On Sentential Decision Diagrams, and their use as a query language for Weighted Model Integration

Anton R. Fuxjaeger

4th Year Project Report
Artificial Intelligence and Mathematics
School of Informatics
University of Edinburgh

2018

Abstract

Graphical Models provide a formalism to represent probability distributions in graphical form, allowing us to use well known results from graph theory to derive properties of such models. Sentential Decision Diagrams (SDD) as such a representation can be used to represent a propositional Knowledge Base as well a Bayesian Belief Networks, but at the same time have been shown to hold desirable properties that other representations such as Sum-Product Networks lack. We will give an in depth review of SDDs, in addition to presenting a linear Model Counting algorithm as well as a polytime Model Enumeration algorithm for such graphical representations. We will then go on to investigate Weighted Model Integration (WMI) as a means of doing probabilistic inference using SDDs as the underlying querying language. A pipeline implemented in python will be presented, to calculate the Model Integration (sharpSMT) as well as the WMI of a given $\mathcal{LRA}$ Knowledge Base.

Acknowledgements

I wish to thank, first and foremost, my Project supervisor Dr. Vaishak Belle for his continuous support and help on this project. His insights and guidance on problems I encountered proved to be indispensable.

Furthermore I want to address a special thank you to my parent, Gerald and Maria-Luise Fuxjaeger for their unwavering support throughout my academic career. Their financial and emotional support throughout the years, not only enabled, but also encouraged me to pursue my interests into new fields and countries, and for that i am deeply grateful.

Finally I want to thank Samuel Kolb for letting me use his data set, that not only made it possible for me to test by algorithm, but also helped develop it.

Table of Contents

Chapter 1

Introduction

Over the past few years we have seen an increased interest in tractable circuit representations [Bekker et al., 2015, Liang et al., 2017, Poon and Domingos, 2011, Kisa et al., 2014, Poon and Domingos, 2011]. That is graphical networks representing probability distributions, that support efficient querying. The formulation of the network polynomial [Darwiche, 2003] and the discovery of local structure, led the research community to arithmetic circuits [Poon and Domingos, 2011], a very powerful tractable circuit representation. In this paper however we were interested in a strict subset of such representations called Sentential Decision Diagrams (SDD). SDD are furthermore a strict superset of Ordered Binary Decision Diagrams, a property that was shown by [Darwiche, 2011], and implies that they support polytime querying such as Model Counting (MC) and Model Enumeration (ME). Because of SDDs having such desirable properties we have seen a number of papers on this particular representation of Propositional Knowledge Bases (KB), including efficient algorithms for learning the structure from data directly [Bekker et al., 2015, Liang et al., 2017] and thus learning the structure of the underlying Markov networks. SDD are however strictly boolean in their formulation, meaning that all the advantages they bring only apply to probabilistic networks in boolean domains. This aspect of the representation is something we wanted to challenge and investigate in this paper.

The idea of extending a graphical model to hybrid domains is by no means new in the Artificial Intelligence community, with well known examples such as XADDs [Sanner et al., 2012]. Generalizing in such a way, that we can represent a mixture of boolean and real valued variables over a given SMT background theory is particularly interesting for applications such as symbolic dynamic programming, health monitoring systems, robotics, planing, human level conception learning. Generally speaking, the use cases compose of applications and situations where tractable inference is of an essence and the universe the problem is formulated in, is hybrid by design.

At the same time, one of the leading mechanisms for probabilistic inference in graphical models over boolean domains, Weighted Model Counting, has been lifted to the more general formulation of Weighted Model Integration (WMI) by [Belle et al., 2015]. WMI, a strict generalization of WMC, made it possible to do exact probabilistic inference in hybrid domains where the queried KB is defined with some background theory in LRA (Linear Arithmetic over the Rationals, a subset of Satisfiability Modulo Theories). Due to being incredibly expressive this formulation has been studied further [Morettin et al., 2017], trying to improve the performance in order to make it more feasible in practice.

Within this paper we present a in depth study of SDDs, proposing different algorithms to perform queries on propositional KBs represented in graphical form as SDDs. We will further present research in combining predicate abstraction, SDDs and WMI in order to perform efficient exact probabilistic inference in hybrid domains. This is done by introducing a pipeline written in python that works in conjunction with the SDD Library published by the Automated Reasoning Group at UCLA. While the results obtained are very promising when it comes to performance, we still have to impose certain restrictions on the input such that only a subset of the LRA theories are considered in this paper.

When it comes to the outline of this paper, we will first present a research review on graphical networks as a means of representing probability distributions, tractable circuit representations and methods for probabilistic inference in such networks in Chapter 2. We will then go on to demonstrate different tractable algorithms for querying SDDs such as Model Counting (MC) and Model Enumeration (ME) providing a proof of concept for the results proved by [Darwiche, 2011, Darwiche and Marquis, 2002] in Chapter 3. Finally in Chapter 4 the pipeline for doing Model Integration (MI) and WMI will be presented, using predicate abstraction and SDDs as the underlying querying language. Using this pipeline we will then demonstrate some results using this pipeline for doing exact probabilistic inference in hybrid domains.

Chapter 2

Background

2.1 Notational Preliminaries

Throughout this paper upper case letters $B \in \mathcal{B}$ will refer to Boolean random variables(RV), while upper case letters $X \in \mathcal{X}$ will denote continuous RV's. Furthermore lower case letters (eg. x, b) will represent instantiation of variables of the same name (eg. B, X respectively). Bold letters then indicate sets of random variables, such that $\mathbf{X}, \mathbf{B}$ is a set of variables and $\mathbf{x}, \mathbf{b}$ is a set of instantiations.
A boolean function $f(\mathbf{X})$ is then a function, taking a set of variables $\mathbf{X}$ and maps each instantiation $\mathbf{x}$ to a value in $\{0, 1\}$. Furthermore a model of a given KB is a finite set of instantiations of variables of size equal to the number of variables in the KB.

Abbreviations:
RV - Random Variable
WMC - Weighted Model Counting
WMI - Weighted Model Integration
SDD - Sentential Decision Diagram
SMT - Satisfiability Modulo Theories SAT - Boolean Satisfiability Problem
MC - Model Counting
ME - Model Enumeration
BN - Bayesian Networks
OBDD - Ordered Binary Decision Diagrams
CNF - Conjunctive Normal Form

2.2 Graphical Models

2.2.1 Bayesian Networks

A Bayesian or belief network $\mathcal{N}$ over boolean Random Variables (RV) $\mathbf{B}$, is a probabilistic graphical model that represents a set $\mathbf{B}$ and conditional dependencies by means of a directed acyclic Graph G in addition to a conditional probability function w for each variable $B \in \mathbf{B}$. Each node $n \in N$ of the Graph $G = (N, E)$ represents a random variable, while each (directed) edge $(n_1, n_2) = e \in E, s.t. n_1, n_2 \in N$ represents a conditional dependency from n_1 to n_2. In other words, a directed edge from variable n_1 to n_2, indicates that the RV corresponding to n_2 is conditionally dependent on n_1. Furthermore each variable $B \in \mathbf{B}$ has a conditional probability function $\theta_{B|B\mathbf{U}}$ defined in terms of its parent nodes $B\mathbf{U}$ in the graph. When dealing with Boolean RVs, these functions are usually defined in terms of tables, where the number of entries is 2^{c+1} whee c denotes the number of parent variables, called conditional probability tables. In such a case, the function $\theta_{B|B\mathbf{U}}$ is described as a piecewise constant function, assigning network parameter values $\theta_{b|\mathbf{u}}$ to each instantiation $B|B\mathbf{U}$ indicated by a row of the joint probability table. An example of such a network is given by Figure 2.1. Here $\mathbf{B} = \{S, R, W\}$, where S,R,W are boolean RV that represent events sprinkler being switched on, its raining outside, and the grass in the garden being wet respectively.

The probability of an instantiation $\mathbf{b}$ in a Bayesian Network, then simply the product of all network parameters $\theta_{b|\mathbf{u}}$, where $b\mathbf{u}$ is consistent with $\mathbf{b}$.

Theorem 1. *Probability in Bayesian Networks*

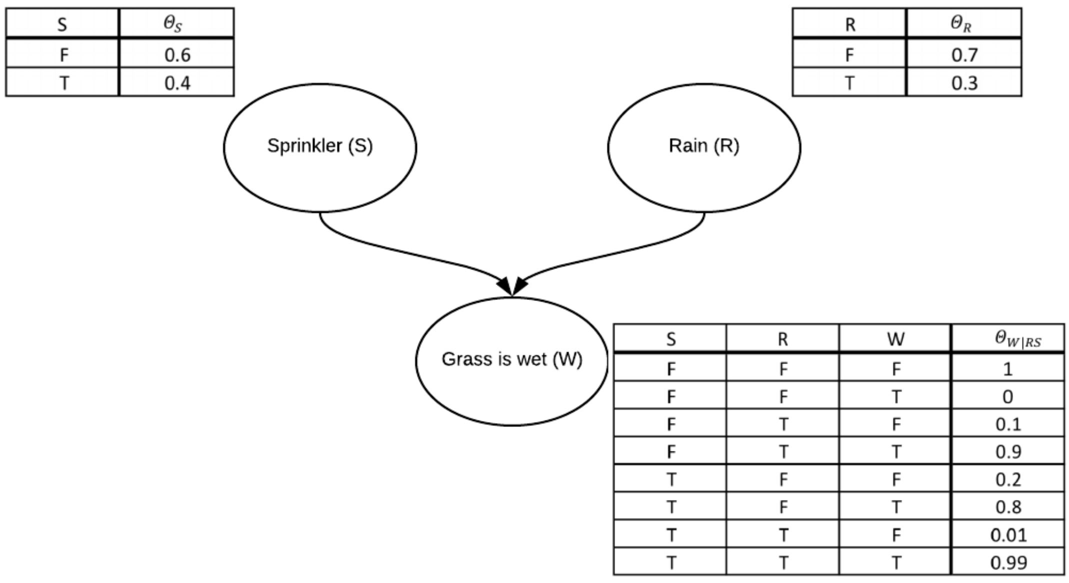
$$Pr(\mathbf{b}) = \prod_{b\mathbf{u} \sim \mathbf{b}} \theta_{b|\mathbf{u}}$$

where $\sim$ is used to denote that instantiation $b\mathbf{u}$ and $\mathbf{b}$ agree on values of their common variables.

Example 1. Consider for example, the probability of it raining outside, given you don't know anything else (and don't have a window). $P(R = True) = 0.3$ as we can read of from the table in Figure 2.1.

2.2.2 The Network Polynomial

The concept of the networks polynomial is, in short a representation of the probability distribution of a given Bayesian Network in the form of a unique polynomial. It has been proven [Darwiche, 2003] that the polynomial evaluated for any evidence (instantiation of variables) is equivalent to the Probability of this evidence. This in combination with partial derivatives allowed us to express a large number of probabilistic queries in the from of the network polynomial. However, this multivariate polynomial where each variable has degree one, has an exponential number of terms (2^c where c denotes the number of variables, to be precise), one term for each instantiation of the network



S	θ_S
F	0.6
T	0.4

R	θ_R
F	0.7
T	0.3

S	R	W	$\theta_{W\|RS}$
F	F	F	1
F	F	T	0
F	T	F	0.1
F	T	T	0.9
T	F	F	0.2
T	F	T	0.8
T	T	F	0.01
T	T	T	0.99

Figure 2.1: Simple Bayesian Network

variables. Formally the Network Polynomial is defined as follows for Bayesian Network N:

Evidence indicators: For each network variable B, we have a set of evidence indicators λ_b .
Network parameters: For each network family $B\mathbf{U}$, we have a set of parameters $\theta_{b|\mathbf{u}}$.

Definition 1. [Darwiche, 2003] Let $\mathcal{N}$ be a Bayesian network over variables $\mathbf{B}$, and let $\mathbf{U}$ denote the parents of variable B in the network. The polynomial of network $\mathcal{N}$ is defined as follows:

$$f = \sum_{\mathbf{b}} \prod_{b\mathbf{u}\sim\mathbf{b}} \lambda_b \theta_{b|\mathbf{u}}$$

The outer sum in the above definition ranges over all instantiations b of the network variables. For each instantiation $\mathbf{b}$, the inner product ranges over all instantiations of families $b\mathbf{u}$ that are compatible with b. The polynomial f of Bayesian network N represents the probability distribution $\mathbf{Pr}$ of N in the following sense; For any piece of evidence $\mathbf{e}$, which is an instantiation of some variables $\mathbf{E}$ in the network, we can evaluate the polynomial f so it returns the probability of $\mathbf{e}$, *Pr(e)* - [Darwiche, 2003]. Coming back to our example in Figure-2.1, the corresponding Network polynomial f would be given by:

$$\begin{aligned}
f =&\lambda_s\lambda_r\lambda_w\theta_s\theta_r\theta_{w|sr} + \lambda_s\lambda_r\lambda_{\bar{w}}\theta_s\theta_r\theta_{\bar{w}|sr} \\
&+ \lambda_s\lambda_{\bar{r}}\lambda_w\theta_s\theta_{\bar{r}}\theta_{w|s\bar{r}} + \lambda_s\lambda_{\bar{r}}\lambda_{\bar{w}}\theta_s\theta_{\bar{r}}\theta_{\bar{w}|s\bar{r}} \\
&+ \lambda_{\bar{s}}\lambda_r\lambda_w\theta_{\bar{s}}\theta_r\theta_{w|\bar{s}r} + \lambda_{\bar{s}}\lambda_r\lambda_{\bar{w}}\theta_{\bar{s}}\theta_r\theta_{\bar{w}|\bar{s}r} \\
&+ \lambda_{\bar{s}}\lambda_{\bar{r}}\lambda_w\theta_{\bar{s}}\theta_{\bar{r}}\theta_{w|\bar{s}\bar{r}} + \lambda_{\bar{s}}\lambda_{\bar{r}}\lambda_{\bar{w}}\theta_{\bar{s}}\theta_{\bar{r}}\theta_{\bar{w}|\bar{s}\bar{r}} \\
=&\lambda_s\lambda_r\lambda_w 0.4 * 0.3 * 0.99 + \lambda_s\lambda_r\lambda_{\bar{w}} 0.4 * 0.30.01 \\
&+ \lambda_s\lambda_{\bar{r}}\lambda_w 0.4 * 0.7 * 0.8 + \lambda_s\lambda_{\bar{r}}\lambda_{\bar{w}} 0.4 * 0.7 * 0.2 \\
&+ \lambda_{\bar{s}}\lambda_r\lambda_w 0.6 * 0.3 * 0.9 + \lambda_{\bar{s}}\lambda_r\lambda_{\bar{w}} 0.6 * 0.3 * 0.1 \\
&+ \lambda_{\bar{s}}\lambda_{\bar{r}}\lambda_w 0.6 * 0.7 * 0 + \lambda_{\bar{s}}\lambda_{\bar{r}}\lambda_{\bar{w}} 0.6 * 0.7 * 1
\end{aligned}$$

Definition 2. [Darwiche, 2003] The *value* of network polynomial f at evidence $\mathbf{e}$, denoted by $f(\mathbf{e})$, is the result of replacing each evidence indicator λ_b in f with 1 if b is consistent with $\mathbf{e}$, and with 0 otherwise.

