A Mixed-Integer Linear Programming Reformulation Approach to Maximum A Posteriori Inference in Sum-Product Networks

Gustavo Perez Katague

Thesis Submitted
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
for the
Degree of Master of Science

Program:
Computer Science

Advisor:
Prof. Dr. Denis Deratani Mauá

This research was supported by
CNPq & CAPES

São Paulo, November, 2020

A Mixed-Integer Linear Programming Reformulation Approach to Maximum A Posteriori Inference in Sum-Product Networks

This version of the thesis contains the corrections and modifications suggested by the Examining Committee during the defense of the original work, performed on February 3rd, 2020. A copy of the original version is available at the Institute of Mathematics and Statistics from the University of São Paulo.

Examining Committee:

- Prof. Dr. Denis Deratani Mauá - IME-USP
- Prof. Dr. Cassio Polpo de Campos - Eindhoven University of Technology
- Prof. Dr. Fabio Gagliardi Cozman - POLI-USP

Acknowledgements

Firstly, I would like to deeply thank my family, in particular my parents, Marcia and Ricardo, for all the support and patience they gave me. Thank you for being always there for me, specially when it came to keeping my worries at bay. Also, I would like to express my sincere gratitude and admiration towards my advisor, Prof. Dr. Denis Deratani Mauá, for countless hours spent on mentoring meetings, for leading me through the research with enthusiasm and sincerity and for being comprehensive about my conditions. To my girlfriend Sayuri, thank you for being up to come with me through such unconventional paths. You are the best companion I could ever have, even on the other side of the world. No matter what hindrances you found in the way, your strength kept us alive. I love you. To my friend Andre, from whom I learn a lot and with whom I work a variety of thoughts, thank you for always giving me advice about how to deal with myself and for all the adventures in these past years. To my friend Diego, with whom I share dreams of a bright world where we do what we love, thank you for advising me about many of our common past experiences and for being so supportive and comprehensive about everything. To my IME friends, in its many groups (B*lelê, BCC10, Vivência), thank you for giving me these awesome years. It definitely would not be so great without you all. To my FFLCH friends, who have heard me complaining night after night, before, between and after classes. Thank you for always being supportive and for always reminding me that this joyful moment would come. ありがとうね。

古池や
蛙飛びこむ
水の音

Abstract

KATAGUE, G. P. **A Mixed-Integer Linear Programming Reformulation Approach to Maximum A Posteriori Inference in Sum-Product Networks**. Thesis (Masters). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2020.

Sum-Product Network (SPN) is a relatively new class of probabilistic graphical models. They differ from other probabilistic graphical models by allowing explicit representation of context-sensitive independence and marginal inference computation in linear time. Bayesian Networks and Markov Networks, for example, require #P-hard effort for performing marginal inference. However, it is still NP-hard to find the most probable configuration for a set of variables in an SPN, and there is currently a shortage of efficient techniques to solve the problem.

A widely employed technique for solving NP-hard optimization problems consists in translating them into Mixed-Integer Linear Programming (MILP) programs, which hence can be solved by highly efficient commercial solvers. Besides harvesting the power of current solvers, formulating the problem as a MILP program immediately allows us to obtain an *anytime* algorithm that continuously improves its solution as more resources are given (time and memory), and can be stopped at any time with a feasible solution with error bounds.

In this work, we developed a new algorithm that finds the most probable configuration for a set of variables in SPNs (Maximum A Posteriori inference) by reformulating it as a MILP program. This translation is rather intricate and relies on several results scattered throughout this field of study, such as the reformulation of SPNs as Bayesian Networks with latent variables, the compact representation of conditional probability tables through Algebraic Decision Diagrams and the symbolic manipulation of multilinear expressions by Parameterized Algebraic Decision Diagrams.

Keywords: probabilistic graphical models, sum-product networks, maximum a posteriori inference, parameterized algebraic decision diagrams, mixed-integer linear programming.

