Universiteit Leiden
The Netherlands

Optimal decision-making under constraints and uncertainty

Latour, A.L.D.

Citation

Latour, A. L. D. (2022, September 13). *Optimal decision-making under constraints and uncertainty*. *SIKS Dissertation Series*. Retrieved from https://hdl.handle.net/1887/3455662

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/3455662

Note: To cite this publication please use the final published version (if applicable).

Part I

Background

2

Logic, probability and inference

2.1 Introduction

As described in Chapter 1, the focus of this work is on developing exact solving methods for *stochastic constraint (optimisation) problems (SCPs)*, that strike a reasonable balance between convenience, generality and speed. Because probability is such an important part of SCPs, we devote this chapter to background on modelling probability distributions and on executing well-known inference tasks. In this work we take a propositional logic-based approach to modelling probability distributions and reasoning about uncertainty. This decision is motivated by the fact that logic-based models of probability distributions generalise many others. Additionally, propositional logic connects very naturally to the *constraint* solving component of the subject matter of this dissertation. Consequently, our focus in this chapter is on probability distribution representations that are based in propositional logic.

We therefore open this chapter in Section 2.2 with some background on propo-

sitional logic, typical problems that can be formulated on propositional formulae, and their associated complexity classes. Since this work aims to solve constraint (optimisation) problems that involve a *stochastic* component in particular, and since the field of probabilistic inference has produced a rich literature on how to solve such problems, in Section 2.3 we give a brief overview of the main probabilistic inference tasks known from this literature.

Moving on to concrete methods for solving probabilistic inference problems, we then discuss the technique of *knowledge compilation* in Section 2.4, and how to use the compilation of propositional formulae to *decision diagrams (DDs)* for probabilistic inference in Section 2.5.

While propositional formulae provide a general way of representing probability distributions, *constructing* these formulae is not always straightforward. We therefore also describe another way of representing probability distributions: the probabilistic logic program, in Section 2.6. This provides a convenient and human-friendly language for programming probability distributions, which can then be converted to DDs or propositional formulae. We conclude this chapter in Section 2.7.

2.2 Propositional logic

In this work, we take a probabilistic logic-based approach to reasoning about uncertainty. In particular, we focus on (literal-weighted) propositional formulae to represent probability distributions. In this section we first give a brief recap of propositional formulae. We then discuss how they can represent probability distributions and how we can compute probabilities from propositional formulae. Finally, we reflect on some relevant complexity classes, using problems formulated on propositional formulae as representative members of those classes. This discussion serves to put the next section (about different kinds of probabilistic queries) into a computational complexity context.

2.2.1 Propositional formulae

Recall the stochastic constraint Equation 1.1 that we defined in Section 1.2. In Section 2.2.3, we show how we can use the formalism of *weighted model counting (WMC)* to compute the probability that an event ϕ occurs. In order to do so, we model ϕ as a *propositional formula*. We now give a brief overview of the notation and terminology that we use in this work. For a detailed description of propositional logic, we refer the reader to the literature, *e.g.*, *Mathematical Logic*

for Computer Science [14].

A propositional formula $\phi(\mathbf{X})$ is defined on Boolean variables $X \in \mathbf{X}$ that are connected in the formula through binary logical connectives $\vee$ ("or") and $\wedge$ ("and"). Variables in the formula may be negated ($\neg$). We define a *literal* as either a variable X or its negation, $\neg X$. In this work, we use *true*, $\top$ and 1 to indicate 'true', and *false*, $\bot$ and 0 to indicate 'false'.

We call $sc(\phi) = \mathbf{X}$ the *scope* of ϕ. In this work, we assume the variables in $sc(\phi)$ to be partitioned into a set of Boolean decision variables $\mathbf{D}$ and a set of Boolean stochastic variables $\mathbf{T}$. We use $D \in \mathbf{D}$ to indicate a Boolean decision variable, and $T \in \mathbf{T}$ to indicate a Boolean stochastic variable. When the type of variable is irrelevant, we use $X \in \mathbf{X}$ to indicate a generic Boolean variable and a generic set of Boolean variables.

When literals are connected through only $\vee$s, we call this a *disjunction* or a (disjunctive) *clause*. When they are connected through only $\wedge$s, we call this a *conjunction*. If a propositional formula consists of a conjunction of clauses, this formula is in *conjunctive normal form (CNF)*. If it consists of a disjunction of conjunctions of literals, the formula is in *disjunctive normal form (DNF)*. For the scope of this thesis we do not assume the input propositional formulae to be in any particular normal form, unless otherwise specified.

We sometimes assign *truth values* to (some of) the variables in a propositional formula ϕ and then consider the *residual formula* that we obtain by replacing those variables by their truth values in ϕ and simplifying the result. We denote such a *partial assignment* by $\pi : \mathbf{X} \mapsto \{\top, \bot\}$ and denote the residual formula as $\phi|_\pi$. In this work in particular, we often want to evaluate the residual propositional formula after we have assigned truth values to the *decision variables* in particular. Since such a partial assignment corresponds to us making a set of decisions, we refer to a partial assignment that only assigns truth values to the decision variables, $\sigma : \mathbf{D} \mapsto \{\top, \bot\}$, as a *strategy*. If σ assigns truth values to only a subset of the decision variables in $sc(\phi)$, we call it a *partial strategy*. When a partial assignment only assigns truth values to stochastic variables, we call $v : \mathbf{T} \mapsto \{\top, \bot\}$ a *scenario* or *possible world*. Any π that assigns a truth value to all variables in $sc(\phi)$ is called an *interpretation* of ϕ. We will sometimes abuse notation and treat a partial assignment or interpretation simply as the set of literals that it sets to $\top$.

We say that a formula evaluates to *true* under σ, in which case we call ϕ *satisfiable*, if $\phi|_\sigma = \top$. We call any interpretation of ϕ that makes ϕ evaluate to *true* a *model*, *witness* or *solution* of ϕ.

The task of deciding whether a propositional formula in CNF has at least one model, is known as the *Boolean satisfiability problem (SAT)* and is known to be

$\mathcal{NP}$-complete [38, 106] (see Section 2.2.4). The task of *counting* the total number of models of a propositional formula, is known as the *propositional model counting problem (#SAT)*, and is the canonical #$\mathcal{P}$-complete problem (see Section 2.2.4). In Section 2.2.3, we discuss how a weighted version of this counting problem relates to computing probabilities, but we first briefly recall some basic concepts of *first-order logic*.

2.2.2 First-order logic

While in this work we limit ourselves to propositional formulae as a basis for reasoning about uncertainty, for the ease of discussion in later sections, we now briefly recall basic concepts and notation of *first-order* formulae. For a formal and more detailed description, we refer the reader to the literature, *e.g.*, *Mathematical Logic for Computer Science* [14].

Like propositional formulae, complex first-order formulae are constructed from simpler formulae using logical connectives such as $\vee$ and $\wedge$, a negation operator $\neg$ and the constants $\top$ and $\bot$. The most simple form of logical formula is the literal, which in the context of first-order logic is more generally defined than in propositional logic. In first-order logic, a literal is an *atomic formula* (*atom*) or its negation. An atom is a *predicate* that operates on *terms*, of which there are different types.

Informally, we can think of terms as objects or entities. The simplest ones are constants, *e.g.*, a and b. They can represent, for example, people. Terms can also be variables. We can interpret variables and constants by giving them values from a domain. The third type of terms are functions. Function symbols are applied on terms to define new terms.

As stated above, the literals of first-order logic are predicates and their negations. These predicates express properties of objects (terms) or relations between objects. A predicate can take any number of arguments, including zero, which is called a nullary predicate. Regardless of the input, a predicate always returns a truth value ($\top$ or $\bot$). Consider for example the domain of friends $\{a, b\}$ and the binary predicate $isFriendOf(X, Y)$. If we interpret X and Y by assigning them the values a and b, respectively, the predicate $isFriendOf(a, b)$ evaluating to *true* may mean that a is a friend of b. Predicates are the reason that first-order logic is sometimes referred to as *predicate logic* [84].

Note that, while expressions involving only constants, variables, and functions are terms, no expression involving a predicate is a term. Rather, an expression involving a predicate is a formula.

In addition, first-order logic uses *quantifiers* to indicate for how many values

the relationship expressed by predicates must hold. The *existential quantifier* $\exists$ ("exists") and the *universal quantifier* $\forall$ ("for all") can be used to express that a certain relationship holds for at least one arbitrary value, or for all values, respectively. For example: $\psi = \forall x \exists y. pred(x, y)$ means that ψ evaluates to *true* iff, for any value x, we can find at least one value y such that the predicate $pred(x, y)$ evaluates to *true*.

Note that first-order logic can express subtle details about the relationships between objects and can be a more compact way to encode information than propositional logic, especially if there is a lot of structure or repetition in the problem or information that we want to model [84]. If the same predicate holds for many (combinations of) values, this can often be more compactly expressed in first-order logic than in propositional logic.

2.2.3 Propositional weighted model counting

In this work, we will use propositional formulae to represent probabilistic models. We then use the formalism of *weighted model counting (WMC)* to compute probabilities from these formulae.

There are many alternative formalisms for representing probability distributions, which we will reflect on in the next chapter. In this work, however, we use a propositional WMC approach, since it generalises many other well-known approaches and is common practice in the domains of probabilistic reasoning, planning and learning [13, 32, 33, 45, 52, 57, 61, 64, 158].

Before we give a formal definition of WMC, we first need to define the notion of *literal-weighted propositional formulae*:

Definition 2.2.1 (Literal-weighted propositional formula)**.** *A* literal-weighted propositional formula *is a tuple* $\langle \phi(\mathbf{X}), W(\mathbf{L}) \rangle$*, of a propositional formula* ϕ *and a* weight function $W : \mathbf{L} \mapsto \mathbb{Q}_0^+$ *that maps each literal in* $\mathbf{L}$ *to a non-negative rational number. Here,* $\mathbf{X}$ *the set of variables of* ϕ*, i.e.,* ϕ*'s scope, and* $\mathbf{L}$ *the corresponding set of literals.*

Specifically, we must define a particular type of literal-weighted formula: one in which all weights can be interpreted as probabilities:

Definition 2.2.2 (Probability-weighted propositional formula)**.** *A* probability-weighted propositional formula *is a tuple* $\langle \phi(\mathbf{T}), W(\mathbf{L}) \rangle$*, with* $\mathbf{T}$ *the set of variables of* ϕ*, i.e.,* ϕ*'s scope, and* $\mathbf{L}$ *the corresponding set of literals. The weight function* $W : \mathbf{L} \mapsto \{w \in \mathbb{Q} \mid (0 \leq w \leq 1) \wedge (\forall L \in \mathbf{L} : W(\neg L) = 1 - W(L))\}$ *assigns a rational weight between* 0 *and* 1 *to each literal* $L \in \mathbf{L}$*, such that the weight of* L *(denoted by* $W(L)$*) and the weight of its negation sum up to* 1*.*

Here, we follow Sato's semantics for probability distributions [160]. In practice, this means that we interpret the probability $W(L)$ to be the probability that L has the value $\top$. We also assume that the probabilities associated with variables are mutually independent. We will use literal-weighted propositional formulae as a way to represent probabilistic models (see Section 2.3.1) and for probabilistic reasoning about those models (see Section 2.2.3). Note that, because of the assumption that $W(\neg L) = 1 - W(L)$, probability-weighted propositional formulae describe Bernoulli distributions.