Theorem 2. *[Darwiche, 2003] Let $\mathcal{N}$ be a Bayesian network representing probability distribution Pr and having network polynomial f. For any evidence (instantiation of variables) $\mathbf{e}$, we have f($\mathbf{e}$) = Pr($\mathbf{e}$).*

As this shows evaluating the network polynomial f at evidence $\mathbf{e}$ for a given network $\mathcal{N}$ is the the probability of that evidence. This made it possible for the research community to think and deal with Bayesian networks purely in the form a multivariate polynomial where each variable is of degree one. However, due to the exponential size of this polynomial, representing this function in a different more compact form is desirable. One such representation is an arithmetic circuit, which we will talk about in more detail in Section 2.4.

2.2.3 The Discrete, Continuous and Hybrid Case with Markov random fields

So far in this paper, when we referred to a random variable B was assumed to be a Boolean Random Variable, only taking two values: true and false. Now we see that the definition of Bayesian Networks is by no mean restricted to such variables. The formulation can easily be extended to Discrete Random Variables, that is RV that take values from a finite set. An example of such a variable is given by the variable C indicating the color of an object, where the color can red, green or blue. Then C has three instantiations corresponding to the three colours it can indicate. For discrete RV's the above definitions and results can be easily extended, and thus we will continue to deal with boolean RV's without loss of generality.

When it comes to continuous RV's, that is variables $X \in \mathcal{X}$ that take values form $\mathbb{R}$, the real numbers, things change a bit but alternations of the same results still hold. Consider a probabilistic model, defined on boolean RV's $B \in \mathcal{B}$ and real-valued continuous RM's $X \in \mathcal{X}$ such that $X \in \mathbb{R}$. Now let $(\mathbf{B}, \mathbf{X}) = (B_1, B_2, ..., B_m, X_1, X_2, ...X_n)$ be one element in the probability space $\{0,1\}^m * \mathbb{R}^n$, denoting a particular assignment to the values in the respective domains. As with Bayesian Networks we can now use a similar graphical model, Markov Networks to define a joint density function of those variables compactly. A Markov Network, in the way we are going to use it is an undirected graphical model, where the notes are Boolean functions taking real and boolean variables, and the edges are logical implications. Furthermore, the network has to satisfy the Markov property, a property referring to the memorylessness property of probability distribution. By [Koller and Friedman, 2009] we then know that we can compactly factorize the joint density function in terms of the cliques of the graph:

$$Pr(\mathbf{B},\mathbf{X}) = \frac{1}{Z} * \prod_k \phi_k(\mathbf{B}_k,\mathbf{X}_k)$$

where $\mathbf{B}_k$ and $\mathbf{X}_k$ are those random variables participating int the kth clique, and $\phi_k(.,.)$ is a non-negative, real-valued potential function. Here ϕ_k is not necessarily denoting a probability, and so Z is used a normalizing constant, also called the partition function defined as:

$$Z = \sum_{B_1} \cdots \sum_{B_m} \int_{X_1} \cdots \int_{X_n} [\prod_k \phi_k(\mathbf{B}_k,\mathbf{X}_k)] dX$$

[Belle et al., 2015].
Similar to the Bayesian Network, the partition function of a Markov Network is full representation of the network, that is however untraceable in practice.

2.3 Inference in Graphical Models

2.3.1 On Probabilistic Inference

Inference in a Bayesian network $\mathcal{N}$ is the task of computing the probability of a given event occurring based on the underlying probability distribution of the Network $\mathcal{N}$. Considering, for example, the Bayesian Network in Figure 2.1, we might be interested in the probability of the grass outside being wet, formally written as $Pr_{\mathcal{N}}(W = w)$. Another more interesting query would be the probability of Rain, given that the grass is wet; written as a conditional Probability $Pr_{\mathcal{N}}(R = r|W = w)$. Because of the expressive power of Bayesian Networks, solving such queries efficiently has become an important field in Computer Science and Probabilistic Reasoning (TODO: EXAMPELS) However desirable such a computation may be, computing such probabilities is still very difficult or NP-Hard, see Section 2.5. For example the brute force approach, where we would create a single table representing all models (a model being a complete instantiation of the variables) of the network with the corresponding probabilities or Network Parameters. This approach is usually infeasible in practice as the size of the table grows exponentially in the number of variables of the network (2^n where n denotes the number of variables). In order to address this problem a number of algorithms have been proposed, that improve the computation time considerably, but are all exponential in the worst case analysis. The most notable of such algorithms include elimination, conditioning and tree-clustering [Chavira and Darwiche, 2008]. All of theses algorithms can compute the Probability $Pr(\mathbf{e})$ in $O(n * exp(w))$ time and space, where $n = |N|$ is the number of nodes of Network $\mathcal{N}$ and w is the bounded

treewidth [Bodlaender and Möhring, 1993].
The inference algorithm we investigated, and will talk about in the following Sections is based on the Weighted Model Counting formulation.

2.3.2 Weighted Model Counting

In this section we will talk about the WMC formulation, its relation to the network Polynomial and how a given Bayesian Network inference problem can be reduced to WMC.
In short the idea is to compile a given Believe Network $\mathcal{N}$ with variables $X = N$ into a Propositional Knowledge base Δ representing the structure, and a weight function $w(x) : \mathbf{x} -> [0,1]$ mapping each variable instantiation to a Probability. The formulation then reproduces the evaluation of the network polynomial for some evidence $\mathbf{e}$.

2.3.2.1 Model Counting Formulation

Given a boolean function f, Model Counting (MC) is the problem of finding all models m such that $f(m) \equiv True$. This is a strict generalization of the Boolean Satisfiability Problem, see Section 2.5 [Biere et al., 2009]. Furthermore, in the language of logic, the models of a given propositional KB Δ are denoted by $\Delta \models m$.

2.3.2.2 Weighed Model Counting Formulation

First introduced by [Chavira and Darwiche, 2008] Weighed Model Counting(WMC) is a strict generalization of Model Counting [Biere et al., 2009]. In WMC, each model of Δ has an associated weight, and we are interested in computing the sum of the weights that correspond to models that satisfy Δ.

In order to create an instance of the WMC problem, a propositional Knowledge base Δ over literals $\mathcal{L}$ is needed as well a weight function $weight : \mathcal{L} \rightarrow \mathbb{R}^{\geq 0}$ mapping literal of Δ, to a weight. We can then use the literals of a given model m to define the weight of that model as well as the Weighted Model Count as follows:

Definition 3. Given a propositional Knowledge base Δ over literals $\mathcal{L}$ and weight function $weight : \mathcal{L} \rightarrow \mathbb{R}^{\geq 0}$ mapping literal instantiations of Δ to non-negative weights, we define the weight of a model as:

$$WEIGHT(m, weight) = \prod_{l \in m} weight(l)$$

Further we define the weighted model count (WMC) as:

$$WMC(\Delta, weigth) = \sum_{m \models \Delta} WEIGHT(m, weight)$$

Based on the above definition, it can be observed that if $weight(l) = 1$ for all l, the weight $WEIGHT(m, weight)$ is always 1, and thus we are just counting the number of satisfying models, or computing the model count.

2.3.2.3 Probabilistic Inference in Bayesian Networks by Weighted Model Counting

The following results will demonstrate that we can use WMC to calculate probabilities of a given Bayesian Network.

Theorem 3. *[Chavira and Darwiche, 2008] For a given Bayesian Network $\mathcal{N}$ over variable $\mathbf{X}$, we can construct a Propositional Knowledge base Δ and weight function weight such that*

$$Pr_{\mathcal{N}}(q|\mathbf{e}) = \frac{WMC(\Delta \wedge q \wedge \mathbf{e}, weight)}{WMC(\Delta \wedge \mathbf{e}, weight)}$$

for some evidence $\mathbf{e}$ and query q, with $\mathbf{e}, q \in \mathbf{x}$ (or $\mathbf{e}, q \subset \Delta$).

Proof. First we compiling the Structure, that is the variable dependencies of $\mathcal{N}$ as a Propositional Knowledge base. Then we define literals $L \in \mathcal{L}$ for every network parameter $\theta_{x|\mathbf{u}}$ such that $L = x \wedge \bigwedge_{u' \in \mathbf{u}} u'$. Finally we define the weight function as $w(L) = \theta_{x|\mathbf{u}}$. Now computing the weighted model count $WMC(\Delta, w)$ of Δ and w is in one-to-one correspondence with computing the Network polynomial $f_{\mathcal{N}}$ of the Network $\mathcal{N}$, and furthermore

$$Pr_{\mathcal{N}}(q|\mathbf{e}) = \frac{Pr_{\mathcal{N}}(q \wedge \mathbf{e})}{Pr_{\mathcal{N}}(\mathbf{e})} = \frac{f_{\mathcal{N}}(q, \mathbf{e})}{f_{\mathcal{N}}(\mathbf{e})} = \frac{WMC(\Delta \wedge q \wedge \mathbf{e}, weight)}{WMC(\Delta \wedge \mathbf{e}, weight)}$$

□

The reduction used to compile a given Bayesian Network $\mathcal{N}$ into a Propositional Knowledge Base Δ is explained in more detail by [Chavira and Darwiche, 2008], but is relatively straight forward in general.

Example 2. Coming back to our example Network in Figure 2.1, with variable $\mathbf{X} = \{S, R, W\}$, the corresponding propositional Knowledge Base (in CNF) is given by:

$$\Delta = (S \rightarrow W) \wedge (R \rightarrow W) = (\neg S \vee W) \wedge (\neg R \vee W)$$

While the weight function would be defined as:

$$weight(L) = \begin{cases} \theta_s & = 0.4 & \text{if } L == s \\ \theta_{\bar{s}} & = 0.6 & \text{if } L == \bar{s} \\ \theta_r & = 0.3 & \text{if } L == r \\ \theta_{\bar{r}} & = 0.7 & \text{if } L == \bar{r} \\ \theta_{w|sr} & = 0.99 & \text{if } L == w \land s \land r \\ \theta_{w|s\bar{r}} & = 0.8 & \text{if } L == w \land s \land \bar{r} \\ \theta_{w|\bar{s}r} & = 0.9 & \text{if } L == w \land \bar{s} \land r \\ \theta_{w|\bar{s}\bar{r}} & = 0 & \text{if } L == w \land \bar{s} \land \bar{r} \\ \theta_{\bar{w}|sr} & = 0.99 & \text{if } L == \bar{w} \land s \land r \\ \theta_{\bar{w}|s\bar{r}} & = 0.2 & \text{if } L == \bar{w} \land s \land \bar{r} \\ \theta_{\bar{w}|\bar{s}r} & = 0.1 & \text{if } L == \bar{w} \land \bar{s} \land r \\ \theta_{\bar{w}|\bar{s}\bar{r}} & = 1 & \text{if } L == \bar{w} \land \bar{s} \land \bar{r} \\ 1 & & \text{otherwise} \end{cases}$$

Then $Pr_{\mathcal{N}}(R = r|W = w) = \frac{WMC(\Delta \land r \land w, weigth)}{WMC(\Delta \land w, weight)} = \frac{\sum_{m \models \Delta \land r \land w} WEIGHT(m, weight)}{\sum_{m \models \Delta \land w} WEIGHT(m, weight)}$
$= \frac{\theta_{\bar{s}}\theta_r\theta_{w|\bar{s}r} + \theta_s\theta_r\theta_{w|sr}}{\theta_{\bar{s}}\theta_r\theta_{w|\bar{s}r} + \theta_s\theta_r\theta_{w|sr} + \lambda_{\bar{s}}\lambda_{\bar{r}}\lambda_w + \lambda_s\lambda_{\bar{r}}\lambda_w} = \frac{0.162 + 0.1188}{0.162 + 0.1188 + 0.224 + 0} = 0.56$

Once again, [Chavira and Darwiche, 2008] showed that all previous results apply for discrete Random Variables as well.

2.3.3 Weighted Model Integration

While Weighted Model Counting is very powerful as an inference tool in discrete and boolean domains, it suffers from the inherent limitation of only admitting inference in discrete probability distributions. This is due to its underlying theory in enumerating all the models (or expanding the complete Network Polynomial), which is exponential in the number of variables, but still finite and countable in the discrete case. For continuous case these properties are not true, and therefore the network polynomial does not exist in the same sense. That being said [Belle et al., 2015] introduced WMI, a strict generalization of WMC for hybrid domains, with the main idea of annotating a logical SMT theory with rational and Boolean variables.

Definition 4. [Belle et al., 2015] Suppose Δ is a SMT theory over Boolean and rational variables $\mathcal{B}$ and $\mathcal{X}$, and literals $\mathcal{L}$. Suppose $weight : \mathcal{L} \to EXPR(\mathcal{X})$, where $EXPR(\mathcal{X})$ are expressions over $\mathcal{X}$. Then the weighted model Integration (WMI) is defined as:

$$WMI(\Delta, weight) = \sum_{m \models \Delta^-} VOL(m, weight)$$

where:

$$VOL(m, weight) = \int_{\{l^+ : l \in m\}} WEIGHT(m, weight) d\mathcal{X}$$

and $WEIGHT$ is defined as described in Def 3

Intuitively the WMI of a SMT theory Δ is defined in terms of the models of its propositional abstraction Δ^-. For each such model we compute its volume, that is, we integrate the $WEIGHT$-values of the literals that are true in the model. The interval of the integral is defined in terms of the refinement of the literal. The w-function is to be seen as mapping an expression e to its density function, which is usually another expression mentioning the variables in e.

2.3.3.1 Probabilistic Inference in Markov Networks by Weighted Model Integration

Theorem 4. *[Belle et al., 2015] Let $\mathcal{N}$ be a Markov network over the Boolean and real-valued random variables $\mathcal{B}$ and $\mathcal{X}$ and potentials $\{\phi_1, \cdots, \phi_k\}$. Let Δ and w be the corresponding encodings. Then for any $q, e \in \mathcal{B} \cup \mathcal{X}$,*

$$Pr_{\mathcal{N}}(q|e) = \frac{WMI(\Delta \wedge q \wedge \mathbf{e}, w)}{WMI(\Delta \wedge \mathbf{e}, w)}$$

Two things are worth noting at this point; Firstly, if Δ is a formula in propositional logic over literals $\mathcal{L}$ and $w : \mathcal{L} \rightarrow \mathbb{R}^{\geq 0}$, then $WMI(\Delta, w) = WMC(\Delta, w)$.

Example 3. [Belle et al., 2015] Suppose Δ is the following formula:

$$B \vee (0 \leq X \leq 10) \text{ where } B_1 \in \mathcal{B} \text{ and } X \in \mathcal{X}$$

For weights, let $w(b_1) = .1$, $w(\neg b_1) = 2x$, $w(b_2) = 1$ and $w(\neg b_2) = 0$, where b_2 is the propositional abstraction of $(0 \leq X \leq 10)$. There are now 3 models of Δ^-:

- $m = \{b_1, \neg b_2\}$: since $w(\neg b_2) = 0$, by definition we have $WEIGHT(m, w) = 0$ and so $VOL(m, w) = 0$
- $m = \{\neg b_1, b_2\}$: $VOL(m, w) = \int_{0 \leq X \leq 10} 2x dx = [x^2]_0^1 0 = 100$
- $m = \{b_1, b_2\}$: $VOL(m, w) = \int_{0 \leq X \leq 10} .1 dx = [.1 * x]_0^1 0 = 1$

Thus, $WMI(\Delta, w) = 100 + 1 = 101$.
Now suppose that we are interested in the probability of the query $X \leq 3$ given that $\neg b_1$ is observed. Suppose B_3 is the abstraction of $X \leq 3$. First, $WMI(\Delta \wedge \neg B_1, w)$ corresponds to the weight of a single interpretation, that of item 2, yielding a value of 100. Next, $WMI(\Delta \wedge \neg b_1 \wedge X \leq 3, w) = WMI(\Delta \wedge \neg b_1 \wedge b_3, w)$ also corresponds to the weight of a single interpretation $m = \{\neg b_1, b_2, b_3\}$, an extension to that in item 2. In this case:

$$VOL(m, w) = \int_{(0 \leq X \leq 10) \wedge (X \leq 3)} 2x dx = [x^2]_0^3 = 9$$

Therefore, the conditional probability is $\frac{9}{100} = 0.09$.

