KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

GeSDD: Learning of tractable SDDs using genetic algorithms

Michiel Baptist

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Artificiële intelligentie

Promotoren:

Prof. dr. ir. Luc De Raedt
Prof. dr. Jesse Davis

Assessoren:

Dr. Bregt Verreet
Pedro Zuidberg Dos Martires

Begeleiders:

Dr. Jessa Bekker
Pedro Zuidberg Dos Martires

Academiejaar 2018 – 2019

Preface

I would like to take this opportunity to thank several people who have guided and helped me throughout the development of this thesis.

First and foremost, I would like to thank Prof. Dr. Ir. Luc De Raedt and Prof. Dr. Jesse Davis for their support, feedback and most importantly suggestions during the year.

I would also like to thank Pedro Zuidberg Dos Martires and Jessa Bekker for giving me the opportunity to show my progression throughout the thesis. Our weekly meetings sparked numerous discussions and resulted in copious amount of new inspiration each time. Additionally, I would like to thank them for being patient with me through my highs and lows. Numerous times I thought my thesis/research was going too slow. However, each time they managed to calm me down. They let me know that research is more about exploring dead ends than it is about obtaining results immediately.

Finally, I would like to thank those close to me. I would like to thank my family for their support and words of encouragement. I would especially like to thank some of my close friends Jorik, Bauwen and Aykut. Our numerous meals at Alma and games of presidenten managed to keep all of us sane during this hectic last year.

Michiel Baptist

Contents

Abstract

Markov logic networks are a staple of modern machine learning. They are intuitive and compact distributions over multivariate categorical variables. A problem arises in the form of performing inference. Indeed, performing inference in Markov logic networks is an exponential problem. This problem is especially prevalent when learning Markov logic networks, as learning them often requires performing inference multiple times.

Nonetheless, Markov logic networks can be learned efficiently in the form of learning a sentential decision diagram (SDD) simultaneously. SDDs are representations of Boolean functions. They allow for inference in polynomial time and in turn enable efficient learning of Markov logic networks. The size of the SDD determines efficiency of inference. This enables the punishment intractable Markov logic networks during the learning process.

This thesis presents GeSDD, a genetic algorithm for learning Markov logic networks simultaneously with its SDD. GeSDD presents a genetic crossover, several genetic mutations and a genetic fitness function. Additionally, two important heuristics are presented and their importance are shown. The first being dynamic SDD minimization, a way to reduce the size of SDDs. The second is the trimming heuristic, where non influential features of a Markov logic networks are removed.

Finally, this thesis presents an exploration of several of the parameters of GeSDD. Their impact on the learning process, convergence and final result are discussed.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

UAI	Uncertainty in Artificial Intelligence
MRF	Markov random field
MLN	Markov logic network
KC	Knowledge Compilation
SDD	Sentential Decision Diagram
MC	Model Counting
WMC	Weighted Model Counting
AMI	Average pairwise mutual information

Symbols

$\mathcal{R}$	The real numbers
$\mathcal{U}(a, b)$	The continuous uniform distribution over interval $[a, b]$.
$\mathcal{U}(S)$	The discrete uniform distribution over the set $S = \{e_1, \ldots, e_n\}$
$\mathcal{B}(p)$	The Bernoulli distribution with parameter p.

Chapter 1

Introduction

1.1 General introduction

Artificial intelligence and machine learning are becoming more and more prevalent in daily life. Take for example Netflix's or Spotify's recommendations, Tesla's self driving car or even fingerprint recognition on the phone in your pocket. These systems are not 100 % accurate because they are dealing with a large amount of uncertainty. Either from gathered data, where we are uncertain whether or not the data is representative. Or from the inherent randomness of life: there are always exceptions to the rule, even to this rule. Yet somehow artificial intelligent systems often manage to perform better than humans in many tasks. Take for example lip-reading, certainly an activity dealing with large amounts of uncertainty. Yet, a task where machines managed to outperform humans [2].

Dealing with uncertainty plays a central role within the AI community. Unsurprisingly, a large subdomain developed studying methods which are able to deal with uncertainty. This subdomain is usually referred to as *uncertainty in AI* (UAI). Within the field of UAI, probability distributions form an integral part of reasoning systems. They allow us to better define what we expect to see from data and with what degree of certainty. One such example are *Bayesian networks*. They are a type of graphical model which allows one to compactly define a distribution over a very large set of variables. However, they often lack the ability to reason over logically related data, such is often the case for data in databases. In 2004 *Markov logic networks* [34] were introduced, in an attempt to merge the two concepts. They allow for probabilistic reasoning over logical domains.

Often - as is the case for MLNs - when a multivariate distribution is dealing with a large number of variables, performing predictions breaks down. This is due to the fact that the number of possible configurations of variables grows exponentially in the number of variables. Take for example the problem of object detection in an image of $p = 50$ pixels. This is a distribution over 2^p possible configurations, either a pixel belongs to the object or it doesn't. If we assume the probability

calculation of a single configuration can be done in 0.000001 seconds, it would take approximately 35.7 years to find which pixels most likely belong to the object. This example demonstrates that while compact probability distributions are easy to work with, they still pose the problem of exponentiality.

As will be discussed in this thesis, learning MLNs usually consists of two sub-problems. The first step traditionally consists of structure learning. This step usually does not require one to perform inference on the MLN. The second step traditionally consists of parameter learning. This step, however, usually requires one to perform inference using the MLN. As performing inference, is an exponential problem, parameter learning is as well.

However, computer science is no stranger to exponential problems. Often, solutions are found by either making trade-offs or by intelligently leveraging properties of the problem. The *travelings salesman problem* is a prime example. A famously NP-hard problem, yet the now renowned Concorde[1] software elegantly tackles the problem nonetheless. Similarly, the domain of inference in graphical models did not remain underdeveloped. Many solutions have been proposed to tackle inference in graphical models nonetheless. There are generally two paradigms to inference in graphical models.

The first paradigm is concerned with *approximate inference*. Approximate inference will sacrifice prediction accuracy in favor of better running times. A nice example of approximate inference include sampling based methods. These methods make use of the law of large numbers and a sufficient[2] number of samples to estimate desired values.

The second paradigm considers *exact inference*. In exact inference one does not sacrifice prediction accuracy for a polynomial running time. Instead one attempts to deal with the exponentiality problem in alternative ways. The most notable method consists of reducing the inference problem to finding a compact representation of a Boolean function. This field of research is known as knowledge compilation (KC). Example representations include binary decision diagram (BDD)[1] and deterministic decomposable negation normal form (d-DNNF)[14]. In this thesis one such representation is used, namely the sentential decision diagram (SDD) introduced in 2011 by A. Darwiche[12]. These representations have properties which allow for tractable inference in graphical models.

1.2 Previous methods

By using smart methods, inference in graphical models becomes tractable. This in turn enables tractable learning of MLNs. MLNs are tremendously usefull, compact

[1] http://www.math.uwaterloo.ca/tsp/concorde/index.html

[2] Where an increase in samples usually means an increased accuracy.

and intuitive. Unsurprisingly, the field concerned with learning MLNs is quite substantial.

Many MLN learning methods have been proposed in the literature. As early as 2005 a search based algorithm for learning MLNs [24] was proposed. Shortly thereafter, many other learning algorithms were proposed [29, 17, 16]. However, they are not based on knowledge compilation methods. As a consequence, they often tend to consider the tractability aspect of learning an MLN as less important. Learning an intractable MLN is often of little use as it is hard to use in real time.

A more recent development in learning MLNs came in the form of simultaneously learning an *arithmetic circuit* (AC) alongside the MLN. This paradigm of learning MLNs was introduced in 2012 and is commonly referred to *circuit learning*[27] . With circuit learning, one is able to incorporate tractability more easily in the MLN learning process.

Arithmetic circuits are one type of circuit one can use. Another type of circuit one could learn alongside an MLN, is the aforementioned SDD. Learning MLNs alongside SDDs, was first proposed by J. Bekker et al. in 2015[4]. This paper presented LearnSDD. LearnSDD greedily builds an MLN and SDD, it does this based on how good (log-likelihood) and how tractable (size) the corresponding SDD is.

1.3 Thesis goals and contributions

Since LearnSDD is a greedy heuristic, the resulting MLN and SDD might be a local optimum. Resulting in a good, but maybe not optimal MLN and SDD. By using genetic algorithms one might escape these local optima in favor of a better solution.

Goals
This thesis explores learning of MLNs alongside its corresponding SDD from binary datasets. Additionally, the learning of the MLNs happen through a genetic algorithm. By doing so we hope to find compacter SDDs and better fitting MLNs. More specifically, this thesis seeks to answer this question:

Is it advantageous to learn MLNs alongside their SDD using genetic algorithms?

Contributions
This thesis presents the following contributions:

- GeSDD: a genetic algorithm for learning tractable MLNs alongside SDDs.
- A study of the effects of the parameters of GeSDD

Firstly, this thesis presents GeSDD. It presents several potential genetic operations for MLNs which go hand in hand with operations that can be performed on SDDs.

It suggests a genetic fitness function. Finally, several other heuristics such as SDD minimization and MLN trimming are discussed.

Secondly, the parameter space of GeSDD is explored. Their impact on MLN learning is discussed. Specifically, their impact on compactness, convergence and final result.

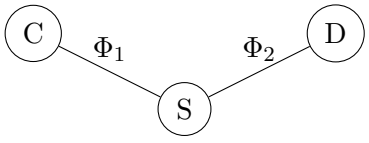
Chapter 2

Background

This section presents the necessary background knowledge. Firstly, a general introduction to MLNs will be provided and their inherent complexity problem will be highlighted. Next, this section provides an introduction to the concept of model counting and its relationship to inference in MLNs. The concept of SDDs will be introduced as a method of solving the model counting problem, and with it a couple of properties will be shown to be important. Finally, this section concludes with an overview of the genetic algorithm framework.

2.1 Markov Random Field

A *Markov random field* (MRF) or *Markov network* is a compact representation of a multivariate distribution. It can be represented as an undirected graph with m nodes, where each node represents a random variable. Every clique in the graph may have a so called *potential function* Φ_i associated with it. Figure 2.1 shows an example of a Markov network. In this figure the nodes C, S and D are random variables. The edges between the nodes Φ_1 and Φ_2 denote the potential functions. In this example they are tabular functions, however they may represent any function such that $\forall C \in \mathcal{D}_C, \forall S \in \mathcal{D}_S : \Phi_1(C, S) \geq 0$ with $\mathcal{D}_C$ the domain of the random variable C and $\mathcal{D}_S$ that of S.



C	S	Φ_1
0	0	10
0	1	2
1	0	0
1	1	5

S	D	Φ_2
0	0	15
0	1	2
1	0	1
1	1	10

Figure 2.1: An example of a Markov random field.

Intuitively, the potentials of the network define how strong a certain configuration of the associated variables is. For example if C is the Boolean random variable denoting the consumption of chicken soup and S is the Boolean random variable

C	S	D	$\prod_i \Phi_i(x)$	$P(\mathbf{X} = x)$
0	0	0	150	$\frac{150}{247}$
0	0	1	20	$\frac{20}{247}$
0	1	0	2	$\frac{2}{247}$
0	1	1	20	$\frac{20}{247}$
1	0	0	0	$\frac{0}{247}$
1	0	1	0	$\frac{0}{247}$
1	1	0	5	$\frac{5}{247}$
1	1	1	50	$\frac{50}{247}$
			Z =247	1.0

Table 2.1: Worked out probabilities for the example in figure 2.1.

denoting being sick. The strength of eating chicken soup and not being sick, i.e. $\Phi_1(C = 1, S = 0)$, is 0. While the strength of eating chicken soup and being sick, i.e. $\Phi_1(C = 1, S = 1)$, is 5. The reason for this intuitive interpretation follows next.

A Markov network defines a certain probability distribution over all m variables in the network. For example, if $m = 3$ and each variable is binary, an MRF defines a distribution over the outcome space $\{0, 1\}^3$. The outcome space is denoted as Ω. This distribution is based on the strengths of the potentials for a given assignment of variables. More specifically, the distribution is given as:

$$P(\mathbf{X} = x) = \frac{1}{Z} \prod_i \Phi_i(\mathbf{X}_{\Phi_i} = x_{\Phi_i}) \tag{2.1}$$

Here $\mathbf{X} = (X_1, \ldots, X_m)$ is a vector of the variables of the MRF and $x = (x_1, \ldots, x_m) \in \Omega$ is a binary vector of values. Furthermore, $\mathbf{X}_{\Phi_i}$ is the subset of $\mathbf{X}$ corresponding to the variables of Φ_i. Logically, x_{Φ_i} is the subset of values of x corresponding to the variables in to $\mathbf{X}_{\Phi_i}$. It is often easier to denote $\Phi_i(\mathbf{X}_{\Phi_i} = x_{\Phi_i})$ as $\Phi_i(x)$. Finally, Z is the normalization constant ensuring that an MRF indeed defines a distribution over its variables:

$$Z = \sum_{x' \in \Omega} \prod_i \Phi_i(x) \tag{2.2}$$

Here, the summation is over the outcome space Ω (i.e. assignment of $\mathbf{X}$). For example, table 2.1 shows the probability of each configuration of the example in figure 2.1.

Looking at equation 2.1, a problem immediately becomes apparent. Namely the summation over Ω, an exponential amount of possible configurations. In any application with a considerable amount of variables, performing inference becomes intractable.

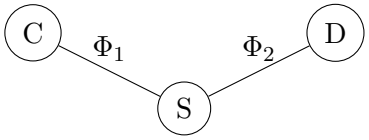
2.2 Markov Logic Networks (MLN)

When using an MRF the potentials are sometimes defined in tabular form as in the example in figure 2.1. However often one defines more structured potentials Φ. In

2004[34] the concept of a *Markov logic network* (MLN) was introduced as a way to combine probabilistic reasoning and logical reasoning. An MLN is an MRF where the potentials are defined as:

$$\Phi_i(x) = \exp\left(w_i f_i(x)\right) \tag{2.3}$$

Here $f_i \in \{0,1\}^m \rightarrow \{0,1\}$ are Boolean functions, usually referred to as *features*. The $w_i \in \mathcal{R}$ are real values, usually referred to as *weights*. For example instead of the potentials in the example in figure 2.1 one could define the potentials as in figure 2.2.



f_i	w_i
$f_1(C,S) = C \Leftrightarrow S$	1
$f_2(S,D) = S \Rightarrow D$	3

Figure 2.2: An example of a Markov logic network.

In the same way that an MRF defines a distribution, an MLN defines a distribution. By substituting Φ_i in equation 2.1 by equation 2.3 we find:

$$\begin{aligned} P(\mathbf{X} = x) &= \frac{1}{Z} \prod_i \exp\left(w_i f_i(x)\right) \\ &= \frac{1}{Z} \exp\left(\sum_i w_i f_i(x)\right) \end{aligned} \tag{2.4}$$

The normalization function Z can then be found by substituting $\Phi_i(x)$ from equation 2.3 into equation 2.2:

$$\begin{aligned} Z &= \sum_{x \in \Omega} \prod_i \exp(w_i f_i(x)) \\ &= \sum_{x \in \Omega} \exp(\sum_i w_i f_i(x)) \end{aligned} \tag{2.5}$$

Equation 2.4 and 2.5 fully define a Markov logic network. It can be considered as a collection of feature-weight pairs:

$$MLN = \{(f_1, w_1), \ldots, (f_k, w_k)\} \tag{2.6}$$

With k the number of weight-feature pairs. Usually, one includes the variables themselves as features in the MLN. For example, an MLN over the variables X_1 and X_2 will typically include $(f_1 = X_1, w_1)$ and $(f_2 = X_2, w_2)$. This thesis will refer to the *empty MLN* or *empty model* E over $\mathbf{X} = \{X_1, \ldots, X_m\}$ as the MLN which contains exactly all features of the form $f_i = X_i$:

$$E = \{(X_1, w_1), \ldots, (X_m, w_m)\} \tag{2.7}$$

As an example, the MLN in figure 2.2 can be formally defined as $M = \{(C \Leftrightarrow S, 1), (S \Rightarrow D, 3)\}$. Table 2.2 shows the probabilities of each configuration of the example in figure 2.2.

C	S	D	$f(C,S)$	$f(S,D)$	$\exp(\sum_c w_c f_c(x_c))$	$P(\mathbf{X} = x)$
0	0	0	1	1	e^{1+3}	$\frac{e^{1+3}}{3e^4+3e^3+e^1+1}$
0	0	1	1	1	e^{1+3}	$\frac{e^{1+3}}{3e^4+3e^3+e^1+1}$
0	1	0	0	0	e^0	$\frac{e^0}{3e^4+3e^3+e^1+1}$
0	1	1	0	1	e^3	$\frac{e^3}{3e^4+3e^3+e^1+1}$
1	0	0	0	1	e^3	$\frac{e^3}{3e^4+3e^3+e^1+1}$
1	0	1	0	1	e^3	$\frac{e^3}{3e^4+3e^3+e^1+1}$
1	1	0	1	0	e^1	$\frac{e^1}{3e^4+3e^3+e^1+1}$
1	1	1	1	1	e^{1+3}	$\frac{e^{1+3}}{3e^4+3e^3+e^1+1}$
					$3e^4 + 3e^3 + e^1 + 1$	1.0

Table 2.2: Worked out probabilities for the example in figure 2.2.