Specifically, we use WMC to compute the probability $P(\phi|_\sigma)$ of an event $\phi(\mathbf{D}, \mathbf{T})$, defined on decision variables $\mathbf{D}$ and stochastic variables $\mathbf{T}$, occurring for a given strategy σ that maps each decision variable in $\mathbf{D}$ to a truth value. Using the above definition of a probability-weighted propositional formula, we define the WMC as follows:

Definition 2.2.3 (Weighted model count)**.** *The weighted model count of a probability-weighted propositional formula $\langle \phi(\mathbf{T}), W(\mathbf{L}) \rangle$ is given by:*

$$wmc(\phi) = \sum_{v \in \mathcal{M}} \prod_{L \in v} W(L), \tag{2.1}$$

where v denotes a model of $\phi(\mathbf{T})$, represented as the set of literals that are set to $\top$ in that model and $\mathcal{M}$ denotes the set of models of $\phi(\mathbf{T})$, and $\mathbf{T}$ and $\mathbf{L}$ as defined in Definition 2.2.2.

We interpret $wmc(\phi)$ as the probability that $\phi(\mathbf{T})$ evaluates to $\top$, given the probabilities as defined by $W(\mathbf{L})$. We therefore also write $P(\phi)$ to denote ϕ's weighted model count. Note that Equation 2.1 shows that we assume that the probabilities associated with positive literals by W are mutually independent. If we had not made this assumption, we would not be able to simply multiply literal weights in Equation 2.1.

The WMC formalism can be used for counting the number of solutions to a propositional formula, and is known to be $\#\mathcal{P}$-complete [155].

We can now use the notion of the weighted model count of a probability-weighted propositional formula to compute the *success probability* $P(\phi|_\sigma)$ of the residual formula that is obtained when conditioning a propositional formula $\phi(\mathbf{D}, \mathbf{T})$ on a full strategy σ, which assigns truth values to all variables in $\mathbf{D}$. We illustrate this with the following example:

Example 2.2.1 (The weighted model count of a propositional formula)**.** *Consider*

the following propositional formula, strategy and weight function:

$$\begin{aligned} \phi_d(\mathbf{D},\mathbf{T}) &:= D_d \vee (D_c \wedge T_{cd}) \vee (D_b \wedge T_{bc} \wedge T_{cd}) \vee (D_a \wedge T_{ac} \wedge T_{cd}) \vee \\ &\qquad (D_b \wedge T_{ab} \wedge T_{ac} \wedge T_{cd}) \vee (D_a \wedge T_{ab} \wedge T_{bc} \wedge T_{cd}), \\ \sigma_d &:= \{D_a := \top, D_b := \top, D_c := \bot, D_d := \bot\}, \\ W(\mathbf{L}) &:= \{p_{T_{ab}} := 0.4, p_{T_{ac}} := 0.8, p_{T_{bc}} := 0.1, p_{T_{cd}} := 0.3\}, \end{aligned} \tag{2.2}$$

with $\mathbf{D} = \{D_a, D_b, D_c, D_d\}$ and $\mathbf{T} = \{T_{ab}, T_{ac}, T_{bc}, T_{cd}\}$. Here, we have only listed the weights of the positive literals for brevity, and denote the weight of a literal L with p_L, since it reflects a probability. Conditioning ϕ_d on σ_d yields the following residual formula:

$$\phi_d|_{\sigma_d} := (T_{bc} \wedge T_{cd}) \vee (T_{ac} \wedge T_{cd}) \vee (T_{ab} \wedge T_{ac} \wedge T_{cd}) \vee (T_{ab} \wedge T_{bc} \wedge T_{cd}). \tag{2.3}$$

We can now 'roll the dice' for each stochastic variable to obtain a truth value for it. The resulting possible world $v : \mathbf{T} \mapsto \{\bot, \top\}$ represents just one scenario of what could happen once the probabilistic truth values of the stochastic variables are revealed. Substituting the truth values specified by v into the residual formula $\phi_d|_{\sigma_d}$ either satisfies that formula, in which case $v \in \mathcal{M}$ is a model of $\phi_d|_{\sigma_d}$, or falsifies it.

We can now use Equation 2.1 *to compute the* success probability *of $\phi_d|_{\sigma_d}$, which we illustrate in Table 2.1, finding a success probability of $P\left(\phi_d|_{\sigma_d}\right)$* 0.2460 *for this formula.*

Table 2.1: An example of how to compute the weighted model count of Equation 2.3. Note that the table only lists interpretations that are models of $\phi_d|_{\sigma_d}$.

model	weight
$\{T_{ab} := \top, T_{ac} := \top, T_{bc} := \top, T_{cd} := \top\}$	$0.4 \cdot 0.8 \cdot 0.1 \cdot 0.3 = 0.0096$
$\{T_{ab} := \top, T_{ac} := \top, T_{bc} := \bot, T_{cd} := \top\}$	$0.4 \cdot 0.8 \cdot 0.9 \cdot 0.3 = 0.0864$
$\vdots$	$\vdots$
$\{T_{ab} := \bot, T_{ac} := \top, T_{bc} := \top, T_{cd} := \top\}$	$0.6 \cdot 0.8 \cdot 0.1 \cdot 0.7 = 0.0336$
	$P(\phi_d|_{\sigma_d}) = 0.2460$

2.2.4 Relevant complexity classes

The problems studied in this work are $\mathcal{NP}$-hard in the general case. They involve solving sub-problems (probabilistic inference problems, specifically) that are shown to be $\#\mathcal{P}$-complete. Later in this chapter, we will describe different types of probabilistic inference and how they relate to the SCPs studied in this

work. In order to put those into (their computational complexity) context, we first provide some definitions.

Recall from Section 2.2.1 that the SAT problem asks, for a given propositional input formula ϕ in CNF, if there is a model of that formula. If there is, we call such a formula *satisfiable*. Informally, problems that are members of the $\mathcal{NP}$ complexity class are decision problems for which we can efficiently verify if a candidate solution to an instance of this decision problem is indeed a solution to this problem instance. Here, the word "efficiently" means "in time polynomial in the input size on a deterministic Turing machine or equivalent model of computation".

In order to define the class $\mathcal{NP}$ more formally, we remark that we can encode a problem instance (a propositional formula, in case of SAT) as a string. We can define the problem size as the length of that string. Using these strings, we can define *languages* that describe a problem class. In the SAT example, the SAT language would consist of all strings that describe propositional formulae that are satisfiable.

We can now imagine the existence of an algorithm that checks (verifies) if an interpretation π of ϕ is indeed a model of ϕ. It thus takes two inputs: a string encoding the propositional formula and a string encoding the interpretation. In the computational complexity literature, this algorithm is typically modelled by a Turing Machine (TM). Using these notions, we can more formally define the class $\mathcal{NP}$ as follows (inspired by the definitions given by Arora & Barak [6] and Martin [124]):

Definition 2.2.4 (The class $\mathcal{NP}$). *Given a finite set of characters (an* alphabet*) A and a language $L \subseteq A^*$, which is a set of strings of arbitrary length, comprising characters from A. This language L is in $\mathcal{NP}$ iff there exists a polynomial $p : \mathbb{N} \mapsto \mathbb{N}$ and a polynomial-time TM M, which we call the* verifier *for L, such that for every string $x \in A^*$,*

$$x \in L \Leftrightarrow \exists u \in A^{p(|x|)} \text{ s.t. } M(x, u) = \top$$

Here, we call u a certificate *for input string x (with respect to L and M) if $x \in L$ and $u \in A^{p(|x|)}$, with $|x|$ the length of the input string x.*

Without loss of generality, we can choose this alphabet to simply be $\{\bot, \top\}$, which we use to both encode the input string x describing the problem instance, and the certificate u. Note that encoding x using an alphabet of $A = \{\bot, \top\}$ is note quite as trivial as encoding u. In the computational complexity literature, it is often assumed that the alphabet also has a *blank* symbol $\square$ and a *start* symbol $\triangleright$ for easier encoding in a Turing Machine. Because a discussion of how to encode x and a discussion of the exact workings of the Turing Machine model of computation are outside the scope of this work, we will ignore these special symbols.

As long as the alphabet is finite, the exact choice of the alphabet is unimportant, since we can always come up with a transformation of one alphabet to another. To get an intuition for this, consider two finite alphabets, $A = \{\bot, \top\}$ and B, with $|A| < |B|$. Now, B can easily simulate A by simply mapping $\top$ to a symbol $b \in B$ and $\bot$ to a symbol $b' \in B$ and not using all other symbols in B. Conversely, A can encode any symbol in B using $\log |B|$ bits.

For the sake of completeness, we also recap the concepts of *hardness* and *completeness*. Informally, a problem P' is called *hard* for a certain time complexity class if any problem P in that complexity class can be *reduced* to P' in polynomial time. Intuitively, this polynomial reduction means that we can transform input string $x \in L$ (the encoding of problem P) into a an input string $x' \in L'$ in time that is polynomial in $|x|$, such that a TM that verifies L' can be used to verify L.

More formally, we repeat the following definitions from Arora & Barak [6]:

Definition 2.2.5 (Reduction). *A language $L \in A^*$ is* polynomial-time Karp reducible to a language L'*, denoted by $L \leq_p L'$, if there is a polynomial-time computable function $f : A^* \mapsto A^*$ such that for every $x \in A^*$, $x \in L$ iff $f(x) \in L'$.*

Note that L and L' need not be languages on the same alphabet A. Rather, f may also be a function from strings on finite alphabet A to strings on a different finite alphabet B, because of the aforementioned possibility of using one alphabet to simulate another.

Using the definition above, we can give definitions for $\mathcal{NP}$-hardness and $\mathcal{NP}$-completeness:

Definition 2.2.6 ($\mathcal{NP}$-hardness and $\mathcal{NP}$-completeness). *We say that L' is* $\mathcal{NP}$-hard *if $L \leq_p L'$ for every $L \in \mathcal{NP}$. We say that L' is* $\mathcal{NP}$-complete *if L' is $\mathcal{NP}$-hard and $L' \in \mathcal{NP}$.*

Here the "NP" in $\mathcal{NP}$ stand for "non-deterministic polynomial". Intuitively, $\mathcal{NP}$-complete problems are those problems for which there cannot be an algorithm that can efficiently (*i.e.*, in polynomial time) and deterministically find a solution to an arbitrary instance of that problem that it is presented with, but for which it is possible to construct an algorithm that efficiently *checks* if a given solution (*e.g.*, a non-deterministic *guess*) to the problem instance is indeed a solution.

At least, that is the hunch that the overwhelming majority of computer scientists share on this topic. Unless we find that $\mathcal{P}$=$\mathcal{NP}$, in which case there would be a way to construct polynomial-time ("efficient") algorithms for solving $\mathcal{NP}$-complete problems, or definitively prove that $\mathcal{P} \neq \mathcal{NP}$, we operate under the *assumption* that no theoretically efficient (polynomial) algorithms exist to find a solution to an arbitrary instance of any $\mathcal{NP}$-complete problem.

Recall that we study stochastic constraint (optimisation) problems in this work, and aim to solve them *exactly*. In the case of the stochastic constraint *satisfaction* problems, it is clear that they are *decision problems*: we ask if the constraint is satisfied. We can turn stochastic constraint *optimisation* problems into stochastic constraint *satisfaction* problems by simply asking if the value that we aim to maximise exceeds a certain threshold (and analogously for minimisation problems). We discuss the relationship between stochastic constraint satisfaction and stochastic constraint optimisation in more detail in Section 4.1.