2.4 Tractable Circuit Representations

So far we have talked about different graphical models for representing probability distributions in discrete and hybrid domains (Bayesian and Markov Networks respectively). In General we define such network in compact form, as a normalized product of factors:

Definition 5. Let $\mathcal{N}$ be a probability distribution in a hybrid domain, then the Partition function used to evaluate the probability of a specific outcome x is defined by:

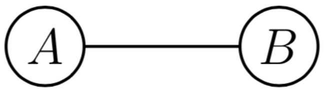
$$P(X=x) = \frac{1}{Z} * \prod_{k} \phi_k(x_k),$$

where $x \in \chi$ is a d-dimensional vector, each *potential* ϕ_k is a function of a subset $x_{\{k\}}$ of the variables (its scope), and $Z = \sum_{x\in\chi} \prod_k \phi_k(x_{\{k\}})$ is the *partition function*. [Darwiche, 2003]

This quite fundamental result by [Darwiche, 2003] was taken further by [Poon and Domingos, 2011], who came up with a way of representing the partition function in graphical from by the formulation of the Sum-Product Network.

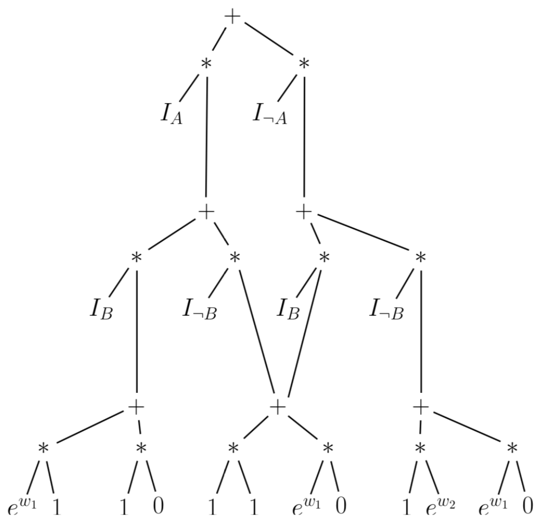
2.4.1 Sum-Product Networks

Sum-Product Networks(SPN) were introduced by [Poon and Domingos, 2011] as a new deep architecture for probabilistic modeling. Building on the idea of the network polynomial as introduced by [Darwiche, 2003], SPN's are directed acyclic graphs of sums and products that efficiently compute partition functions and marginals of high-dimensional distributions. Generally speaking, the SPN represents a graphical representation of the partition function Z. In comparison to other models SPN's prove to be exponentially more compact than hierarchical mixture models as well as junction tree models. Furthermore SPN's are far more general in their design, such that they allow discrete variables as well as continuous (straightforward as long as computing the max and argmax of p(x) is easy) ones. [Darwiche, 2003] even showed in their paper, that computing the probability is linear in its circuit size. It is argued [Darwiche, 2003] that SPN's are theoretically more well-formed, at least on order of magnitude faster in both learning and inference, and much more effectively in learning compared to other models. Figure 2.2 shows an example of a Markov Network and the corresponding SPN.



weight	feature	parameter variable
w_1	$A \wedge B$	P_1
w_2	$\neg A \wedge \neg B$	P_2

(a) Markov Network



(b) Arithmetic Circuit (SPN)

Figure 2.2: A Markov network over variables A,B, and its tractable AC representation [Bekker et al., 2015]

2.4.2 Sentential Decision Diagram

Sentential Decision Diagrams (SDD), first introduced by [Darwiche, 2011] are graphical representations of propositional knowledge bases. SDD's are shown to be a strict subset of deterministic, decomposable negation normal form (d-DNNF), a poplar representation for probabilistic reasoning applications [Chavira and Darwiche, 2008], due to their desirable properties. Decomposability and determinism ensure tractable probabilistic (and logical) inference, as they enable MAP queries in Markov networks. SDDs however satisfy two even stronger properties found in Ordinary Binary Decision Diagrams (OBDD), namely structured decomposability and strong determinism. Thus, they are strict supersets of OBDDs as well, inheriting their key properties; canonicity and a polynomial time support for Boolean combination. Finally SDD's also come with an upper bound on their size in terms of tree-width.

2.4.2.1 Structured Decomposability, Strong Determinism and Vtrees:

Consider the boolean function $f(\mathbf{Z})$ such that $\mathbf{Z} = \mathbf{X} \sqcup \mathbf{Y}, \mathbf{X} \cap \mathbf{Y} = \emptyset$. Now if p_i and s_i are further boolean functions and $f = (p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})) \vee ... \vee (p_n(\mathbf{X}) \wedge s_n(\mathbf{Y}))$, then $\{(p_1, s_1), ..., (p_n, s_n)\}$ is called an $(\mathbf{X}, \mathbf{Y})$-decomposition of f since it allows us to express f purely in terms of functions on $\mathbf{X}$ and $\mathbf{Y}$ [Pipatsrisawat and Darwiche, 2010]. Formally, a conjunction is decomposable if each pair of its conjuncts share no variables. Now if $p_i \wedge p_j \equiv false$ for $i \neq j$ the decomposition is considered to be strongly deterministic. In such a case the structures pair (p_i, s_i) is called an *element* of the decomposition and p_i, s_i the elements prime and sub respectively [Darwiche, 2011]. But the decomposition used by SDDs has structural properties as well, that build on the notion of the vtrees [Pipatsrisawat and Darwiche, 2010].

Definition 6. A vtree for a set of variables **Z** is a full, rooted binary tree whose leaves are in one-to-one correspondence with the variables in **Z**.

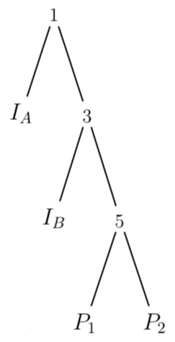


Figure 2.3: Vtree for the Markov network in figure 2.2 [Bekker et al., 2015]

Figure 2.3 represents a possible vtree for the networks depicted in Figure 2.2. While we are going to use v for a vtree node, v^l and v^r are used to represent the left and right child respectively of the a node v. Furthermore, each Vtree induces a total variable order that is obtained by a left-right traversal of the tree.

2.4.2.2 The Syntax and Semantics of SDDs:

Here we will used $\langle . \rangle$ to specify a mapping from an SDD to a boolean function.
Definition 7. α *is an SDD that resprects vtree v iff:*

$$\begin{aligned}
- \quad & \alpha = \bot \text{ or } \alpha = \top \\
& \text{Semantics:} \langle \bot \rangle = false \text{ and } \langle \top \rangle = true \\
- \quad & \alpha = X \text{ or } \alpha = \neg X \text{ and } v \text{ is a leaf with variable } X \\
& Semantics : \langle X \rangle = X \text{ and } \langle \neg X \rangle = \neg X \\
- \quad & \alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}, v \text{ is internal,} \\
& p_1, \ldots, p_n \text{ are SDDs that respect subtrees of } v^l, \\
& s_1, \ldots, s_n \text{ are SDDs that resprect subtrees of } v^r, \text{and} \\
& \langle p_1 \rangle, \ldots, \langle p_n \rangle \text{ is a partition} \\
& \text{Semantics: } \langle \alpha \rangle = \bigvee_{i=1}^{n} \langle p_i \rangle \wedge \langle s_i \rangle
\end{aligned} \tag{2.1}$$

The size of SDD α, denoted $|\alpha|$, is obtained by summing the sizes of all its decompositions.

Bool- and Literal-SDD-Nodes are called *terminal Nodes* and *decomposition/decision Nodes* otherwise. Graphically we represent a decision node by a circle with a number indication the vtree node it respects, and elements of the decision node by boxes.

Figure 2.4 depicts an SDD and the vtree it respects for the specified boolean function. Figure 2.5 on the other hand gives an graphical representation of the SDD for the Markov network in figure 2.2.

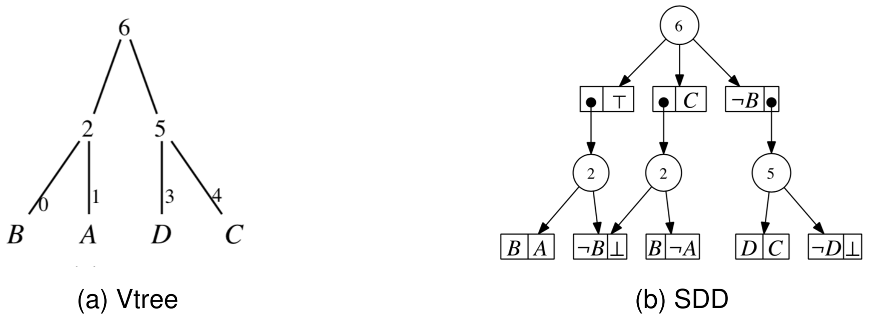


Figure 2.4: Function: $f = (A \wedge b) \vee (B \wedge C) \vee (C \wedge D)$ [Darwiche, 2011]

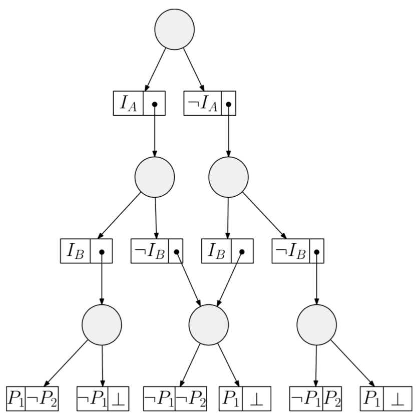


Figure 2.5: Tractable SDD representation of the Markov network in figure 2.2 [Bekker et al., 2015]

Thus we see that SDDs can be used to represent a boolean function in compact form.

2.4.2.3 Canonicity:

Canonicity of SDDs, is an especially interesting property introduced and proved by [Darwiche, 2011]. We are going to show the theorem here, but will refer to the original paper for the proof. First however we have to introduce a few definitions:

Definition 8. A Boolean function f essentially depends on vtree node v if it is not trivial and if v is a deepest node that includes all variables that f essentially depends on.
Definition 9. A non-trivial function essentially depends on exactly one vtree node.
Definition 10. An SDD is *compressed* iff for all decompositions $\{(p_1, s_1), \cdots, (p_n, s_n)\}$ in the SDD, $s_i \not\equiv s_j$ when $i \neq j$. It is *trimmed* if and only if it does not have decompositions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$.

In order to trim an SDD, one has to traverse it bottom up, replacing decompositions $\{(\top,\alpha)\}$ and $\{(\alpha,\top),(\neg\alpha,\bot)\}$ with α. Thus, any SDD that is compressed and trimmed respects a unique vtree node, assuming it is not equal to $\top$ or $\bot$. Furthermore two equivalent SDDs that are compressed and trimmed, respect the same unique vtree node. As an example we see that the SDD in Figure 2.4b is compressed and trimmed, where each Decision-Node respects the vtree node indicated by the number. Now to the actual result:

Theorem 5. *Let α and β be compressed and trimmed SDDs respecting nodes in the same vtree. Then $\alpha \equiv \beta$ iff $\alpha = \beta$ [Darwiche, 2011]*

2.4.2.4 Polytime Apply Operation:

The *Apply* operation is one of great importance, as it allows us to construct an SDD from any propositional KB in CNF by simply converting each clause into an SDD, and then recursively combining all of the SDDs. The *Apply* algorithm, as well as its proof can be found in the paper by [Darwiche, 2011]. The argument however generally follows along the same lines as for the *Apply* operation of OBDDs. The result however tells us, that any two SDDs can be conjoined as well as disjoint in polynomial time with respect the size of the SDD. Furthermore we see that we can negate an SDD in polytime as well, by doing an exclusive-or with $\top$ [Darwiche, 2011]. Formally [Darwiche, 2011] showed that using the apply operation for a given boolean operation and SDDs α_1 and α_2 returns the combined SDD in $O(|\alpha_1| * |\alpha_2|)$ time, where $|\alpha_x|$ denotes the size of an SDD as usual.

2.4.2.5 Upper Bound on SDDs:

Finally we want to talk about the upper bound of SDDs, a result again proven by [Darwiche, 2011]. By first laying out further definitions on vtrees, they prove that the size of an SDD for a given function f is asymptotically bound by the product of the number of variables and the width of a special kind of vtree. This result is then taken further by the following theorem:
Theorem 6. *A CNF with n variables and treewidth w has a compressed and trimmed SDD of size $O(n2^w)$, where the treewidth of a CNF formulation is related to the pathwidth pw in the following sense: $pw = O(wlogn)$ [Darwiche, 2011]*

While this means that the size of the SDD is still exponential in w, it is linear in the number of variables n. This means that SDDs come with a tighter bound on their size then Binary Decision Diagrams (BDD), which is squared exponential in treewidth.

2.5 On theoretical Complexity

In this section we want to give a brief overview of well defined problems in complexity theory, and their correspondence the formulation of previous sections, such as WMC.

2.5.1 Boolean Satisfiability Problem

The boolean satisfiability Problem (SAT), is one of the best known problems in complexity theory. Here we are interested in determining if a given boolean function has some assignment of variables that make the function evaluate to true, or if such an assignment does not exist. If it does, this function is then considered satisfiable, a term that comes form mathematical logic. By the famous Cook-Levi theorem, the SAT problem is indeed NP-Complete [Cook, 1971], meaning that all problems in complexity class NP-Complete are as most as difficult to solve as SAT. Notable algorithms for solving the SAT problem include the David-Putnam-Logemann-Loveland algorithm (DPLL), which solves the problem by utilizing a systematic recursive search procedure. While asymptotic runtime of solving this problem has decreased over the years, it still is exponential in the input size, corresponding to the NP complexity class.

As a quick reminder, the complexity class NP, an abbreviation for nondeterministic polynomial time, is used to classify the difficulty of a given decision problem. If a problem is said to be NP, this means that a (Yes/No) solution to the problem can be computed by a nondeterministic Turing Machine in polynomial time. The complexity class P on the other hand is the class of decision problems that can be computed by a deterministic Turing Machine in polynomial time. Finding an algorithm that solves the SAT problem in polytime would by extension show that $P = NP$, which is not considered to be the case (but has not been proven).

2.5.2 Sharp SAT

If SAT is the problem of determining if a given boolean function is satisfiable or not, #(sharp)SAT is the problem of finding all such models that satisfy the function. When it comes to the complexity of this problem, it corresponds to #P-Complete. Usually algorithms used to solve #SAT are extensions of the DPLL algorithms, that enumerate all models, rather than stopping when the first on is found. Algorithms computing #SAT are clearly exponential, however when the boolean function f is pre-compiled into an d-DNNF or OBDD, this changes. Once the function is represented in such a way, computing the number of satisfying assignments can be done in polynomial time [Darwiche and Marquis, 2002]. Furthermore we see that computing the number of satisfying assignments of a boolean function is just the model count as described in Section 2.3.2. This result is particularly interesting to me, as [Darwiche, 2011] showed that every OBDD is an SDD. Thus it follows that model counting in SDDs can be done in polynomial time as well, which we will provide a proof a concept for in Chapter 3.