Resumo

KATAGUE, G. P. **Uma abordagem de reformulação à Programação Linear Inteira Mista para a Inferência de Máximo a Posteriori em Redes Soma-Produto**. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2020.

Rede Soma-Produto (SPN) é uma classe de modelos probabilísticos baseados em grafos relativamente nova. Elas diferem de outros modelos probabilísticos por permitir a representação explícita de independência sensível a contexto e a computação de inferência marginal em tempo linear. Redes Bayesianas e redes de Markov, por exemplo, exigem esforço #P-difícil para computar inferência marginal. Entretanto, continua sendo NP-difícil encontrar a configuração mais provável para um conjunto de variáveis em uma SPN, e atualmente há uma escassez de técnicas eficientes que solucionam o problema.

Uma técnica amplamente utilizada para solucionar problemas de otimização NP-difíceis consiste em transformá-los em um programa de Programação Linear Inteira Mista (MILP), que então poderia ser solucionado por otimizadores de alta performance disponíveis comercialmente. Além de aproveitar o potencial dos otimizadores atuais, formular o problema como um programa MILP nos permite obter um algoritmo *anytime* que continuamente encontra soluções melhores quanto mais recursos (tempo e memória) forem disponibilizados, e pode ser interrompido a qualquer momento obtendo-se uma solução válida com margens de erro.

Neste trabalho nós desenvolvemos um novo algoritmo que soluciona o problema de computar a configuração mais provável (inferência de Máximo A Posteriori) para um conjunto de variáveis em SPNs através de sua reformulação como um programa MILP. Esta reformulação é consideravelmente complexa e se baseia em diversos resultados dispersos pela literatura, tal como a reformulação de SPNs em Redes Bayesianas com variáveis latentes, a representação compacta das tabelas de probabilidades através de Diagramas de Decisão Algébrica, e manipulações simbólicas de expressões multilineares na forma de Diagramas de Decisão Algébrica Parametrizados.

Palavras-chave: modelos probabilísticos baseados em grafos, redes soma-produto, inferência de máximo a posteriori, diagramas de decisão algébrica parametrizados, programação linear inteira mista.

Contents

List of Abbreviations

ADD	Algebraic Decision Diagram
AMP	Argmax-Product
BDD	Binary Decision Diagram
DAG	Directed Acyclic Graph
MAP	Maximum A Posteriori
MILP	Mixed-Integer Linear Programming
MIS	Maximal Independent Set
MP	Max-Product
MS-FC	MaxSearch with Forward Checking
MS-MC	MaxSearch with Marginal Checking
PADD	Parameterized Algebraic Decision Diagram
PGM	Probabilistic Graphical Models
SPN	Sum-Product Network

List of Figures

List of Tables

Chapter 1

Introduction

> "The studies reported here have been concerned with the programming of a digital computer to behave in a way which, if done by human beings or animals, would be described as involving the process of learning. While this is not the place to dwell on the importance of machine-learning procedures, or to discourse on the philosophical aspects, there is obviously a very large amount of work, now done by people, which is quite trivial in its demands on the intellect but does, nevertheless, involve some learning." [Sam59]

The citation above was taken from the introduction of an article called "Some Studies in Machine Learning Using the Game of Checkers", written in 1959 by Arthur Lee Samuel, a pioneer regarding to artificial intelligence research, who spent research time on programming self-learning computers and based much of his work on the famous game *checkers* [MF90]. As far as we know, it is likely to be the first mention ever made about the term Machine Learning.

Since then, several improvements in terms of computational resources have arisen, and new statistical techniques and new models were introduced or existing ones adapted to perform Machine Learning. Among these models, there is a particular class called Probabilistic Graphical Models (PGMs), that encode complex joint multivariate probability distributions using graphs, and given that graphs have been a research subject for long time, the knowledge gathered around them becomes fairly convenient when dealing with such models. Among PGMs, Bayesian Networks, introduced by Judea Pearl in 1985 [Pea85], and Markov Networks are probably the most known nowadays. Most recently, Hoifung Poon and Pedro Domingos proposed a new PGM called Sum-Product Network [PD11], which is one of the focuses of this work.