While the SCPs that we study in this work are shown to be $\mathcal{NP}$-hard in the general case, they need not be in $\mathcal{NP}$, and thus need not be $\mathcal{NP}$-complete. This is because real-world examples of SCPs (such as the ones studied in this work) often-times contain a lot of structures and symmetries that can be analysed to define *special cases* of SCPs for which we *can* design theoretically efficient solving algorithms.

As briefly touched upon in Section 1.3, in order to exactly solver SCPs, we need to evaluate the quality of different strategies, and evaluating the quality of a strategy is $\#\mathcal{P}$-complete. The complexity class $\#\mathcal{P}$ (pronounced "sharp p") captures the class of problems in which we are not just interested in *finding* a solution, but rather in *counting* all solutions to a problem instance. Perhaps unsurprisingly, the canonical problem of this complexity class is the *problem of counting the number of models of a propositional formula (#SAT)*, the *model counting problem* for short, in which we count all models of a given propositional formula, as discussed in Section 2.2.1.

More formally, we again give a definition analogous to the one given by Arora & Barak [6]:

Definition 2.2.7 (The class $\#\mathcal{P}$). *A function $f : A^* \mapsto \mathbb{N}$ is in $\#\mathcal{P}$ if there exists a polynomial $p : \mathbb{N} \mapsto \mathbb{N}$ and a polynomial-time TM M such that for every $x \in A^*$:*

$$f(x) = \left|\left\{u \in A^{p(|x|)} : M(x,u) = \top\right\}\right|.$$

Note that this definition implies that the count itself can be encoded in a string whose length is polynomial in the length of the input string x that encodes the problem instance. The weighted version of this problem, as described in Section 2.2.3, is also shown to be $\#\mathcal{P}$-complete in the general case [155].

Perhaps unsurprisingly, counting can be harder in practice than deciding. In particular, some problems that are easy to decide have an associated counting version that is hard. A well-known example are propositional formulae in DNF. Their satisfiability can be decided easily (while deciding their falsifiability

is hard), but counting the number of models of a DNF formula is not easier than counting the number of models of a CNF formula.

In order to define completeness for $\#\mathcal{P}$, we must first define $\mathcal{FP}$, the class of functions $f : A^* \mapsto \mathbb{N}$ that are computable by a deterministic polynomial-time Turing Machine [6]. Intuitively, computable functions are those that can be computed by an algorithm or computer without going into infinite loops. A formal definition of $\#\mathcal{P}$-completeness is outside the scope of this work, but informally, a function f is $\#\mathcal{P}$-complete if it is in $\#\mathcal{P}$ and the existence of a polynomial-time algorithm for f implies that $\#\mathcal{P} = \mathcal{FP}$. We refer the reader to the literature for more details, *e.g.*, *Computational Complexity*, by Arora & Barak [6].

Finally, we point to the class of problems that represents *decision versions* of problems in $\#\mathcal{P}$: $\mathcal{PP}$. Intuitively, and taking the #SAT problem as an example, the question asked in these problems is whether the model count of a propositional formula meets a certain threshold. More formally [6]:

Definition 2.2.8 (The class $\mathcal{PP}$)**.** *A language L is in $\mathcal{PP}$ if there exists a polynomial-time TM M and a polynomial $p : \mathbb{N} \mapsto \mathbb{N}$ such that for every $x \in A^*$,*

$$x \in L \Leftrightarrow \left|\left\{u \in A^{p(|x|)} : M(x,u) = \top\right\}\right| \geq \frac{1}{2} \cdot 2^{p(|x|)},$$

where we assume that $A = \{\bot, \top\}$.

Here, $2^{p(|x|)}$ represents the total number of possible certificates for the input instance x, and it is assumed that the length of a certificate u is polynomial in the length of x.

In the language of Turing Machines, we can think of L as a language whose strings are accepted by the *majority* of paths in non-deterministic TM M. Unlike problems that are in $\mathcal{NP}$, and taking the SAT problem as an example, we do not need to find just one model, but we must check that the majority of interpretations are models of the input formula.

We finally point to the concept of *oracles*, which are used in computational complexity theory to reason about different complexity classes. Informally, an oracle is a black box that answers queries. Even though those queries represent problems in a specific complexity class, it costs a Turing Machine only one time step to query the black box. For example, $\mathcal{NP}^{PP}$ is the class of problems that can be decided by a non-deterministic, polynomial-time Turing Machine, provided that it has access to an oracle that decides problems that are in $\mathcal{PP}$.

In the next section, we describe a number of probabilistic inference tasks that are members of the complexity classes described above.

2.3 Probabilistic inference

As we described in Chapter 1, we study problems that involve some sort of stochastic component, and thus require us to perform some kind of probabilistic reasoning. Given a probabilistic model, we use the term *probabilistic inference* to refer to answering *probabilistic queries* about the model, such as "what is the probability that it will be windy and rainy when I go outside?" In this section we describe several probabilistic queries known from the probabilistic inference literature.

Specifically, we identify three main types of inference tasks: *max-inference* tasks, *sum-inference* tasks and *mixed-inference* tasks [143]. They are listed here in increasing order of difficulty, their complexity classes ranging from $\mathcal{NP}$-complete, to $\#\mathcal{P}$-complete to $\mathcal{NP}^{PP}$-complete, respectively [47, 139], with $\mathcal{NP} \subseteq \#\mathcal{P} \subseteq \mathcal{NP}^{PP}$.

Before we describe these tasks, their complexities, and techniques that have been developed to solve them, we first introduce some notation and terminology related to the probabilistic models on which these tasks are formulated.

2.3.1 Probabilistic models

Probabilistic models can be represented in different ways. In the previous section, we discussed a specific way of modelling probability distributions: literal-weighted propositional formulae. A popular alternative approach to representing probabilistic models are graphical models [97, 140].

A well-known example of a graphical model is the *Bayesian network (BN)* [140], whose name was coined by Pearl in the 1980s. BNs are directed-acyclic graphs in which each node represents a variable and the directed edges indicate dependence relationships. With each node they also associate a *conditional probability table (CPT)* that describes that relationship. Another well-known example is the *Markov Network (MN)*, or *Markov Random Field (MRF)* [140], which is an undirected version of the Bayesian network.

However, in this work we choose *literal-weighted propositional formulae* to model probability distributions (as described in Section 2.2), since they generalise other approaches [32, 158]. Because of that choice, in this work, we consider problems formulated on Boolean variables. For ease of discussion, here we therefore assume that all variables have Boolean domains, but this need not be the case in general (in fact, they need not even be discrete).

The contents of this section are agnostic of any specific probabilistic model representation, unless indicated otherwise.

In our discussion of probabilistic inference tasks below, we consider a probabilistic model $\mathcal{P} = \langle \mathbf{E}, \mathbf{S}, \mathbf{Q}, \Phi \rangle$ on variables that are partitioned into three disjoint sets: **E**, **S** and **Q**. Intuitively, we can think of these sets as variables whose values represent *evidence* (**E**), variables that must be marginalised out (**S**), which is done by *summation*, and variables whose values we want to *query* (**Q**). Taking the spread of influence problem as described in Section 1.1 as an example, we can think of the *evidence* as a decision on which people get a free sample. Maybe we want to target a specific group (*e.g.*, women in tech, people who like running, reading enthusiasts, ...). We then *query* members of that group to predict the expected reach of our marketing campaign to that specific target audience, leaving members outside that target audience (who might still participate in spreading the word about our product) to be *marginalised out*.

Joint probability distributions on these variables are, in the graphical model literature, typically defined using a set **F** of *potentials* f. These potentials map sets of (truth value) assignments to variables in **X** to real numbers: $f : \{\top, \bot\}^{|\mathbf{X}|} \mapsto \mathbb{R}$. In general, each potential must take a nonnegative value for at least one set of truth value assignments [187]. In the context of graphical models, we can typically interpret these potentials as *conditional probability tables (CPTs)* that associate a probability with an assignment of truth values to the variables in the scope of the potential (**X**). We thus consider all the potential's values to be nonnegative in this chapter.

Recall the discussion of literal-weighted propositional formulae and their weighted model counts from Section 2.2.3. Similar to the CPTs in the context of graphical models, we can see Table 2.1 as a potential. Instead of having multiple potentials that define a probability distribution over subsets of all variables involved, now we have just one potential. This potential maps models of the literal-weighted propositional formula to their weights, and any interpretation of the formula that is not a model to 0.

We now continue with a description of the three main probabilistic tasks that can be formulated on probabilistic models.

2.3.2 Max-inference tasks

Max-inference tasks typically aim to find the most probable configuration of a joint probability distribution. A typical max-inference task is the *most probable explanation (MPE)* task. In some of the literature, this task is also known as the *maximum probability assignment (MPA)* task [20], or the *maximum a posteriori (MAP)* task, *e.g.*, in [1, 35, 91, 104, 112, 126, 153, 167, 184]. Given a probabilistic model $\mathcal{P}$ as described above, with $\mathbf{S} = \varnothing$, and some *evidence* **e** in the form of truth

assignments to the variables in $\mathbf{E}$, $\mathbf{e} : \mathbf{E} \mapsto \{\bot, \top\}$, the MAP task aims to find an assignment of truth values $\mathbf{q}$ to the variables in $\mathbf{Q}$, $\mathbf{q} : \mathbf{Q} \mapsto \{\bot, \top\}$, that maximises $P(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e})$.

Then, in the context of graphical models, for each possible $\mathbf{q}$ we can write:

$$P\left(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}\right) := \frac{\prod_{f \in \mathbf{F}} f\left(\mathbf{e}, \mathbf{q}\right)}{P(\mathbf{E} = \mathbf{e})} \tag{2.4}$$

Note that, in the context of literal-weighted propositional formula representations of probabilistic models, instead of Equation 2.4, for each possible $\mathbf{q}$ we would write:

$$P\left(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}\right) := \begin{cases} \prod_{L \in \mathbf{q}} W(L) & \text{if } \mathbf{e} \cup \mathbf{q} \in \mathcal{M}(\phi) \\ 0 & \text{otherwise,} \end{cases} \tag{2.5}$$

where $W(L)$ represents the weight of literal L and $\mathcal{M}(\phi)$ is the set of models of propositional formula ϕ. Note that this is simply the weight of the interpretation defined by $\mathbf{e} \cup \mathbf{q}$.

The goal of the MPE task is then to find an assignment of truth values to variables in $\mathbf{Q}$ that maximises this probability:

$$\mathbf{q}^* := \arg\max_{\mathbf{q}} P\left(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}\right), \tag{2.6}$$

where in fact we can also simply compute $\mathbf{q}^* \in \arg\max_{\mathbf{q}} \prod_{f \in \mathbf{F}} f(\mathbf{e}, \mathbf{q})$ (in the graphical model view), since $P(\mathbf{E} = \mathbf{e})$ is independent of $\mathbf{q}$. Where, for simplicity, we assume that the arg max function in the above formula returns just one assignment, even if more than one maximise $P\left(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}\right)$.

Note that, because $\mathbf{S} = \varnothing$, the MPE task can be seen as finding the most likely configuration of a set of variables $\mathbf{Q}$, given evidence about the variables in the complement of $\mathbf{Q}$, namely those in $\mathbf{E}$. As such, it is a very useful task for diagnostic purposes. For example: given a set of symptoms ($\mathbf{E} = \mathbf{e}$), a physician may want to ask what the probability is that the patient has a certain disease ($\mathbf{Q} = \mathbf{q}$).