2.5.3 Satisfiability Modulo Theories and LRA

While the SAT formulation has been proven to be very expressive and useful in many areas of computer science. An instance of Satisfiability Modulo Theory, is then a strict generalization of a SAT instance by allowing variables as well as predicates to come from an underlying pre-defined back ground theory. This means that the satisfiability of a given formula is determined with respect to the background theory, such as First-Order Logic. However, First-Order logic being the most general case, in many applications such expressive power is not needed, so the background theory fixes the interpretation of certain predicate and function symbols. An example is given by determining if the formula:

$$X \leq Y \wedge \neg(X \leq Y + 0)$$

is satisfiable, where $X, Y \in \mathcal{X}$. [Biere et al., 2009]
$\mathcal{LRA}$ being an abbreviation for quantifier-free Linear arithmetic over the rationals is an example of a Background theory. $\mathcal{LRA}$ is the fraction of first-order logic over the signature $(0, 1, +, \leq)$. For the hybrid distributions and networks we will consider the combination of $\mathcal{LRA}$ and propositional logic throughout the paper. This means that variables such as B_j range over the propositional variables, while X_j range over the constrains of the language. Therefore, ground atom of a function are of the form $B_0, \neg B_1, (X_0 + 1 \leq 10)$.

2.5.4 #SMT

We see that a satisfying model for a given function f with respect to some background theory is a instance of $x \in \mathbb{R}^n$. Thus # (sharp) SMT or computing the number of such models is usually infinite as variables are continues, and thus the formulation is not very informative in the usual case. However, by using the definition of WMI 2.3.3, integrating over all ground variables (with weight 1), we can think about the WMI with weight 1 as a density of the $\mathcal{NRA}$ formula. For example computing WMI of a given function f with n continuous variables, the WMI simplifies to computing the volume of the polytope encoded in f. With additional Boolean random variables it might be best to think about it as a hybrid version of #SAT. [Bekker et al., 2015, Ma et al., 2009, Chistikov et al., 2017]

Chapter 3

Proof of Concept for Querying SDDs

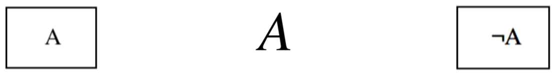
In the following chapter we will present a proof of concept for querying SDDs. [Darwiche and Marquis, 2002] showed that Model Counting (MC) as well as Model Enumeration (ME) are both polynomial time operations for d-DNNFs and OBDDs with respect to the size of the tree structure. In extension [Darwiche, 2011] proved that SDDs are a strict subset of d-DNNFs. Thus MC and ME are polytime operations for SDDs as well. As part of the project we developed a algorithm, implemented in python, that can compute the MC, as well as the ME in polynomial time with respect to the size of the SDD.

3.1 On MC and ME in SDDs

In order to understand how we enumerate the satisfying models of a given SDD we want to give a few toy examples going over the basics of how the models are constructed. Now first of all we see that a given SDD node can ether be a Bool-Node, a Literal-Node or a Decision-Node, where a Literal- and Decision-Node both have a corresponding vtree node they represent. Each Decision-Node holds sub-prime pairs denoted as elements, where each sub/prime is again a SDD-Node. It should also be noted at this point that we will denote the satisfying models of a given function as a set of dictionaries, that is each model is a dictionary; e.g. $\{A: true, B: false\}$ is a model for variables A and B, such that A and B are instantiated to "true" and "false" respectively. A set of such models then gives the satisfying models for a given function; e.g $\{\{A: true, B: false\}\}$ is the set of all satisfying models for the function $f = A \wedge \neg B$.

3.1.1 Literal-Node:

Now assume our boolean function is $f = A$ where f denotes the boolean function, and A is a propositional variable. This function would correspond to the SDD in Figure 3.1a, with corresponding vtree depicted in Figure 3.1b. Clearly this SDD node (and thereby also the function f) has $\{\{A: True\}\}$ as the satisfying models, and furthermore we see that if $f = \neg A$ the corresponding SDD is given in Figure 3.1c with satisfying models $\{\{A : False\}\}$. Thus each Literal Node has only 1 model (model count), which is the Literal it represents set to true or false corresponding to the SDD-Node.



(a) SDD for $f = A$ (b) Vtree for f (c) SDD for $f = \neg A$

Figure 3.1: SDD and Vtree for function f

3.1.2 Bool-Node:

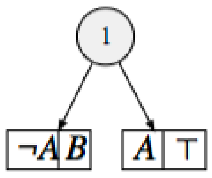
When it comes to Bool-Nodes we clearly have two different ones, true and false. If a given SDD-Node is a false Bool-Node, the corresponding set of satisfying models is empty, as there is no model that satisfies false. This also corresponds to a model count of 0. However if the SDD-Node is a true Bool-Node, things get a bit more complicated. Consider for example the boolean function $f = A \vee true$ which corresponds to $f = true$ but binds one variables, namely ”A”. So the set of satisfying models is given by $\{\{A : True\}, \{A : False\}\}$ as both of these instantiations satisfy the function f. Generalizing this result we see that the models for a true Bool-Node are all possible instantiations of the variables is binds, and thus the model count is 2^m where m is the number of variables it binds. To illustrate this, the corresponding sdd-Node and vtree are depicted in Figure 3.2.

(a) SDD (b) Vtree

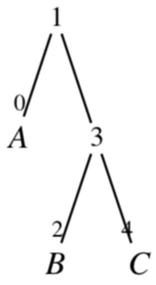
Figure 3.2: SDD and Vtree for function $f = A \vee true = true$

3.1.3 Decision-Node:

Given a SDD-Node which is a Decision-Node, we have 3 things to consider. First we have to conjoin the sub-prime pairs denoted as elements of a Node, then we have to complete the models for each element, and finally we have to disjoin all elements of the node. Consider for example the function $f = A \vee B$ over variables $A, B, C \in \mathcal{B}$ for which the corresponding vtree and SDD are depicted in Figure 3.3.



(a) SDD



(b) Vtree

Figure 3.3: SDD and Vtree for function $f = A \vee B$

Now we already know that the prime and sub of the first element are Literal-Nodes with models $\{\{A : False\}\}$ and $\{\{B : True\}\}$ respectively. As they are conjoined we get the $\{\{A : False, B : True\}\}$ as the set of models for the first element. This again can be generalized to the computation of adding each model in the set of models of the sub-Node to each model of the prime-Node models. For example, if the prime-Node already has two models, while the sub-Node only has one, the models for this element will be constructed by just adding the model of the sub-Node to each model of the prime-Node. This is possible by the strongly deterministic decomposition property of SDDs, meaning that the variables ”bound” by sub and prime are strictly exclusive; no variable can be bound by sub and prime at the same time. For this reason we can also compute the model count of an element simply by multiplying the model count of the sub and prime.

However this does not necessarily have to give us all the models of the element, as the Decision-Node might be referencing a vtree-Node that bounds more variables as in Figure 3.3. Here we see that the prime-Node of $element_1$ (A, $prime_1$) is referencing vtree node 0, and the sub-Node of $element_1$ (B, sub_1) is referencing vtree node 2, but the Decision-Node is referencing vtree node 1 which bind 3 variables A, B and C. Thus we have to complete the models for $element_1$ of the Decision-Node by adding all possible instantiations of C, to each model of the so far computed set of models for $element_1$. This means that we have to combine models $\{\{A : False, B : True\}\}$ (Models of $element_1$) with $\{\{C : True\}, \{C : False\}\}$ completing the models for $element_1$ as:

$$models_{element_1} = \{\{A : False, B : True, C : True\}, \{A : False, B : True, C : False\}\}$$

Again this can be generalized in the following way: After computing the models for $element_x$ we compute the ”unbound” or missing variables of the decision-Node, by simply subtracting the set of variables bound by the vtree-Node respecting sub and prime from the variables bound by the vtree-Node respecting the Decision-Node. To put this in mathematical terms let $sub_x, prime_x$ denote the sub and prime of $element_x$ and $vars_{sub_x}$, $vars_{prime_x}$ and $vars_{element_x}$ denote the set of variables bound by the vtree respecting sub_x, $prime_x$ and $element_x$ respectively. Then the models we have to add to each model of $prim_x$ and sub_x combined are all different instantiations of the variables in $missing_x = vars_{element_x} - vars_{sub_x} - vars_{prime_x}$. Considering the number of models or model count of $element_x$, we see that it similarly given by:

$$modelcount_{element_x} = modelcount_{prime_x} * modelcount_{sub_x} * 2^{missing_x}$$

It should be noted, that this formulation allows us to deal with true Bool-Nodes that appear in conjunctions such as the second element of the SDD in Figure 3.3 as well. This element, also respecting vtree-Node 1, has models:

$$\begin{aligned} models_{element_2} =& \{\{A : True, B : True, C : True\}, \{A : True, B : True, C : False\} \\ & , \{A : True, B : False, C : True\}, \{A : True, B : False, C : False\}\} \end{aligned}$$

Coming back to computing all models of a given Decision-Node x ($node_x$), we still have to combine the models of the individual elements of the node. In our example this would correspond to combining the $models_{element_1}$ and $models_{element_2}$ to get $models$. This is done by simply chaining the two sets of models together, as ether one or the other satisfies the node. Thus we get:

$$\begin{aligned} models_{element_2} =& \{\{A : False, B : True, C : True\}, \{A : False, B : True, C : False\} \\ & , \{A : True, B : True, C : True\}, \{A : True, B : True, C : False\} \\ & , \{A : True, B : False, C : True\}, \{A : True, B : False, C : False\}\} \end{aligned}$$

As a final remark one might be interested in why it is possible to have an SDD-Node that is not a true Bool-Node reference a vtree-Node that binds more variables than the node itself. This is a minimization property to the best of my knowledge, giving us the power to express boolean function such as $f = (A \lor B) \land (C \lor \neg C)$ simply by $f = (A \lor B)$.

3.2 The sharpLRA Python Library

Over the course of the last year we implemented multiple algorithms to perform queries on Sentential Decision Diagrams. All algorithms are implemented in python and are combined into one library which can be found at [1].

As a starting point for my work on SDDs we used the C-Library developed by the Automated Reasoning Group at UCLA, available at [2]. The library provides implementations for compiling a Propositional knowledge base in CNF/NNF to an SDD. Furthermore they implemented dynamic minimization of SDDs [Choi and Darwiche, 2013], considerably improving the size of the tree structure. They also provide functionality for performing model Counting as well as Weighted Model Counting. That being said, they do not provide an algorithm for enumerating the satisfying models of

[1] https://github.com/anton-musrevinu/thesisproject
[2] http://reasoning.cs.ucla.edu/sdd/

a given propositional KB. As we were interested in using the SDDs to perform Model Integration, by abstracting an LRA formula, this was a very important functionality to us. Therefore we used MC as a staring point developing a python Library that work as an extension of the UCLA Library. To be more precise, we are using the bash command ”sdd-Linux” provided by the package to compile CNF formulas into minimized SDDs. These are then saved to a file, and provides the input for my code.

SDDs are internally represented as an object of the class SDDNode [3], which can be a Literal-None, Decision-Node or Bool-Node. Each node has individual properties such as a variable-id for Literal-Nodes or a list of elements (sub, prime pairs, each being an index referencing other nodes) for Decision-Nodes. However Decision- and Literal-Nodes share is an attribute indicating the vtree node they represent. This in turn is used to compute the scope of a given SDD-Node, meaning the variables the vtree-Node corresponding to this particular SDD-Node binds. This is a very important property used for Model Counting and Model Enumeration, and is computed during parsing of the SDD. Furthermore it should be noted that all SDD Nodes are then stored in one array, which well be denote as ”_nodes” within the code samples.

3.3 Benchmarks

To visualize results we tested the performance of multiple examples, from well known #SAT Libraries. This includes Problems from the following sources: Plan_Regonition [4], DIMACS Benchmark Instances, blocksworld, the Beijing competition, as well as problems from [Belle et al., 2015] and the UCLA SDD Package. As to the size of the KBs, we were bound by the SDD compiler of the UCLA SDD package. We compiled all Problems in the table 3.1 for 24 hours on the University of Edinburgh Undergraduate Cluster, and continued working with those problems that finished compiling in the given time. Throughout this chapter we will use exactly these problems (the ones that finished within 24h) to demonstrate the performance of different model counting and model enumeration algorithms.

To give an idea of the cost of compiling a propositional KB into an SDD, the graph 3.4 gives an indication of this. In order to visualize the time it takes to construct the SDD, we are plotting the time with respect to the euclidean distance of the number of variables and number of clauses in the CNF formulation of the KB.

While we do not know exactly how the code works in practice, we can think about the procedure described in [Darwiche, 2011]. First each clause of the CNF formulation is compiled into an SDD, which is polynomial in the number of variables in that clause. In the next instance the different clause-SDDs are combined using the polynomial apply function (see section 2.4.2.4), where the apply function has $O(|SDD1| * |SDD2|)$ asymptotic runtime. What follows is that constructing the SDD from CNF is

[3] File: sharpsmt.sdd.SddStructure.py

[4] http://www.cs.rochester.edu/u/kautz/Cachet/Model_Counting_Benchmarks/index.htm

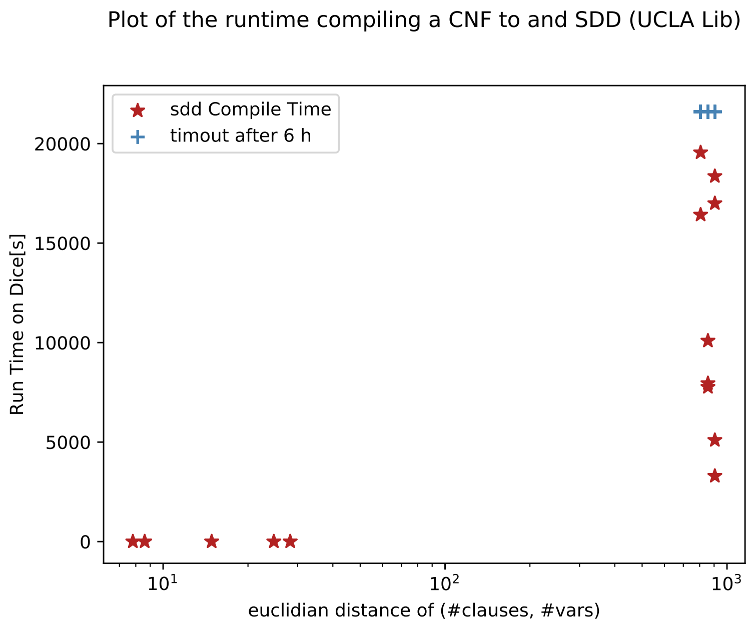


Figure 3.4: Compile Time of CNF to SDD

$O(c^{\#clauses})$ where $c \in \mathbb{R}^{>1}$ is a constant, eg. the size of the largest clause in the form of an SDD.