Intuitions for MLNs

From the MLN equations 2.4 and 2.5 a couple of derivations can be made, making MLNs more intuitive. This intuition will become quite important in section 4, particularly in sections 4.7 and 4.8.2.

Firstly, an MLN allows every possible configuration of variables x to have a non zero probability:

$$\forall x \in \Omega : P(\mathbf{X} = x) \propto \exp\left(\sum_i x_i f_i(x_i)\right) > 0 \tag{2.8}$$

This is due to the fact that exp(.) is a positive function in its domain. Secondly, when the weight w_i of a feature f_i approaches the limit to infinity, the configurations x not satisfying f_i do have a zero probability:

$$\forall x \in \Omega : f_i(x) = 0 : \lim_{w_i \to \infty} P(\mathbf{X} = x) = 0 \tag{2.9}$$

The distribution defined by the MLN converges to the configurations which satisfy f_i. This means that the feature f_i is a non-probabilistic fact. Thirdly, when the weight w_i of a feature f_i tends to minus infinity, all configurations satisfying f_i have a probability of 0:

$$\forall x \in \Omega : f_i(x) = 1 : \lim_{w_i \to -\infty} P(\mathbf{X} = x) = 0 \tag{2.10}$$

The distribution defined by the MLN converges to the configurations which do not satisfy f_i. Finally, when the weight w_i of a feature f_i is equal to 0, the distribution

of the MLN is not affected:

$$\begin{aligned} P(\mathbf{X} = x) &= \frac{\exp(\sum_j w_j f_j(x_j))}{\sum_{x'} \exp(\sum_j w_j f_j(x'_j))} \\ &= \frac{\exp(\sum_{j \neq i} w_j f_j(x_j) + 0)}{\sum_{x'} \exp(\sum_{j \neq i} w_j f_j(x'_j) + 0)} \end{aligned} \tag{2.11}$$

Consequently, for any MLN $M = \{(f_1, w_1), \ldots, (f_i, 0), \ldots, (f_n, w_n)\}$ it holds that:

$$\begin{aligned} \langle M \rangle &= \langle \{(f_1, w_1), \ldots, (f_i, 0), \ldots, (f_n, w_n)\} \rangle \\ &= \langle \{(f_1, w_1), \ldots, (f_n, w_n)\} \rangle \\ &\equiv \langle M' \rangle \end{aligned} \tag{2.12}$$

Where $\langle M \rangle$ and $\langle \{.\} \rangle$ denote the distribution induced by MLN M.

From the above reasoning, we may find the following intuition: as the weight w_i of a feature f_i approaches 0, the feature f_i captures almost no information about the distribution. As $|w_i|$ is larger, the feature captures more information about the distribution. In fact as $|w_i| \to \infty$, feature f_i can be considered as a logical fact or impossible. As such, one might find that features f_i with a high absolute weight are "more important" than features with a weight closer to 0.

Learning of MLNs
MLNs define a distribution over a set of Boolean random variables. This means that given a dataset $\mathcal{X}$, consisting of Boolean data, we could attempt to learn an MLN M such that the distribution $\langle M \rangle$ reflects the empirical data $\mathcal{X}$.

Learning MLNs consist of two parts. The first part are the features $f = (f_1, \ldots, f_k)$. The second part are the weights $w = (w_1, \ldots, w_k)$. It is therefore traditional to learn MLNs iteratively in two separate steps. The first part consists of learning f. Commonly this step is referred to as *structure learning*. The second step then involves the weights w and is usually referred to as *weight learning*. As these weights largely determine the distribution of the MLN, learning them usually requires performing inference, i.e. calculate $P(x)$ for some x. It is therefore necessary to be able to compute probabilities efficiently. Section 2.3, 2.4, 2.5 and 2.6 delve deeper into how this can be achieved.

2.3 Model Counting and Weighted Model Counting

Here the concepts of *model counting* (MC) and *weighted model counting* (WMC) are presented. They are a crucial concept in the field of exact inference in graphical models as can be seen in sections 2.4 and 2.6. It is one step in enabling efficient inference and learning MLNs as a result. However, (weighted) model counting may be considered as a separate problem.

2.3.1 Model Counting (MC)

The model count of a Boolean function $f : \{0,1\}^m \to \{0,1\}$ is the number of models this function has. A model of a Boolean function f is an assignment of variables x such that f is satisfied. More concretely this means:

$$MC(f) = \Big|\{x \in \Omega | f(x) = 1\}\Big| \tag{2.13}$$

Where $\Omega = \{0,1\}^m$ is the domain of the Boolean function f. As an example, the model count of $f(a,b) = a \wedge b$ is $MC(f) = |\{(a,b)\}| = 1$, while that of $g(a,b) = a \vee b$ is $MC(g) = |\{(a,b),(a,\neg b),(\neg a,b)\}| = 3$.

2.3.2 Weighted Model Counting (WMC)

Weighted model counting is essentially model counting, but with a slight twist. In WMC every assignment of a variable has an associated weight i.e. $W(X_i = x_i)$, where x_i is an assignment to variable X_i. For example, if X_i is a Boolean variable then $W(X_i = 0)$ might be 2 and $W(X_i = 1)$ might be -1. Based on these weights, a numeric value is computed. Concretely, the weighted model count of a given Boolean function $f : \{0,1\}^m \to \{0,1\}$ and weighted assignments, is the sum of the weights of the models this function has. The weight of a model of f is the product of the weights of the assignments of $\mathbf{X} = (X_1, \ldots, X_m)$. The weighted model count then is simply the sum of the weights of all models. Written as a formula the weighted model count of f is:

$$\begin{aligned} WMC(f) &= \sum_{x \in \Omega : f=1} W(\mathbf{X} = x) \\ &= \sum_{x \in \Omega : f=1} \prod_{i=1}^{m} W(X_i = x_i) \end{aligned} \tag{2.14}$$

Here, $\Omega = \{0,1\}^m$ is the domain of f and $f = 1$ is shorthand for $f(x) = 1$. We usually refer to the weights of the assignment $X_i = x_i$ as $W(x_i)$ if no confusion is possible. As an example, the weighted model count of $g(a,b) = a \vee b$ given $W(a) = W(b) = 2$, $W(\neg a) = 3$ and $W(\neg b) = 1$:

$$\begin{aligned} WMC(g) &= W((a,b)) + W((\neg a,b)) + W((a,\neg b)) \\ &= W(a)W(b) + W(\neg a)W(b) + W(a)W(\neg b) \\ &= 4 + 6 + 2 = 12 \end{aligned} \tag{2.15}$$

From this example, one could note that we can perform a slight optimization of this calculation. While in this example we used 3 additions and 3 multiplications, we could for example factor out $W(a)$ or $W(b)$. This would result in 3 additions and 2 multiplications. In smaller examples like these, it will not save a great deal of computation. However, as a Boolean function uses more variables, the amount of models it has might grow exponentially. As such, in order to compute the WMC of

a more complex function, one has to leverage its structure somehow. In order to do so, a whole range of methods have been developed. Section 2.5 delves deeper into one such knowledge compilation method.

2.4 Solving inference in MLNs with WMC

While the WMC problem is an interesting problem on its own, there is a tight relationship between WMC and performing inference in MLNs. This section highlights that relationship.

Earlier it was noted that performing inference in MLNs is intractable. However, by exploiting structure in the MLNs, it is still remarkably often possible to do inference anyways. One of the ways to exploit structure, is by reducing the inference problem in MLNs to a WMC problem.

This section shows that performing inference in MLN (i.e. computing Z from equation 2.5) can be reduced to calculating the WMC of a derived theory Δ. Concretely, given an MLN M with partition function Z as defined in equation 2.5, for the derived theory Δ is should hold that:

$$Z = WMC(\Delta) \tag{2.16}$$

In the literature, Δ is commonly referred to as the *knowledge base*. By doing so, the highly researched and well developed methods of WMC computation can be leveraged.

There exists many methods of constructing Δ. In the following, a frequently used and intuitive method is presented. It is also used throughout this thesis. This derivation of Δ was presented by A. Darwiche and P. Marquis [6].

Constructing the knowledge base Δ
Looking at the form of Z in equation 2.5, there is a striking similarity with the form of the WMC in equation 2.14. Indeed, in both cases there is a summation over a product. While the WMC is summing over the models of f, the partition function Z is summing over all possible configurations of $\mathbf{X}$, i.e. over Ω. While constructing the theory Δ, we construct a one-to-one mapping between the models of Δ and the possible configurations of $\mathbf{X} = (X_1, \ldots, X_m)$. Such a Δ can be constructed as follows:

Step 1: For every variable X_i, introduce a new *indicator variable* I_{X_i}. For every feature[1] f_i in the MLN, introduce a new *attribute variable* F_i.

Step 2: For every feature f_i in the MLN, introduce a new equivalence $F_i \Leftrightarrow f_i'(I_X)$. Here, f_i' is the same Boolean function as f_i where every X_i is replaced by its indicator variable I_{X_i}. The knowledge base Δ is simply the conjunction of the new

[1] This does not apply to features of the form $f_i = X_i$.

equivalences:

$$\Delta = \bigwedge_{i=1}^{k} F_i \Leftrightarrow f_i'(I_X) \tag{2.17}$$

Where k denotes the number of features f_i in the MLN.

Step 3: For every indicator variable I_{X_i} choose weight 1, i.e. $W(I_{X_i}) = 1$. For every indicator variable $\neg I_{\mathbf{X}_i}$ choose weight 1, i.e. $W(\neg I_{X_i}) = 1$. For every attribute variable F_i choose weight $\exp(w_i)$, i.e $W(F_i) = \exp(w_i)$. Finally, for every attribute variable $\neg F_i$ choose weight 1, i.e. $W(\neg F_i) = 1$.

Indeed, applying step 1 and 2, results in a one-to-one mapping of the possible configurations of $\mathbf{X}$ and the models of Δ. As an example, applying the three steps to the example in figure 2.2 results in the following:

Step 1: The indicator variables I_C, I_S and I_D are introduced. Additionally, the attribute variables F_1 for $f_1(C, S) = C \Leftrightarrow S$ and F_2 for $f_2(S, D) = S \Rightarrow D$ are introduced.

Step 2: Replacing C, S in $f_1(C, S)$ by I_C and I_S results in $I_C \Leftrightarrow I_S$. Replacing S, D in $f_2(S, D)$ by I_S and I_D results in $I_S \Rightarrow I_D$. The conjunctions of equivalences results in Δ:

$$\Delta = (F_1 \Leftrightarrow (I_C \Leftrightarrow I_S)) \wedge (F_2 \Leftrightarrow (I_S \Rightarrow I_D)) \tag{2.18}$$

Step 3: We obtain the weights as in table 2.3.

Literal	Weight $W(I_j)$
I_C, $\neg I_C$	1
I_S, $\neg I_S$	1
I_D, $\neg I_D$	1
F_1	$\exp(1)$
F_2	$\exp(3)$
$\neg F_1$, $\neg F_2$	1

Table 2.3: Example of transforming the MLN in example 2.2 to Δ.

Indeed, one can now verify the equivalence. This method of transforming the problem of the computation of Z from equation 2.5, to the problem of WMC as in equation 2.14 is commonly referred to as the *encoding*. The encoding of an MLN M is denoted as $enc(M)$.

2.5 Sentential decision diagram

The result of reducing an inference problem to a WMC is already a step in the right direction. However, as mentioned in section 2.3, computing the WMC of a theory is no trivial task. In fact it remains a # P-complete problem [10] and as such, tackling it requires smarter methods which leverage the structure present in the knowledge base Δ.

Two main paradigms developed around (weighted) model counting. The first is based on DPLL-like SAT methods. Famous examples of these kinds of methods include Cachet [36] and sharpSAT [38]. Here, the models of a theory are enumerated by iteratively searching for satisfying assignments of the knowledge base. They are highly optimized and use methods such as clause learning and component caching.

The second main paradigm consists of *knowledge compilation* methods. Here the WMC is not computed by iteratively searching for models of the knowledge base. Instead, one tries to find a representation of the knowledge base in a way that allows (weighted) model counting. Indeed, a theory has more than a single way of being written. The function $f = a \wedge (b \vee c)$ may just as well be represented as $f = (a \wedge b) \vee (a \wedge c)$. While logically equivalent, they are not equal. They belong to different classes of Boolean functions. In fact, some representations allow for efficient model counting while others don't. In order to perform model counting, one finds an equivalent representation of the knowledge base in a different class of Boolean functions. This is usually referred to as *compiling* the knowledge base to a *target language*. Examples of such target languages include conjunctive normal form (CNF), ordered binary decision diagrams (OBDDs) and deterministic decomposable negation normal form (d-DNNF)[14].

While considering target languages, three properties are considered important[14]. Firstly the succinctness, which determines how big the representation is. Secondly which operations can be performed in polynomial time. Lastly, which queries may be performed in polynomial time.

The remainder of this section introduces a crucial concept for this thesis. Namely the sentential decision diagram (SDD), introduced in 2011 by A. Darwiche [12]. SDDs are one such target language for Boolean functions. The advantage of SDDs will become clear later on and lies in the ability to perform operations in polynomial time.

2.5.1 Definition

An SDD is a recursive structure, it is either a *constant SDD*, a *literal SDD* or a *composite SDD*. Each type of SDD is introduced next.

Constant SDDs
A constant SDD is as the name implies, constant. It is not a recursive structure and

is called a *terminal* SDD. A constant SDD α can be one of the following SDDs:

$$\begin{aligned} \alpha &= \top \\ \alpha &= \bot \end{aligned} \tag{2.19}$$

This is the SDD equivalent of *true* and $false$ respectively. To denote which Boolean function an SDD represents we write $\langle \alpha \rangle$. For example the SDDs from equation 2.19 represent $\langle \top \rangle = true$ and $\langle \bot \rangle = false$. Finally, the size of a constant SDD is 0.

Literal SDDs
A literal SDD consists of a literal, and is also not a recursive structure. As it is not recursive, it is also called a terminal SDD. A literal SDD α is denoted as:

$$\begin{aligned} \alpha &= X \\ \alpha &= \neg X \end{aligned} \tag{2.20}$$

A literal SDD represents the Boolean function $\langle X \rangle = X$ and the Boolean function $\langle \neg X \rangle = \neg X$. Finally, the size of a literal SDD is 0.

Composite SDDs
A composite SDD α is a collection of SDD pairs. It is denoted as:

$$\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\} \tag{2.21}$$

Here the p_i are SDDs and are referred to as *primes*, the s_i are SDDs and are referred to as *subs*. A composite SDD represents the Boolean function

$$\langle \{(p_1, s_1), ..., (p_n, s_n)\} \rangle = \bigvee_{i=1}^{n} \langle p_i \rangle \wedge \langle s_i \rangle \tag{2.22}$$

Using the composite SDD, complex SDDs can be constructed and represent complex Boolean functions. Finally the size of a composite SDD is the sum of the sizes of its decompositions and its number of elements n.

2.5.2 Vtrees, compilation and minimization

Vtrees and compilation
By the definition of SDDs, it does not immediately become clear how to write a given Δ as an SDD α, i.e. $\Delta \equiv \langle \alpha \rangle$. For that, another structure is required. This being the so called *vtree*. Originally introduced in 2008 [33], its definition is as follows. Given a set of variables $\mathbf{X} = (X_1, \ldots, X_m)$, a vtree for $\mathbf{X}$ is a full binary tree whose leaves are in a one-to-one correspondence to the variables in $\mathbf{X}$. As an example, two vtrees for $\mathbf{X} = (A, B, C, D)$ can be seen in figure 2.3.
Given a vtree v, then v^l denotes the vtree rooted in the *left* child of v. Similarly v^r then denotes the vtree rooted in the *right* child of v. When a vtree v has no children, we usually make no distinction between v and the variable corresponding to the leaf

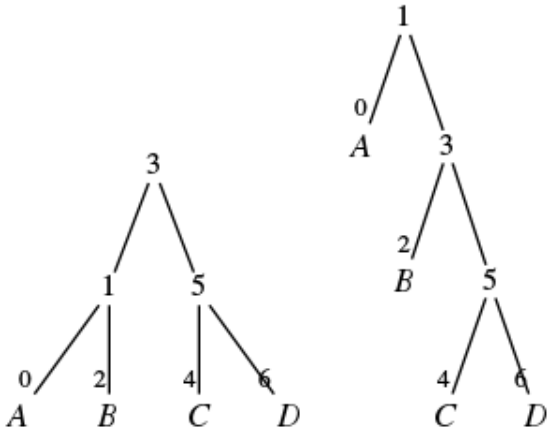


Figure 2.3: Two example vtrees for the variables $\mathbf{X} = (A, B, C, D)$.

v. For example the vtree rooted at 0 in figure 2.3 in both cases is $v = A$. While the vtree rooted in 5 in both cases is $v = (C, D)$. Quite interestingly, a vtree over variables $\mathbf{X}$ will partition the variables in two groups. This property is quite useful for constructing SDDs.