The MPE inference task of determining if there exists a configuration $\mathbf{q}$ such that $P\left(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}\right) > \theta$ (with rational threshold $0 \leq \theta \leq 1$) is shown to be $\mathcal{NP}$-complete [20, 167]. In fact, even finding a solution to the MPE problem whose quality is guaranteed to be within a constant ratio ρ_{MAP} of the optimal solution, is shown to be $\mathcal{NP}$-hard [1].

The earliest exact methods for solving MPE were *join-tree* algorithms, devised in the context of Bayesian networks, where potentials are computed one by one, in a strict order that is determined by the (structure of the) problem, often using the potentials that were computed earlier [90, 101, 140]. In the late 1990s, Dechter

proposed the *bucket elimination* framework as a generalisation of *variable elimination algorithms*, typically used for *mixed-inference* tasks, which we will briefly discuss in Section 2.3.4. Because the max-inference and mixed-inference tasks are so closely related, Dechter proposed a general framework for solving these task. It provides functionality for balancing the space and time requirements of variable elimination algorithms [53]. Note that the efficiency of variable elimination algorithms is determined by the variable order inherent to the specific problem instance.

Alternative approaches to exact MPE solving include one based on a modification of the DPLL algorithm, using a dynamic programming approach [159], and one that is based on *integer linear programming (ILP)* [153]. Another class of algorithms encodes the problem in an AND/OR diagram (which exploits the independencies in the graphical model), and uses a depth-first branch-and-bound search to traverse that diagram in order to solve the MPE task [115, 116].

Finally, we point to the existence of approximation methods for MAP, based on, *e.g.*, local search [139], mini-bucket elimination algorithm [54], (hybrid) message passing algorithms [91, 112], weighted search [67] and others [103, 119, 138, 143, 186]. A detailed discussion of these techniques is outside the scope of this work, since we focus on exact SCP solving.

Note that solving SCPs as described in Section 1.1 never requires max-inference *only*. Instead, we always require a form of inference that performs some kind of aggregation over probabilistic paths in networks, and thus *sum-inference*.

2.3.3 Sum-inference tasks

The *probabilistic logic programming (PLP)* literature identifies an inference task in which the set $\mathbf{Q}$ contains only one variable ($\mathbf{Q} = \{Q\}$), there are no observed variables, and thus there is no evidence ($\mathbf{E} = \varnothing$), but there are latent (*i.e.*, unobserved or uninteresting) variables that must be marginalised out ($\mathbf{S} \neq \varnothing$). In this case, we call $P(Q = \top)$ the *success probability* of *query* Q, which, in the graphical model context, is computed as:

$$P(Q = \top) := \frac{1}{Z} \dot{\sum_{\mathbf{s}}} \prod_{f \in \mathbf{F}} f\left(\mathbf{s}, Q = \top\right), \tag{2.7}$$

where Z is a constant needed for normalisation, often referred to as a *partition function*. In the literal-weighted propositional formula context, computing $P(Q = \top)$ corresponds to computing the weighted model count of a formula that can only be satisfied if $Q = \top$, using Equation 2.1.

This task is known in the PLP literature as the *marginal distribution (MARG)* [52, 64, 65] or simply *MAR* [47] task, or the *PROB* task [96].

This task is known to be $\#\mathcal{P}$-complete in the general case [155, 176, 177]. The decision version of this problem asks if $P(Q = \top) > \theta$ holds for a given threshold $0 \leq \theta \leq 1$. This problem is $\mathcal{PP}$-complete [6, 47].

Since the task of computing the success probability of a query is such an important task in the PLP community, many methods for solving this task either exactly or approximately have been developed over the years. In particular, it has been shown that the MARG task can be reduced to the WMC problem [8, 9, 43], which we briefly referred to in Section 2.2.4. Consequently, a lot of the literature on solving this task is based on model counting techniques.

An early example of this is the suggestion by Bacchus and Dalmao to adapt the Davis-Putnam-Logemann-Loveland (DPLL) algorithm such that it can be used for model counting [9]. They observed that computing the conditional probability distributions for variables in a Bayesian Network (the *BAYES* problem), and counting the number of model of a propositional formula (#*SAT*) are instances of the same *SUMPROB* problem, as identified by Dechter [53]. They thus proposed to solve the BAYES problem with their #DPLL algorithm [8, 9], which eventually led to weighted model counter Cachet, which employed *conflict-driven clause learning (CDCL)* for efficient weighted model counting [157, 158]. Somewhat more recent model counters of this type include weighted versions of miniC2D [134] and sharpSAT [173], specifically our weighted version[1], a version called sharpSAT-TD[2], and a weighted version of GANAK [166].[3] Finally, Chakraborty *et al.* showed that, under certain assumptions, literal-weighted propositional formulae can be transformed into unweighted propositional formulae whose model counts can then be transformed back into probabilities, thus allowing any unweighted model counter to be transformed into a weighted one [29].

These model counting techniques typically require the input to be in a certain propositional language (typically CNF). However, a large part of the probabilistic inference literature assumes the probabilistic model to be encoded as a graphical model, thus requiring some kind of encoding step before solving. Several methods for encoding probabilistic graphical models into (literal-)weighted CNFs have been proposed [13, 29, 33, 43], but a detailed discussion of them is outside the scope of this work.

A weighted model counter takes a (literal-)weighted CNF as input and returns

[1] Available at `bitbucket.org/latower/weighted-sharpsat`.

[2] Available at `github.com/Laakeri/sharpsat-td`.

[3] Weighted version available at `github.com/meelgroup/ganak/tree/wmc`.

a probability. In some contexts, this might be enough. However, in some other contexts, we may want to query the same model multiple times, only with different weights, or with different decisions. For example, in the context of the power grid reliability problem, experts may have provided us with upper and lower bounds on the survival probabilities of the power lines. Using those bounds, we may want to formulate queries for best-case and worst-case scenarios. The underlying model is the same for these two scenarios, but the weights are different. Taking the spread of influence problem as another example, we may just want to determine the expected number of eventual customers given different 'seed' sets of people who receive a free sample. Again, the underlying model is the same, but some decisions (and thus the values of decision variables) are different.

In these examples, we want to compute the success probability of a literal-weighted formula multiple times, only with slightly different weights. Using the DPLL-based algorithms, we would have to run the algorithm again for each different weight function, and we cannot reuse any results.

This single-use property of model counters is a drawback if we may want to perform multiple queries on the same model. These observations are addressed by the field of *knowledge compilation* [48, 123, 165], where a propositional formula is compiled into a *decision diagram (DD)*. These data structures capture the model into a data structure that allows for repeated querying in time that is polynomial (typically linear) in the size of the DD. Since this work relies heavily on knowledge compilation, we discuss this in more detail in Section 2.4.

Due to the complexity of the MARG task, there is also a class of bounding and approximation algorithms for solving this problem. One of the earliest methods simply uses a SAT solver to generate sample solutions, whose weights can be used to estimate the total weighted model count of a literal-weighted input formula [182], taking care to take into account the weight of the samples themselves [28]. Later methods take a hashing-based approach, which adds random XOR constraints to the formula, which cuts down the solution space until it is small enough to count. The total model count is then estimated by repeating this procedure, resulting in probabilistic upper and lower bounds on the weighted model count [30]. Another method is an anytime approach from the PLP literature simply generates partial proofs by partially grounding a probabilistic logic program, generating lower and upper bounds on the success probability that get closer as the algorithm continues to run [151, 152].

Finally, we mention parallelised methods for weighted model counting, such as ones that utilise GPUs [62] or implement parallel DD compilation algorithms [40].

2.3.4 Mixed-inference tasks

Recall the probabilistic model we described above: $\mathcal{P} = \langle \mathbf{E}, \mathbf{S}, \mathbf{Q}, \mathbf{\Phi} \rangle$. For the definition of the MPE problem, we assumed that $\mathbf{S} = \varnothing$. For the definition of MARG inference, we assumed that $\mathbf{E} = \varnothing$. We now discuss a generalisation of the MPE in which $\mathbf{S} \neq \varnothing$ and $\mathbf{E} \neq \varnothing$, known as the MAP problem. As we mentioned in Section 2.3.2, some literature calls the MPE task the MAP task. In these works, the mixed-inference task is called *marginal MAP (MMAP)* (*e.g.*, in [1, 35, 91, 104, 112, 126, 153, 167, 184]) or partial MAP [100].

The MAP task is to find an assignment of truth values $\mathbf{q}$ to the variables in $\mathbf{Q}$ that maximises $P(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e})$, and thus given only *partial* evidence on the variables in the complement of $\mathbf{Q}$.

In the MAP setting, we have to marginalise out the variables in $\mathbf{S}$, which present the 'hidden' variables that we neither know, nor care about, resulting in the following expression for the probability of an instantiation $\mathbf{q}$ of the variables in $\mathbf{Q}$, in the context of graphical models:

$$P\left(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}\right) := \frac{1}{P\left(\mathbf{E} = \mathbf{e}\right)} \cdot \sum_{\mathbf{s} \in \{\top, \bot\}^{|\mathbf{S}|}} \prod_{f \in \mathbf{F}} f\left(\mathbf{e}, \mathbf{q}, \mathbf{s}\right). \tag{2.8}$$

In the context of literal-weighted propositional formulae, this would correspond to computing the weighted model count of the residual formula $\phi|_{\mathbf{q},\mathbf{e}}$, obtained by substituting the variables in $\mathbf{Q} \cup \mathbf{E}$ with their truth values according to $\mathbf{q}$ and $\mathbf{e}$, and simplifying the resulting formula. Consequently, the sum in Equation 2.1 then only runs over interpretations $\mathbf{s}$ that are models of $\phi|_{\mathbf{q},\mathbf{e}}$. Similar to the MPE task, MAP aims to find the solution to Equation 2.6, but uses Equation 2.8 to compute $P\left(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}\right)$ instead of Equation 2.4:

$$\mathbf{q}^* := \arg\max_{\mathbf{q}} \sum_{\mathbf{s} \in \{\top, \bot\}^{|\mathbf{S}|}} \prod_{f \in \mathbf{F}} f(\mathbf{e}, \mathbf{q}, \mathbf{s}), \tag{2.9}$$

where we note that, again, the factor $P(\mathbf{E} = \mathbf{e})$ is unimportant for the purposes of finding $\mathbf{q}^*$.

A generalisation of the MAP task is the *maximum expected utility (MEU)* task [53], which is formulated on a probabilistic model $\mathcal{P} = \langle \mathbf{E}, \mathbf{S}, \mathbf{Q}, \mathbf{\Phi}, \mathbf{U} \rangle$, where we are given a utility function that associates a utility $u(\mathbf{e} \cup \mathbf{s} \cup \mathbf{q})$ with each possible instantiation of the variables in $\mathbf{E} \cup \mathbf{S} \cup \mathbf{Q}$. Instead of maximising the marginal probability, this problem maximises the expected utility. As such, it is a popular setting in the field of optimisation, planning and scheduling [5, 102]. Clearly, we can formulate SCPs as MEUs, provided that we can encode the constraints into the probabilistic model.