When it comes to the size of the SDD with respect to the propositional KB in CNF we are using, the following graph depicted in Figure 3.5 plots exactly that. Again looking at the figure we observe the exponential behavior of the size of an SDD for a given propositional KB.

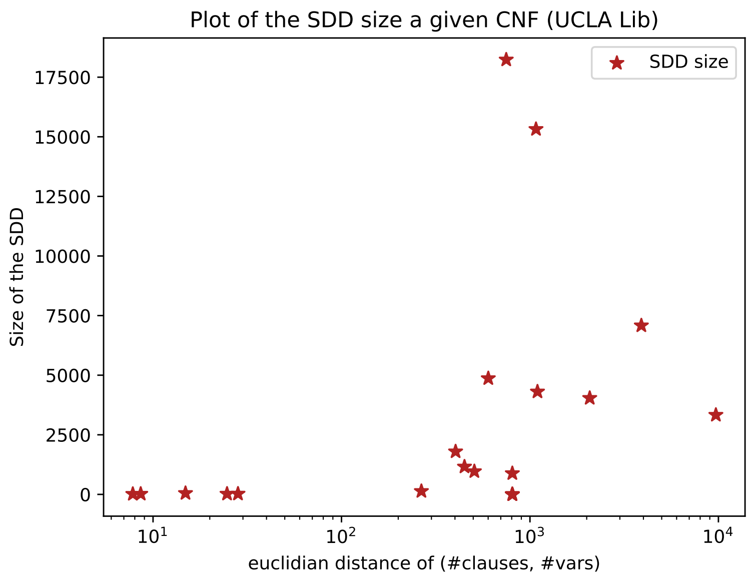


Figure 3.5: SDD Size for propositional KBs

3.4 Model Counting with SDDs

Model Counting was the starting point of my code base, and the simplest algorithm to implement. By traversing the tree we want to find the number of logical models that satisfy the given boolean function.

3.4.1 The MC Algorithm

The way we designed the algorithm, is by a recursive function (named ”_getModelCount” in Listing 3.1) that returns the model count of a given Node(id). Generally speaking the algorithms follows the procedure described in section 3.1 very closely:

The function ”_getModelCount” in Listing 3.1 is a recursive function computing the model count for a given node, with respect to the vtree Node it respects. First a node is retrieved from the internal storage space of the SDD nodes, before it is processed according to what kind of node(Decision, Literal or Bool) is represents.

If the Node is a Bool-Node, and set to ”False” then the corresponding model count is 0, as there is no model that satisfies ”False”. If it is ”True” the function returns 1, which will be used more as a sort of place holder since the actual model count is undefined at that point, until the models are completed at line 23 in Listing 3.1.

Now if the Node is a Literal-Node, then the corresponding model count is 1, as there is one model that satisfies this Node with respect to the variables it binds (1 as it is literal).

As we want to compute each node only once, we store the model-count for a given Decision-Node in the Node, as the variables ”node.modelCount”. Thus we have to check if the model count for this node has already been computed and return the the value on line 14 in Listing 3.1 if that is the case.

If none of the above is the case however, the node is a Decision-Node and we have to compute the model count of all elements (sub - prime pairs) of the node and return the sum of these counts. To compute the model count of a given element, we first compute the temporal model count by recursively calling ”_getModelCount” on the prime and the sub and multiply them together on line 19 in Listing 3.1. Next we compute the number of unreferenced variables by subtracting the referenced variables of the current Node by the number of referenced variables of prime and sub Node. Completing the model count for one element we multiply the temporal model count by 2 to the power of missing variables, corresponding to the size of the truth table of the missing variables (line 23 in Listing 3.1). Finally we add the model count of the element to the model count of the parent node (corresponding to the disjoint elements of the node) which was initialized to 0 on line 16 in Listing 3.1. Once all elements have been computed, we set the indicator variable for the node being computed before returning the resulting model count. Corresponding to Listing 3.1 this describes the behavior of the method "_getModelCount", which computes the model count for a given sdd-Node.

Listing 3.1: Recursive Model Counting Algorithm in Python

```
def _getModelCount(self,nodeId):
  #retrieve node from internal array
  node = self._nodes[nodeId]

  if isinstance(node, BoolNode):
    if node.true:
      return 1
    else:
      return 0
  elif isinstance(node, LitNode):
    return 1
  elif node.computed:
    #return model count of the node if already processed
    return node.modelCount

  node.modelCount = 0
  for (p,s) in node.elements:
    #Process Element
    tmpModelCount = self._getModelCount(s) * self.
        _getModelCount(p)

    #Complete Model Count for element
    missing = node.scopeCount - self._nodes[s].scopeCount -
        self._nodes[p].scopeCount
    node.modelCount += tmpModelCount * 2**missing

  node.computed = True
  return node.modelCount
```

To compute the model count of the whole propositional KB, the method "getModelCount" depicted in Listing 3.2 starts the algorithm "_getModelCount" with the stored root node of the SDD (line 4 in Listing 3.2). Now since the root node of the SDD does not have to coincide with the root node of the vtree, we again need to compute the missing variables, and multiply the result by 2 to the power of the missing variables. This is then the full model Count of the propositional KB, stored as an SDD.

Listing 3.2: Call Function for the Model counting algorithm

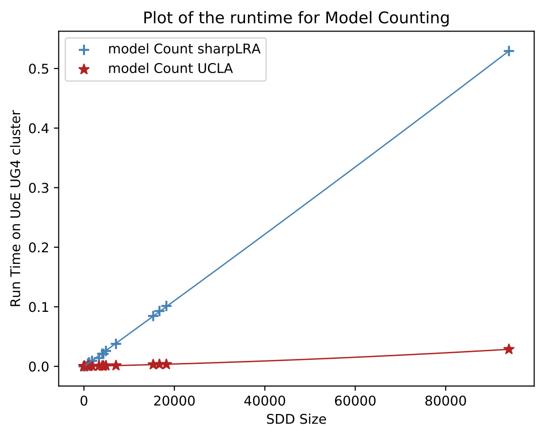
```
   def getModelCount(self):
 2
     #Retrieve model Count of root node
 4   modelCount = self._getModelCount(self._root)

 6   #complete models for the root of the vtree
     if len(self._vtreeMan.getScope(self._vtreeMan.getRoot())) !=
         self._nodes[self._root].scopeCount:
 8     mask1 = self._nodes[self._root].scope #List of Varids
       mask2 = self._vtreeMan.getScope(self._vtreeMan.getRoot())
10     missing = len(list(set(mask2) - set(mask1)))
```

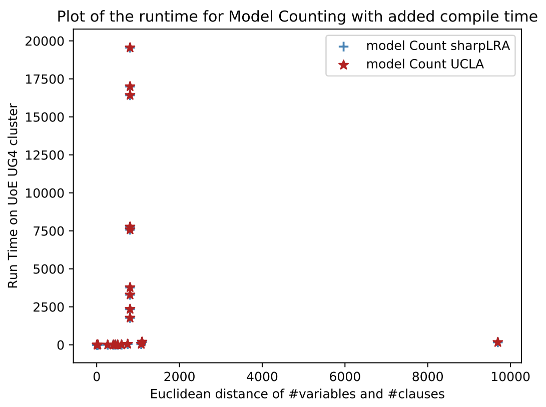
```
        modelCount = modelCount * 2**missing
    return modelCount
```

3.4.2 MC Performance

When it comes to the performance of the algorithm, we see that every node is computed once, with a constant amount of computations needed, corresponding to everything around the recursive call. Thus the algorithm is not only polynomial, but moreover linear in the size of the SDD, a result which is nicely demonstrated in Figure 3.6a. Here the blue '+'s correspond to my algorithm in comparison to the still superior one by the UCLA automated reasoning group denoted by red stars. The graph depicted in Figure 3.6b gives the same comparison only with the the compile time added to the model counting execution time.



(a) Model Counting sharpNRA vs UCLA



(b) Model Counting sharpNRA vs UCLA with Compile Time added

Figure 3.6: Model Counting sharpNRA vs UCLA

3.5 Model Enumeration with SDDs

Enumerating all the satisfying models of a given propositional KB represented as an SDD, proved the most difficult practical aspect of my project. While the basic algorithm is fairly similar to the model counting algorithm from section 3.4, we encountered various scalability problems. We are going to demonstrate and quickly outline the baseline algorithm used. Furthermore we will discuss the issues we have encountered, the different versions of the algorithm we have implemented and their performance afterwards.

3.5.1 The ME Algorithm

Very similar to the algorithm for model counting the algorithm for model enumeration follows along the same line, with a recursive function that enumerates the models for a given node. This function is called *_getModels* and is depicted in Listing 3.3.

What is important to note here is how models are dealt with in code. Starting off with python dictionaries we quickly realized that size is of an issue here and switched to using Bit-Vectors, each representing one model. A Bit-Vector is an array of bits (or python object) as the name suggests, giving a simple space efficient way to store assignments of propositional variables in my case. As each Literal already has an index between 0 and the number of Literals, a 0 or 1 at index x in a given model (BitVector) indicates that literal x is set to false or true respectively. To implement this, we used the Python BitVector package which can be found at [5].

As this algorithm, again follows the procedure described in Section 3.1 we will point out the main differences to the MC code in Listing 3.1, rather then explaining the procedure again.

First and foremost, the algorithm depicted in Listing 3.3 does only take Literal- and Decision-Nodes as inputs. This makes it possible to save some time, and process a Bool-Node within an element if it occurs. We then first check if the node has already been computed and return the corresponding models if that is the case. Otherwise we go on to check if it is a Literal-Node, create a new Model if it is indeed a Literal-Node. The Model is created as a Bit-Vector of the size "self._varFullModelLength". These variables created at parsing gives the number of total variables referenced by the vtree of the SDD we are querying. Once the model is created, and initialized to have 0's in all positions we set the bit corresponding to the Literal in the Bit-Vector to 1 or 0 if the Literal-Node is true or false. This is accomplished by using the variable ("node.varId") the Literal Node represents and the "node.negated" Boolean Variable, indicating if the given Literal-Node is negated or not. Once the corresponding bit is set in the model, the model is put into a list and returned.

Within the loop enumerating the elements, we first check if prime or sub are false Bool-Nodes, and skip the current element if that is the case, as a conjunction with "False" is always False and does not have any models. Furthermore, if prime or sub are "True", we only need to compute the oder node by a recursive call, completing the models on line 34 in Listing 3.3.

As described in Section 3.1 we only need to complete the models if the number of variables bound by prime and sub combined is not equal the number of variables bound be the Decision-Node we are currently processing. If that is the case the "tmpModels" are overwritten by the result of completing the so far computed tmpModels on Line 34 of Listing 3.3.

Finally we add the models of the current element to the models of the given node, before continuing to the next element of the node. Once all elements have been pro-

[5] https://engineering.purdue.edu/kak/dist/BitVector-3.4.8.html

cessed the resulting models are already stored in ”node.models” and are returned by the function.

Listing 3.3: Recursive Model Enumeration Algorithm in Python

```python
def _getModels(self,nodeId):
  #retrieve node from internal nodes array
  node = self._nodes[nodeId]

  if node.computed:
    #If node has already been computed return the corresponding models
    return node.models

  elif isinstance(node,LitNode):
    #If node is a Literal-Node initialize the model, save and return it as list
    model = BitVector(size = self._varFullModelLength)
    model[node.varId] = (0 if node.negated else 1)

    node.models = list([model])
    node.computed = True
    return node.models

  #If none of the above is the case the node is a Decision-Node
  node.models = []
  for (p,s) in node.elements:
    #compute models for the given element (p,s)
    tmpModels = []
    if self._isFalse(p) or self._isFalse(s):
      continue
    if self._isTrue(p):
      tmpModels = self._getModelsRAM(s)
    elif self._isTrue(s):
      tmpModels = self._getModelsRAM(p)
    else:
      tmpModels = self._product(self._getModelsRAM(p),self._getModelsRAM(s))

    #------ complete the computed models
    if node.scopeCount != self._nodes[p].scopeCount + self._nodes[s].scopeCount:
      tmpModels = self.completeModels(tmpModels,node.scope,p, s)

    #add the models of the lement to the models of the node
    node.models.extend(tmpModels)

  node.computed = True
  return modelsReturn
```

Similar to the algorithm for model counting, we need to call the recursive function in Listing 3.3 with the root Node of the SDD. The method responsible for this is depicted in Listing 3.4. However as previously mentioned the recursive ”_getModels” function only takes Literal- and Decision-Nodes, so we first have to make sure only such nodes are passed to the function. Afterwards the models are again completed for the variables bound by the full vtree. This then gives all the satisfying models of the given SDD.

Listing 3.4: Call Function for the Model Enumeration Algorithm

```python
def getModels(self):
  #check if kb is unat
  if self._isFalse(self._root):
    return []
  #check if sdd is true
  if self._isTrue(self._root):
    models = list([])
  else:
    #retrieve models for the root node
    models = self._getModels(self._root)

  #complete models for root of vtree
  if len(self._vtreeMan.getScope(self._vtreeMan.getRoot())) !=
      len(self._varMap):
    mask1 = self._nodes[self._root].scope
    mask2 = self._vtreeMan.getScope(self._vtreeMan.getRoot())
    missing = list(set(mask2) - set(mask1))

    models = self._completeModels(models,missing)
```

3.5.2 ME Algorithm Complexity

Looking at the algorithm in Listing 3.3 we see that for each element of a given node we might have to compute the product of the models of its sub and prime, before moving on to to complete the models if necessary. Thus we have to do to products of models where the complexity of the product is $O(|models1| * |models2|)$. Completing the models can be thought of computing the product of the so far computed models with models corresponding to all instantiations of unbound variables. Thus for each element x we have to do

$$O(|models_{x_{prime}}| * |models_{x_{sub}}|) + O((|models_{x_{prime}}| * |models_{x_{sub}}|) * 2^{missing})$$
$$\equiv O((|models_{x_{prime}}| * |models_{x_{sub}}|) * (2^{missing} + 1))$$

computations. As the number of models a given node has is bound by the number of variables is binds, we see that we can find a constant c, depending on the variables such that

$$|(|models_{x_{prime}}| * |models_{x_{sub}}|) * (2^{missing} + 1)| \leq c(\#vars)$$

for all elements x in the SDD. Now each element is processed exactly once and the computations performed for a given element have an upper bound, the number of computations it takes is polynomial in number of elements. As the number of elements a given SDD has, corresponds exactly with the definition of size, the algorithm for computing all given models of a propositional KB is polynomial in the size of the SDD corresponding to the KB.

