for inference in probabilistic logic programs". In: *Proceedings of the 1st Workshop of Beyond NP*

A. Dries, A. Kimmig, W. Meert, J. Renkens, G. Van den Broeck, J. Vlasselaer, and L. De Raedt (2015). "ProbLog2: Probabilistic logic programming". In: *Lecture Notes in Computer Science, ECML PKDD*, pp. 312–315

J. Vlasselaer, J. Renkens, G. Van den Broeck, and L. De Raedt (2014). "Compiling probabilistic logic programs into sentential decision diagrams". In: *Proceedings of the Workshop on Probabilistic Logic Programming (PLP)*

J. Vlasselaer, W. Meert, G. Van den Broeck, and L. De Raedt (2014). "Efficient probabilistic inference for dynamic relational models". In: *Proceedings of the 4th International Workshop on Statistical Relational AI (StarAI)*

J. Renkens, D. Shterionov, G. Van den Broeck, J. Vlasselaer, D. Fierens, W. Meert, G. Janssens, and L. De Raedt (2012). "ProbLog2: From probabilistic programming to statistical relational learning". In: *Proceedings of the NIPS Probabilistic Programming Workshop*

Curriculum Vitae

Jonas Vlasselaer was born in Leuven, Belgium on December 24th 1986. In 2009, he obtained his Bachelor of Industrial Sciences degree and in 2010 his Master of Industrial Sciences degree (specialized in Electronic Engineering). Both degrees were obtained from the Leuven Engineering College (GroupT) with magna cum laude. His master thesis, entitled "Design and implementation of an energy management system for residential living", was awarded the Baudouin Elleboudt Award (2011) and was chosen as *most innovative project* on the "Day of an Engineer" (an initiative of USG Professionals). He graduated in 2011 as a Master of Science in Artificial Intelligence, major subject Engineering and Computer Science, at the KU Leuven with magna cum laude.

In 2011, he started as a doctoral student at the DTAI lab (Declarative Languages and Artificial Intelligence) at the KU Leuven under the supervision of Professor Luc De Raedt. He received a personal doctoral scholarship in 2012 from IWT (agency for Innovation by Science and Technology) for his proposal, entitled "Temporal Statistical Relational Learning for Machine Monitoring". From February till April 2016, he was a visiting student of Guy Van den Broeck and the Automated Reasoning Group at the University of California, Los Angeles. In December 2016 he will defend his dissertation, entitled "Probabilistic Inference for Dynamic and Relational Models".

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DECLARATIVE LANGUAGES AND ARTIFICIAL INTELLIGENCE
Celestijnenlaan 200A box 2402
B-3001 Leuven
jonas.vlasselaer@kuleuven.be
https://dtai.cs.kuleuven.be