Note that solving the MAP problem involves both maximisation and summation. Since the maximisation and summation operators do not commute, mixed-inference tasks are typically harder than either sum-inference or max-inference alone. In fact, Park and Darwiche proved that MAP is $\mathcal{NP}^{PP}$-complete in the general case [139], whereas MPE is 'only' $\mathcal{NP}$-complete.

There is a rich literature on solving the MAP task, with methods for solving the problem exactly, approximately, or for computing bounds on the solution. We highlight a few common methods.

The MAP task is naturally solved by using some form of variable elimination [53]. Early exact algorithms for MAP use variable elimination in a branch-and-bound algorithm to find an optimal solution [138, 186], in some cases even in combination with knowledge compilation [83]. Just like AND/OR diagrams can be used in algorithms to solve the MPE problem, they can also be used to solve MAP, by combining them with depth-first search [117] or best-first search [118].

In order to employ branch-and-bound algorithms, we need to compute actual upper bounds on the conditional probability. Perhaps unsurprisingly, techniques that have been developed for this are quite similar to those that have been developed for solving MPE tasks, and the MPE approximation methods that we mentioned in Section 2.3.2 can be used for this purpose.

Given the applicability of AND/OR diagrams (combined with a search algorithm) to solve both MPE and MAP exactly, it is unsurprising that anytime variants of these algorithms have also been developed over the last years [104, 119, 120, 122]. Some methods not only use MPE approximation methods to obtain meaningful upper bounds on the optimal probability of a MAP, but also employ weighted search methods [67] to also obtain meaningful lower bounds [121].

Other approximation methods rely on decomposition and approximate variable elimination [35, 143], on repeatedly performing the MARG task on each variable in the MAP to compute a lower bound [4], gradient-based methods [39], to name a few. Because of its hardness and usefulness to model a wide range of problems, many MAP approximation methods have been developed over the years, and still are being developed. Since our focus is on exact solving, we do not expand further on approximate MAP, considering it to be outside the scope of this work.

In this section, we gave a very brief overview of the main inference tasks in the probabilistic inference literature. This literature displays a strong focus on graphical models of probability distributions. As we motivated earlier, in our work we focus on propositional representations of probability distributions. In the next

section, we give a brief introduction to a useful tool in the realm of propositional inference: knowledge compilation.

2.4 Knowledge compilation

Historically, *knowledge compilation* [48, 123, 165] has been a popular method for making online WMC computation more tractable in the field of probabilistic inference and planning [33, 45, 52, 64, 82]. As we mentioned in Section 2.3.3, computing the success probability of a query is a well-known task in the field of PLP. In stochastic optimisation problems, it is only natural to want to repeatedly compute a conditional success probability, conditioned on different evidence (or strategies). However, recall from Section 2.3.3 that WMC computation can be done by using a 'single-use' weighted model counter. Consequently, if we would want to recompute the weighted model count of a probabilistic model after changing the evidence, we would also have to rerun the solver, discarding any partial results that might have been reusable.

2.4.1 Decision diagrams

Knowledge compilation represents a solution to this problem. Most knowledge compilers are essentially DPLL-based model counters that record their *trace* (the search tree) while counting. Taking care to create the trace in such a way that it has specific properties, the result is saved in a language that supports tractable (meaning "in time that is polynomial in the size of the string in that language") inference operations. A formula (or *sentence*) in this language can be represented as a *decision diagram (DD)*, and is then said to be 'compiled'. This DD can be seen as a compact representation of the truth table of the input formula.

Recall Table 2.1 in Section 2.2.3. In essence, this table is a truth table (albeit one that only lists the rows that represent models of ϕ_d). It will not surprise the reader that, by adding appropriate *weights* to a DD representation of the truth table of a literal-weighted propositional formula, we can use that DD to compute the WMC of that formula. Note also that, once the diagram is compiled, these weights can be changed to, *e.g.*, represent different assumptions about the exact probabilities in the probabilistic model, or to represent different strategies that we need to evaluate in order to solve a SCP.

Examples of DDs that are used for tractable MARG inference include *binary decision diagrams (BDDs)* and *ordered binary decision diagrams (OBDDs)* [42, 45], *negation normal forms (NNFs)* [32], *deterministic decomposable negation normal forms*

(d-DNNFs) [34], *sentential decision diagrams (SDDs)* [37], *weighted positive binary decision diagrams* [41], and *algebraic decision diagrams (ADDs)* [60]. In this work, we focus on *ordered binary decision diagrams (OBDDs)* and *sentential decision diagrams (SDDs)*, specifically. For a good overview of the different properties of some of these languages and on how they relate to each other, we refer the reader to Darwiche's *A knowledge compilation map* [48].

Note that, while all these diagrams support inference operations that take time polynomial in the size of the diagram, this diagram must still be compiled. Consequently, for a single query, the total time complexity would not be reduced if using knowledge compilation instead of running a model counting algorithm for that query. However, for repeated querying under different assumptions, this computational effort is 'shared' among all those queries.

An additional potential time saver is the fact that propositional formulae can have sub formulae in common. This is particularly likely to happen when these formulae originate from the same system. In Section 4.2 we will show how queries about real-world systems can be translated into propositional formulae. Because these formulae represent questions about the same stochastic system, they are likely to overlap in part. In this case, we can choose not to construct an individual DD for each formula, but instead compile one DD with multiple roots, each root corresponding to a different formula. This way, we can potentially save compilation time and memory, by avoiding to repeatedly recompile the same sub formulae, but instead re-using those compiled sub formulae. For the sake of simplicity, the discussion below will be limited to single-rooted DDs.

While inference operations can be done in time polynomial in the size of the diagram, the size of the diagram may still be exponential in the size of the input CNF in the worst case [48]. The task of finding a minimal-sized DD is typically also hard. In fact, finding a minimal-size OBDD is known to be $\mathcal{NP}$-hard [22], and we expect the same to hold for SDDs, although we are not aware of a published proof of this. There is a rich literature on how to compile succinct diagrams, the discussion of which is outside the scope of this work.

Note also that, since the knowledge compilation process involves storing the full trace (search tree) of the DPLL-based model counter that the compiler is built on, it may require a lot of memory, which can be prohibitive.

To summarise: compiling CNFs to DDs gives us data structures that we can use for tractable inference *if* we are able to make these DDs compact enough, and *if* we have enough memory to compile the diagram. Additionally, this effort is only useful if we need to answer multiple queries. Taking this into consideration, the question arises of why we use knowledge compilation in this work. In short:

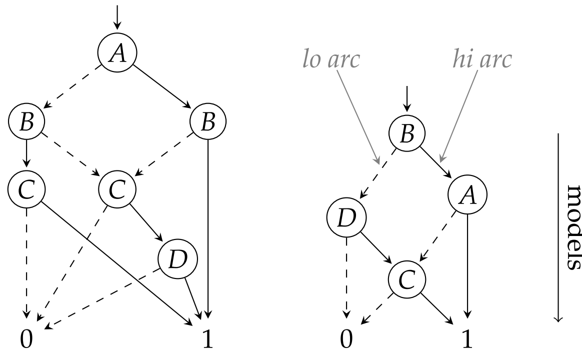


Figure 2.1: *Two small OBDDs, each encoding the propositional logic formula $\phi = (A \land B) \lor (B \land C) \lor (C \land D)$. The left OBDD has variable order $A \prec B \prec C \prec D$, while the right one has variable order $B \prec A \prec D \prec C$.*

DDs help us to model relationship between variables and exploit those relationships to solve SCPs. We will answer this question in detail in Chapters 5 and 6.

In the remainder of this section, we will first provide some background on two specific types of DDs: *ordered binary decision diagrams (OBDDs)* and *sentential decision diagrams (SDDs)*. In the next section, we show how to use them specifically for the task of WMC for probabilistic inference.

2.4.2 Ordered binary decision diagrams

Figure 2.1 shows two examples of OBDDs, each representing the truth table of the same propositional formula. An OBDD is a *directed acyclic graph (DAG)* with two leaf nodes that represent the values *true* (1) and *false* (0). In an OBDD $\Delta(\phi)$ that encodes a formula ϕ, each internal node n is labelled with a variable $X \in sc(\phi)$. There can be multiple nodes with the same label, but never multiple nodes with the same label *on a path* from root to leaf. Each internal node n has two outgoing arcs: a *lo arc* that corresponds to $X = \bot$ and a *hi arc* that corresponds to $X = \top$, where $X \in sc(\phi)$ labels n.

A path from the root of the diagram to the leaf node labelled with 1 corresponds to a *model* of ϕ, a mapping $sc(\phi) \mapsto \{\bot, \top\}$ of truth values on the variables in the scope of ϕ. Note that not all variables may be encountered in a path from root to leaf, since assignments of truth values to variables may make the satisfiability of the resulting residual formula agnostic to the truth values of some of the other variables. As a consequence, OBDDs can be used to very compactly encode all the models of a propositional formula.

The size and shape of an OBDD are determined by its *variable order* $\mathcal{O}$, which indicates in which order we encounter the variables in $sc(\phi)$ on a path from the root of the OBDD to a leaf. The two OBDDs in Figure 2.1 are shaped by two different variable orders.

2.4.3 Sentential decision diagrams

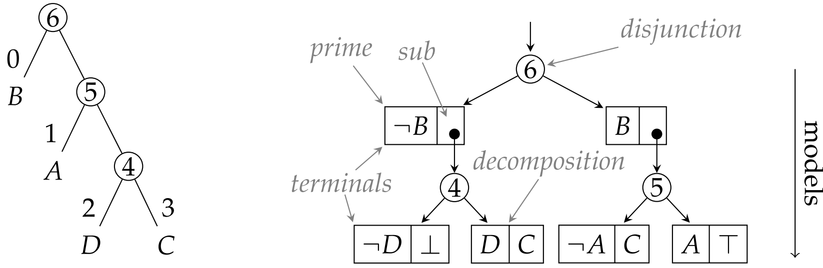
Figure 2.2 shows two examples of SDDs. Like OBDDs, SDDs are compact representations of truth tables.

In Figure 2.2 the circular nodes represent disjunctions and rectangular nodes represent decompositions of a *prime* p (rooted in the left half of the node) and a *sub* s (rooted in the right half), such that a decomposition node represents the formula $p \wedge s$. A single variable or constant in a prime or a sub is called a *terminal*. Naturally, a disjunction node is *true* if at least one of its children is *true*. A conjunction node is *true* if both the prime and the sub evaluate to *true*.

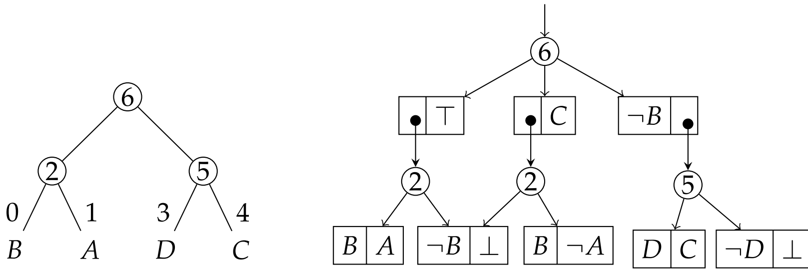
As with OBDDs, there is typically no unique SDD representation for a propositional formula. Rather, the shape and size of an SDD is determined by the *vtree* [144] that it *respects*. A vtree is a full binary tree that generalises the concept of a variable order. In particular: each disjunction node in an SDD respects a subvtree of the subvtree that the entire SDD respects, rooted at an internal node in that vtree. The left and right children of that vtree node determine the scopes of respectively the primes and subs of the children of the disjunction node in de SDD.