There are several tasks related to PGMs, such as learning, classification, and inference. Learning concerns using data to estimate the parameters of probability functions and even to assemble the graph structure itself. Classification consists in categorizing unseen instances based on the knowledge stored in the graph structure. Inference relates to computing the marginal distribution of one or more random variables, which is also based on the graph structure. This work does not have learning neither classification under its scope, but focuses upon a specific type of inference called Maximum A Posteriori, which consists in finding the most likely combination of values for a set of variables, given previous observed data.

1.1 Objective

In this research, we aim the development of an anytime algorithm for Maximum A Posteriori (MAP) inference in Sum-Product Networks (SPNs) through its reformulation as a Mixed-Integer Linear Programming (MILP) optimization problem. The first part of this algorithm is based on the ideas proposed in [ZMP15] to efficiently extract a set of Algebraic Decision Diagrams (ADDs) from an SPN, while the second part, where the extracted ADDs are used to build the MILP program, comes as our main contribution.

1.2 Motivation

In order to develop the proposed algorithm, we need to deal with several, and eventually scattered, fields of study within computer science. We believe that it becomes easier to comprehend the applications of this research by giving a concrete and simple example where the algorithm could be useful. Later in this section, we also provide answers for major questions such as why SPNs, MAP inference and the MILP reformulation approach were chosen.

1.2.1 Example: Statistical Imputation

The following situation is described in the book "Missing Data", by Paul Allison [All01]:

> "Suppose you have collected data on a sample of 1000 people and you want to estimate a multiple regression model with 20 variables. Each of the variables has missing data on 5% of the cases, and the chance that data are missing for one variable is independent of the chance that it is missing on any other variable. You could then expect to have complete data for only about 360 of the cases, discarding the other 640. If you merely downloaded the data from a web site, you might not feel too bad about this, although you might wish you had a few more cases. On the other hand, if you had spent $200 per interview for each of the 1000 people, you might have serious regrets about the $130000 that was wasted (at least for this analysis). Surely there must be some way to salvage something from the 640 incomplete cases, many of which may lack data on only one of the 20 variables."

Missing data are a considerable frequent problem when dealing with statistical analysis, although most of the standard statistical methods presume that every case has information on all the variables to be included in the analysis. Although there are proper statistical techniques used to fill the gap of missing data, such as maximum likelihood imputation and multiple imputation, considering the topic of this research, another practicable solution is to find the missing data through MAP inference in a PGM.

Looking back at the situation described above, it would be possible to use the 360 complete instances in the data set to define the structure and probability functions of a PGM, and then performing MAP inference for each 640 incomplete instances of the data set to find the most likely value for each lacking variable.

1.2.2 Example: Structured Prediction

To exemplify these terms in practical matters, let us suppose there is a database with a considerable number of black and white images of human faces, following certain criteria of size, orientation, and inclination. It would be possible to make an SPN learn the patterns of these images, that is, how the variables (in this case, pixels) are related to store in its structure relevant information regarding the dependence between them.

After learning, to verify how accurately the knowledge was stored in the SPN structure, new face images as previously described can be provided to the SPN, but now with a particular condition: missing data. Let us say that half of the image is missing, although the missing part does not necessarily have to be defined as a slice of the image, but any pixels scattered throughout the image. Thus, by performing an inference process (in this case, MAP inference), the SPN fills in the missing data.

Figure 1.1: Examples of SPN image completion without prior face knowledge made by the package GoSPN [Geh]. The grey half is the known part of the picture given to an SPN, and the green half is the most probable combination of values inferred by the SPN.