Next, a crucial notion that connects SDDs and vtrees follows. For a given SDD α and a vtree v, α *respects* v if [12]:

1. $\alpha = \bot$ or $\alpha = \top$
2. $\alpha = X_i$ or $\alpha = \neg X_i$ and v is a leaf with $v = X_i$
3. $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$ and $p_1, \ldots, p_n$ respect v^l and $s_1, \ldots, s_n$ respect v^r. Additionally, $\langle p_1 \rangle, \ldots, \langle p_n \rangle$ form a partition.

Using the notion of respecting vtrees, some nice properties of SDDs can be proven. One property is particularly interesting, namely the following property.

Theorem: Given two SDDs α and β such that they are *compressed*, *trimmed* and respecting the same vtree. Then $\langle \alpha \rangle = \langle \beta \rangle$ if and only if $\alpha = \beta$.

The definitions of compressedness and trimmedness will not be discussed here. However, the full definitions can be found in [12]. This property is quite useful. Intuitively it means that if two SDDs define the same function, respecting the same vtree, they are the same SDDs. In other words, given a vtree v and a Boolean function f, there exists exactly one trimmed and compressed SDD α such that $\langle \alpha \rangle = f$. Consequently, there exists a function

$$S : \mathcal{V} \times \mathcal{F} \to \mathcal{A} : (v, f) \mapsto S(v, f) \tag{2.23}$$

Such that $\langle S(f, v) \rangle = f$. Here, $\mathcal{V} = \{v | v \text{ is a vtree over } \mathbf{X}\}$ is the space of all vtrees over $\mathbf{X}$. $\mathcal{F} = \{f | f \text{ is a Boolean function over } \mathbf{X}\}$, is the set of all Boolean functions

over $\mathbf{X}$. Finally, $\mathcal{A} = \{\alpha | \alpha \text{ is an SDD over } \mathbf{X}\}$ is the space of all SDDs over $\mathbf{X}$.

This property, among others, allows for two crucial results:

1. **Result 1:** Boolean operations on SDDs can be done in polynomial time. Conretely, given α, β respecting vtree v and $\circ$ a Boolean operator. The SDD γ such that it respects vtree v and $\langle\gamma\rangle = \langle\alpha\rangle \circ \langle\beta\rangle$ can be found in $\mathcal{O}(|\alpha||\beta|)$. Usually, when no confusion is possible, γ will be written simply as $\gamma = \alpha \circ \beta$

2. **Result 2:** There exists an algorithm[2] given a vtree v, a Boolean function f both over $\mathbf{X}$, which can computes an SDD α for f.

The second point allows us to *compile* any function f to an SDD, given a vtree v. As a result, we can compile the knowledge base Δ constructed in section 2.4.

A note on minimization
While any Boolean function f can be compiled to an SDD given vtree v, not all vtrees are created equal. In fact, when compiling f using vtree v_1 or v_2, we might find vast differences in the sizes $|\alpha_1|$ and $|\alpha_2|$ in SDDs $S(f, v_1) = \alpha_1$ and $S(f, v_2) = \alpha_2$. It is therefore important to choose your vtree wisely. As discussed in section 2.6, the size of an SDD is quite important. Additionally, above it was mentioned that performing Boolean operations $\circ$ on two SDDs is polynomial in the size of both SDDs. Therefore A. Choi and A. Darwiche introduced a minimization algorithm for SDDs[8]. It is based on exploring the space of vtrees $\mathcal{V}$ in a smart manner, by rotating and swapping subtrees. The algorithm will not be covered here, as this would lead us too far.

2.5.3 Graphical representation

Often it's easier to represent an SDD using a graphical notation. Graphically an SDD is represented as a directed acyclic graph with two types of nodes. An example of an SDD can be seen in figure 2.4.

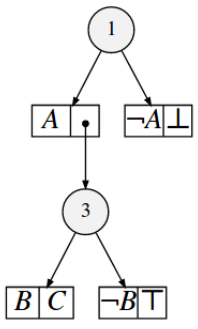


Figure 2.4: Example of the graphical notation of an SDD.

[2]Originally presented in [12], which this text will not cover. However, the knowledge of its existence suffices.

Here, every circle represents a composite SDD and every rectangle represents an SDD prime-sub pair. The SDD in the left box represents the prime and the SDD in the right box represents the sub. The number inside the circle denotes which vtree the SDD respects. The size of an SDD is the same as the number of edges in its graphical representation. For example the circle denoted with 1 in figure 2.4 represents the SDD.

$$\alpha_1 = \{(A, \alpha_2), (\neg A, \bot)\} \tag{2.24}$$

The circle denoted with 3 in figure 2.4 represents the SDD

$$\alpha_2 = \{(B, C), (\neg B, \top)\} \tag{2.25}$$

By recursively using equation 2.22 we find that the SDD represented in the example in figure 2.4 represents the Boolean function

$$\begin{aligned} \langle \alpha_1 \rangle &= (A \wedge \langle \alpha_2 \rangle) \vee (\neg A \wedge \bot) \\ &= (A \wedge ((B \wedge C) \vee (\neg B \wedge \top))) \vee (\neg A \wedge \bot) \end{aligned} \tag{2.26}$$

This can then be simplified to:

$$\langle \alpha_1 \rangle = A \wedge (\neg B \vee C) \tag{2.27}$$

Even though the function in equation 2.26 is equivalent to the function in equation 2.27, 2.26 is an SDD and 2.27 is not.

2.5.4 SDD properties

This section discusses several SDD properties in more detail. It is not necessary to understand these in order to understand this thesis. They do, however, provide more insight into SDDs. This will become especially useful for section 2.6.

Property 1: the primes and subs do not share variables
Let α be an SDD defined as in equation 2.22. Let the SDD be defined over the variables in $\mathbf{Z}$. The primes $p_1, ..., p_n$ are defined over the same set of variables $\mathbf{X} \subset \mathbf{Z}$. This means that any prime p_i can be written as $p_i(\mathbf{X})$, and all variables of p_i are in $\mathbf{X}$. The subs are also defined over the same set of variables $\mathbf{Y} = \mathbf{Z} \setminus \mathbf{X}$. In addition the two subsets $\mathbf{X}$ and $\mathbf{Y}$ form a partition of $\mathbf{Z}$.

1. $\mathbf{X}$ and $\mathbf{Y}$ share no variables, i.e. $\mathbf{X} \cap \mathbf{Y} = \emptyset$
2. $\mathbf{X}$ and $\mathbf{Y}$ forms $\mathbf{Z}$, i.e. $\mathbf{X} \cup \mathbf{Y} = \mathbf{Z}$

This property is particularly interesting, since for any configuration of $\mathbf{Z}$ whether or not the prime $p_i(\mathbf{X})$ is satisfied, is independent from whether or not the sub $s_i(\mathbf{Y})$ is satisfied. This also provides more intuition as to how a Boolean function f is compiled given a vtree v. The vtree v decides how the variables are recursively partitioned.

Property 2: the primes form an X-partition over X
Let α be an SDD as defined in equation 2.22. Then this SDD is strongly deterministic. Intuitively when a decomposition is strongly deterministic, for every configuration of $\mathbf{X}$ there is exactly one prime satisfied. More concretely this means:

1. Every prime is consistent. This means that every prime $p_i(\mathbf{X})$ is satisfiable by some assignment of $\mathbf{X}$.
$$\forall i : \exists X : p_i(X) = true \tag{2.28}$$

2. Every pair of distinct primes is mutually exclusive. Intuitively this means that no pair of primes may be true at the same time.
$$\forall i, j : i \neq j : p_i \wedge p_j = false \tag{2.29}$$

3. The disjunction of the primes is valid. Intutively this means that for any configuration of $\mathbf{X}$, there is at least one prime p_i such that this prime is satisfied $p_i(\mathbf{X}) = true$.
$$\forall \mathbf{X} : \bigvee_{i=1}^{n} p_i(\mathbf{X}) = true \tag{2.30}$$

Rule 2 and 3 means that there is exactly one prime satisfied for any configuration of $\mathbf{X}$. Rule 1 means that every prime has at least one configuration of $\mathbf{X}$ such that it is satisfied. This property is especially important for section 2.6.

Property 3: the size of SDD α remains exponential: $|\alpha| = \mathcal{O}(n2^w)$
The compilation of functions f may result in an SDD α with a small size $|\alpha|$. However, this is not always the case. In [12] it was shown that the worst case size for an SDD is still exponential. More specifically, it is exponential in the tree width w of the vtree.

Property 4: conditioning SDDs
In section 2.5.2, result 1 showed that arbitrary Boolean operations $\circ$ can be performed on SDDs α and β. This same result allows one to condition an SDD α on any variable X_i in the SDD. Indeed, assume that α is an SDD containing the variable X_i. Conditioning on X_i can be done as follows:

$$\beta = \alpha \wedge X_i \tag{2.31}$$

Using this logic, an SDD representing an encoded MLN can be conditioned into an SDD representing the encoding of a subset of the MLN. Concretely, given an MLN $M = \{(f_1, w_1), \ldots, (f_i, w_i), \ldots, (f_k, w_k)\}$ and SDD α such that $enc(M) = \langle \alpha \rangle$. The SDD α' of a subset MLN $M' = \{(f_1, w_1), \ldots, (f_{i-1}, w_{i-1}), (f_{i+1}, w_{i+1}) \ldots, (f_k, w_k)\} \subset M$ such that $enc(M') = \langle \alpha' \rangle$ can be found by conditioning on the attribute variables of the encoding as follows:

$$\alpha' = (\alpha \wedge F_i) \vee (\alpha \wedge \neg F_i) \tag{2.32}$$

As an example, consider $M = \{(C \Leftrightarrow S, 1), (S \Rightarrow D, 3)\}$ with $\Delta = (F_1 \Leftrightarrow (I_C \Leftrightarrow I_S)) \wedge (F_2 \Leftrightarrow (I_S \Rightarrow I_D))$ from the example in section 2.4. Assume $\Delta = \langle\alpha\rangle$. Say we wish to find the SDD of the encoding of $M' = \{(S \Rightarrow D, 3)\} \subset M$. This can be done as follows:

$$\alpha' = (\alpha \wedge F_1) \vee (\alpha \wedge \neg F_1) \tag{2.33}$$

Intuitively, this allows us to remove features of an MLN and its corresponding SDD without having to recompile the new SDD from scratch. This property is crucial for the entirety of section 4.

As equation 2.32 is quite cumbersome, the notation of conditioning in this thesis is denoted as follows. Assume M is an MLN $M = M' \cup M''$ with SDD α such that $\langle\alpha\rangle = enc(M)$, then the SDD α' such that $\langle\alpha'\rangle = enc(M')$ is denoted as $\alpha|M''$.

2.6 WMC reduced to SDD compilation

The most important feature of SDDs, or any circuit such as d-DNNFs or OBDDs, is that they allow for model counting in time linear to the circuit size. In the case of SDDs only a single pass is needed to compute the weighted model count of a SDD α. In fact, a single pass has a time complexity of $\mathcal{O}(|\alpha|)$. It is therefore quite desirable to have $|\alpha|$ as small as possible.

In essence, the way SDDs are able to model count, is by using the strong determinism property discussed in section 2.5.4. The full details of how SDDs allow for model counting is beyond scope. More details can be found in [13] and [9]. However, in the context of this thesis, the knowledge that SDDs allow WMC suffices.

Together with section 2.4, which showed how to transform an inference problem to a WMC problem, the WMC problem can be reduced even further. Indeed, by finding an SDD for the logical theory Δ the WMC can be found efficiently. By the fact that $WMC(\Delta) = Z$, inference in an MLN is essentially reduced to finding the SDD of some theory and passing over the result a single time.

In essence what this reduction entails is leveraging the fact that we introduced structure into the potentials Φ of an MRF by transforming it into an MLN. This structure helped us find a more efficient way of inference, however the upper bound on inference is still exponential in the worst case.

2.7 Genetic Algorithms

Section 2.2 described learning MLNs as a two part process. This thesis also tackles learning MLNs as a two part process. More specifically, the structure learning part of MLN learning is done through genetic algorithms.

Therefore, this section explains the framework that is, a genetic algorithm. The

ingredients that make up a genetic algorithm will be discussed is some detail. A more comprehensive overview can be found in [30].

2.7.1 Genetic algorithms as an optimization heuristic

First and foremost, genetic algorithms are a type of optimization methods [30]. More specifically, they are categorized as meta-heuristic optimization methods. They are quite unique as it is based on the principles of evolution. They imitate the natural selection process. In an attempt to do so, they iteratively select the *fittest* solutions and combine them to produce new solutions. Biology often acts in a similar way: giraffes with a longer neck are better equipped for their environment and as a result are healthier/fitter with a substantially higher chance of reproducing than their shorter-necked peers. As opposed to reproducing, in a genetic algorithm, the operation of combining multiple individuals to produce offspring is referred to as *crossover*. Similarly, as is the case in the natural world, individuals in a population usually all differ in some way or another. Some of these differences may be detrimental to survival or provide a considerable advantage. In biology, these differences are called *mutations*. Much in the same way, genetic algorithms implement the concept of mutations. They are random changes to an individual which might improve a solution or worsen it.

Mutation, crossover and fitness form the base of a genetic algorithm. In order to make the concept more clear, algorithm 1 shows the genetic algorithm in its most general form.

Algorithm 1: The general outline of a genetic algorithm.

output : P = Final population
F = Final fitness
input : n = population size
p_m = mutate probability
$P \leftarrow$ Initialize(n)
$F \leftarrow$ fitness(P)
while *Not done* **do**
  $S \leftarrow$ selectParents(P, F)
  $C \leftarrow$ crossover(S)
  $P \leftarrow$ mutate(P, p_m)
  $F \leftarrow$ fitness(P)
  $P \leftarrow$ thinning($P \cup C$)
end
return P, F

In algorithm 1 P is the vector of individuals of the population, F is a vector of the fitness of each individual in P and C a vector of offspring individuals. Here $n \in \mathcal{N}$ is the number of individuals in the population, $p_m \in [0, 1]$ is the probability of mutation.

2.7.2 An individual and a population

Individual As explained above, a genetic algorithm strives to find an optimal solution of some sort of representation. Such a representation is referred to as an *individual*. Examples of such a representation are a path in a graph, a string, a vector of bits or even a real number. In what follows, the set of all possible of individuals will be denoted as $\mathcal{I}$. A genetic algorithm attempts to find the individual $x \in \mathcal{I}$ for which the fitness is maximized:

$$x = \arg\max_{i \in \mathcal{I}} f(i) \tag{2.34}$$

The concept of the fitness f is further explained in section 2.7.6.

Population A population P is simply a collection of individuals:

$$P \subset \mathcal{I}, |P| = n \tag{2.35}$$

The population size parameter n in algorithm 1 determines how many individuals can belong in the population at one time. This parameter has a large impact on both the convergence of the population to a good solution and the computation time required for finding it. That is, when the parameter n is larger, one usually observes faster convergence. However, all the operations in a genetic algorithm might take a substantial amount of time. In reality one has to make a trade-off between computation time and population size.

Example: Consider an individual to be a bit vector of length 4, an individual might be $[0100] \in \{0,1\}^4$. A population would then consist of a subset $P \subseteq \{0,1\}^4 = \mathcal{I}$.

2.7.3 Mutations

As explained in section 2.7.1, a mutation is a random change to an individual. Within a genetic algorithm is provides the large jumps in a search space and ensures diversity within a population. More formally, a mutation is defined as a function:

$$m : \mathcal{I} \to \mathcal{I} \tag{2.36}$$

It takes in an individual and returns a, possibly, different individual. The mutation function usually incorporates some form of randomness. While the mutation changes the individual, it might be for the better or the worse. That is there is no guarantee that:

$$f(i) \leq f(m(i)) \tag{2.37}$$

And in fact, one of the reasons a genetic algorithm is able to work is because the mutation does not adhere to the above property [30].

Example: Take an individual i as in example 2.7.2. One way a mutation m could be defined as:

$$m(i) = (i + j) (\text{mod } 2) \qquad \text{with } j \sim \mathcal{U}(\{j \in \{0,1\}^4 | \, \|j\|_1 = 1\}) \tag{2.38}$$

Here (mod 2) is applied for each element of the vector $i + j$. This mutation operation basically changes one bit of the vector. And it chooses so uniformly at random.

2.7.4 Crossovers

While mutations provide a way of making large jumps in search space, some convergence is still required. The crossover operation is one element to ensure this convergence. The goal of the crossover is to combine two or more individuals in a certain way, in the hopes of producing offspring with an even better fitness. Formally, it is a function defined as follows:

$$c : \mathcal{I}^p \to \mathcal{I}^c \tag{2.39}$$

Where p is the amount of parents used to perform crossover and c is the amount of children produced. Often we choose $p = c = 2$.

Example: Again, using the individuals of example 2.7.2 one way to define a cross-over could be

$$\begin{aligned} c(i,j)_1 &= i_{\{1,\dots,n\}} + j_{\{n,\dots,4\}} \\ c(i,j)_2 &= j_{\{1,\dots,n\}} + i_{\{n,\dots,4\}} \qquad \text{with } n \sim \mathcal{U}(\{1,2,3\}) \end{aligned} \tag{2.40}$$

Here $i_{\{l,\dots,r\}} \in \{0,1\}^4$ denotes the 0 vector where every entry $k \in \{l, ..., r\}$ is replaced by i_k. Intuitively, this crossover selects one point between 1 and 3 and splits the vectors and re-combines them. As n is a random variable, the crossover operation is a random function.