Consider the SDDs and corresponding vtrees in Figure 2.2. Each disjunction node in the SDD is labelled with the index of the internal vtree node that is the root of the subvtree respected by that disjunction node. The root disjunction node of the SDD respects the entire vtree. Let ϕ_p and ϕ_s be the propositional formulae represented by the prime and the sub of any decomposition node that is a descendant of a disjunction node Δ. Let ℓ and r be the left child and the right child of an internal vtree node n, respectively. We say that Δ respects n if the following holds: $sc\left(\phi_p\right) \subseteq \{X_l \in \mathcal{T}_\ell\}$ and $sc\left(\phi_s\right) \subseteq \{X_l \in \mathcal{T}_r\}$, where $\mathcal{T}_\ell$ and $\mathcal{T}_r$ represent the sub vtrees rooted at ℓ and r, respectively, and X_l represents the variable that labels a leaf in those sub vtrees.

Thus, in the (sub) SDD rooted at Δ, the sub formulae corresponding to the *primes* of the decomposition nodes that are Δ's children only contain variables that occur in the vtree rooted at the *left* child of internal vtree node n, and the sub formulae corresponding to the *subs* of the decomposition nodes that are Δ's children only contain variables that occur in the vtree rooted at the *right* child of n.



***(a)** An SDD (right) that respects a right-linear vtree (left).*



***(b)** An SDD (right) that respects a balanced vtree (left). Example from Darwiche [46]*

Figure 2.2: *Two examples of SDDs encoding the truth table of propositional formula $\phi = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$, and their corresponding vtrees. Both internal nodes and leaves in the vtrees are labelled with an index to indicate their place in the variable order. Disjunction nodes in SDDs are labelled with the index of the internal vtree node they respect.*

The figure shows two examples of how vtrees influence the size and shape of SDDs. In particular, we distinguish three types of vtrees: *right-linear*, *left-linear* and *balanced*. Figure 2.2a shows an example of an SDD that respects a *right-linear* vtree, while Figure 2.2b shows an example of an SDD that respects a *balanced* vtree.

Note that we can distill a total order from vtrees by doing a left-right traversal and noting the order in which we encounter the variables. Unlike OBDDs, the size and shape of SDDs are not defined by a total order, but by a vtree. This is also illustrated in Figure 2.2. Both vtrees have the total order $B \prec A \prec D \prec C$, but they have different shapes and thus correspond to differently shaped SDDs.

In a top-down traversal of an SDD, we can interpret the primes as conditions: if the prime evaluates to *true*, then the condition in the sub must evaluate to *true* in order to make the formula evaluate to *true*. Right-linear vtrees have the special property that the SDDs that respect them are equivalent to OBDDs. Comparing the right OBDD in Figure 2.1 and the SDD in Figure 2.2a, we see that the primes

in the SDD correspond to the values of the outgoing arcs in the OBDD.

SDDs are a strict superset of OBDDs [46]. When an SDD respects a right-linear vtree, its primes can only condition on truth assignments to single variables (as is shown in Figure 2.2a). For other vtrees, however, primes might represent entire sub formulae (as is shown in Figure 2.2b), which are called *sentences* in the knowledge compilation literature (hence the name *sentential* decision diagrams). Because of this, truth tables can potentially be encoded more efficiently when the vtree is not right-linear, and thus SDDs can be made at least as small as OBDDs [24].

We remark that SDDs and OBDDs are not the only DDs that can be used for conditional probability computation in time that is linear in the size of the diagram. Other examples include *negation normal forms (NNFs)* [95], d-DNNFs [33, 145], *smooth deterministic decomposable negation normal forms (sd-DNNFs)* [95] and *affine decision trees (ADTs)* [98]. A detailed discussion of these is outside the scope of this work. We point the interested reader to Darwiche & Marquis's *A Knowledge Compilation Map* [48] on how most of these languages relate to each other.

2.5 Inference with decision diagrams

We now describe, mainly with help of examples, how we can use OBDDs and SDDs for probabilistic inference. The propositional formula that we will be using in these examples is the following:

$$\begin{aligned}\phi_d(\mathbf{D},\mathbf{T}) := \; & D_d \vee (D_c \wedge T_{cd}) \vee (D_b \wedge T_{bc} \wedge T_{cd}) \vee (D_a \wedge T_{ac} \wedge T_{cd}) \vee \\ & (D_b \wedge T_{ab} \wedge T_{ac} \wedge T_{cd}) \vee (D_a \wedge T_{ab} \wedge T_{bc} \wedge T_{cd}) \, ,\end{aligned} \tag{2.10}$$

with $\mathbf{D} = \{D_a, D_b, D_c, D_d\}$ a set of Boolean decision variables and $\mathbf{T} = \{T_{ab}, T_{ac}, T_{bc}, T_{cd}\}$ a set of Boolean stochastic variables. We will discuss the origin of this formula in more detail in Section 4.2. Recall that $\phi_d|_\sigma(\mathbf{T})$ is the residual formula obtained by taking $\phi_d(\mathbf{D},\mathbf{T})$ and replacing the variables in $\mathbf{D}$ by their values specified by strategy σ and simplifying. Since the goal of solving Equation 1.1 is finding a strategy σ that satisfies that constraint, we are going to describe how to use OBDDs and SDDs to compute $P(\phi_d|_\sigma)$, particularly.

As discussed in the previous section, DDs are data structures that summarise truth tables of propositional formulae, and we can use DDs to compute the (weighted) model count of a formula in time that is linear in the size of the DD, instead of simply listing all the models of the formula, determining their individual weights and summing those, as we did in Table 2.1. In order to employ a DD to compute the success probability of a probability-weighted propositional

formula, we must transform it into an *arithmetic circuit (AC)*. In the following, we will not explicitly show the *arithmetic circuits (ACs)*, but describe how to transform an OBDD or SDD into one.

Note that we compile the entire formula ϕ, and then compute $P(\phi|_\sigma)$ by setting the appropriate weights in the AC to reflect σ. Once those weights are defined, a bottom-up traversal of such an AC computes the success probability of $\phi|_\sigma$. For the sake of brevity, we will sometimes abuse terminology and say that we traverse the DD to compute that probability, where we actually mean that we traverse the AC that is obtained from the DD. Note that, for OBDDs and SDDs, the size of the AC is linear in the size of the DD it was constructed from.

As we discussed in Section 2.4.1, a DD may have multiple roots, each corresponding to a different formula. Consequently, an AC constructed from such a multi-rooted DD may also have multiple roots, where each returns the success probability of a different query. For the sake of simplicity, the discussion below will be limited to single-rooted ACs.

2.5.1 Inference with OBDDs

In this section, we briefly discuss how to compute conditional probabilities using an OBDD representation of a (weighted) propositional formula, in time that is linear in the size of the OBDD.

To see how we can compute $P(\phi_d|_\sigma)$ using an OBDD [42, 45] in linear time, consider Figure 2.3. This OBDD has two types of internal nodes. The square nodes are labelled with decision variables $D_i \in \mathbf{D}$, and we refer to those nodes as *decision nodes*. The circular nodes are labelled with stochastic variables $T_{ij} \in \mathbf{T}$, and we refer to those nodes as *stochastic nodes*. The two leaf nodes are labelled with 0 and 1, which represent the values *false* and *true*, respectively. A path from the root of an OBDD to the leaf labelled with 1 corresponds to a (sub)set of the set of (variable, truth value) pairs that form a model for the formula encoded by the OBDD. This subset is sufficient for satisfying the formula, and its weight equals the sum of the weights of all models that are its supersets. Each model of the formula is defined by exactly one such path/(sub)set.

We map this OBDD to an AC to compute the probability that ϕ_d in Example 4.2.3 evaluates to *true* under a strategy σ as follows. The weights on the outgoing arcs of a stochastic node correspond to the probability that the variable that labels that node is *true* (for the solid, or *hi*, arcs) or *false* (dashed, or *lo*, arcs). We add a strategy σ on the OBDD by adding weights of 0 and 1 to the appropriate outgoing arcs of the decision nodes. The OBDD in Figure 2.3 does not reflect any specific strategy.

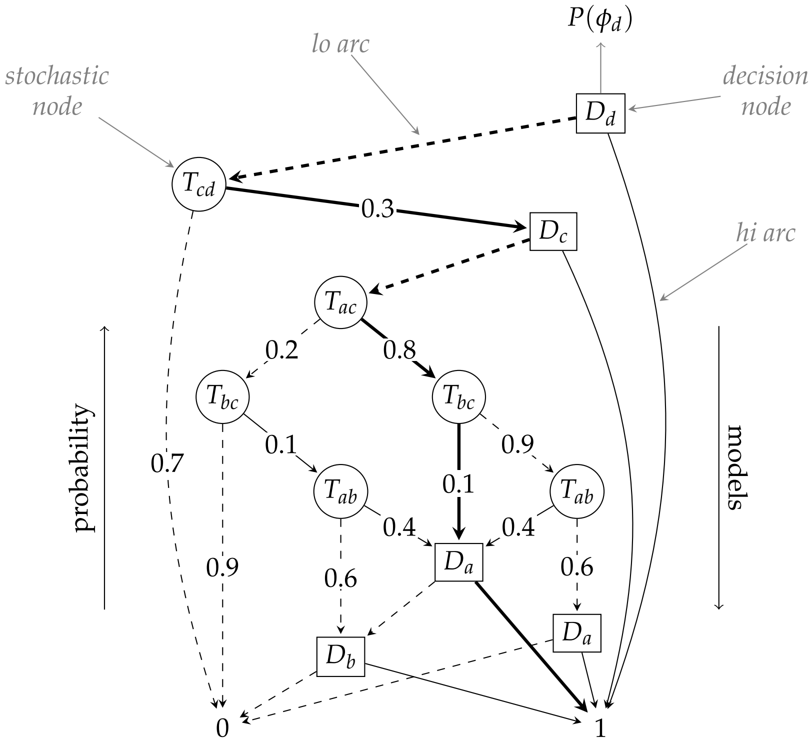


Figure 2.3: *An OBDD representation of Equation* 2.10*, mapped to an AC. This OBDD has variable order* $D_d \prec T_{cd} \prec D_c \prec T_{ac} \prec T_{bc} \prec T_{ab} \prec D_a \prec D_b$*. Bold arcs represent one of the models of* ϕ_d*:* $\{D_d := \bot, D_c := \bot, T_{ac} := \top, T_{bc} := \top, D_a := \top\}$.

We can now compute $P(\phi_d|_\sigma)$ as follows. In a bottom-up traversal of the OBDD, each node r is assigned the following *score*:

$$v(r) := w(r) \cdot v\left(r^+\right) + (1 - w(r)) \cdot v\left(r^-\right), \tag{2.11}$$

where $0 \leq w(r) \leq 1$ represents the weight of the variable that labels r, r^+ (r^-) is the *hi* (*lo*) child of r, i.e., the child connected through the solid (dashed) outgoing arc of r; $v(r) := 0$ for the negative leaf and $v(r) := 1$ for the positive leaf. Observe that $v(root) = P\left(\phi|_\sigma\right)$. Note that it takes one *bottom-up* traversal of this AC to compute the score of the root.

In the interest of brevity, in the remainder of this work, we will sometimes abuse terminology and refer to the OBDD when we actually mean the AC that the OBDD is mapped onto.