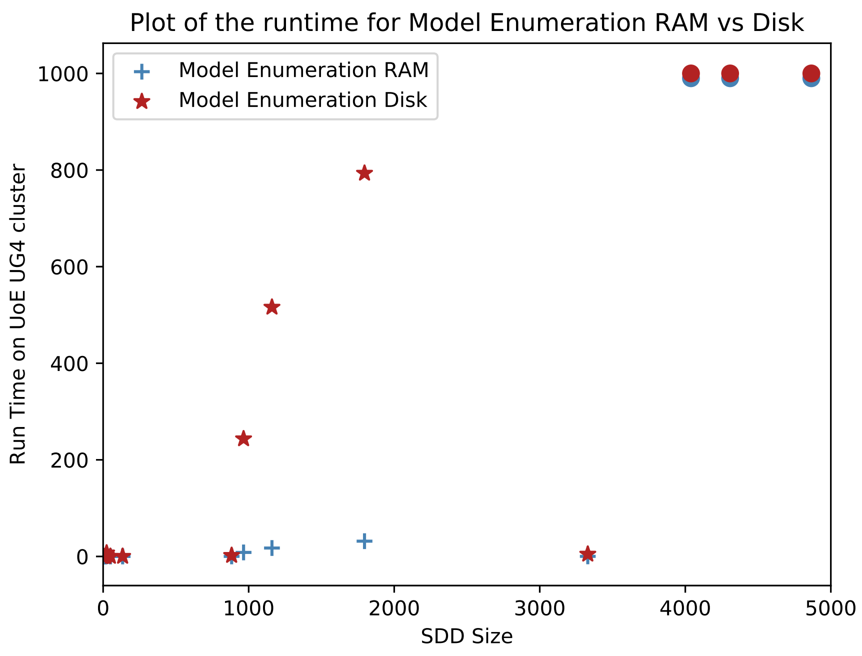


Figure 3.7: Model Enumeration Benchmark Ram vs Disk

3.5.3 ME Performance

Due to the large amount of models a given SDD can have, this algorithm has proven very difficult to implement without overflowing the internal memory of the computer. As the models are stores as a list of Bit-Vectors at each node, the total memory used can grow quickly and makes it difficult to scale the algorithms to larger SDD's.

The first design choice we made at this point was to use python generators [6] as much as possible which significantly improving runtime and space used. While generators do represent lists, they do so in a more abstract sense. Generally speaking they are more a stack trace than a list, that is only expanded and followed when the generator iterated.

However the models of a given node still have to be stored in memory, thus we developed two algorithms, one as depicted in Listing 3.4, which stores the models of each Node into the Ram, denoted by ModelEnumeration-RAM. The other algorithm, used the read and write functionality of the BitVector package to write the models of each Node to a file on the disk, which will be denoted by ModelEnumeration-DISK. Storing the models on the disk improves the scalability of the algorithm, however when computing the models of a new Decision-Node, the models for prime and sub still have to be combined before saving them to the file again. Thus we still see a memory overflow when using the DISK algorithm, due to loading the models of the elements of a node into memory before combining them appropriately.

When it comes to the performance of the algorithm, we quickly see looking at Figure 3.7 that storing the models in RAM is considerably faster, while storing them on

[6] https://wiki.python.org/moin/Generators

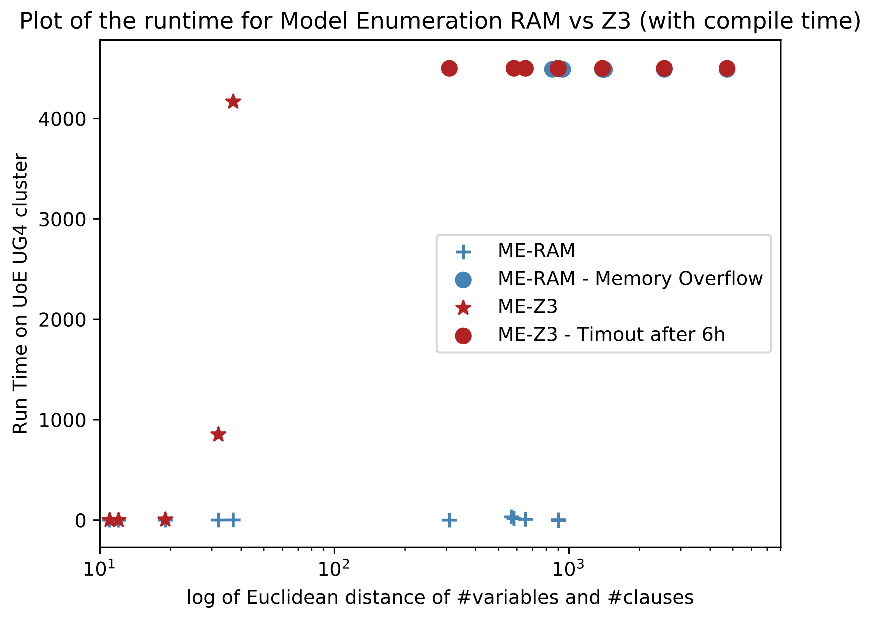


Figure 3.8: Model Enumeration with z3 vs Ram with Compile Time added

the disk makes it possible to compute the models of larger SDDs. To give a better indication of the speed we are dealing here with, and the polynomial behavior of the algorithm, Figure 3.8 demonstrates a comparison between my algorithm (RAM) with added compile time and one using the Z3 Package Developed my Microsoft Research [7]. This algorithm implemented with Z3 makes use of the Z3 Solver class, iteratively checking for models of the given propositional KB. Thus even adding the execution time of compiling the CNF to SDD, enumerating works considerable faster, while the Z3 solver suffers from the exponential increase where the euclidean distance is about 700. After that the algorithm exceeds the 6 hour timeout and is stopped. As an interesting side node, we also ran the algorithms on an 2.3 GHz Intel Core i5 processor with a Solid State Drive, with considerable improves the runtime of the DISK algorithm in comparison to the RAM one. This is interesting as it shows that writing and reading the models to and from the disk, is a considerable bottleneck of the algorithm.

3.6 Benchmark Problems with results:

[7] https://github.com/Z3Prover/z3

Table 3.1: Benchmarks used for MC and ME

Problem	NBClauses	Vars	Size	Compile Time [Dice]	ModelCount	MC [Dice] - UCLA	MC [Dice] - sharpNR
APPROXWMI/1023˙data	5	14	50	0.010067463	1023	0.000061	0.000474215
APPROXWMI/32768˙data	9	23	22	0.00722909	32768	0.000013	0.000274897
APPROXWMI/3˙data	5	6	14	0.007462978	3	0.000008	0.000210524
APPROXWMI/65536˙data	11	26	24	0.012256861	65536	0.000014	0.00028944
APPROXWMI/7˙data	5	7	19	0.007666349	7	0.000011	0.000241041
Plan˙Recognition/4step	418	165	1160	4.233083487	86432	0.000337	0.006740332
Plan˙Recognition/5step	475	177	965	4.604432344	81300	0.00029	0.005070448
Plan˙Recognition/log-1	3785	939	7081	-	5.64E+20	0.001331	0.037787914
Plan˙Recognition/tire-1	1038	352	4308	191.5772183	726440820	0.000724	0.021032572
Plan˙Recognition/tire-2	2001	550	4039	-	7.39E+11	0.000719	0.021210432
Bejing/3blocks	9690	283	3330	165.3674376	174	0.000459	0.014006376
blocksworld/anomaly	261	48	134	0.188941956	1	0.000029	0.001218796
jnh/jnh10	800	100	0	7763.482601	0	0.000017	0.000172853
jnh/jnh205	800	100	883	19551.47249	468	0.000249	0.004508018
jnh/jnh20	800	100	0	7576.116427	0	0.000031	0.000190973
jnh/jnh216	800	100	0	16422.72203	0	0.000021	0.000214338
jnh/jnh2	800	100	0	2344.920862	0	0.000019	0.00013113
jnh/jnh302	800	100	0	16996.33688	0	0.000017	0.000129461
jnh/jnh305	800	100	0	3774.856774	0	0.000018	0.000124931
jnh/jnh307	800	100	0	1771.127552	0	0.000017	0.000138521
jnh/jnh309	800	100	0	3293.717427	0	0.000018	0.000137806
sddUCLA/c8	713	227	18232	48.48982501	268435456	0.003341	0.101371765
sddUCLA/c432.isc	986	432	15318	42.63233328	68719476736	0.002972	0.084534883
sddUCLA/count	425	425	4866	6.411350965	68719476736	0.001181	0.025858402
sddUCLA/s208.1.scan	285	285	1796	1.341817856	262144	0.000487	0.00934124
complexQueryExperiments/voting	-	1961	16699	-	1.25824E+409	0.003351	0.092869759
complexQueryExperiments/movie	-	2390	93989	-	1.07E+301	0.028613	0.529385805

Chapter 4

On Probabilistic Inference by Weighted Model Integration using SDDs

Over the past few years we have seen a number papers on exact probabilistic inference [Bekker et al., 2015, Morettin et al., 2017, Sanner et al., 2012] using the formulation of Weighed Model Integration. What I am proposing in this chapter is a novel way of doing Weighted Model Integration by using SDDs as the underlying querying language. Here the Model enumeration algorithm presented in chapter 3.5, is making to possible to efficiently retrieve all satisfying models for a given propositional KB Δ^{-}. This in combination with predicate abstraction and the WMI formulation makes it possible to do Probabilistic Inference.

After explaining the restrictions we impose on the $\mathcal{NRA}$ formula Δ as an input, we will expand on two different formulations used, namely Model Integration(MI) and Weighted Model Integration (WMI). We will then go on to discuss the implementation in python and finally demonstrate some performance results.

4.1 Restrictions on the Input

As previously mentioned our formulation imposes certain restrictions on the set of SMT formulas Δ we are able to use.

Fist and foremost we are only considering $\mathcal{NRA}$ formulas with the signature $(0,1,+,-,*,/,\leq,<,\geq,>)$ as a subset of all SMT formulas.

Furthermore we only consider KBs Δ such that each literal in Δ can be abstracted such that the propositional variables B_x is of the form $B_x = (a \leq X_x \leq b)$ where $a, b \in \mathcal{R} \cup \{-inf, inf\}$ and X_x is a ground variable. This means that every continuous variables X_x is defined form some interval $[a,b]$ where a, b are constants.

Finally the examples we used for evaluation all have a weight-function defines as a single function over all ground atoms $\mathcal{X}$ and some subset of propositional variables $\mathbf{B} \subseteq \mathcal{B}$. This however is by now means a restrictions on the weight function w, but rather a notational remark.

4.2 Probabilistic Inference with SDDs - The Concept

Doing probabilistic inference composes of calculating the WMI of two separate, but related SMT formulations. Meaning we have to compute the WMI of a given KB Δ conjoined with some evidence e and the query q, divining it by the WMI of Δ conjoined with the evidence e. This formulation introduced by [Belle et al., 2015] and explained in more detail in Section 2.3.3, can be written as:

$$Pr_{\Delta}(q|\mathbf{e}) = \frac{WMI(\Delta \wedge \mathbf{e} \wedge q)}{WMI(\Delta \wedge \mathbf{e})}$$

Hence once we are able to compute the WMI of a given $\mathcal{LRA}$ formula Δ, the sought after probability is a simple devision of two such computations. The following sections will therefore focus on computing the WMI and the MI, as a simplification of the former.

4.2.1 WMI/MI with SDDs - The Pipeline

As a basis for doing probabilistic inference, we first have to be able to calculate the WMI of a given *LRA* formula Δ efficiently. As we are interested in doing so by using SDDs as a query language, the WMI breaks down into a sequence of sub-computations depicted as the sharpNRA pipeline. This pipeline is graphically represented in Figure 4.1.

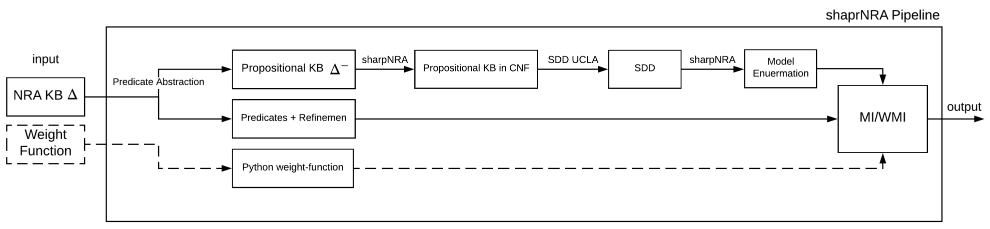


Figure 4.1: Graphical depiction of the pipeline for MI/WMI

Abstraction: The goal of this aspect of the pipeline is to abstract a $\mathcal{LRA}$ KB Δ into a propositional KB Δ^{-}. Parsing the formula, the deepest possible boolean formulations of the formula are one-by-one abstracted into propositional variables, while at the same time a Propositional KB is build with the resulting predicates. In practice this is achieved by parsing the formula as a string, recursively building the propositional KB using the Z3 Prover [1] developed by the Microsoft Research Community. Repre-

[1] https://github.com/Z3Prover/z3/wiki

senting the formulas internally as z3 object makes it possible to easily rearrange and decompose the formula (e.g for the CNF compiler).

In general, we see that the abstractions is done very similar to the abstraction explained in the paper [Belle et al., 2015]. Suppose for example Δ is the following formula:

$$B_1 \vee (0 \leq X_1 \leq 10) \vee (0 \leq X_2)$$

then the propositional abstraction will give a KB Δ^- of the form:

$$B_1 \vee B_2^- \vee B_3^- \text{ where } B_2^+ = (0 \leq X_1 \leq 10), B_3^+ = (0 \leq X_2 \leq inf)$$

As previously mentioned the underlying condition here is that Δ is restricted to a certain subset of $\mathcal{LRA}$ formulas as described in Section 4.1.

Now once the full formula is abstracted we have a propositional formula, as well a list of mappings for each propositional variable B_x to the corresponding $\mathcal{LRA}$ refinement B_x^+, and a mapping for each $\mathcal{LRA}$ ground atom $x \in \mathcal{R}$ to a list of propositional variables that reference it.

Weight Function Parsing: This only applies to the case when we are doing exact probabilistic inference, and there is a joint weight function given. If so, we need to read that function from a file (in SMT 1.0- [Choi and Darwiche, 2013] or SMT 2.0 format) and parse it into a Python object of the function type. Once parsed the function is stored with the predicate abstractions in the Abstraction Manager of our python Library.

Converting the CNF: Once we retrieved the propositional KB for a given $\mathcal{LRA}$ KB Δ, we have to convert it to Conjunctive Normal Form(CNF) and write it to a file, as this is the required input format for the SDD-compiler.

Compiling to SDD: In the next instance we compile the propositional KB in CNF into an SDD α and the corresponding Vtree. Both will again be written to a file and stored in a temporal location to serve as an input for the sharpLRA package.

Parsing the SDD and Vtree: Once we finished compiling we can read the vtree and SDD into our internal data structure. As mentioned before, the scope of each vtree- and SDD-Node is computed at this point as well.

Model Enumeration: Now using the RAM algorithm introduced in Section 2.3.3 we enumerate all the models for the given SDD, and store them for later use.

Computing WMI: As a final step in our pipeline we have to enumerate all the models (we computed at an earlier point) summing the volume of each model with respect to the refinements of referenced boolean variables. Two methods are going to be proposed for computing the volume of a given model, one being a very fast approximation method that ignores all the weights, the other being a formulation using n dimensional integrals to compute the exact volume.

4.3 Computing the Volume of a given Model - Theory

As can be seen in the previous section calculating the volume for a given model is the crucial part of the procedure described for probabilistic inference (the pipeline). All previous steps in the pipeline such as compiling and model enumeration have been shown to work very fast, even for larger KBs, see Chapter 3. Thus the problem we are still facing is the one of computing the volume for a given model efficiently, with respect to the background theory, the weight function and the propositional refinements. The importance for finding an efficient way to do this becomes apparent, when one considers that we have to calculate the volume for *every* model that satisfies a given KB, to retrieve the WMI, and do probabilistic inference as an extension. Based on the formulation by [Belle et al., 2015] the volume of a model is defined in the following way:

$$VOL(m,w) = \int_{\{l^+ : l \in m\}} WEIGHT(m,w)dX$$

where the $WEIGHT$ is defined in the usual way. In order to tackle this problem we will propose two methods of computing this number, one being an approximation.