In fact, this is an exercise of Structured Prediction, which consists in predicting a whole structured object instead of just values. Hence, an artificial missing data occasion, made up for testing the learning process. The most important aspect in our case, the MAP inference executed by the SPN when dealing with the missing data. The completion likelihood naturally depends on how well the learning algorithm captured the dependencies between variables and how accurate is the inference algorithm. Nevertheless, the image completion task itself can be formulated as a MAP inference problem: the computer must seek the combination of the unknown variables that is most likely to occur given a previous configuration of the known variables.

1.2.3 Why Sum-Product Networks?

It would be convenient to have proper models to represent statistical information considering the computer architecture, since our purpose was doing probabilistic reasoning in an automated fashion. There are several established probabilistic models that represent information under uncertainty, as mentioned previously in this chapter, but some complications arise in regard to computational processing. For example, marginal inference, which is to compute the probability of a variable taken to a certain value, takes #P-hard effort both in Bayesian Networks and in Markov Networks [Rot96].

Sum-Product Networks (SPNs) were chosen because of their simple representation of probabilistic distributions in computational terms. They encode a complicated mixture model by a rooted graph whose internal nodes denote arithmetic operations (sum and product) and whose leaves are associated with univariate distributions. They differ from Bayesian Networks and Markov Networks in many characteristics, such as the ability to enable linear-time marginal inference in its size and to capture event-level independence, thus allowing the modeling of complex phenomena such as determinism, context-specific independence and symmetry.

Also, since SPNs have been proposed recently if compared with other well-known established probabilistic graphical models, there are considerably fewer researches developed about it and hence there are several topics related to SPNs that are still to be explored.

1.2.4 Why Maximum A Posteriori inference?

Maximum A Posteriori inference, sometimes referred as Most Probable Explanation depending on the context, consists in finding the most probable configuration for a set of variables over a probabilistic distribution. Computationally speaking, we would like to have a Machine Learning based system that seeks the combination of values for variables that is most likely to occur given a previous configuration of the valued variables. Generally speaking, given a part of a pattern, the computer must seek the most probable pattern that fits the given one. It is still a NP-hard problem in SPNs and there is currently a shortage of efficient techniques to solve the problem. As far as we know, there are few approaches for approximating MAP inference in SPNs. We provide further explanations about these approaches later in this work.

Many tasks can be solved by computing the most probable configuration under a probabilistic model consistent with an evidence, such as image segmentation [GG84], 3D image reconstruction [BVZ98], natural language processing [KRC+10], speech recognition [PKMP14], sentiment analysis [ZNSS11], protein design [SZS+08] and multicomponent fault diagnosis [SS04].

1.2.5 Why Mixed-Integer Linear Programming?

The reformulation of optimization problems through Linear Programming and its derivatives is a very well-known and established technique for solving such kind of problems, and we believe that it brings us several advantages.

Firstly, it allows us to describe our original problem by using the language that describes MILP problems, that is, by using constraints expressed as equations. This can be considered a gain in matters of expressiveness in a way that, for example, we become able to explore

constraints relaxation to have solutions in less time and with error bounds, or even to add new constraints to restrict the scope of possible answers.

Secondly, this approach allows us to develop an anytime algorithm. An anytime algorithm satisfies the following criteria: i) it can be stopped at any time with a valid solution; and ii) the quality of the solution it finds improves monotonically as more resources (time and memory) are given. For our case, MAP inference in SPNs, we might understand ii) such that the probability of the found solution increases monotonically as more resources are given.

And last, since many commercial and efficient optimizers are available nowadays, it allows us to use of any of them to find a reliable solution for our problem.

1.3 Contributions

Our theoretical contribution lies on the second part of the algorithm, which is based on manipulating Algebraic Decision Diagrams (ADDs) and Parameterized Algebraic Decision Diagrams (PADDs) obtained from the original SPN to generate a MILP program that solves the original problem. These ADDs and PADDs will be seen as functions, and thus used to build a domain graph. Then, based on the cliques of this domain graph, a path decomposition can be built and used for message passing. Each message corresponds to a constraint on the MILP program, and the last message specifies the objective function.