Example 2.5.1 (WMC on an OBDD)**.** *Consider the OBDD in Figure 2.3. Suppose we want to compute* $P(\phi_d|_\sigma)$*, with* $\sigma := \{D_d := \bot, D_c := \bot, D_a := \top, D_b := \top\})$.

We label the dashed outgoing arcs of nodes labelled with D_d and D_c, as well as the solid outgoing arcs of nodes labelled with D_a and D_b with the value 1. Similarly, we label the solid outgoing arcs of nodes labelled with D_d and D_c, as well as the dashed outgoing arcs of nodes labelled with D_a and D_b with the value 0. Then, we perform a bottom-up sweep of the diagram to compute the score of each node, by computing the weighted sum of its children, as per Equation 2.11.

This yields a score of 1 *for the nodes labelled with D_a, D_b and T_{ab}, and for the right node labelled with T_{bc}. The left T_{bc} node has a score of* 0.1. *The T_{ac} and D_c nodes each have a score of* 0.82, *and the nodes labelled with T_{cd} and D_d each have a score of* 0.246. *Because the node labelled with D_d is the root node of the diagram, we conclude that* $P(\phi_d \mid \{D_d := \bot, D_c := \bot, D_a := \top, D_b := \top\}) = 0.246$.

2.5.2 Inference with SDDs

Just like OBDDs, we can use SDDs to compute the success probability of a residual propositional formula, by mapping the SDD onto an AC. Note that here, too, the time it takes to compute those probabilities is linear in the size of the DD, in this case an SDD. Since SDDs can be made more succinct than OBDDs through minimisation [24], the ACs we derive from them can also be more succinct, and therefore more efficient tools for repeated querying, than ACs obtained from OBDDs.

To compute $P\left(\phi_d|_\sigma\right)$ with an SDD, we construct an AC as follows. We replace the subs and the primes in Figure 2.4 with their weight according to the corresponding probability (in case of stochastic variables T) or their assignment (in case of decision variables D). We compute $P(\phi_d|_\sigma)$ in a bottom-up traversal of the SDD, where each disjunction node r takes score

$$v(r) := v\left(p^\ell\right) \cdot v\left(s^\ell\right) + v\left(p^r\right) \cdot v\left(s^r\right), \tag{2.12}$$

where p^ℓ (p^r) denotes the prime of the left (right) child of r, and s^ℓ (s^r) the sub of the left (right) child, and $v(p)$ or $v(s)$ is the weight of the terminal in the corresponding prime or sub, or the score of the sub formula rooted in that prime or sub. Again, $v(root) = P\left(\phi_d|_\sigma\right)$. And again, in the interest of brevity, in the remainder of this work, we will sometimes abuse terminology and refer to the SDD when we actually mean the AC that the SDD is mapped onto.

Example 2.5.2 (WMC on an SDD)**.** *Consider the SDD in Figure 2.4. Suppose that, again, we want to compute* $P\left(\phi_d|_{\{D_d:=\bot, D_c:=\bot, D_a:=\top, D_b:=\top\}}\right)$.

The disjunction node indexed with 10 *has score* $0 \cdot 0.4 + 1 \cdot 1 = 1$, *and the one indexed with* 11 *has score* $1 \cdot 0.4 + 0 \cdot 0 = 0.4$. *Continuing in our bottom-up traversal*

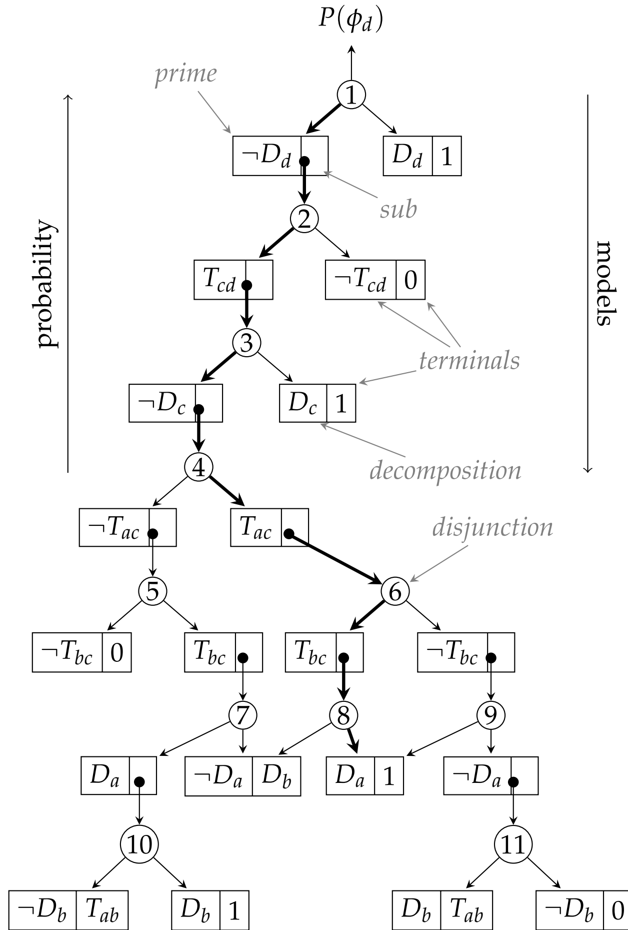


Figure 2.4: *An SDD representation of Equation* 2.10 *with variable order* $D_d \prec T_{cd} \prec D_c \prec T_{ac} \prec T_{bc} \prec D_a \prec D_b \prec T_{ab}$. *Bold arcs represent one of the models of* ϕ_d: $\{D_d := \bot, D_c := \bot, T_{ac} := \top, T_{bc} := \top, D_a := \top\}$. *Probabilities are omitted. Disjunction nodes are indexed for reference.*

and applying Equation 2.12, *disjunction nodes* 6–9 *each have score* 1, *while disjunction node* 5 *has score* 0.1. *Disjunction node* 4 *has score* $0.2 \cdot 0.1 + 0.8 \cdot 1 = 0.82$, *and so does node* 3. *Finally, node* 2 *has score* $0.3 \cdot 0.82 + 0.7 \cdot 0 = 0.246$, *and so does node* 1. *Since node* 1 *is the root of the SDD, we find that* $P\left(\phi_d|_{\{D_d:=\bot, D_c:=\bot, D_a:=\top, D_b:=\top\}}\right) = 0.246$.

2.6 Probabilistic logic programming

As the previous section shows, DDs such as OBDDs and SDDs represent convenient data structures for performing tractable probabilistic inference. This begs the question: how do we go from a mathematical model of a probability distribution to a DD-representation of that distribution?

Note that this question really has two components. First, we need some convenient way to model the probability distribution in a computer. Second, we then need to compile this model into a DD.

To illustrate the first challenge, recall the two example problems described in Section 1.1: spread of influence and power grid reliability. Both of them are formulated on probabilistic networks (where edges exist with a certain probability), and both of them require some form of reasoning about paths in those networks. As a consequence, the resulting probability distributions can become quite complex, since there are often many different paths between two nodes in a real-world network, especially in the case of social networks, which are the types of network that spread of influence problems are formulated on. Additionally, these paths may partially overlap, divert from each other, and maybe even join again later on. This makes the resulting probability distributions quite non-trivial to define.

This is a problem, since we aim to develop solving methods for problems like the spread of influence problem, that are *convenient* to use. We want it to be easy for a user to specify them, and to not require (much) technical knowledge, aiming to democratise the technology that we develop in this dissertation as much as possible, making it as accessible as possible to anyone who needs it.

In this section we discuss a convenient tool for modelling probability distributions: the probabilistic logic programming language ProbLog [52]. This Prolog-based language provides a simple way for users to model probability distributions, and is particularly suited for modelling distributions that arise from probabilistic networks, thus addressing the first challenge. Conveniently, ProbLog has functionality for compiling these distributions into several different DDs built in, thus addressing the second challenge. In the remainder of this section, we give a brief introduction to the logic programming Prolog, which is a precursor to ProbLog, and then to ProbLog itself.

2.6.1 Logic programs

Prolog is a *rule-based*, *logical* programming language that is *declarative* in nature. Declarative programming paradigms are very user-friendly in the following way. *Imperative* programming languages (such as C, C++, Java, Python, Go, etc.) require

the user to specify *how* the computer should solve a problem. Declarative programming languages (like Prolog, Datalog and SQL), on the other hand, only ask the user to specify everything they *know* about the problem, and then simply ask a question. The computer will then figure out how to get to the answer.

Our choice to build on Prolog is motivated by the fact that it is declarative. While a study of how different types of intended users experience modelling problems in Prolog compared to other languages is outside the scope of this work, we are confident that its declarative nature can appeal to a wide range of users. In Section 4.3, we present a language to model constraint optimisation problems by querying databases. Arguably the most-used programming paradigms for these elements are *constraint programming (CP)* for modelling linear and non-linear constraints (see Section 3.3), *mixed integer programming (MIP)* for modelling linear constraint optimisation problems (see Section 3.4) and SQL for querying databases. Since all three of these paradigms are declarative in nature, we expect our target audience to find a declarative programming tool convenient to use and easy to learn. After all, there is probably a good reason that declarative languages are so popular, and due to their popularity, a potential user is likely to be familiar with the declarative programming paradigm.

It is important to us that our tools are easy to use by a wide variety of people, even if they have no previous coding experience, because that helps to democratise computing power and technology. However, there are possible downsides such as added expressive complexity for specifying certain tasks and less speed. At the same time, it is a very common practice in Computer Science to design languages for specific goals, and thus with limited applicability, so these downsides are only a natural consequence of the fact that we design task-specific tools.

Recall the goal that we specified in Section 1.3: to find SCP solving methods with reasonable trade-offs between convenience, generality and speed. Our reason for choosing the declarative ProbLog (probabilistic Prolog) language as a basis for our new SCP modelling language is primarily rooted in the first criterion: convenience. As mentioned above, the basics of Prolog, which provide the user with enough expressibility to model a wide range of problems, are quick and easy to learn. Additionally, it is very convenient for modelling *relations* between entities. In this work, our focus is on SCPs that are formulated on probabilistic networks. Networks are very easy to model in Prolog, because edges can be seen as relations between nodes. To illustrate this, we now first present a few basics about the syntax and semantics of Prolog, and then provide a small example.

Prolog is a *rule-based* programming language. To construct rules, we can use *terms* and *predicates*. Terms are either *constants*, or *variables*. Constant symbols in

Prolog start with a lower case letter, or are a string enclosed in single quotes. Variable symbols are capitalised and can take arbitrary constants as values. Prolog uses *predicates* to express relationships between constants or variables, in the same way as the predicates in first-order logic (Section 2.2.2). The name of a predicate also starts with a lowercase letter, and is followed by comma-separated terms in brackets: the *arguments* of the predicate. Any Prolog predicate can also be negated.

Each rule is of the form: `Head :- Body.`, which means "`Head` is *true* if `Body` is *true* ." The body contains one or more predicates and terms, which are known as *goals*, separated either by "`,`" for conjunction, or "`;`" for disjunction. We also refer to these rules as *clauses*. A special property of the rules in Prolog is that the head only contains one predicate, which makes these clauses *Horn clauses* [81].

In addition to rules, there are also facts. An example is: `dir(alexa,behrouz).`, which is a shorthand for `dir(alexa,behrouz) :- true.`, and means that there is a directed relationship between the constant `alexa` and the constant `behrouz`.