2.7.5 Selection

When performing the crossover, one has yet to decide which individuals are fit enough to be crossed in the first place. For this we introduce the selection of a genetic algorithm. Selection is the act of choosing a subset of individuals among a population of individuals. Often times this selection is based on the fitness of the individuals in the population. While not always the case, most selection methods incorporate elements of randomness.

Example: A classic example of selection is the so called roulette wheel selection[23]. In simple terms it selects m individuals proportionately to the fitness of the individual:

$$\begin{aligned} p(i) &\propto f(i) \\ p(i) &= \frac{f(i)}{\sum_j f(j)} \end{aligned} \tag{2.41}$$

2.7.6 Fitness

Arguably the most important ingredient of the genetic algorithm. It determines what is a good or bad solution, or more generally which solution is considered

better than another solution. It should in the best case incorporate all that is deemed important to a solution. Moreover it defines the shape of the search space and the convergence behavior of the algorithm. A badly chosen fitness function might be infeasible to compute, and makes the algorithm converge to undesired solutions. A well chosen fitness function may lead to desirable or optimal solutions[30].

A basic fitness function f is formally defined as:

$$f : \mathcal{I} \to \mathcal{R}^+ \tag{2.42}$$

While not strictly necessary, usually f only assumes non-negative values. There exists many variations on this basic fitness function, for example one popular extension if a fitness function based on which generation the algorithm is currently in:

$$f : \mathcal{I} \times \mathcal{N} \to \mathcal{R}^+ \tag{2.43}$$

Example: Again looking at an individual as in example 2.7.2. Say our desire is to be as close as possible to the following criteria:

- We want the ratio between the amount of 0 elements and the amount of 1 elements to be as close to 1 as possible.
- We want our solution to have so called *regions* of 1's and 0's. Concretely, for our function, it should hold that

$$f(\begin{bmatrix}0\\0\\0\\1\end{bmatrix}) \geq f(\begin{bmatrix}0\\0\\1\\0\end{bmatrix}) \tag{2.44}$$

  because the first vector has more uniform regions than the second.

Guided by our desires for the solution we might come up with the following fitness function

$$f(x) = \alpha(x) - \beta(x) \tag{2.45}$$

Where α could be a term rewarding equal balance between 0 and 1

$$\alpha(x) = 8\frac{\|x\|_1}{4}(1 - \frac{\|x\|_1}{4}) \tag{2.46}$$

Here α reaches a maximum in $\|x\|_1 = 0.5$. While β might be a term punishing solutions with small regions

$$\beta(x) = \frac{1}{4}\sum_{i=1}^{4} |\{x_j \neq x_i | j \in \{1, ..., 4\}, j = i \pm 1\}| \tag{2.47}$$

Intuitively β represents the average neighbor-incompatibility of a solution. e.g.

$$\beta(\begin{bmatrix}0 & 1 & 0 & 1\end{bmatrix}) = \frac{1+2+2+1}{4} = \frac{3}{2} \tag{2.48}$$

In this example it's clear to see the optimal solution would be either $\begin{bmatrix}0 & 0 & 1 & 1\end{bmatrix}$ or $\begin{bmatrix}1 & 1 & 0 & 0\end{bmatrix}$, however in real situations it is not always clear if f reflects correctly what kind of solution we would like.

Chapter 3

Relevant work

This section presents the relevant literature behind the thesis.

3.1 Graphical models, MLNs and Model Counting

Markov random fields, more generally, graphical models are a staple of modern probabilistic and logical reasoning. Already in 1950 Loéve defined the concept of *conditional independence*, which would later be formalized by Phil Dawid [28]. Around 1970, these conditional independence concepts were introduced in *undirected probabilistic graphs* (UG) as a tool for statistical physics[28]. A mere 10 years later, directed acyclic graphs (DAGs) knew their rise to fame, as a tool in decision theory [28]. However, in the field AI, the popularity of graphical models lay mostly dormant until 1980. In 1985 the Uncertainty in Artificial Intelligence (UAI) conference was first held in Los Angeles, further boosting the popularity of graphical models. However, it was only in 2004 where MLNs were introduced[34].

While graphical models were gaining in popularity, it was well known that performing inference in graphical models is #P-hard [10, 35]. This did not deter the development of many algorithms for performing inference. A popular example is variable elimination [40]. The idea of using techniques from the SAT and model counting problems was originally presented in [11]. In his work A. Darwiche presented the encoding of a Bayesian model which can be found in section 2.4. An overview of different possible encodings can be found in [6].

3.2 Knowledge compilation and SDDs

The rise in popularity of graphical models and encoding techniques, meant that the field of knowledge compilation gained considerable interest. As briefly mentioned, each target language has specific properties. A general overview of target languages can be found in [14]. The target language used in this thesis is the SDD, which was introduced by A. Darwiche in 2011[12]. In this work, Darwiche introduced SDDs as a representation and a way to compile theory bases. Additionally it showed that

performing operations on SDDs can be done in polynomial time in the size of the SDDs. It was later shown by S. Bova that SDDs are in fact exponentially more succint[5]. In a later work A. Choi and A. Darwiche proposed a way of minimizing SDDs, increasing their usefulness[8].

3.3 MLN learning and circuit learning

Many MLN learning techniques have been developed after the introduction of MLNs [24, 29, 17, 16]. Usually these are search based methods in the MLN space. By searching in this space, these methods usually don't consider tractability as very important. However, with the advent of the new encoding and compilation techniques, a lot of interest has developed in learning MLNs using these target representations directly.

In [27], it was suggested to use arithmetic circuits (AC) directly in order to learn distributions. ACs have the property that allows to compute the WMC in polynomial time to their size. This development allowed searching while keeping tractability in mind. This lead to more target representations to be explored. Especially target representations which allow for polynomial operation, such as SDDs, are quite interesting in this regard. In [4] J. Bekker et al. developed a greedy learning algorithm for SDDs. The crux of this approach is iteratively finding the best feature and conjoining it to the existing SDD. A variant of SDDs is the *probabilistic sentential decision diagrams* (PSDDs). In [26], LearnPSDD is presented, a method of learning PSDDs.

In the past genetic based algorithms for learning graphical models have been proposed. In [7, 25], genetic algorithms were applied to Bayesian network learning. However, these methods do not learn circuits simultaneously. As a result tractability of the learned model suffers. This thesis attempts to bridge the gap between genetic algorithms on the one hand and tractable MLN and circuit learning on the other.

3.4 Genetic algorithm

Already in 1944 the now famous Alan M. Turing had spoken about programming a child-like brain and then educating it [39]. However, he also realized there are numerous ways of programming one such child-like brain. Naturally, in a certain sense, one would outperform another. Making small changes to one child-like brain might result in a vastly different performance. Even though he remained rather vague what exactly it means for one to outperform another, he drew a remarkable parallel to evolution of a species. He noted the following connections:

aspect of learning	aspect of evolution
Structure of child brain	Hereditary material
Changes to a child-like brain	Mutation
Judgment of performance	Natural selection

In 1954 Nils Barricelli published the first paper on genetic algorithm software [3]. This paper sparked more interest in the field of genetically based algorithms. However, the field took off only starting from the 1970's with the work of John Holland and his book *Adaptation in Natural and Artificial Systems* [22]. In 1985, the first International Conference on Genetic Algorithms (ICGA) took place in Pittsburg, USA [20]. Even in the advent of the recent deep learning craze, genetic algorithms stay prevalent, with some authors attempting to combine both worlds[15, 37].

A introductory guide to genetic algorithms can be found in [30]. This guide helped me greatly throughout the thesis. A large and comprehensive overview of selection methods can be found in [19].

Chapter 4

GeSDD

This section delves deeper into the specifications of GeSDD. As mentioned in the problem statement in section 1.3, GeSDD is an algorithm for learning MLNs simultaneously with its SDD. MLNs are comprised of two parts. The first part are the features $f = (f_1, \ldots, f_k)$ and the second are their corresponding weights $w = (w_1, \ldots, w_k)$. Therefore, learning MLNs usually consist of *feature* or *structure learning* and *weight learning*. In the case of GeSDD, the structure learning is done through a genetic algorithm. Each generation of GeSDD, the weights of the population are trained.

This section is structured as follows. Firstly, a general overview of the algorithm is presented. Secondly, the weight learning is discussed. Next, the feature space in which GeSDD searches is discussed alongside an evaluation metric for features in said space. Then, the genetic structure learning algorithm is discussed. Finally, a note on other heuristics.

The algorithm is implemented in python, using the PySDD library [1]. This is a wrapper for the SDD library originally written by A. Darwiche in C[12]. However, this entire section may be read independently from any programming language or even any SDD library.

GeSDD is freely available on github: https://github.com/MichielBaptist/GeSDD.git

4.1 GeSDD

Here, I present the general structure of GeSDD. It follows a very similar approach as in the introduction to genetic algorithms in section 2.7. The pseudocode of GeSDD is shown in algorithm 2 below. In GeSDD, however, the weights of the population have to be trained each generation. The method used for training the MLNs is described in section 4.2. Sections 4.6, 4.7 and 4.5.1 respectively discusses the mutation, crossover and fitness. The code below does not show every single parameter, as this would make

[1] Freely available at https://github.com/wannesm/PySDD and maintained by W. Meert of the DTAI research group in KULeuven.

the pseudocode unreadable. However, each parameter in GeSDD will be discussed in their relevant sections.

Algorithm 2: Pseudocode of GeSDD

```
input : n = population size
        p_m = mutate probability
        g = number of generations
        D_t = binary training dataset
Best ← (E, ⊤) //Keep track of the best model
P ← Initialize(n) //Initialize the population
P ← trainWeights(P, D_t)
F ← fitness(P, D_t)
for i = 1, 2, ..., g do
    S ← selectParents(P, F)
    P ← P ∪ crossover(S)
    P ← mutate(P, p_m)
    P ← trainWeights(P, D_t)
    F ← fitness(P, D_t)
    P ← thinning(P, F)
    if max(F) > fitness(Best, D_t) then
        Best ← max(P)
    end
end
return Best
```

An individual and population in GeSDD
As a genetic algorithm works in a certain search space, it is important to specify how it is defined. In the context of this thesis an individual is a single MLN, along with its corresponding SDD:

$$\mathcal{I} = \{(M, \alpha) | enc(M) \equiv \langle \alpha \rangle, |M| \leq k_f\} \tag{4.1}$$

Here M is an MLN as defined in 2.2 and $|M| < k_f$ denotes that the search space of MLNs is limited to MLNs which have less than k_f features. α is an SDD as defined in section 2.3 and $enc(M)$ denotes the encoding of the MLN as defined in section 2.4. This representation allows for efficient combining and mutation of individuals. Moreover, by keeping track of the SDD, inference remains tractable without the need for re-compilation. A key factor which proves quite necessary for weight learning.

4.2 Weight Learning of MLNs

In order for an MLNs distribution $\langle M \rangle$ to be reflective of a given dataset $\mathcal{X}$, the weights $w = (w_1, \ldots, w_k)$ first have to be learned. In order to do so, there are many estimators available. However, since the data we are dealing with is assumed to

be independently sampled, it might be appropriate to make use of the maximum likelihood estimator. Therefore, GeSDD uses the maximum likelihood estimator. For a weight w_i and binary dataset $\mathcal{X} = (x_1, \ldots, x_n)$, it can be written as:

$$\begin{aligned} \hat{w_i} &= \arg\max_{w_i} \mathcal{L}(w|\mathcal{X}) \\ &= \arg\max_{w_i} \prod_{x \in \mathcal{X}} P(x|w) \\ &= \arg\max_{w_i} \sum_{x \in \mathcal{X}} \log\left(P(x|w)\right) \end{aligned} \tag{4.2}$$

Equation 2.4 shows the definition of $\langle M \rangle = P(x|w)$. Plugging this into equation 4.2 then results in:

$$\begin{aligned} \hat{w_i} &= \arg\max_{w_i} \sum_{x \in \mathcal{X}} \log\left(\frac{1}{Z} \exp\left(\sum_j w_j f_j(x)\right)\right) \\ &= \arg\max_{w_i} \sum_{x \in \mathcal{X}} \sum_j w_j f_j(x) - \sum_{x \in \mathcal{X}} \log(Z) \\ &= \arg\max_{w_i} \sum_j w_j \sum_{x \in \mathcal{X}} f_j(x) - n \log(Z) \\ &= \arg\max_{w_i} \sum_j w_j * count(f_j, \mathcal{X}) - \log(Z) \end{aligned} \tag{4.3}$$

Where, in equation 4.3 $count(f_j, \mathcal{X})$ is defined as follows:

$$count(f_j, \mathcal{X}) = \frac{1}{n} \sum_{x \in \mathcal{X}} f_j(x) \tag{4.4}$$

It denotes the fraction of the data, in which f_j is satisfied, i.e. $f_j(x) = 1$. A necessary condition for a function to reach its optimum, is that the gradient vanishes. In the case of the maximum likelihood estimator in 4.3, the gradient in w_i is equal to the following:

$$\begin{aligned} \frac{\partial \mathcal{L}(w|\mathcal{X})}{\partial w_i} &= count(f_i, \mathcal{X}) - \frac{1}{Z} \frac{\partial}{\partial w_i}(Z) \\ &= count(f_i, \mathcal{X}) - \frac{1}{Z} \frac{\partial}{\partial w_i} \sum_{x' \in \Omega} \exp\left(\sum_j w_j f_j(x')\right) \\ &= count(f_i, \mathcal{X}) - \frac{1}{Z} \sum_{x' \in \Omega} \exp\left(\sum_j w_j f_j(x')\right) \frac{\partial}{\partial w_i}\left(\sum_j w_j f_j(x')\right) \\ &= count(f_i, \mathcal{X}) - \frac{1}{Z} \sum_{x' \in \Omega} \exp\left(\sum_j w_j f_j(x')\right) f_i(x') \end{aligned} \tag{4.5}$$

The step from line 1 to line 2 is obtained by plugging in Z from equation 2.5. By setting the gradient with respect to w_i, equal to zero we find the following:

$$count(f_i, \mathcal{X}) = \frac{1}{Z} \sum_{x' \in \Omega} \exp\left(\sum_j w_j f_j(x')\right) f_i(x') \tag{4.6}$$

This resulting condition is indeed a logical and intuitive one. This can be seen by re-writing the right hand side of the above equation:

$$\begin{aligned}\frac{1}{Z}\sum_{x'\in\Omega}\exp\Big(\sum_j w_j f_j(x')\Big)f_j(x') &= \sum_{x'\in\Omega}\Big(\frac{\exp\big(\sum_j w_j f_j(x')\big)}{Z}\Big)f_i(x') \\ &= \sum_{x'\in\Omega} P(x'|w)f_i(x') \\ &= P(f_i = 1|w)\end{aligned} \tag{4.7}$$

Indeed, the right hand side denotes the probability that feature f_i is satisfied. The gradient condition in equation 4.5 states that a weight w_i of an MLN maximizes the likelihood if the observed probability that f_i is satisfied in the data, is equal to the predicted probability that f_i is satisfied according to the MLN. This result is quite satisfying, however it does not offer a closed form solution to the maximization problem.

A workaround to this problem is using gradient based methods. Luckily, the gradient in equation 4.5 can be computed. The first term can be computed by counting the probability that a feature f_i is satisfied in the data $\mathcal{X}$. The second term can also be computed. This can be done by computing Z.

Due to the fact that the gradient can be computed, a gradient based method can be used. The implementation of GeSDD uses scikit-learn[32] general optimization functionality[2].

4.3 The feature space of f_i

Until now, we have considered a feature f_i as an abstraction. Assuming that f_i belonged to the class of all possible Boolean functions over $\mathbf{X} = (X_1, \ldots, X_m)$:

$$f_i \in \{0,1\}^m \to \{0,1\} = \mathcal{F}_m \tag{4.8}$$

While this is useful from a theoretical perspective, in practice these features might need to be restricted to a certain class of Boolean functions. Especially in the case of genetic algorithms, where features will be manipulated into different features. In order to do so, the structure of a feature should be known, easy to work with and not restricting.

Choosing a subspace of the feature space is not an easy task. The subspace has to be simple enough to work with but also not too restricting. GeSDD is restricted to the space of recursive Boolean functions of conjunctions of literals and negated features of the same space. Additionally, each literal may only appear once in the feature.

[2]More specifically the function minimize in the python package scipy.optimize and the BFGS method.