Finally, the values assumed by the set of variables after optimizing the MILP problem correspond to the most probable configuration of the variables in the original SPN. The optimum value for the objective function corresponds to the probability value of such configuration in the original SPN.

Since the previous available packages for manipulation of Decision Diagrams would not embed PADD functionality, as a side contribution, we developed an extension software package that supports PADD manipulation into one of the existing ADD packages.

Once the developed algorithm has the characteristic of being anytime, the generated MILP problem can be solved in an anytime fashion by available solvers, that is, this method can be used to compute the variable configuration that leads to the maximum probability in an SPN, but since solvers work continuously improving a feasible solution, it also might give an approximation if the algorithm is prematurely stopped.

Finally, the overall contributions accomplished within this work are:

- the development of a PADD support package
- the development of a anytime algorithm that translates the MAP inference in SPNs as a MILP optimization problem
- a performance analysis of the developed method in comparison with other existing ones
- the publication of the article "Two Reformulation Approaches to Maximum A Posteriori Inference in Sum-Product Networks" [MRKA20], whose content is partly based on this research.

1.4 Text Organization

In this chapter, we present the motivation of our work, explaining why we chose to use SPNs as a probabilistic model, why we chose to direct our research towards the MAP inference problem and why we chose to reformulate it as a MILP problem, thus presenting our contributions within this research area.

- In Chapter 2 we explore the formal definitions of all the models, structures and concepts related with the scope of this work: Sum-Product Networks, Binary Decision Diagrams, Algebraic Decision Diagrams, Parameterized Algebraic Decision Diagrams, Domain Graphs, Path Decompositions and Mixed-Integer Linear Programming.

- In Chapter 3 we discuss the formal definition of the problem we want to solve: the MAP inference. Also, we provide explanations about the existing methods to compute the MAP inference in SPNs: Max-Product, Argmax-Product and MaxSearch.

- In Chapter 4 we present and explain the proposed algorithm for MAP inference in SPNs. Since it can be divided in two major sequential steps, firstly we show how to generate ADDs and PADDs from an SPN restricted by its variables (the reference for this step can be found in [ZMP15] as part of the procedure which converts an SPN in a Bayesian Network). Secondly, we show how to use the set of ADDs and PADDs generated in the first step to obtain the MILP program that translates the MAP inference in the original SPN. As an example, we provide an execution of a simple case throughout the algorithm in both steps of the explanation. Finally, we present a high-level code that summarizes the whole algorithm.

- In Chapter 5 we provide the tests done through synthetic and real learned SPNs, to assure the validity and efficiency of the proposed algorithm.

- In Chapter 6 we recap the concepts and ideas presented in this document as a whole, discussing its bottlenecks and relating to possible future works.

- In Appendix A we present some implementation details about this research that are secondary but still could help readers to understand better our work, such as how we developed part of software during this research.

Chapter 2

Fundamentals

To ensure that this document is self-contained, in this chapter, we present all the models and structures used in this work, as well as the necessary concepts of each of them for fully understanding of the algorithm described in Chapter 4. Firstly, there is a section for the notation used in this work. Later, there are sections dedicated to explain essential concepts of Sum-Product Networks, Decision Diagrams (including Binary Decision Diagrams, Algebraic Decision Diagrams and Parameterized Algebraic Decision Diagrams), Domain Graphs, Path Decomposition and Mixed-Integer Linear Programming.

2.1 Notation

Finding a proper notation when dealing with structures from many different areas of mathematics and computer science is a quite complex task. In this section, we define a standard notation, which however will have its exceptions throughout this document, since we have to deal, for example, with the situation where a constant in one structure becomes a variable in another. We will try to be as clear as possible whenever a notation exception occurs.