The user can use rules and facts to describe everything they know about the problem, and ask questions by specifying *queries* of the form `?- influences(alexa,daniel).`. Queries can be seen as rules without a head, where we ask if the goals in the body are *true*. The Prolog system then uses an inference process called *selective linear definite (SLD) clause resolution*. The resolution process tries to find constants in the rest of the program that can be used to substitute variables such that rules are satisfied (the arguments in the predicates in the head and the body match) and new facts are proven from these rules.

We can now formulate a small Prolog program that describes some relationships between some people.

Program 2.1: *A Prolog program describing influence relationships between four people.*

```
   % Relation facts
1. dir(alexa,behrouz).
2. dir(alexa,claire).
3. dir(behrouz,claire).
4. dir(claire,daniel).

   % Relation rules
5. influences(X,Y) :- dir(X,Y).
6. influences(X,Y) :- dir(Y,X).

   % Query
7. ?- influences(alexa,daniel).
```

Example 2.6.1 (A simple Prolog program)**.** *Consider the small Prolog program in Program 2.1. In lines 1–4, it describes the direct influence relationships between four peo-*

ple who are represented by the constants `alexa`*,* `behrouz`*,* `claire` *and* `daniel`*. Line 6 serves to make the relationships symmetric, and line 7 is a query asking if Alexa influences Daniël. Evaluating this program tells us that she does not, since neither the predicate* `dir(alexa,daniel).` *nor* `dir(daniel,alexa).` *is in the knowledge base of relation facts, meaning that the rules in line 5 and 6 cannot be used to prove that Alexa influences Daniël.*

The reader may have noticed that the fact that Alexa *does not* influence Daniël is proved by *failing to prove* that she *does*. This property, absence of truth meaning negation of truth, is called the *closed-world assumption* and central to the semantics of Prolog. The closed-world assumption is very powerful because it drastically limits the size of program you need to model a problem.

Note that in Program 2.1, lines 1–4 are essentially first-order logic formulae. Using predicates, we could rewrite these lines as $R(a,b) \wedge R(a,c) \wedge R(b,c) \wedge R(c,d)$, where $R(a,b)$ indicates that there is a directed relationship between constants a (Alexa) and b (Behrouz), and analogously for the other predicates and constants. We could replace each predicate by a Boolean variable, and then rewrite these lines as as the *propositional* logic formula $R_{ab} \wedge R_{ac} \wedge R_{bc} \wedge R_{cd}$, where R_{ab} is a Boolean variable that is *true* iff there is a directed relationship from Alexa to Behrouz, and analogously for the other variables. This latter expression is not written in first-order logic, since it does not use any predicates or quantifiers.

Lines 5 and 6, on the other hand, are written in *first-order* logic and cannot be rewritten by replacing predicates with Boolean variables. We can rewrite line 5 as $\forall X, Y. \, (I(X,Y) \vee \neg R(X,Y))$, where X and Y are variables that could be replaced by a, b, c or d, and $I(X,Y)$ is a predicate that is *true* iff X influences Y.

It helps our discussion later on to introduce the concept of *grounding* a logic program. A predicate that does not contain any variables, is called a *ground predicate*. A ground logic program is one in which all predicates are ground by substituting the variables by constants. Note that this corresponds to turning the first-order logic formula that is implied by the program into a propositional formula.

Program 2.2: *Ground relation rules.*

```
5.1. influences(alexa,behrouz).
6.1. influences(behrouz,alexa).
5.2. influences(alexa,claire).
6.2. influences(claire,alexa).
5.3. influences(behrouz,claire).
6.3. influences(claire,behrouz).
5.4. influences(claire,daniel).
6.4. influences(daniel,claire).
```

Example 2.6.2 (Ground Prolog program). *For example, we could ground the program in Program 2.1 by replacing lines 5 and 6 by Program 2.2. Here, we have replaced the `X`s and `Y`s in these lines by combinations of constants that are allowed according to the relation facts in lines 1–4. Note again that `influences(alexa,daniel).` is not in the ground program, and thus we cannot prove that Alexa influences Daniël, as is asked in line 7 of Program 2.1.*

Note that in the example above, we can replace the ground predicates of lines 5.1–6.4 by converting them into Boolean variables, and write these lines as $I_{ab} \wedge I_{ba} \wedge I_{ac} \wedge I_{ca} \wedge I_{bc} \wedge I_{cb} \wedge I_{cd} \wedge I_{dc}$, which is a propositional logic formula.

Grounding an entire Prolog program requires substitution of variables into all possible (combinations of) constants that are allowed by the program. This typically results in not only a large program, but one with many ground facts that are irrelevant for answering the query, as is very clearly shown in Example 2.6.2, where all eight ground relation rules turn out to be irrelevant for answering the query. Therefore, grounding typically happens in a top-down rather than bottom-up manner, such that only those predicates are ground that are needed for answering the query.

A detailed description of Prolog, and of SLD and techniques for efficient grounding, is outside the scope of this work. For more information on the syntax, semantics and inner workings of Prolog, we refer the reader to the literature, *e.g.*, Peter Flach's *Simply Logical: Intelligent Reasoning by Example* [66].

2.6.2 Probabilistic logic programs

The probabilistic logic programming ProbLog [52] is a language for programming probability distributions, built on Prolog. We can turn the Prolog program in Program 2.1 into a ProbLog program by adding probabilities to facts and rules. For example, by replacing lines 1–4 by the following, we can make the influence relationships between people probabilistic:

Program 2.3: *Probabilistic facts.*

```
1. 0.4::dir(alexa,behrouz).
2. 0.8::dir(alexa,claire).
3. 0.1::dir(behrouz,claire).
4. 0.3::dir(claire,daniel).
```

Here, we assume that the associated probabilities are independent of each other. We can think of these probabilistic facts as the (positive) literals in a probability-weighted propositional formula, as described in Section 2.2.3.

We can think of a ProbLog program as one that defines a distribution of 'underlying' Prolog programs, or *possible worlds*, where each probabilistic fact is non-deterministically present in a program that was randomly sampled from this distribution, according to that fact's associated probability. This probability is the one that annotates the corresponding rule or fact. A probabilistic version of the rule in line 5 in Program 2.1 would be, *e.g.*, `0.2::buys(X) :- influences(Y,X), buys(Y).`. This is shorthand for `0.2::buys(alexa) :- influences(alexa,alexa), buys(alexa).`, `0.2::buys(alexa) :- influences(behrouz,alexa), buys(behrouz).`, and so on. Each of these rules has a probability of 0.2 to be included in a Prolog program that is randomly selected from the distribution defined by the ProbLog program. In other words: if there is a way in which the body of the rule can be made *true*, the chance that the head is *true*, is 0.2.

Following the closed-world assumption, we assume that the probability that a fact is true (and thus present in a Prolog program randomly sampled from the distribution) and the probability that the negation of that fact is true (and thus not present in that Prolog program), sum up to one. For example: we assume that the probability that Alexa *does not* influence Behrouz is $1 - 0.4 = 0.6$. Now, instead of asking *if* Alexa influences Daniël, line 7 now asks *with what probability* she does. We also refer to this probability as the *success probability* of the query.

The success probability of a query Q is defined as follows:

$$P(Q \mid \mathscr{P}) = \sum_{v \in \mathscr{P} | v \models Q} P(v), \tag{2.13}$$

where $\mathscr{P}$ is a ProbLog program, v is a *possible world*, *i.e.*, a Prolog program that can be obtained from $\mathscr{P}$ by including all deterministic facts and rules and including a subset of the probabilistic facts and rules, and $P(v)$ is the probability that v is randomly sampled from $\mathscr{P}$. Here, we use the notation $v \in \mathscr{P}$ to indicate that v is a possible world that can be obtained from $\mathscr{P}$, and the notation $v \models Q$ to denote that Q is a *logical consequence* of v. Note that we simply sum the probabilities of all possible worlds in which $Q = \top$. In the example above, there are no world in which the query is *true*, so the probability that Alexa influence Daniël is 0.

Because of the assumption that the probabilities annotating the probabilistic facts are mutually independent, we can follow Sato's distributions semantics [160] and define the probability of a possible world as:

$$P(v) = \prod_{f \in v} P(f), \tag{2.14}$$

where f is a (probabilistic) fact in the *ground* Prolog program v, and $P(f)$ its associated probability, according to $\mathscr{P}$.

Equations 2.13 and 2.14 should look familiar to the reader, as they bear a clear and non-accidental similarity to Equation 2.1, which defines the weighted model count of a literal-weighted propositional formula. Indeed, we can think of ProbLog as a tool to program literal-weighted propositional formulae, whose model counts reflect success probabilities of the associated queries.

This is also reflected in the inner workings of ProbLog. Like Prolog, ProbLog uses SLD resolution in order to provide an answer to the query, which comes in the form of that query's success probability. Conceptually, in doing so, it generates all possible ground logic programs that one could create from the probabilistic logic program, and summarises them into a DD, which can then be used to compute a query's success probability, as described in Section 2.5. That knowledge compilation step helps us to achieve our requirement that SCP solving methods are fast, as well as convenient.

However convenient probability distributions are to model using ProbLog, they still require the user to be smart about *how* they model the problems exactly. As with many problems in Computer Science, we expect the way that we model an SCP to have a large impact on the speed with which we can solve it. We will reflect on this some more in the next chapter.

2.7 Conclusion

In this chapter we discussed topics related to probabilistic inference and how to express probabilistic models and queries using logic and a probabilistic logic programming language. We motivated why, in this work, we chose to model probability distributions using probability-weighted propositional formulae, which we discussed in Section 2.2, along with related topics such as first-order logic and relevant complexity classes. We specifically described how we can apply *weighted model countings (WMCs)* to weighted propositional formulae to compute the probability that our probabilistic model is in a certain configuration.

In order to put the (sub-)problems described in this dissertation into context, we also gave a high-level overview of the different probabilistic inference tasks known in the probabilistic reasoning literature. Specifically, we discussed the $\mathcal{NP}$-complete task of determining the most likely truth evaluation of each of a set of queries, given a truth value assignment to the complementary set of variables in the probabilistic model. The decision version of this max-inference task, where we do not ask what the most likely truth evaluation is, but rather if a given assignment is more likely than a certain threshold value, is relevant to this work, as it can be seen as asking if a specific strategy σ satisfies Equation 1.1.

We then discussed the $\#\mathcal{P}$-complete sum-inference task, which is to compute the success probability of a query, such as the ones described in Section 2.6.2. Finally, we discussed the $\mathcal{NP}^{PP}$-complete mixed-inference task that aims to find an assignment to a subset of variables, given truth assignments to another, disjoint set, and without knowing the truth values of the variables in the complement of these two sets. Provided we can encode the constraints into the probabilistic model, we can cast SCPs as instances of MMAP.

We then argued that knowledge compilation techniques can help us achieve our goal of developing exact SCP solving methods that strike a reasonable balance between convenience, generality and speed. Specifically, we argued that DD representations of probability distributions can be compact and can be used for tractable probabilistic inference. We closed this chapter with a description of probabilistic logic programming language ProbLog, a tool that not only allows us to conveniently model probability distributions that arise from the probabilistic networks on which the SCPs that we study in this work are formulated, but also provides support for converting the resulting probabilistic logic programs into DDs for fast inference.

The focus of this chapter was on the probabilistic reasoning part of SCP solving. In the next chapter, we focus on the constraint optimisation part of SCP solving. Together, the techniques and frameworks described in this chapter and the next will help us develop SCP solving tools in Part II of this dissertation.