Once we outlined both methods, more detail will be given on design choices and implementation of the pipeline, showing that a great amount of computations can be saved due to smartly reusing evaluations. This drastically improves the performance of the algorithm as a whole which will be demonstrated in Section 4.5.

4.3.1 Approximate Volume Computation - MI

The idea used to approximate the volume of a given model is based on the idea of neglecting the weight-function completely, and thereby computing the (unweighted) Model Integration of a given $\mathcal{LRA}$ formula. This approximation has been studied before [Chistikov et al., 2017] and is generally referred to as sharpSMT or #SMT in the research community. Similar to #SAT being a generalization of the SAT problem, the #SMT is to be thought of as an extension of the SMT problem. Yet another way to think about this from a more algorithmic view point, is to think of #SMT as the Model Integration, computing the n-dimensional polytop of the a given $\mathcal{NRA}$ formula Δ, similar to Model Counting for a given propositional formula Δ^-. Now, these relations between the different formulations and problems will become more apparent with the following Definition and Lemma.

Definition 11. Let Δ be a $\mathcal{NRA}$ theory over Boolean and Rational Variables $\mathcal{B}$, $\mathcal{X}$ and Literal $\mathcal{L}$. Furthermore let $w_1 : \mathcal{L} \rightarrow 1$, be the function mapping every Literal to 1. Then the Model Integration (or #SMT) is defined as follows:

$$MI(\Delta) = \sum_{m \models \Delta^-} Vol(m, w_1)$$

where $Vol(m,w)$ is defined as in Definition 4:

$$VOL(m,w) = \int_{\{l^+ : l \in m\}} WEIGHT(m,w)dX$$

What we can see here is that this formulation is more general than necessary, since the weight of every Model will be 1 due to using w_1 as the weight function. However it is general enough to demonstrate the connection to the WMI formulation, and we will prove in the next lemma how this can be reduced to a much simpler formulation.

Lemma 7. *Let Δ be a $\mathcal{NRA}$ theory over Boolean and rational variables $\mathcal{B}$ and $\mathcal{X}$, and literals $\mathcal{L}$ such that each every refinement of $l^+ : l \in M$, for $M \models \Delta^-$ has the form $l^+ = (a \leq x \leq b)$ for some variable $x \in \mathcal{X}$ and values $a, b \in \mathbb{R} \cup \{inf, -inf\}$. Suppose further that $w : \mathcal{L} \to 1$ for all $c \in \mathbb{R}$, then the Volume of a model is given by:*

$$VOL(m,w) = \int_{\{l^+:l\in m\}} WEIGHT(m,w)dX = WEIGHT(m,w) * \prod_{\{x\in l^+:l\in m\}} INTERVAL(x) \tag{4.1}$$

where: $INTERVAL(x) = max(b_1, \cdots, b_n) - min(a_1, \cdots, a_n)$, for all $(a_i \leq x \leq b_i) = l^+ \in m$ such that $x \in l$.

Proof. This is a direct consequence of the properties of integrals. As the function is constant for all literal $l \in \mathcal{L}$, $WEIGHT(m,w)$ of a given model m is constant as well. Thus the the volume integration with respect to a single literal l^+ is equivalent to:

$$\int_{l^+} WEIGHT(m,w)dx = WEIGHT(m,w) * x|_a^b = (b-a) * WEIGHT(m,w)$$

Since multiple literals can reference the same variable x, we have to intersect all such intervals that are corresponding to the model we want to find the volume of. If the variable x is referred to by more than one literal we get the following definition of the integral:

$$\begin{aligned}
VOL(M,w) &= \int_{l^+:l\in m} WEIGHT(m,w)dX \\
&= \int_{l^+:l\in m\wedge x\notin l} \int_{(a_1\leq x\leq b_1)\wedge\cdots\wedge(a_n\leq x\leq b_n)} WEIGHT(m,w)dxd(X - \{x\}) \\
&= \int_{l^+:l\in m\wedge x\notin l} \int_{(min(a_1,\cdots,a_n)\leq x\leq max(a_1,\cdots,a_n)} WEIGHT(m,w)dxd(X - \{x\}) \\
&= \int_{l^+:l\in m\wedge x\notin l} INTERVAL(x) * WEIGHT(m,w)d(X - \{x\}) \\
&= WEIGHT(m,w) * \prod_{\{x\in l^+:l\in m\}} INTERVAL(x)
\end{aligned}$$

thus the integral over the weights reduces to a simple product of real intervals, as $WEIGHT(m,w)$ is a constant for all m and the result follows. □

Now we can use the more general Lemma 7, with our definition of MI Def 11 to derive the following formulation for MI:

$$MI(\Delta) = \sum_{m\models\Delta^-} VOL(m,w_1) = \sum_{m\models\Delta^-} \prod_{\{x\in l^+:l\in m\}} INTERVAL(x)$$

This is possible due to the fact that the $WEIGHT(m,w)$ is always 1 in the definition of MI. Furthermore it should be noted at this point again, that the above definitions is only defined for $\mathcal{NRA}$ and the additional restriction that the refinement of every Literal l is of the form: $l^+ = (a \leq x \leq b)$ for $x \in \mathcal{X}$ and $a,b \in \mathbb{R} \cup \{-inf, inf\}$.

That being said, we are now able to compute the MI of a given KB Δ without integration, but rather by a sum of products of real intervals. Section 4.4.1 will go into more detail on how this has been implemented in python, while Section 4.5 will demonstrate the results and performance obtained.

4.3.2 Exact Volume Computation - WMI

Computing the exact volume of a given Model m, weight-function w and KB Δ is generally straight forward, following Definition 4 by integrating over the weight-function with respect to all referenced ground variables. However, as already mentioned, we are considering a weight-function over real and boolean variables. Thus instead of only computing a single integral we have to compute the volume for all possible instantiations of boolean variables the weight function takes as input, and sum them together. This is basically a simple way of integrating over a function with discrete variables such as boolean ones. As a final remark we want to point out that is not necessary to integrate over all ground variables in $\mathcal{X}$, as the definition might suggest, but only over the subset the weight-function is defined for.

4.4 Python Implementation

4.4.1 Model Integration Implementation

Implementing the MI for a given $\mathcal{NRA}$ formula Δ is mainly concerned with the intervals of the refinements of propositional literals and their manipulation. Thus while parsing all resulting propositional literal (with or without corresponding refinement) are stored as objects of a custom class denoted as Predicate [2] within the code base. This means that each literal in a given Model of a KB Δ^-, corresponds to exactly one such Predicate. Furthermore, if such a Predicate object has a refinement, it is stored within the object as an interval with an maximum and minimum value in $\mathbb{R} \cup \{-inf, inf\}$. This once again is done when the KB Δ is parsed, at the abstraction stage of the pipeline.

Now when we are computing the volume (without the weight) for a given model we first have to iterate over all ground variables $x \in \mathcal{X}$ computing the interval corresponding to the model. How this is achieved in code can be seen in Listing 4.1. Here the python dictionary *self._groundVarReferences* maps each ground variable to a list of Predicates that reference this ground variable in their refinement. Thus we iterate over

[2]file: sharpNRA.Predicate.py

all such ground variables, computing the difference in intervals with respect to the current model, using the function *self._get_Interval_Float*. All such differences are then multiplied together, giving the volume of a model disregarding the weight-function. This volume is then returned, and summed together with the volumes for each other satisfying model of the given KB Δ^-, giving the $MI(\Delta)$ for the KB Δ by Section 4.3.1.

Listing 4.1: Computing the Volume of a given model for MI

```python
def vol_no_weight(self,model):
  vol = 1
  for groundVar in self._groundVarRefernces.keys():
    diff =  self._get_Interval_Float(groundVar, model)
    vol *= diff
  return vol
```

Now when it comes to computing the difference in intervals for a given ground variables x and model m, function *self._get_Interval_Float* depicted in Listing 4.2 gives the python implementation used to achieve this. Within this function we first retrieve all Predicates that reference the given ground Variable on line 2 of Listing 4.2. This list of Predicates is then used to compute the subset of this list that is true in the given model denoted as *trueRefPredicates* in the code.

If this list (*trueRefPredicates*) of True-referenced-Predicates is empty the identity of a product (1) is returned as the difference. Otherwise we create a new Interval object (initialized to -inf, inf) and combine it with all intervals corresponding to the Predicates which are referenced by the ground Variable and are true in the given model (Line 6,7 in Listing 4.2). Once the desired interval is computed we return it as a number corresponding to the difference $b-a$ for a given interval $[a,b]$.

The sum of the volumes for each satisfying model of the KB Δ^- then gives the MI of the KB Δ.

Listing 4.2: Computing the Interval for a given ground Variable

```python
  def _get_Interval_Float(self,groundVar, model):
2   refPropositionalVars = self._groundVarRefernces[groundVar]
    trueRefPredicates = [item for item in refPropositionalVars if
        model[item - 1] == 1]
4   if not trueRefPredicates:
      return 1
6   interval = Interval()
    for varId in trueRefPredicates:
8     interval.combine(self._predicates[varId].interval)
    return interval.asFloat()
```

4.4.2 Weighted Model Integration Implementation

Retrieving the exact WMI for a given $\mathcal{NRA}$ formula Δ is mainly concerned with the exact volume computation for a given Model m and weight-function w. Computing this volume we have to integrate over the weight-function with respect to the ground variables, and the model. In practice we see that the weight-function is defined for a subset of the ground variables in addition to a subset of the Boolean predicates of the KB Δ^{-}. Our aim here is then to integrate this function over both Real and Boolean variables, which can be formulated as the sum of integrals over the Real ground variables for each instantiation of the Boolean variables (with respect to the model).

Our implementation of this is based on the *integrate.nquad* [3] method of the *scipy* [4] python package. This function allows us to compute an n-dimensional defined integral of a given python function object, within certain bounds and additional arguments (variable instantiations).

So in order to use this method, the first challenge was to construct a python function object of the weight-function given as an SMT string. While the construction of the function is done similar to the SMT KB parser, special attention has to be paid to the variable order in the function declaration. When this function is constructed (in the abstraction stage of the pipeline), it is done so that all Boolean variables follow the last Real variable in the declaration, enabling us to pass the Boolean values as arguments to the integration method rather than integrating over those as well. An example of this is given in Listing 4.3 showing a possible weight-function as a python function. Furthermore it can be observed that all Boolean variables (denoted by A_x) are at the very end of the function declaration. As the order of variables in the function declaration is of great importance it is recorded and saved, when the function is constructed.

Listing 4.3: Example of a possible weight function, as as a python function

```
def weight_func(x_3,x_1,x_0,x_2,A_2):
  return ( ( ( ( 5.0 * x_3 * x_1 ) + ( -8.0 * (x_1**2.0) * (x_0
      **2.0) ) + ( 4.0 * x_2 * (x_1**0.0) * (x_3**0.0) * x_0 ) )
       * ( -8.0 * x_2 * (x_1**0.0) ) ) * (( 10.0 * x_0 * (x_1
      **2.0) ) if ( A_2 and ( ( ( 4.0 * x_1 ) + ( -8.0 * x_0 ) +
       ( -6.0 * x_2 ) ) < -9.0 ) and A_2 ) else ( ( 6.0 * (x_0
      **2.0) ) + ( 8.0 * (x_0**2.0) * (x_1**0.0) * (x_2**2.0) )
      + ( 10.0 * x_2 * (x_0**2.0) ) )) )
```

Now to actually do the integration for a given model, we first have to compute the intervals with respect to a model in a similar manner to the MI formulation 4.4.1. One key difference here however, is that we want to retrieve the intervals as intervals (lists of two values) rather than a difference as a real value. The other distinction to the MI method is that the order in which the intervals are stored has to correspond to the variable order of the weight-function declaration. Thus we first retrieve the variable order of the Real and Boolean Variables from the custom function object, and then retrieve the intervals in that order, as depicted in Listing 4.4. Boolean intervals are basically a list such as $[True, False]$ if the variables is not a Predicate of the KB Δ^{-}. Otherwise the interval is a list with a single value corresponding to the the instantiation of that predicate in the model.

[3] https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.integrate.nquad.html#scipy.integrate.nquad

[4] https://www.scipy.org/

Listing 4.4: Computing the volume for a given model - WMI

```
   def vol_single_weight(self,model):
 2   vol = 1
     variableOrder = self._weightFunction.getVariableOrder()
 4   variableOrderBool = self._weightFunction.getVariableOrderBool
         ()

 6   intervalsReal = []
     for groundVar in variableOrder:
 8     interval =  self._get_Interval_List(groundVar,model)
       intervalsReal.append(interval)
10
     intervalsBool = []
12   for boolVar in variableOrderBool:
       interval = self._get_Bool_Interval(boolVar, model)
14     intervalsBool.append(interval)

16   (vol, error) = self._volume_for_Model(intervalsReal,
         intervalsBool)

18   return (vol, error)
```

Once all intervals (Real and Boolean) have been constructed for a given model, we call the function *self._volume_for_Model*. What this function does in a nutshell is to construct a list of all possible instantiations of the Boolean variables the weight-function takes with respect to the Boolean Intervals (*intervalsBool* object). Then for every such instantiation we compute the integral over the ground variables with the Boolean variable assignments as arguments. Finally all such integrals are added together and returned as a tuple $(vol, error)$ giving the exact volume of a given model up to some integration error, depicted in Line 16-18 of Listing 4.4.

The sum of volumes for all satisfying models computed in the *QueryManager* class of our Library, is then the WMI for a given KB Δ.

4.5 Performance

When it comes to the performance of the pipeline, we were able to test it on a data set provided by Samuel Kolb [Sanner et al., 2012]. This comprises of a formula in NRA, such that all the imposed restrictions (Section 4.1) hold. The formula tested consists of 8 clauses and 12 variables once abstracted and converted to CNF. The resulting SDD has a size of 33. Furthermore the data provided also gives a weight-function and a set of queries, so we are able to do probabilistic inference for the 100 queries all asking the probability of a certain ground variable x being withing an interval. In order to run the experiment we start the pipeline with the original KB Δ conjoined with the query q $(\Delta \wedge q)$, and divide the result be the value obtained starting the pipeline only on the KB Δ. We will then measure the runtime of running such a query as well as the accuracy for the approximation method MI.

It should be noted at this point, that in practice some computations of the pipeline can be neglected, as they would compute the same values multiple times. Thus for each query the denominator of the fraction, given by the WMI for the whole KB Δ does not change, and can therefor be reused. Furthermore the volume for a given model is stored in a dictionary were the keys are the exact intervals corresponding to the model. In practice we see that many models have the same intervals or bound to integrate over, and by storing them, we are able to reuse a given computed value if the intervals are the same.

4.5.1 Model Integration Performance

Testing our formulation of Model Integration, we found that we are able to compute the sharpLRA or MI of a given KB Δ very efficiently. However MI being an simplification or abstraction of WMI, we found that it is a very crude one.

In Figure 4.2 the runtime of computing the probability of a given query (of the from q: $(X_x \leq c)$ for some ground variable X_x and some constant $c \in [0, 100]$ is depicted. Here we are using the MI formulation, only working with intervals, and thus see a very efficient performance considering that for each instance the whole pipeline has to be computed.