Then, we use as a standard:

- *lowercase letters* for constants and functions (the latter denoted usually by the letter f, and eventually both with lower numerical indexes, such as f_1, x_i). We also denote vertices of a graph as i and j, since they can be seen as numerical indexes;
- *uppercase letters* for variables and random variables (also eventually followed by lower numerical indexes, such as X_1, X_i). For Boolean variables, we denote $\bar{X_1}$ as its negation;
- *bold uppercase letters* for sets and vectors of variables, whose dimensions are clear by context;
- *sans serif uppercase letters* for sets, vectors and matrices of constants (also eventually followed by lower numerical indexes, such as A_1, I_i), whose dimensions are clear by context. Exceptionally, we denote the set $\{1, 2, \ldots, n\}$ as $[n]$. We denote the instances of vector as mentioned before, so that x_1 is then the first instance of a vector $\mathsf{X} = (x_1, \ldots, x_n)$. Note that X_1 and X_2 are two distinct vectors;

- *blackboard bold uppercase letters* for mathematical spaces such as integer ($\mathbb{Z}$) and real ($\mathbb{R}$) sets;
- *calligraphic uppercase letters* for structures and models such as SPNs, PADDs, graphs, MILP programs (also followed by lower numerical indexes when necessary, such as $\mathcal{G}_1$). Structures such as SPNs (including their internal nodes) and PADDs can also be seen as functions, so we might eventually denote them as $\mathcal{S}(X_1, \ldots, X_n)$ as well.

Still, the Greek letter λ is usually adopted to denote the indicator function in most articles related to the subject. For a variable X and a fixed value a, the indicator function λ_a is such that

$$\lambda_a(X) = \begin{cases} 1 & \text{if } X = a \text{ or } X \text{ is unknown} \\ 0 & \text{otherwise}. \end{cases}$$

For our purposes, we denote λ_X and $\bar{\lambda}_X$ as indicator variables derived from the indicator functions $\lambda_1(X)$ and $\lambda_0(X)$ when X is a Boolean variable.

2.2 Sum-Product Networks

Generally speaking, Sum-Product Network (SPNs) is a relatively new class of probabilistic graphical model, proposed by Hoifung Poon and Pedro Domingos in 2011 [PD11]. It consists in a Directed Acyclic Graph (DAG) with weighed arcs, leaves that represents variables and internal nodes that represents sum or product operations. They rely on the ideas of Darwiche [Dar03] specially for the notion of network polynomial, and we might as well think of an SPN as a multilinear function of indicator variables. Although we know that a network polynomial has exponential size of terms in the number of variables, SPNs will allow us to represent it and evaluate the probability of evidences in polynomial space and time. As said in [CMdC17], SPNs have received increasing popularity in applications of machine learning due to their ability to represent complex and highly multidimensional distributions, such as in [AT12], [AT16], [CKP+14] and [ND16]. For simplicity, we focus on SPNs with Boolean variables in this work.

Formally speaking, the following definition is close to the one in the SPNs debut paper [PD11]:

Definition 2.2.1. *A* Sum-Product Network (SPN) *is a rooted DAG whose leaves are indicators variables* $\lambda_{X_1}, \ldots, \lambda_{X_n}$ *and* $\bar{\lambda}_{X_1}, \ldots, \bar{\lambda}_{X_n}$ *and whose internal nodes are sums and products. Each arc* (i, j) *emanating from a sum node* i *has a non-negative weight* w_{ij}*.*

As we might note in Definition 2.2.1, we mentioned indicator *variables* instead of indicator *function* as in Section 2.1. We chose this nomenclature because we could interpret an SPN $\mathcal{S}$ as a function whose value is calculated given a configuration of its indicator variables $\lambda_{X_1}, \ldots, \lambda_{X_n}$ and $\bar{\lambda}_{X_1}, \ldots, \bar{\lambda}_{X_n}$. Still, each indicator variable λ_{X_i} and $\bar{\lambda}_{X_i}$ is associated with a univariate probability distribution X_i.