Written as a formula, GeSDD restricts the feature space over $\mathbf{X} = (X_1, \ldots, X_m)$ to the following class:

$$\mathcal{F}_g = \{f_1 \wedge \cdots \wedge f_l | f_i = X_i \text{ or } f_i = \neg f_i' \text{ with } f_i' \in \mathcal{F}_g \text{ and } \forall s, t : L(f_s) \cap L(f_t) = \emptyset \text{ and } l > 1\} \tag{4.9}$$

Where $L(f_s)$ denotes all literals which are present in f_s:

$$L(f_s) = \begin{cases} \{X_s\} & \text{if } f_s = X_s \\ L(f_s') & \text{if } f_s = \neg f_s' \\ \bigcup_{i=1}^{l} L(f_i) & \text{if } f_s = f_1 \wedge \cdots \wedge f_l \end{cases} \tag{4.10}$$

The empty subset condition in equation 4.9 states that a given literal X_i may only be used a single time in a given feature. This can either be as the literal itself $X_i \in L(X_i)$, as a negation $X_i \in f(\neg X_i)$ or it is used in a conjunction $X_i \in L(f_1' \wedge, \ldots, \wedge f_l')$.

The use of this feature space is motivated by the fact that it is customary to use conjunctions in case of Boolean data[21]. Additionally, conjunctions are easy to understand and easy to manipulate. Finally, from my empirical observations, this class seems both expressive enough and results in the most compact SDDs.

4.4 Evaluating and selecting features f_i

Looking at the feature space in equation 4.9, it seems to be rather large. It is not infinite due to the fact that literals may only be used once. Nevertheless, it is exponential in the amount of literals. Searching randomly in such a space might turn out to be inefficient. It is therefore customary to *select* which features might perform better than others. The field of research concerned with evaluating and selecting features is know as *feature selection*. It forms an important part of many machine learning algorithms. This section describes a feature evaluation heuristic used by GeSDD. Here we will not discuss how and where in GeSDD it is used, this will be discussed in sections 4.6 and 4.7.

Given a feature $f = f_1 \wedge f_2$ and a dataset $\mathcal{X}$. How can we know if feature f captures any information about $\mathcal{X}$? One way to measure this is through the covariance of f_1 and f_2. As an example, assume that f_1 and f_2 are completely independent in $\mathcal{X}$. Then, if we know f_1 is true, we gain no information about f_2. In this sense, f doesn't capture any information about $\mathcal{X}$. As such, f would perform miserably as a feature in an MLN. Conversely, if we assume now that f_1 is highly correlated with f_2. In this case, given information about f_1 would give us a very good indication about f_2. This is precisely what f would then represent. So in a sense, f captures the information that f_1 and f_2 are highly correlated.

The concept explained above, takes on many forms. In the well known principal component analysis[31], one often speaks of finding the direction that *captures*

most variance. The same principle of capturing variance is used in sparse dictionary learning. This method attempts to find a change of basis such that less components provide the same information as the original basis.

In GeSDD, features are also evaluated for the amount of information they capture. The metric used to measure how much information they capture, is the *mutual information*. For Boolean variables, mutual information between two variables is defined as follows:

$$I(X;Y) = H(X) + H(Y) - H(X,Y) \tag{4.11}$$

Here $H(X)$ is the entropy of the variable X and $H(X,Y)$ is the joint entropy. They are defined as follows:

$$H(X) = -\sum_{x \in B} P(x) \log(P(x)) \tag{4.12}$$

$$H(X,Y) = -\sum_{y \in B} \sum_{x \in B} P(x,y) \log(P(x,y)) \tag{4.13}$$

Where $B = \{0,1\}$ denotes the outcome set of X and Y. Often we do not know the distributions $P(X = x)$ and $P(Y = y)$. Instead, as is the case for GeSDD, it is often estimated as $\hat{P}_x = count(x, \mathcal{X})$.

Note that mutual information in equation 4.11 is defined for two variables. If mutual information is used, we can only evaluate features of the form $f = f_1 \wedge f_2$. In the literature, many estimators for multivariate information gain has been proposed[18]. However, often they lack an intuitive meaning and are hard to interpret. Therefore, GeSDD uses an easier but less robust alternative.

Given a feature $f = f_1 \wedge \cdots \wedge f_l$ and a dataset $\mathcal{X}$, the *average pairwise mutual information* (AMI) is defined as:

$$AMI(f, \mathcal{X}) = \frac{2}{l(l-1)} \sum_{i=1}^{l} \sum_{j=i+1}^{l} I(f_i; f_j) \tag{4.14}$$

As the name suggests, it is the average mutual information between all pairs of f_i and f_j of f. As this is a pairwise method, a lot of higher order information is not considered and therefore is less robust than a traditional measure of multivariate mutual information.

4.5 Fitness and selection

4.5.1 Fitness

The fitness function is arguably the trickiest part in designing a genetic algorithm. It completely decides what defines a good solution or a bad solution. A good fitness function should incorporate all properties of what defines a good solution. In the

case of simultaneously learning MLNs alongside SDDs and in the context of this thesis, these two factors are considered most important:

1. The learned MLN should ideally reflect the data well. This way, the MLN is useful in performing inference and making predictions. In order to quantify this property, some measure for distribution distance should be used.

2. The resulting SDD should be tractable. As seen in 2.6, the size of the SDD determines the running time of performing inference.

To adress the first point, it is customary in the literature to measure the fit of the distribution using the log-likelihood $\mathcal{L}(M, \mathcal{X}) = \log(P(\mathcal{X}|w))$. This is possible, as the data $\mathcal{X} = (x_1, \ldots, x_n)$ is usually assumed independent. The second point can addressed by incorporating $|\alpha|$, for an MLN with SDD α.

Taking these two key ingredients, and in an attempt to be as unbiased as possible, GeSDD uses the following fitness function:

$$f(i, \mathcal{X}, \beta) = f((M, \alpha), \mathcal{X}, \beta) = \max\Big(\mathcal{L}(M_p, \mathcal{X}) - \mathcal{L}(E_p, \mathcal{X}) - \beta|\alpha|, 0\Big) \tag{4.15}$$

Here, $\mathcal{L}(M_p, \mathcal{X})$ denotes the log-likelihood of the distribution defined by M for the data $\mathcal{X}$. E denotes the empty model, as defined in 2.7. Lastly, β is a parameter which decides preference for larger or smaller SDDs.

Intuitions behind the fitness function
A couple of remarks about the fitness function follow:

- The term $\mathcal{L}(M_p, \mathcal{X}) - \mathcal{L}(E_p, \mathcal{X})$ can be interpreted as the *gain in log-likelihood* compared to the empty model.

- In most cases, the value in the left hand side of the max(.,.) will be a positive value. However, this may not always be the case. To ensure a positive value, the max(.,.) operand is used.

4.5.2 Selection

GeSDD uses a variant of tournament selection[19]. This method consists of multiple *tournament rounds*. In each round a fixed number k_t, the *tournament size* , of individuals are selected. Next, one individual gets selected with a probability proportional to its fitness. This means that given k_t randomly selected individuals I_i with fitness f_j, each individual is selected with a probability of:

$$P(I_i \text{ is selected }) = \frac{f_i}{\sum_{j=1}^{k_t} f_j} \tag{4.16}$$

4.6 Mutation

Now that we know the search space of features, it becomes possible to define mutations. In this section, the mutation of GeSDD will be discussed. In GeSDD there is not a single mutation, but 6 mutations. Each time an individual is mutated, GeSDD selects one out of these 6 at random. The mutations are divided into 3 categories:

1. Adding features, this category has 1 mutation
2. Removing features, this category has 1 mutation
3. Altering features, this category has 4 mutations

4.6.1 Adding features

The first category of mutations is the adding of features. This mutation will take in an individual and add one or several new features.

Given an individual $I = (M, \alpha)$, with $M = \{(f_1, w_1), \ldots, (f_k, w_k)\}$ and the corresponding SDD α. This mutation will add one or several new features.

$$\begin{aligned} m(I) &= \Big(M \cup M', \alpha \wedge \beta\Big) \\ M' &= \{(f'_1, w'_1), \ldots, (f'_l, w'_l)\} \text{ and } \langle\beta\rangle = enc(M') \end{aligned} \tag{4.17}$$

Here, β is the SDD corresponding to the new features in M' using the same vtree as α. The amount of added features l is drawn uniformly between 1 and a percentage $\gamma \in [0, 1]$ of the size of the MLN M:

$$l \sim \mathcal{U}\Big(\{1, \ldots, \Big\lceil \gamma * |M| \Big\rceil\}\Big) \tag{4.18}$$

γ defines how big of a jump this mutation makes in the search space. This line of thought might seem quite peculiar at first. Why would it be necessary to make larger jumps as the MLN M is larger? The answer lies in convergence behavior. If a fixed number of features were added to a large MLN, the jump might end up being too weak. As a result we might get stuck in a local optimum.

The features added to the MLN first need to be created. In order to do so GeSDD follows a general two step approach: generate and select. Algorithm 3 shows the procedure for generating a single new feature. Here, the generated *candidates* are denoted by c_i. All candidates together form the *candidate set* C. AMI is the average pairwise mutual information as defined in equation 4.14. C_n denotes the amount of features to be generated in the candidate set: $|C| = C_n$. This parameter somewhat determines how random the feature generation is. This can be seen by setting $C_n = 1$, in this case the only generated feature will be selected. If C_n is set to a high value, it is more likely that a feature with a higher AMI value will be sampled.

Algorithm 3: The generate and select procedure.

Input : $\mathcal{X}$ = binary training data
Output : f = generated feature
Parameters: C_n = size of the candidate set

// Step 1: generate and evaluate
for $i = 1, 2, ..., C_n$ **do**
  $c_i \leftarrow g(\mathcal{X})$
  $a_i \leftarrow AMI(c_i, \mathcal{X})$ //Average pairwise mutual information
end

// Step 2: select
for $i = 1, 2, \ldots, C_n$ **do**
  $p_i \leftarrow a_i / \sum_j a_j$
end
$f \leftarrow$ draw randomly with probabilities p_i

return f

Finally, $g(.)$ denotes a function generating candidates. In theory $g(.)$ may generate any Boolean function belonging to the feature space $g(.) \in \mathcal{F}_g$. GeSDD uses an approach based on the work of J. Van Haaren et al[21]. The generation procedure $g(.)$ is described in algorithm 4.

Algorithm 4: The candidate generation procedure $g(.)$.

Input : $\mathcal{X}$ = binary training data
Output : c = candidate feature

// Step 1: generate initial feature set.
$In \leftarrow \{f | count(f, \mathcal{X}) > 0\}$

// Step 2: sample randomly from initial feature set.
$f \leftarrow \mathcal{U}(In)$

// Step 3: randomly sample feature length.
$l \leftarrow \mathcal{U}(\{2, \ldots, |f|\})$

// Step 4: randomly sample l conjunctors from $f = f_1 \wedge \cdots \wedge f_s$.
$G \leftarrow$ sample l conjunctors without replacement from f
$c \leftarrow \bigwedge_{f_i \in \mathcal{G}} f_i$
return c

4.6.2 Removing features

The second category of mutations is the opposite of adding features. This mutation will take in an individual and will remove one or several existing features.

Given an individual $I = (M, \alpha)$, with $M = \{(f_1, w_1), \ldots, (f_k, w_k)\}$ and the corresponding SDD α. This mutation will remove one or several present features:

$$\begin{aligned} m(I) &= \Big(M \setminus M', \alpha|M'\Big) \\ M' &= \{(f'_1, w'_1), \ldots, (f'_l, w'_l)\} \end{aligned} \tag{4.19}$$

Note that the notation $\alpha|M'$ denotes the SDD corresponding to $enc(M \setminus M')$ as defined in property 4 in section 2.5.4. A natural condition for equation 4.19 is that the removed features M' is a subset of the original MLN M, $M' \subseteq M$. As before, the amount of removed features l is drawn uniformly between 1 and a percentage $\gamma \in [0, 1]$ of the size of the MLN:

$$l \sim \mathcal{U}\Big(\{1, \ldots, \Big\lceil \gamma * |M| \Big\rceil\}\Big) \tag{4.20}$$

The features selected to be removed are based on the weights w_i of the features. More specifically, a feature is selected to be removed based on a softmax:

$$P(f_i \text{ is selected } |w) = \frac{\exp(w_i)}{\sum_j \exp(w_j)} \tag{4.21}$$

4.6.3 Altering features

Finally, the third category of mutations is the altering mutations. This mutation will take in an individual and select one or several features to be removed. Then *alter* or *transform* them in a certain way. Finally, the transformed features are again added to the individual.

This mutation takes in an individual $I = (M, \alpha)$ and changes one or several features. Given an individual with $M = \{(f_1, w_1), \ldots, (f_k, w_k)\}$:

$$\begin{aligned} m(I) &= \Big(M \setminus M_1 \cup M_2, \alpha|M_1 \wedge \beta\Big) \\ &M_1 = \{(f'_1, w'_1), \ldots, (f'_l, w'_l)\} \subset M \\ &M_2 = \{(T(f'_1), w_1), \ldots, (T(f'_l), w'_l)\} \text{ and } \langle\beta\rangle = enc(M_2) \end{aligned} \tag{4.22}$$

Where the amount of features altered l is drawn uniformly between 1 and a percentage $\gamma \in [0, 1]$ of the size of the MLN:

$$l \sim \mathcal{U}\Big(\{1, \ldots, \Big\lceil \gamma * |M| \Big\rceil\}\Big) \tag{4.23}$$

Furthermore, in equation 4.22, $T(.)$ is a random function that transforms a feature:

$$T : \mathcal{F}_g \to \mathcal{F}_g : f \mapsto T(f) \tag{4.24}$$

The general transformation procedure can be seen in algorithm 5. It follows the same generate and select approach as the adding features mutation. The difference being that this time the generated features are based on the input feature, while the adding mutation does not receive an input feature. In the *transform* step of the procedure, $t(.)$ is a function which takes in a feature f and generates a *candidate* c_i. All the candidates together form the *candidate set* C. The C_n is a parameter denoting the size of the generated candidate set $|C| = C_n$. In the *select* step of the procedure, a feature from the candidate set is chosen based on the AMI, defined in equation 4.14.

Algorithm 5: The transform and select procedure $T(.)$.

Input : $f = f_1 \wedge \cdots \wedge f_s$ input feature
$\mathcal{X}$ = binary training data
Output : $T(f)$ = transformed feature
Parameters: C_n = size of the candidate set

// Step 1: transform and evaluate
for $i = 1, 2, ..., C_n$ **do**
  $c_i \leftarrow t(f)$
  $a_i \leftarrow AMI(c_i, \mathcal{X})$ //Average pairwise mutual information
end

// Step 2: select
for $i = 1, 2, \ldots, C_n$ **do**
  $p_i \leftarrow a_i / \sum_j a_j$
end
$g \leftarrow$ draw randomly with probabilities p_i
return g

The definition of $t(.)$ in algorithm 5 was left rather vague. However, it is simply a transformation function:

$$t : \mathcal{F}_g \to \mathcal{F}_g : f \mapsto t(f) \tag{4.25}$$

It serves as a way to generate features from an existing feature f. GeSDD has 4 such transformation functions. In the following each of these will be discussed together with their parameters.

Lengthening features
This feature transformation will lengthen the input feature $f = f_1 \wedge \cdots \wedge f_s$:

$$t(f) = f_1 \wedge \cdots \wedge f_s \wedge f_{s+1} \wedge \cdots \wedge f_{s+a} \tag{4.26}$$

The f_{s+i} are uniformly sampled without replacement from all literals that were previously not already present:

$$f_{s+i} \sim \mathcal{U}\Big(\{X_1, \ldots, X_m\} \setminus \bigcup_{j=1}^{s+i-1} L(f_j)\Big) \tag{4.27}$$

The amount of added literals a is uniformly sampled between 1 and m_a:

$$a \sim \mathcal{U}(\{1, \ldots, m_a\}) \tag{4.28}$$

Where m_a is a parameter that can be chosen. A higher value of m_a will result in larger jumps. GeSDD employs a constant value of $m_a = 5$. The amount of literals in the new feature can never be greater than than m, the amount of variables $X_1, \ldots, X_m$. This is to ensure the condition of the feature space $\mathcal{F}_g$.

Shortening features
As opposed to lengthening, this feature transformation will shorten the input feature $f = f_1 \wedge \cdots \wedge f_s$. It does this by randomly selecting a subset of the conjunction:

$$t(f) = f'_1 \wedge \cdots \wedge f'_{s-d} \tag{4.29}$$

Where the $f'_i \in \{f_1, \ldots, f_s\}$ are uniformly sampled without replacement from the existing conjunction:

$$f'_i \sim \mathcal{U}\Big(\{f_1, \ldots, f_s\} \setminus \bigcup_{j=1}^{i-1} f'_j\Big) \tag{4.30}$$

The amount of literals dropped d is uniformly sampled between 1 and m_d:

$$d \sim \mathcal{U}(\{1, \ldots, m_d\}) \tag{4.31}$$

Again, m_d is a parameter that can be chosen. A higher value of m_d will result in larger jumps. GeSDD employs a constant value of $m_d = 5$. A minimum of 2 features are kept in the new conjunction.

Negating individual conjunctors
This transformation will negate individual elements of a conjunction with a probability of 50 %. Given a feature $f = f_1 \wedge \cdots \wedge f_s$, the transformation looks as follows:

$$t(f) = n_{0.5}(f_1) \wedge \cdots \wedge n_{0.5}(f_s) \tag{4.32}$$

Where $n_{0.5}(.)$ negates its argument in 50% of the time:

$$n_{0.5}(f_i) = \begin{cases} \neg f_i & \text{if } \mathcal{B}(0.5) = 1 \\ f_i & \text{else} \end{cases} \tag{4.33}$$

Here, $\mathcal{B}(0.5)$ denotes a Bernoulli random variable with probability parameter $p = 0.5$.