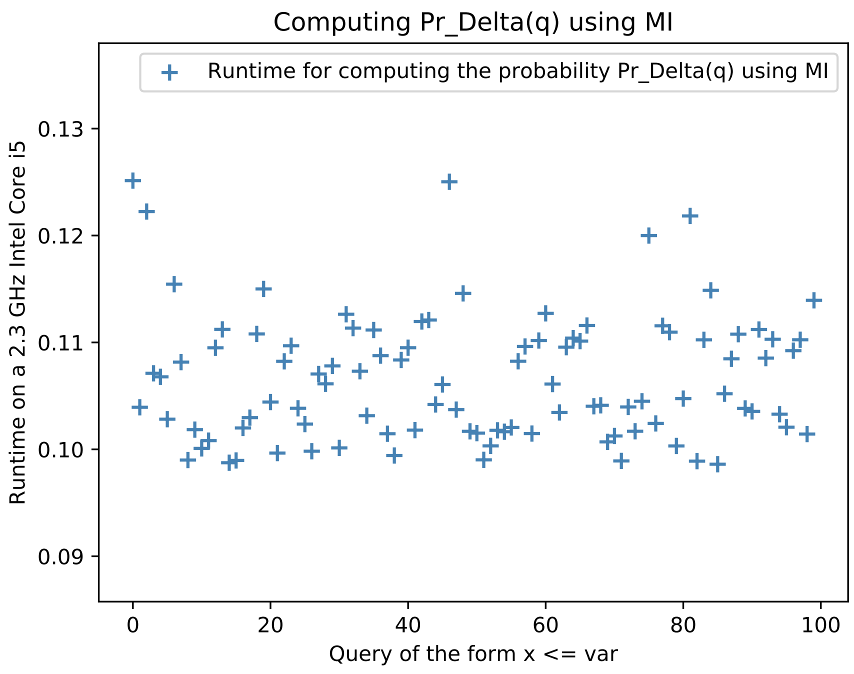


Figure 4.2: Runtime of Computing the Probability Pr(q) using MI

This performance is however bound to a considerable error when it comes to the resulting probability as can be seen in Figure 4.3. Here the absolute error between the computed probability of the given query and the actual probability is depicted on the y axis. As we are dealing with probabilities here, an absolute error of up to 0.6 makes this approximation very unreliable in practice. This is due to the complicated weight function that is neglected doing MI.

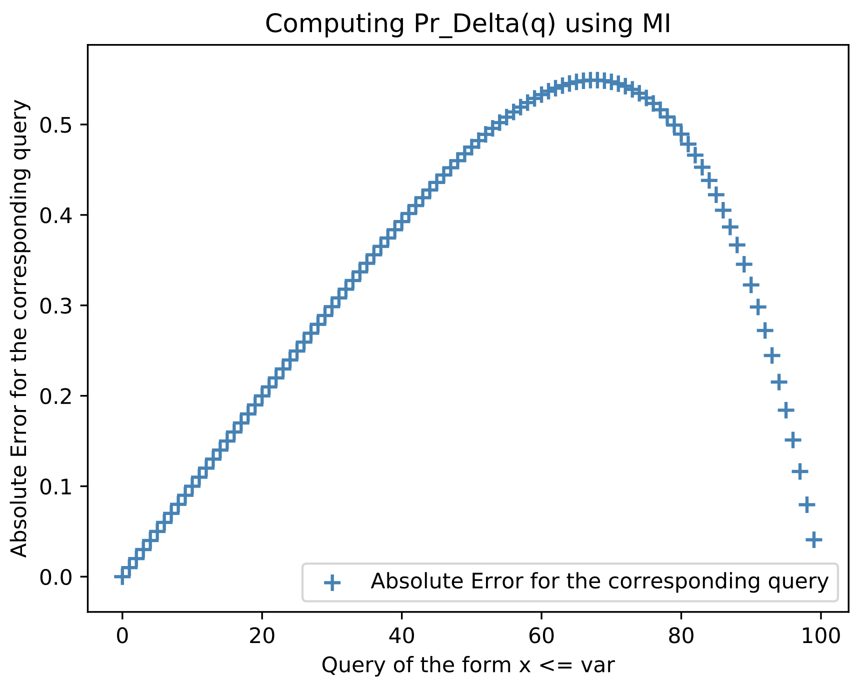


Figure 4.3: Error of Computing the Probability Pr(q) using MI

4.5.2 Weighted Model Integration Performance

As previously mentioned we are using the *scipy.integration* python package to evaluate the defined n-dimensional integral of the weight-function with respect to the bounds corresponding to the model. This function comes with an absolute error tolerance, corresponding to the numeric error of computing a Fourier integral, when we have infinite bounds. However running the examples we see that absolute error, in comparison to the solutions we were given for the data, is in the region of 10^{-12} for all queries tested and thereby not considered.

When it comes to the runtime of computing the probability of a given query, Figure 4.4 is depicting the runtime corresponding to each query tested. While the blue '+'s indicate the total runtime needed to compute the probability of the query, the red '*'s indicate how much time was spend integrating. Figure 4.4 then clearly shows that the majority of the query time is spend integrating over the weight function. Furthermore we see that the execution time decreases as the interval for our variable X in the query is decreasing. This corresponds to the integral having a tighter bound, and thus resulting in a faster runtime. Overall we observed that for each query only 4 integrations (or 4 calls of the *scipy.integrate.nquad* method) have to be carried out on average, due to reusing computed volumes.

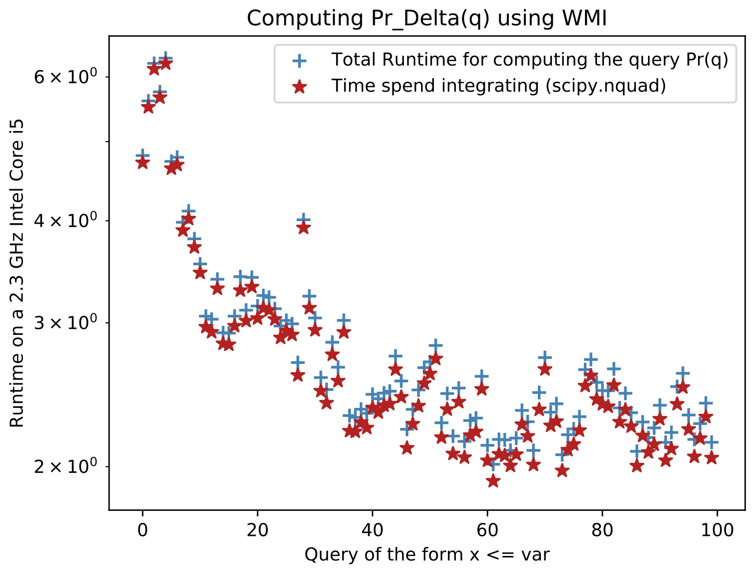


Figure 4.4: Runtime of Computing the Probability Pr(q) using WMI

Finally in Figure 4.5 a comparison for computing the probability of query q between our pipeline and wmi-pa[5] by [Sanner et al., 2012] is given. Here we can see that the algorithm by [Sanner et al., 2012] still outperforms the one we wrote by taking about half the time for a given query. However, as we have seen in Figure 4.4, most of the execution time of our algorithm is spend on doing 4 integrations only, thus we believe the next step here would be to test a different method of integration. As a final remark it should be noted that we sadly did not have time to test the scalability of the algorithms, as this is where the strength of our algorithm actually lies.

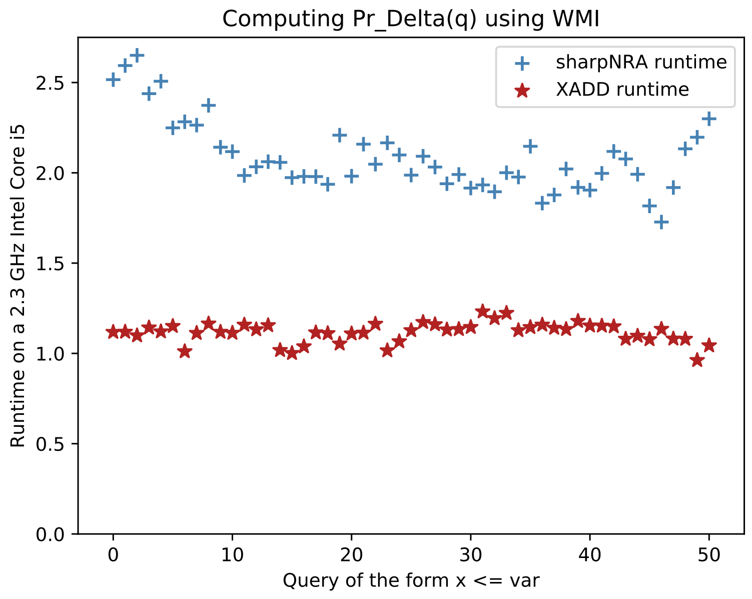


Figure 4.5: Runtime of Computing the Probability Pr(q) Comparison

[5]https://github.com/unitn-sml/wmi-pa

Chapter 5

Discussion and Conclusion

Over the course of this paper we introduced different algorithms for querying SDDs and went on to use this methods for probabilistic inference by predicate abstraction and WMI. The algorithms for querying SDD showed promising results, competing with the UCLA SDD library, when it comes to model counting. That being said, the current implementation of model counting is running on a single thread, where we would expect a drastic increase in performance implementing a multi-threaded algorithm. Furthermore, no method has been implemented for weighted model counting as an extension. This however would prove to be a rather simple extension, that could use the formulation for Model Enumeration as a basis.

When it comes to the performance of Model Enumeration we demonstrated the behavior of two different algorithms, one storing the models in RAM the other on the disk. As already mentioned computing the models of a given node, both algorithms have to load the models of the elements of that node into memory before combining them in the appropriate way. This leads to an limitation on the size of the SDD that can be queried due to the limitations of the RAM of the computer used to run the algorithm. Here we would suggest implementing batch processing, making it possible to solve this issue effectively. When it comes to the runtime performance of the ME algorithm, we have seen that the version storing model on the disk, it dramatically slower as read-/write operations tend to be very slow for HDD disks. A very simple, but expensive solution for this issue is using a computer with an SDD hard drive, improving the runtime dramatically. Furthermore, similar to the MC algorithm the ME also runs on a single thread at this point, where a parallelized version is expected to improve runtime considerably. Implementing this, should be relatively straight forward in practice due to the structural decomposability property of SDDs.

In Chapter 4 we went on to introduce a novel way of doing Model Integration (or sharpNRA) and Weighted Model Integration using SDDs as the underlying querying language. To recap, the concept here is to abstract a given $\mathcal{NRA}$ KB Δ into a propositional KB with corresponding propositional variable mappings. The abstracted KB can then be compiled into an SDD and queried as such, making it possible to use the algorithms proposed in Chapter 3 to retrieve the satisfying models efficiently. Finally we use these models to calculate the MI or WMI based on the formulation introduced by

[Belle et al., 2015], which can then be used to calculate the probability of a given query. When it comes to the results demonstrated for this pipeline we see that sharpLRA or MI, is a very inaccurate approximation of the WMI, and should generally not be used as such. However computing the sharpLRA of a given formula is in itself interesting as well, as a property of the formula, especially considering the very efficient implementation.

When it comes to WMI, the proposed pipeline makes it possible to compute this value very accurately but still rather slow. As our results have shown this is mainly due to the integration, which is responsible for most of the execution time needed to retrieve the result. Thus it would be very interesting to try a different method for integration, ideally a symbolic formulation such that we can compute the exact integral. However, needing an average of 3 seconds for computing the probability of a given formula, the proposed pipeline is still showing promising results, especially considering that about 98% of that time is spend integrating.

That being said the main limitation of the pipeline, and the main focus of future work should be on lifting the restrictions imposed on the input. In practice we see that the restriction on literals being of the from $l = (a \leq X_x \leq b)$ where $a, b \in \mathbb{R} \cup \{-inf, inf\}$ is very limiting with respect to possible use cases. For example, when it comes to symbolic programming and decision-theory planning problems we see that very few examples actually comply with such a restriction on literals. While we did not have time to further research this problem a possible solution might comprise of using multivariate functions as the bounds of integration.

Finally it would be very interesting to see how well this formulation applies to real world data sets and problems. As the mayor part of this project was spend on the theoretical formulation and practical implementation, use cases and real world applications had to be neglected due to the limited time of the project. However, we believe that the results demonstrated are interesting by themselves, giving an indication of the performance and usefulness of the proposed formulation and method.

Bibliography

[Bekker et al., 2015] Bekker, J., Davis, J., Choi, A., Darwiche, A., and Van den Broeck, G. (2015). Tractable learning for complex probability queries. In *Advances in Neural Information Processing Systems*, pages 2242–2250.

[Belle et al., 2015] Belle, V., Passerini, A., and Van den Broeck, G. (2015). Probabilistic inference in hybrid domains by weighted model integration. In *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2770–2776.

[Biere et al., 2009] Biere, A., Heule, M., and van Maaren, H. (2009). *Handbook of satisfiability*, volume 185. IOS press.

[Bodlaender and Möhring, 1993] Bodlaender, H. L. and Möhring, R. H. (1993). The pathwidth and treewidth of cographs. *SIAM Journal on Discrete Mathematics*, 6(2):181–188.

[Chavira and Darwiche, 2008] Chavira, M. and Darwiche, A. (2008). On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799.

[Chistikov et al., 2017] Chistikov, D., Dimitrova, R., and Majumdar, R. (2017). Approximate counting in smt and value estimation for probabilistic programs. *Acta Informatica*, 54(8):729–764.

[Choi and Darwiche, 2013] Choi, A. and Darwiche, A. (2013). Dynamic minimization of sentential decision diagrams. In *AAAI*.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM.

[Darwiche, 2003] Darwiche, A. (2003). A differential approach to inference in bayesian networks. *Journal of the ACM (JACM)*, 50(3):280–305.

[Darwiche, 2011] Darwiche, A. (2011). Sdd: A new canonical representation of propositional knowledge bases. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 819.

[Darwiche and Marquis, 2002] Darwiche, A. and Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17(1):229–264.

[Kisa et al., 2014] Kisa, D., Van den Broeck, G., Choi, A., and Darwiche, A. (2014). Probabilistic sentential decision diagrams. In *KR*.

[Koller and Friedman, 2009] Koller, D. and Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

[Liang et al., 2017] Liang, Y., Bekker, J., and Van den Broeck, G. (2017). Learning the structure of probabilistic sentential decision diagrams. In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*.

[Ma et al., 2009] Ma, F., Liu, S., and Zhang, J. (2009). Volume computation for boolean combination of linear arithmetic constraints. In *International Conference on Automated Deduction*, pages 453–468. Springer.

[Morettin et al., 2017] Morettin, P., Passerini, A., and Sebastiani, R. (2017). Efficient weighted model integration via smt-based predicate abstraction. *def*, 1(x1):x2.

[Pipatsrisawat and Darwiche, 2010] Pipatsrisawat, T. and Darwiche, A. (2010). A lower bound on the size of decomposable negation normal form. In *AAAI*.

[Poon and Domingos, 2011] Poon, H. and Domingos, P. (2011). Sum-product networks: A new deep architecture. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 689–690. IEEE.

[Sanner et al., 2012] Sanner, S., Delgado, K. V., and De Barros, L. N. (2012). Symbolic dynamic programming for discrete and continuous state mdps. *arXiv preprint arXiv:1202.3762*.