The value of each node in $\mathcal{S}$ depends on a given configuration of the input and are recursively calculated as follows:

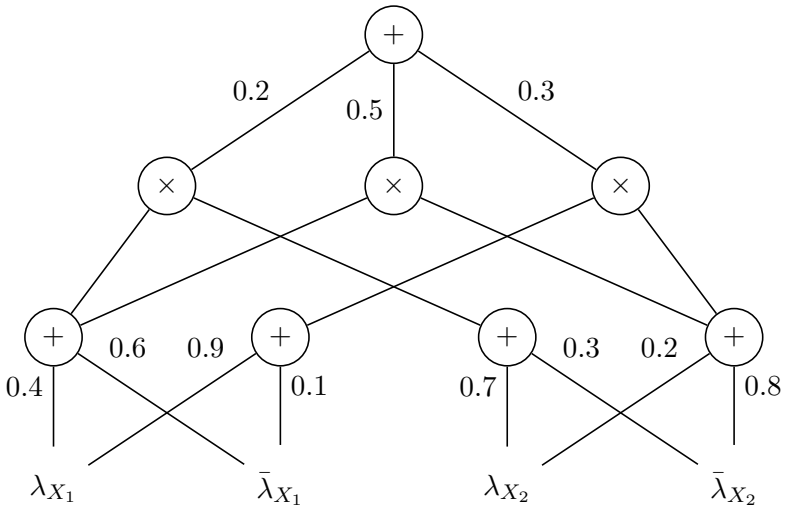


Figure 2.1: A very similar example of a Sum-Product network as showed in [PD11] over variables of X_1 and X_2. The value along the arcs is their weight and for the arcs without a value it is implicit that their weight is 1. Each internal node is associated with a symbol ($+$ or $\times$) when they represent *sum* and *product* operations, respectively. Each leaf node λ_{X_i} or $\bar{\lambda}_{X_i}$ is an indicator variable of X_i in a specific context.

- The value of a leaf node, which will always be an indicator value, is the value of the input itself;
- The value of a product node is the product of the values of its children;
- The value of a sum node i is $\sum_{j \in \mathrm{Ch}(i)} w_{ij} v_j$, where $\mathrm{Ch}(i)$ are the children of i, v_j is the value of the node j and w_{ij} is the arc weight from i to j.

The value of an SPN is the value of its root node. We denote the value of an SPN $\mathcal{S}$ by $\mathcal{S}(\lambda_{X_1}, \ldots, \lambda_{X_n}, \bar{\lambda}_{X_1}, \ldots, \bar{\lambda}_{X_n})$.

The *scope* $\mathrm{Sc}(\cdot)$ of a node in an SPN is defined as the set of variables whose indicators appear among the node's children. For instance, if u is a leaf node, it is an indicator variable λ_{X_i} or $\bar{\lambda}_{X_i}$ over the variable X_i, then we have $\mathrm{Sc}(u) = X_i$. If u is an internal node, then we have it recursively as $\mathrm{Sc}(u) = \bigcup_{v \in \mathrm{Ch}(u)} \mathrm{Sc}(v)$.

The *height* of an SPN is defined as the size of the longest path (that is, the number of arcs) in its graphical representation between the root and a leaf node.

It is shown in Figure 2.1 an intuitive graphical representation of an SPN. Since the value of a product node is the product of the value of its children, it is implicit that the arcs it emanates have weight 1, therefore the weights of the product nodes are omitted. Its height is 3, since it is the number of arcs in the longest path from the root to a leaf node. The leftmost sum node has the scope $\{X_1\}$, since its children are leaf nodes and indicators λ_{X_1} and $\bar{\lambda}_{X_1}$ over the variable