Negating groups of conjunctors
Similar as the previous transformation, this transformation will negate conjunctors. However, instead of negating individual conjunctors, it will negate groups of conjunctors. Given a feature $f = f_1 \wedge \cdots \wedge f_s$:

$$t(f_s) = f'_1 \wedge \cdots \wedge f'_{s-g} \wedge \neg\Big(f'_{s-g+1} \wedge \cdots \wedge f'_s\Big) \tag{4.34}$$

Where the negated group and the non negated group form a partition of the original elements in the conjunctions:

$$\{f_1, \ldots, f_s\} = \{f'_1, \ldots, f'_{s-g}\} \cup \{f'_{s-g+1}, \ldots, f'_s\} \tag{4.35}$$

Here, the size of the negated group g is uniformly sampled between 2 and the $s-1$:

$$g \sim \mathcal{U}\Big(\{2, \ldots, s-1\}\Big) \tag{4.36}$$

The -1 is needed such that this transformation is closed over the feature space $\mathcal{F}_g$. Indeed, if $g = s$ were possible, a feature of the form $t(f) = \neg(f') \notin \mathcal{F}_g$ could be generated.

4.7 Crossover

The crossover serves as a way for individuals of the population to interact. By sharing information of the best performing individuals, and by doing so creating new individuals who rival or even best their parents.

In GeSDD, the crossover operation is quite straight forward. In a crossover 2 individuals will exchange one or more features with each other. Concretely, given two individuals $I_2 = (M_1, \alpha_1)$ and $I_2 = (M_2, \alpha_2)$ the crossover is defined as:

$$\begin{aligned} c(I_1, I_2) = &\Big((M_1 \cup M'_2, \alpha_1 \wedge \beta_2), (M_2 \cup M'_1, \alpha_2 \wedge \beta_1)\Big) \\ &M'_1 = \{(f_1, w_1), \ldots, (f_{l_1}, w_{l_1})\} \subset M_1 \text{ and } \langle \beta_1 \rangle = enc(M'_1) \\ &M'_2 = \{(g_1, w'_1), \ldots, (g_{l_1}, w'_{l_1})\} \subset M_2 \text{ and } \langle \beta_2 \rangle = enc(M'_2) \end{aligned} \tag{4.37}$$

Here β_1 respects the same vtree as α_2 and β_2 respects the same vtree as α_1. Again, as in most previous mutations, the amount of rules traded l_1 and l_2 is sampled uniformly between 1 and a percentage $\gamma \in [0, 1]$ of the MLNs.

$$\begin{aligned} l_1 &\sim \mathcal{U}\Big(\{1, \ldots, \lceil \gamma * |M_1| \rceil\}\Big) \\ l_2 &\sim \mathcal{U}\Big(\{1, \ldots, \lceil \gamma * |M_2| \rceil\}\Big) \end{aligned} \tag{4.38}$$

The method of selecting the rules which will be swapped is also of importance. If the swapped rules are of bad quality, both new MLNs are likely to perform worse

than their parents. However when good rules are traded the opposite is more likely. Therefore, the rules chosen to be swapped are sampled without replacement based on the softmax of its weight:

$$P(f_i \text{ selected } |w) = \frac{exp(|w_i|)}{\sum_j exp(|w_j|)} \tag{4.39}$$

4.8 Other heuristics

This section discusses two other heuristics used by GeSDD.

4.8.1 Minimization of SDDs

In section 2.5.2, minimization of SDDs was discussed. In order to leverage the power of SDD minimization, GeSDD uses dynamic minimization. Concretely, each generation GeSDD minimizes the SDDs before the selection step.

4.8.2 Trimming of MLNs

In section 2.2 it was shown that when a feature f_i has a weight of $w_i = 0$, it is equivalent to the MLN without f_i. A similar idea can be applied to features with a relatively small weight. Therefore GeSDD employs MLN *trimming*. At the end of every generation all features with an absolute weight smaller than τ are removed from an MLN M:

$$\begin{aligned} Trim((M, \alpha)) &= \Big(M \setminus M', \alpha | M'\Big) \\ &\text{where } M' = \{(f, w) \in M || w| < \tau\} \end{aligned} \tag{4.40}$$

Chapter 5

Experiments

GeSDD has many parameters to adjust. In this section we explore the effects of different parameters on the results of learned MLN and SDD.

The experiments shown here only form a small subset of possible experiments one could perform on GeSDD. However, the ones shown here are perhaps among the more informative and insightful. The questions this section wishes to answer:

1. What are the effects of the β parameter in the fitness function defined in equation 4.15?

2. Throughout GeSDD the γ parameter is used to define a percentage of a MLN. What are the effects of larger or smaller jumps in the search space by adjusting the γ parameter for all mutations and the crossover?

3. What is the effect of the MLN trimming heuristic and its τ parameter from section 4.8.2?

4. What are the effects of the dynamic SDD minimization?

5. How does GeSDD compare in terms of fit and compactness to other learners?

Throughout the experiments, the NLTCS datset is used. This dataset has 16 binary variables. The datset is divided into a training set, validation set and a test set. The training set is used to train the MLNs and compute the fitness for selection. The validation set is used to manually find the best parameters and for the experiments. Finally, the test set is used after GeSDD ends, to compute the test log-likelihood using the best found model by GeSDD.

5.1 Experimental evaluation metrics

Throughout the following experiments, there are a couple of factors that need to be considered:

- How well does the learned MLN reflect the data? This will be measured by the validation log-likelihood or the test log-likelihood as is customary in the literature.
- How compact is the SDD of the learned MLN?
- How long did it take to train the MLN?
- What is the convergence behavior of the learning process?

By adjusting the parameter settings, one or more of these metrics will change for the better or worse.

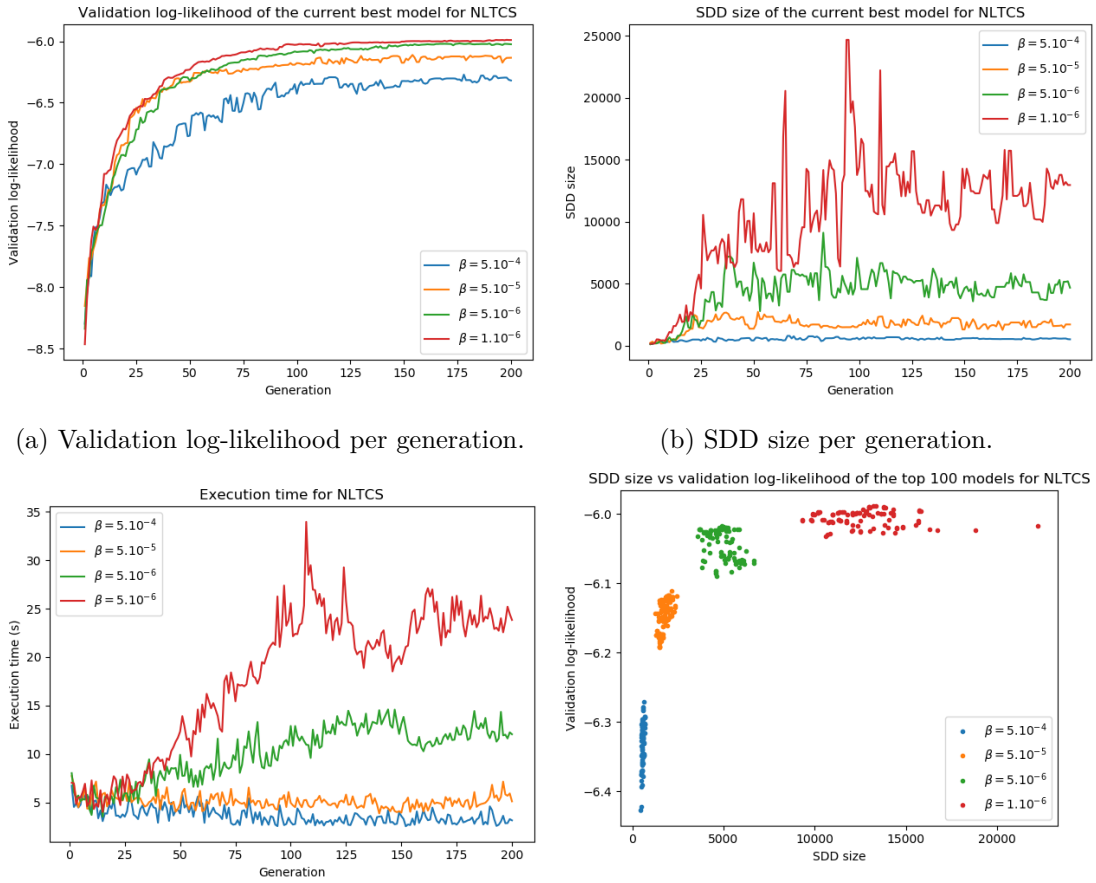
5.2 Effects of the fitness parameter β

Experimental setup
In this experiment, β is varied and all other parameters remain constant. For each value of β considered, GeSDD is run for 200 generations. The validation log-likelihood, SDD size and generation time are plotted against the generation number. Additionally, validation log-likelihood is plotted against the SDD size for the best 100 generations. The full parameter setting for this experiment is shown in table A.1 in the appendix. Figure 5.1 shows the results of this experiment on the NLTCS dataset for $\beta \in \{5.10^{-4}, 5.10^{-5}, 5.10^{-6}, 1.10^{-6}\}$.

Results and findings
From figure 5.1a we can clearly see that when a β is chosen too large, the validation log-likelihood converges to a lower value. From the same figure, we can see that a β chosen smaller will generally increase the value to which the validation log-likelihood will converge. However, there does seem to be diminishing returns. This can especially be seen in figure 5.1d. Additionally, from figure 5.1b and 5.1c a too large β will result in substantially larger SDDs and training time. Especially the SDD size may become a concern, a too large β usually results in a very mild increase in validation log-likelihood but usually a factor increase in SDD size.

Another interesting observations is the fact that, for all β, the variance of the SDD size in figure 5.1b seems substantially larger for the first 125 generations than the last 75. This is most likely due to the fact that the log-likelihood of the models in the first iterations are smaller. This makes the gain in log-likelihood in the fitness function more important than the size. In the later iterations, where the log-likelihood has more or less converged, the size becomes the primary factor in the fitness funtion.

Finally, a last observation one could make, can be seen in figure 5.1d. Here for $\beta \in \{5.10^{-5}, 5.10^{-6}, 1.10^{-6}\}$ seperate clusters seem to form. More notably, however, is the fact that within each cluster there seems to form plateaus or subgroups. This seems to indicate large jumps to individuals with a higher log-likelihood.



(a) Validation log-likelihood per generation.

(b) SDD size per generation.

(c) Execution time per generation.

(d) SDD size versus log-likelihood of the 100 best models

Figure 5.1: Results of varying β on the NLTCS dataset.

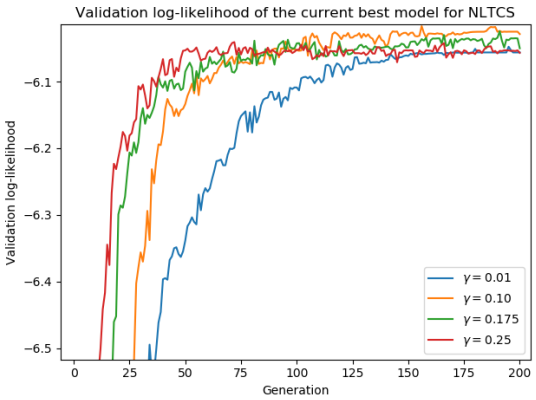
Conclusion
As could have been expected, a trade-off needs to be made between validation log-likelihood and SDD size. However, a drastic decrease in β usually results in unreasonably larger SDDs as a result.

5.3 Effects of the jump size parameter γ

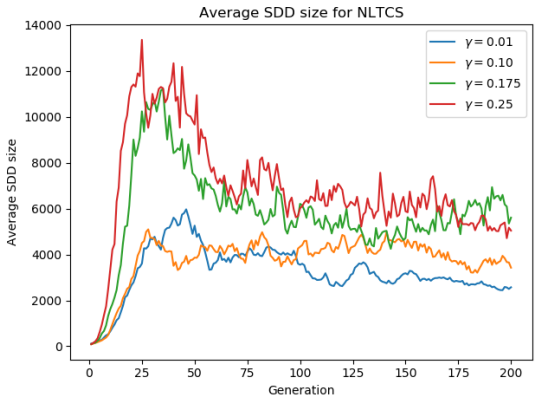
Throughout GeSDD γ was used to denote a percentage of a MLN, as used in section 4.6 for each mutation and in section 4.7 for the crossover. Essentially, this parameter defines how big of a jump we make in the search space (mutation) and how much information is shared among individuals (crossover). In theory each mutation and the crossover has a separate γ which can be fine tuned. However, as this would lead to an explosion of parameter settings and for the sake of simplicity we consider a single γ.

Experimental setup

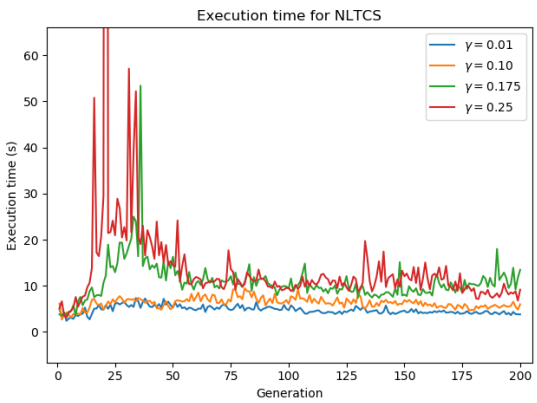
In this experiment, we consider the effects of changing the γ parameter. We do so by keeping all parameters as a constant and varying γ. Then, for each value of γ considered, GeSDD is run for 200 generations. The validation log-likelihood, average SDD size and execution time is plotted against the generation number. Additionally, the validation log-likelihood is plotted against the SDD size for the best 100 generations. The full parameter setting for this experiment is shown in table A.2 in the appendix. Figure 5.2 shows the results of this experiment on the NLTCS dataset for $\gamma \in \{0.01, 0.10, 0.175, 0.25\}$.



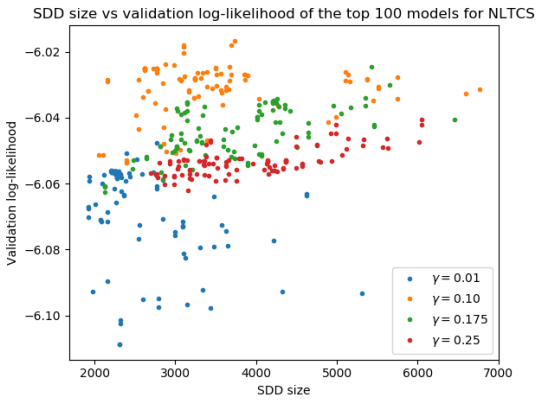
(a) Validation log-likelihood per generation.



(b) SDD size per generation.



(c) Execution time per generation.



(d) SDD size versus log-likelihood of the 100 best models

Figure 5.2: Results of varying γ on the NLTCS dataset.

Results and findings

Figure 5.2 shows some interesting results. As can be seen from figure 5.2a, choosing a low value, e.g. $\gamma = 0.01$ results in generally the slowest convergence. This is especially the case in the earlier generations. Additionally, when $\gamma = 0.01$ GeSDD seems to

converge towards a lower value than $\gamma = 0.10$ and $\gamma = 0.175$. This is perhaps due to the fact that the population is not sufficiently diverse anymore. This coupled with a small jump in the search space results in convergence to a local minimum. On the other hand, choosing a high value, e.g. $\gamma = 0.25$, generally results in faster convergence. Especially so in the earlier generations. However, as a result of the perhaps too large jumps, the validation log-likelihood converges to a smaller value as opposed to $\gamma = 0.175$ or $\gamma = 0.10$. Again, a middle ground must be found for γ.

The trade-off for choosing a low γ can be seen in figure 5.2b, 5.2c and 5.2d. Indeed, a lower γ generally results in a training speedup and a smaller SDD size. While a higher γ generally means larger SDDs and a longer training time. Additionally, the higher γ values generally indicate a more sporadic behavior of the average SDD size throughout the generations. Take for example $\gamma = 0.01$, by generation 150 the average SDD size seems to be steadily declining without many sudden increases. However $\gamma = 0.25$ and $\gamma = 0.175$ show a more sporadic up and down behavior in average SDD size.

The same figures shows a phenomenon which was noted earlier as well. In the earlier generations, the SDD size is significantly larger than the later generations. This in turn causes the training times to become significantly longer. Additionally, they cause the strange anomalies/peaks in training time in figure 5.2c. They are most likely the result of being out of memory and having to swap to disk, causing huge efficiency loss.

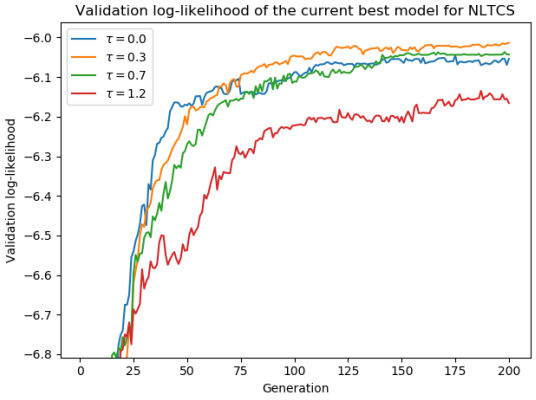
Conclusion
γ should be chosen such that the jumps in the search space are large enough to avoid premature convergence. However, it should be chosen such that convergence is still possible. A too high γ value results in substantially higher training times and SDD size.

5.4 Effects of the threshold parameter τ

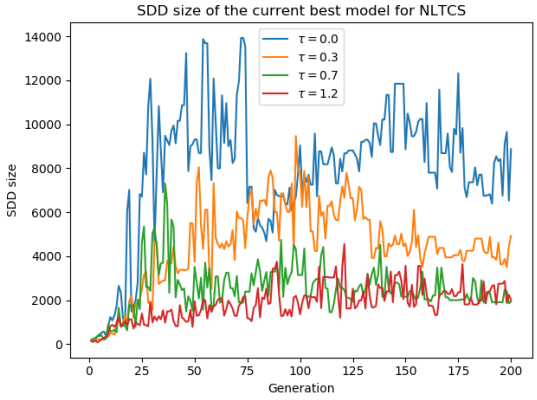
In this section we take a closer look at the MLN trimming heuristic defined in section 4.8.2. Indeed, is it even justified to remove features with a low weight in the first place? After all, shouldn't the fitness function make sure that individuals with large SDDs never survive for a long time?

Experimental setup
In order to study the effects of the heuristic, the following experiment is performed on the NLTCS dataset. All parameters are kept constant and we only vary the threshold parameter τ. Then for each τ considered, the validation log-likelihood, SDD size and execution time are plotted against the generation number. Finally, in order to see how the validation log-likelihood and SDD size interact, they are plotted against each other for the 100 best individuals across the algorithm. The full param-

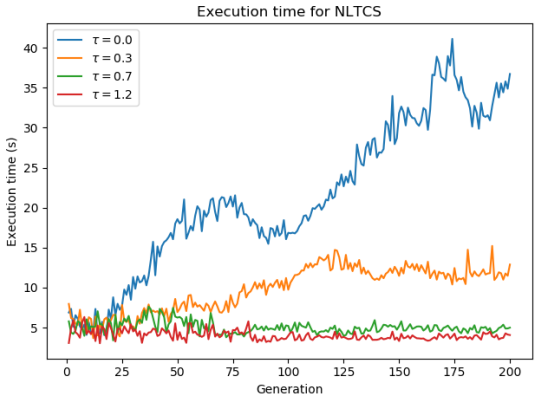
eter setting for this experiment is shown in table A.3 in the appendix. Figure 5.3 shows the results of this experiment on the NLTCS dataset for $\tau \in \{0.0, 0.3, 0.7, 1.2\}$.



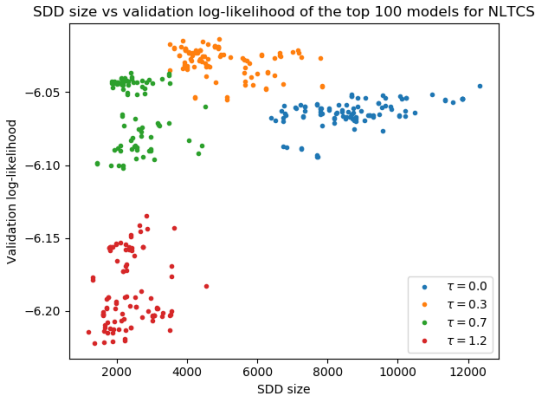
(a) Validation log-likelihood per generation.



(b) SDD size per generation.



(c) Execution time per generation.



(d) Sdd size versus log-likelihood of the 100 best models

Figure 5.3: Results of varrying τ for the NLTCS dataset.

Results and findings

From these experiments we can clearly see the advantage of having a threshold τ that is greater than 0. From figure 5.3a and 5.3d we can see that when $\tau = 0$ the log-likelihood tends to converge to a smaller value than $\tau = 0.3$ and $\tau = 0.7$. This is most probably due to the fact that without trimming, SDDs tend to be larger. This in turn makes the size aspect of the fitness more important than the gain in log-likelihood. While a τ of zero is not advised, a τ too large results in a detriment in validation log-likelihood as well. From figure 5.3a, it becomes clear that $\tau = 1.2$ is substantially too large. It is removing features which clearly contain a lot of information.

The effects of not trimming MLNs also becomes clear in the SDD sizes in fig-

ure 5.3b. In general, the higher τ value the smaller the SDDs per generation. This is to be expected, as high τ values mean that only high informative and non-redundant features are not removed. The same is reflected in figure 5.3c, a τ value of 0 results in longer iteration times without any gain. Perhaps most interesting is the value of $\tau = 0.7$. In this case some validation log-likelihood is sacrificed compared to $\tau = 0.3$ as can be seen in figure 5.3a and 5.3d. However, the SDDs tend to be about twice as small. This additionally has a nice effect of a smaller training time.

Conclusion
From this experiment we can conclude that trimming SDDs is an effective additional heuristic ensuring less informative rules are removed. Trimming MLNs substantially reduces SDD size while mostly not affecting the validation log-likelihood negatively when chosen correctly. The experiments suggest using a value of $\tau \in [0.3, 0.7]$.

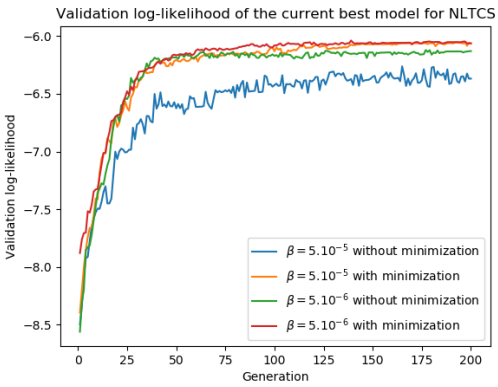
5.5 Effects of the dynamic SDD minimization

This experiment takes a closer look at dynamic SDD minimization mentioned in section 4.8.1. Is the computation cost of minimization worth it?

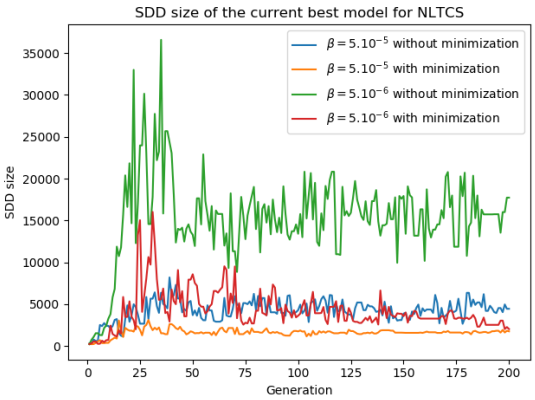
Experimental setup
In order to study the effects of dynamic minimization, the following experiment is performed on the NLTCS dataset. GeSDD is run for 200 generations with two different β values. Both β values are run once with dynamic minimization and then once without dynamic minimization. For each run, the validation log-likelihood, SDD size and execution time are plotted against the generation number. Finally, the SDD size and the validation log-likelihood are plotted against each other for the 100 best individuals across the algorithm. The full parameter setting for this experiment is shown in table A.4 in the appendix. Figure 5.4 shows the results of this experiment on the NLTCS dataset for $\beta \in \{5.10^{-5}, 5.10^{-6}\}$.

Results and findings
From the results in figure 5.4, the effects of dynamic minimization become immediately apparent. Figure 5.4a and 5.4d clearly show the effects on the validation log-likelihood. In both runs where minimization is turned on, the convergence validation log-likelihood increases substantially. This is most likely because the SDD sizes without minimization are substantially larger. As a result, the size penalty in the fitness function is substantially higher. This in turn results in a lower convergence validation log-likelihood.

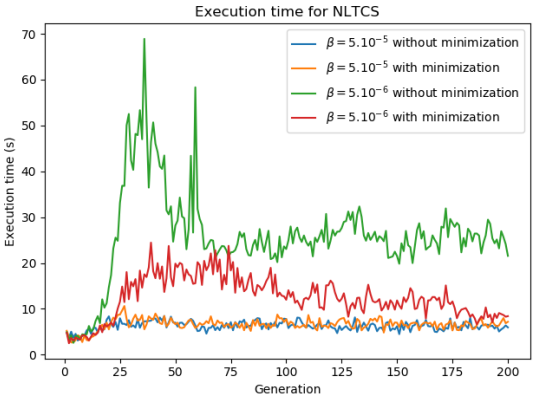
Whether or not dynamic minimization is worth it, is clearly answered by figure 5.4c. Even though minimization itself is quite a computationally heavy task, in the long run it is worth it. This is especially apparent for the case where $\beta = 5.10^{-6}$. This is most likely because weight training, mutations and the crossover computation time scale with the SDD size as described in section 2.5. Thus, minimizing reduces the



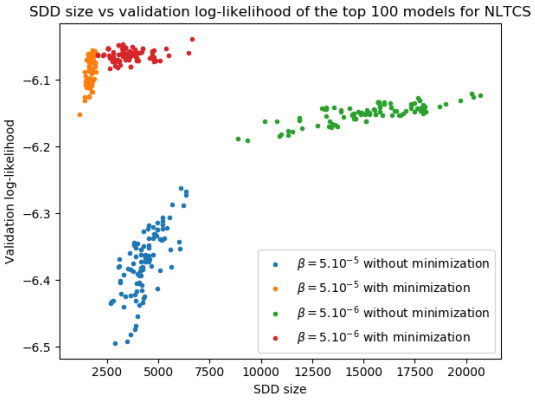
(a) Validation log-likelihood per generation.



(b) SDD size per generation.



(c) Execution time per generation.



(d) Sdd size versus log-likelihood of the 100 best models

Figure 5.4: Results of enabling dynamic minimization.

computation cost of the other operations substantially more than the time required to minimize itself.

Conclusion
Dynamic minimization is extremely important in the context of GeSDD. There is no trade-off to be made and dynamic minimization should always be turned on. The computation time of minimization is negligible because this reduces the computational cost of the genetic operations and weight learning.

5.6 Comparison

In this section a brief comparison is made with LearnSDD, by J. Bekker et al. [4]. It seeks to answer if GeSDD, using genetic algorithms, can potentially provide an advantage when learning MLNs and its SDD.

In order to make a comparison, five different datasets are used. Each dataset is split into a training set, validation set and test set. Table 5.1 shows the different sets with the number of variables and the sizes of each subset. The synth5 and synth8 are synthetic datasets sampled from custom made MLNs. For more information on these MLNs, consult section A.2 in the appendix.

Name	n_{tr}	n_{vl}	n_{ts}	m
Synth5	102.000	9.000	9.000	5
Synth8	102.000	9.000	9.000	8
NLTCS	16181	2157	3236	16
MSNBC	291326	38843	58265	17
Plants	17412	2321	3482	69

Table 5.1: The datasets used for the comparison experiments.

In order to compare both GeSDD and LearnSDD, we consider three evaluation criteria. The first being the log-likelihood of the test set. Secondly, the size of the resulting SDD. Finally, the total training time.

Table 5.2 shows the results of the comparison. GeSDD was run for 300 iterations for each dataset. LearnSDD was run until completion. For the sake of reproducibility, the full parameter settings for each dataset can be found in table A.5, A.6, A.7, A.8 and A.9.

Name	Size G	Size L	TLL G	TLL L	Time (s) G	Time (s) L
Synth5	140	206	-2.415	-2.425	342	34
Synth8	574	3125	-4.475	-4.470	1458	1717
NLTCS	3381	10989	-6.060	-6.030	5871	9084
MSNBC	3288		-6.454		5553	
Plants	3127		-17.20		12868	

Table 5.2: The results of the comparison of GeSDD and LearnSDD.

In this table two entries for LearnSDD are missing. This is due to a segmentation fault that occurs after a certain amount of iterations for the Plants and MSNBC datasets. We were not able to resolve this unexplained segmentation fault in a timely manner due to a planning error on my part.

From the table it can be seen that LearnSDD generally outperforms GeSDD in terms of test log-likelihood. Especially so for the datasets with more variables. On the contrary, GeSDD tends to result in smaller SDDs in general. From the table 5.2 one might suggest GeSDD simply does not achieve a higher log-likelihood than LearnSDD due to a too high β parameter. However, this is not the case as generally LearnSDD achieves a better log-likelihood to size ratio. This can be seen in figure 5.5, where the results of the NLTCS dataset are studied further. Here, the validation log-

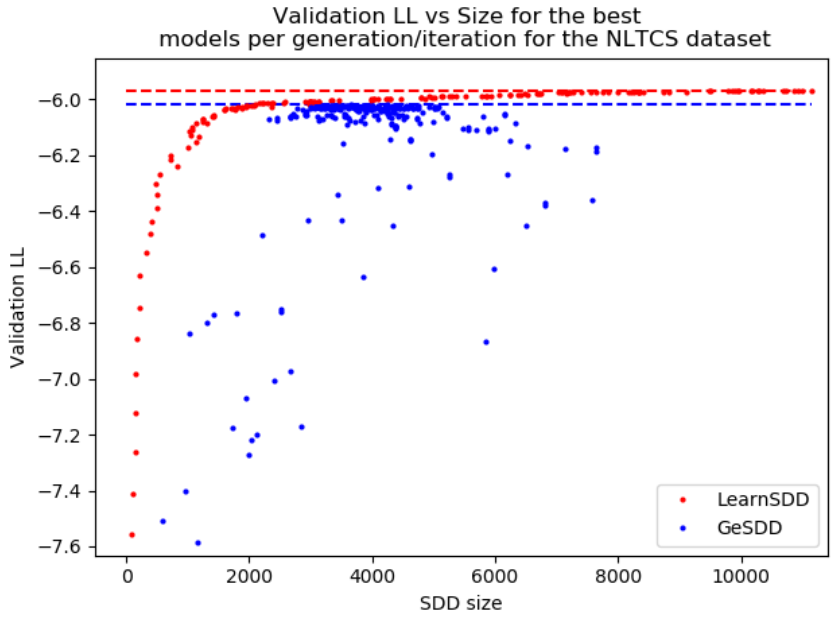


Figure 5.5: Validation log-likelihood vs SDD size of the best models per iteration/generation for the NLTCS dataset.

likelihood is plotted against SDD size for the best model of each generation/iteration of GeSDD/LearnSDD respectively. From this figure it is clear that GeSDD performs worse than LearnSDD on the NLTCS dataset. Due to the random nature of genetic algorithms, the convergence to a good validation log-likelihood is slower. However, due to the greedy nature of LearnSDD, its validation log-likelihood graph grows very rapidly in the early stages. Additionally, the convergence validation log-likelihood of GeSDD is lower than LearnSDD. Finally, from the figure it can be seen that LearnSDD achieves the same validation log-likelihood as GeSDD with a slightly smaller SDD.

In the case of smaller sets, however, GeSDD manages to obtain a similar validation log-likelihood while still being a factor smaller in size than LearnSDD. This can be seen in figure 5.6, where the results of the synth8 dataset are studied further. Again, the validation log-likelihood is plotted against SDD size for the best model of each generation/iteration of GeSDD/LearnSDD respectively. From it we can observe that GeSDD manages to obtain a better validation log-likelihood to size ratio.

From the above observations, we can conclude that the random nature of a genetic algorithm seems to pose a problem. This was the original reason for introducing the feature evaluation method as described in section 4.4. This would enable a more directed and heuristic based search. However, while a feature evaluation method helped, GeSDD still performs worse than LearnSDD.

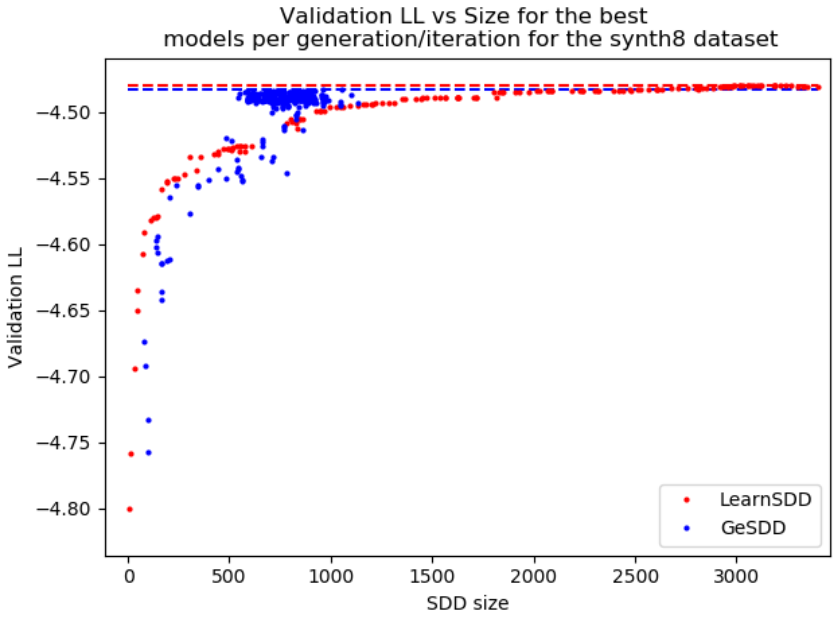


Figure 5.6: Validation log-likelihood vs SDD size for the best models per iteration/generation for the synth8 dataset.

Chapter 6

Conclusion and future work

This section provides a brief conclusion, a recap on the main contributions of this thesis and finally a note on future work.

Distributions form an integral part of most modern machine learning systems. One popular family of distributions comes in the form of Markov Logic Networks. This is a compact, intuitive method of defining a distribution over a set of discrete random variables. However, performing exact inference with MLNs poses a significant problem.

Nevertheless, this problem can be reduced to that of weighted model counting. It can then be further reduced to a knowledge compilation problem. These two reductions means that exact inference is often still tractable. Therefore, a lot of interest has develpoded in learning MLNs directly through circuits such as ACs and SDDs. However, most of the circuit learners are based on greedy approaches.

In this thesis, a closer look was taken into circuit learning using genetic algorithms.

6.1 Main contributions

In this thesis GeSDD was presented, a genetic algorithm for learning MLNs alongside a tractable SDD. Several genetic operators were presented such as several mutations, one crossover and one fitness. Additionally several other heuristics such as MLN trimming and SDD minimization were employed.

Additionally, several parameters and heuristics of GeSDD were inspected in greater detail. More specifically their impact on the learning process, convergence, compactness and validation log-likelihood were discussed.

6.2 Conclusion

This section answers the question:

Is it advantageous to learn MLNs alongside their SDD using genetic algorithms?

While GeSDD seems to offer an advantage in datasets with a small amount of variables, the contrary is true for datasets with a larger amount of variables. The most plausible explanation for this behavior is the inherent random nature of genetic algorithms. This, combined with an exponential search space, prevents GeSDD from finding a better solution within reasonable time.

6.3 Limitations

This section discusses a couple of key limitations concerning GeSDD:

- While GeSDD shows its strong suit in datasets with a small number of variables, it performs considerably worse in datasets with a larger number of variables. This can be seen in section 5.6. The most likely reason for this, is the vast search space GeSDD works in. This combined with the fact that the genetic operations, especially the mutations, are not directed enough. In a sense, finding good features comes down to luck.
- Genetic algorithms are know to be memory and CPU intensive. Even though GeSDD employs parallelism for the mutation, crossover and weight learning, the amount of processing time and memory needed often formed a hurdle.
- In its current form, GeSDD is only able to process Boolean variables efficiently. GeSDD can also process categorical variables by transforming the data using a one-hot encoding, which is usually the way circuit learners handle categorical data. By doing so, some subsets of the transformed data (the ones corresponding to the one-hot encoding) are mutually exclusive. GeSDD does not employ a strategy for dealing with this mutual exclusivity.
- GeSDD employs, in my opinion, a primitive feature evaluation scheme. A better scheme might work in GeSDDs favor.

6.4 Future work

The following are a couple of points of improvement and potentially further research:

- As GeSDD proved worse in the datasets with a large amount of variables, a hybrid approach of some sorts might be worthwhile. For example, it might prove beneficial to bootstrap the initial population of GeSDD with a different learner. This way, GeSDD potentially starts off in a local minimum and would then be able to escape it.

- As an alternative to the above, another suggestion of further research could attempt to iteratively merge a local greedy search with a randomized genetic based search. This would ensure stronger convergence, while potentially escaping the local minima.
- The fitness function of GeSDD is static in the sense that the β parameter don't evolve as the generations go up. However, one could implement a fitness function where β varies through the generations, e.g. slightly decreasing it. This way, we could for example attempt to ensure fast convergence in the early generations and make sure β is slightly smaller in the later generations ensuring convergence to a desired log-likelihood.
- As mentioned in the limitations above, the feature evaluation method GeSDD currently employs seems quite primitive. An improvement in this regard might prove substantial.
- GeSDD limits the feature space to that defined in equation 4.9. However, while a few different spaces were tried, a different feature space might prove more effective in terms of log-likelihood, compactness and convergence speed.
- In the development of GeSDD, a whole range of genetic operations were tried. However, a specific combination of mutations and crossovers might again improve log-likelihood, compactness and convergence speed.

Appendices

Appendix A

Appendix

A.1 Parameter settings

parameter	meaning	value
β	size punishing factor in fitness	varies
$\|P\|$	population size	52
k_f	maximum number of features	100
m_p	mutation probability	0.8
k_t	tournament size	3
γ_m	jump size for mutations	0.05
γ_c	jump size for crossover	0.05
C_n	candidate size for feature generation	3
τ	threshold for trimming	0.2
minimization	minimization enabled?	Yes

Table A.1: Parameter settings for the β experiment on the NLTCS dataset from section 5.2

parameter	meaning	value
β	size punishing factor in fitness	5.10^{-6}
$\|P\|$	population size	52
k_f	maximum number of features	100
m_p	mutation probability	0.8
k_t	tournament size	3
γ_m	jump size for mutations	varies
γ_c	jump size for crossover	varies
C_n	candidate size for feature generation	3
τ	threshold for trimming	0.65
minimization	minimization enabled?	Yes

Table A.2: Parameter settings for the γ experiment on the NLTCS dataset from section 5.3

parameter	meaning	value
β	size punishing factor in fitness	5.10^{-6}
$\|P\|$	population size	52
k_f	maximum number of features	100
m_p	mutation probability	0.8
k_t	tournament size	3
γ_m	jump size for mutations	0.05
γ_c	jump size for crossover	0.05
C_n	candidate size for feature generation	3
τ	threshold for trimming	varies
minimization	minimization enabled?	Yes

Table A.3: Parameter settings for the τ experiment on the NLTCS dataset from section 5.4

parameter	meaning	value
β	size punishing factor in fitness	varies
$\|P\|$	population size	52
k_f	maximum number of features	100
m_p	mutation probability	0.8
k_t	tournament size	3
γ_m	jump size for mutations	0.1
γ_c	jump size for crossover	0.1
C_n	candidate size for feature generation	3
τ	threshold for trimming	0.65
minimization	minimization enabled?	varies

Table A.4: Parameter settings for the minimization experiment on the NLTCS dataset from section 5.5

parameter	meaning	value
β	size punishing factor in fitness	5.10^{-6}
$\|P\|$	population size	52
k_f	maximum number of features	100
m_p	mutation probability	0.8
k_t	tournament size	3
γ_m	jump size for mutations	0.1
γ_c	jump size for crossover	0.1
C_n	candidate size for feature generation	3
τ	threshold for trimming	0.2
minimization	minimization enabled?	Yes

Table A.5: Parameter settings for the synth5 dataset from section 5.6.

parameter	meaning	value
β	size punishing factor in fitness	5.10^{-6}
$\|P\|$	population size	52
k_f	maximum number of features	100
m_p	mutation probability	0.8
k_t	tournament size	3
γ_m	jump size for mutations	0.1
γ_c	jump size for crossover	0.1
C_n	candidate size for feature generation	3
τ	threshold for trimming	0.2
minimization	minimization enabled?	Yes

Table A.6: Parameter settings for the synth8 dataset from section 5.6.

parameter	meaning	value
β	size punishing factor in fitness	5.10^{-6}
$\|P\|$	population size	52
k_f	maximum number of features	250
m_p	mutation probability	0.8
k_t	tournament size	3
γ_m	jump size for mutations	0.08
γ_c	jump size for crossover	0.08
C_n	candidate size for feature generation	3
τ	threshold for trimming	0.5
minimization	minimization enabled?	Yes

Table A.7: Parameter settings for the NLTCS dataset from section 5.6.

parameter	meaning	value
β	size punishing factor in fitness	5.10^{-6}
$\|P\|$	population size	52
k_f	maximum number of features	250
m_p	mutation probability	0.8
k_t	tournament size	3
γ_m	jump size for mutations	0.1
γ_c	jump size for crossover	0.1
C_n	candidate size for feature generation	3
τ	threshold for trimming	0.5
minimization	minimization enabled?	Yes

Table A.8: Parameter settings for the MSNBC dataset from section 5.6.

parameter	meaning	value
β	size punishing factor in fitness	5.10^{-6}
$\|P\|$	population size	52
k_f	maximum number of features	250
m_p	mutation probability	0.8
k_t	tournament size	3
γ_m	jump size for mutations	0.07
γ_c	jump size for crossover	0.07
C_n	candidate size for feature generation	3
τ	threshold for trimming	0.6
minimization	minimization enabled?	Yes

Table A.9: Parameter settings for the Plants dataset from section 5.6.

A.2 Custom MLNs for synth5 and synth8

The synth5 and synth8 datasets were constructed by sampling the distributions of two custom MLNs. One over 5 variables and one over 8. The custom MLNs are defined in table A.11 and A.10.

Feature	weight
$(X_1 \wedge X_2 \wedge \neg X_3) \Rightarrow (\neg X_4 \wedge X_5)$	3
$X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge \neg X_8$	1
$\neg X_1 \vee X_6 \vee \neg X_7$	-2
$(X_8 \wedge X_7 \wedge X_4) \Rightarrow (X_6 \wedge X_4)$	2
$(\neg X_6 \wedge X_4) \Rightarrow \neg X_8$	1.5
$(\neg X_7 \wedge \neg X_4) \Rightarrow (\neg X_8 \wedge X_1)$	-1
$(X_1 \vee \neg X_2) \wedge (\neg X_6 \Rightarrow X_7)$	2

Table A.10: Custom MLN over 8 variables.

Feature	weight
$(X_1 \wedge X_5) \Rightarrow X_2$	2
$(X_2 \wedge X_5 \Rightarrow X_1$	2
$X_1 \Rightarrow X_3$	3
$X_2 \Rightarrow X_4$	3
$(X_1 \wedge X_2) \Rightarrow X_5$	1
$(\neg X_1 \wedge \neg X_4) \Rightarrow (X_2 \wedge X_5)$	2
$(\neg X_2 \vee (\neg X_1 \wedge X_3)) \Rightarrow (X_4 \wedge (X_5 \vee \neg X_2))$	2.5

Table A.11: Custom MLN over 5 variables.

Bibliography

[1] Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[2] Y. M. Assael, B. Shillingford, S. Whiteson, and N. de Freitas. Lipnet: Sentence-level lipreading. *CoRR*, abs/1611.01599, 2016.

[3] N. A. Barricelli. Esempi numerici di processi di evoluzione. *Methodos*, pages 45–68, 1954.

[4] J. Bekker, J. Davis, A. Choi, A. Darwiche, and G. Van den Broeck. Tractable learning for complex probability queries. In *Advances in Neural Information Processing Systems 28 (NIPS)*, Dec. 2015.

[5] S. Bova. Sdds are exponentially more succinct than obdds. *CoRR*, abs/1601.00501, 2016.

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

[7] F. Chen, X. Wang, and Y. Rao. Learning bayesian networks using genetic algorithm* *this project was supported by the national natural science foundation of china (70572045). *Journal of Systems Engineering and Electronics*, 18(1):142 – 147, 2007.

[8] A. Choi and A. Darwiche. Dynamic minimization of sentential decision diagrams. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, pages 187–194. AAAI Press, 2013.

[9] A. Choi, D. Kisa, and A. Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In *Proceedings of the 12th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, ECSQARU'13, pages 121–132, Berlin, Heidelberg, 2013. Springer-Verlag.

[10] G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42(2):393 – 405, 1990.

[11] A. Darwiche. A logical approach to factoring belief networks. In *Proceedings of the Eights International Conference on Principles of Knowledge Representation*

and Reasoning, KR'02, pages 409–420, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[12] A. Darwiche. Sdd: A new canonical representation of propositional knowledge bases. In *IJCAI*, 2011.

[13] A. Darwiche. A differential approach to inference in bayesian networks. *CoRR*, abs/1301.3847, 2013.

[14] A. Darwiche and P. Marquis. A knowledge compilation map. *CoRR*, abs/1106.1819, 2011.

[15] E. David and I. Greental. Genetic algorithms for evolving deep neural networks. *GECCO 2014 - Companion Publication of the 2014 Genetic and Evolutionary Computation Conference*, 07 2014.

[16] Q. Dinh, M. Exbrayat, and C. Vrain. Heuristic method for discriminative structure learning of markov logic networks. In *2010 Ninth International Conference on Machine Learning and Applications*, pages 163–168, Dec 2010.

[17] Q.-T. Dinh, M. Exbrayat, and C. Vrain. Discriminative markov logic network structure learning based on propositionalization and x2-test. pages 24–35, 11 2010.

[18] G. Doquire and M. Verleysen. A comparison of multivariate mutual information estimators for feature selection. *ICPRAM 2012 - Proceedings of the 1st International Conference on Pattern Recognition Applications and Methods*, 1:176–185, 01 2012.

[19] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *FOGA*, 1990.

[20] J. J. Grefenstette, editor. *Proceedings of the 1st International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 1985*. Lawrence Erlbaum Associates, 1985.

[21] J. V. Haaren and J. Davis. Markov network structure learning: A randomized feature generation approach. In *AAAI*, 2012.

[22] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992.

[23] K. Jebari. Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, 3:333–344, 12 2013.

[24] S. Kok and P. Domingos. Learning the structure of markov logic networks. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 441–448, New York, NY, USA, 2005. ACM.

[25] P. Larranaga, M. Poza, Y. Yurramendi, R. H. Murga, and C. M. H. Kuijpers. Structure learning of bayesian networks by genetic algorithms: a performance analysis of control parameters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(9):912–926, Sep. 1996.

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

[27] D. Lowd and P. M. Domingos. Learning arithmetic circuits. *CoRR*, abs/1206.3271, 2012.

[28] M. Maathuis, M. Drton, S. Lauritzen, and M. Wainwright. *Handbook of Graphical Models*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2018.

[29] L. Mihalkova and R. J. Mooney. Bottom-up learning of markov logic network structure. In *Proceedings of 24th International Conference on Machine Learning (ICML-2007)*, Corvallis, OR, June 2007.

[30] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[31] K. Pearson. LIII. On lines and planes of closest fit to systems of points in space, Nov. 1901.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[33] T. Pipatsrisawat and A. Darwiche. New compilation languages based on structured decomposability. volume 1, pages 517–522, 01 2008.

[34] M. Richardson and P. Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, Feb. 2006.

[35] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1):273 – 302, 1996.

[36] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *SAT*, 2004.

[37] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning, 2019.

[38] M. Thurley. sharpsat – counting models with advanced component caching and implicit bcp. In A. Biere and C. P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 424–429, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[39] A. M. TURING. I.-computing machinery and intelligence. *Mind*, LIX(236):433–460, 1950.

[40] N. L. Zhang and D. Poole. Exploiting causal independence in bayesian network inference. *J. Artif. Int. Res.*, 5(1):301–328, Dec. 1996.

Fiche masterproef

Student: Michiel Baptist

Titel: GeSDD: Learning of tractable SDDs using genetic algorithms

Nederlandse titel: GeSDD: Het leren van efficiënte SDDs gebruikmakend van genetische algoritmen

UDC: 681.3

Korte inhoud:

Markov netwerken zijn een parel van moderne machine learning. Ze zijn een compacte en intuïtieve manier om kansverdelingen voor te stellen over multivariate categorische variabelen. Een groot nadeel van zulke netwerken is de moeilijkheid van statistische inferentie, dit is namelijk een exponentieel probleem. Dit probleem stelt zich vooral bij het leren van Markov netwerken. Dit vergt immers statistische inferentie als een subprocedure.

Toch kan men deze netwerken efficient leren door gelijktijdig gebruik te maken van *sentential decision diagrams* (SDDs). SDDs zijn voorstellingen van Booleaanse functies. Ze staan inferentie toe in polynomiale tijd en daardoor ook het efficiënt leren van Markov netwerken. Hierbij bepaalt de grootte van de SDD de efficiëntie van inferentie. Dit maakt het mogelijk om bij het leerproces inefficiënte Markov netwerken te bestraffen.

Deze thesis presenteert GeSDD, een genetisch algoritme voor het leren van efficiënte Markov netwerken samen met hun overeenkomstige SDDs. GeSDD stelt een genetische kruising, meerdere mutaties en een fitness functie voor. Verder worden er twee belangerijke heuristieken voorgesteld en wordt hun belang in het algoritme benadrukt. De eerste is de dynamische minimalisatie, dit kan de grootte van de SDD verkleinen. De tweede is het trimmen van Markov netwerken, deze heuristiek verwijdert weinig invloedrijke attributen.

Tot slot, presenteert deze thesis een studie van de verschillende parameters van GeSDD. Hun impact op het leerprocess, convergentie en uiteindelijk resultaat worden besproken.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Artificiële intelligentie

Promotoren: Prof. dr. ir. Luc De Raedt
Prof. dr. Jesse Davis

Assessoren: Dr. Bregt Verreet
Pedro Zuidberg Dos Martires

Begeleiders: Dr. Jessa Bekker
Pedro Zuidberg Dos Martires