X_1. We can represent this SPN as the function

$$\begin{aligned} S(\lambda_{X_1}, \lambda_{X_2}, \bar{\lambda}_{X_1}, \bar{\lambda}_{X_2}) = 0.2((0.4\lambda_{X_1} + 0.6\bar{\lambda}_{X_1})(0.7\lambda_{X_2} + 0.3\bar{\lambda}_{X_2})) \\ + 0.5((0.4\lambda_{X_1} + 0.6\bar{\lambda}_{X_1})(0.2\lambda_{X_2} + 0.8\bar{\lambda}_{X_2})) \\ + 0.3((0.9\lambda_{X_1} + 0.1\bar{\lambda}_{X_1})(0.2\lambda_{X_2} + 0.8\bar{\lambda}_{X_2}))\,. \end{aligned}$$

For instance, the probability of the complete state $X_1 = 1$ and $X_2 = 0$, where $\lambda_{X_1} = 1$, $\lambda_{X_2} = 0$, $\bar{\lambda}_{X_1} = 0$ and $\bar{\lambda}_{X_2} = 1$, can be computed as

$$\begin{aligned} S(1, 0, 0, 1) &= 0.2((0.4)(0.3)) + 0.5((0.4)(0.8)) + 0.3((0.9)(0.8)) \\ &= 0.024 + 0.16 + 0.216 \\ &= 0.4\,. \end{aligned} \tag{2.1}$$

Moreover, it is important to mention the following definitions so that we can restrict the class of SPNs on which we focus to work.

Definition 2.2.2 (Completeness)**.** *An SPN is* complete *iff all children of the same sum node have the same scope.*

Definition 2.2.3 (Consistency)**.** *An SPN is* consistent *iff no variable appears negated in one child of a product node and non-negated in another.*

Definition 2.2.4 (Decomposability)**.** *An SPN is* decomposable *iff no variable appears in more than one child of a product node.*

We might note that decomposability is more restricted than consistency, which makes SPNs more general than several other representations, such as arithmetic circuits [Dar03], probabilistic context-free grammars [Col03], mixture models [Zha04], junction trees [CG08], which all require decomposability. We might also note that, hence, a decomposable SPN implies a consistent SPN. Peharz et al. [PTPD15] showed that consistent SPNs over discrete random variables can be transformed into equivalent decomposable SPNs with a polynomial increase of size.

With the purpose of relating SPNs to probability distributions, we define an evidence e as

$$\mathsf{e} = \{X_i = b_i : i \in \mathsf{I}\}\,, \tag{2.2}$$

where b_i is a Boolean value and $\mathsf{I} \subseteq [n]$ is a set of indexes. Specifically, in the case where $\mathsf{I} = [n]$, we say that e is a complete evidence. For us, it means that the indicator variables are defined as

$$\lambda_{X_i} = \begin{cases} b_i & \text{if } i \in \mathsf{I} \\ 1 & \text{if } i \notin \mathsf{I} \end{cases} \qquad \bar{\lambda}_{X_i} = \begin{cases} 1 - b_1 & \text{if } i \in \mathsf{I} \\ 1 & \text{if } i \notin \mathsf{I} \end{cases}$$

Thus, we say that $\mathcal{S}(\mathsf{e}) = \mathcal{S}(\lambda_{X_1}, \ldots, \lambda_{X_n}, \bar{\lambda}_{X_1}, \ldots, \bar{\lambda}_{X_n})$, such that λ_{X_i} and $\bar{\lambda}_{X_i}$ are defined as above. For example, if we assume $\mathcal{S}$ to be the SPN shown in Figure 2.1 with $\mathsf{I} = \{1, 2\}$ and $(b_1, b_2) = (1, 0)$, we have that $\mathcal{S}(\mathsf{e})$ has the same result as the one shown in Equation 2.1.

An SPN $\mathcal{S}$ can represent a probability distribution $\Pr_{\mathcal{S}}(X_1, \ldots, X_n)$ over Boolean variables by $\Pr_{\mathcal{S}}(X_1 = b_1, \ldots, X_n = b_n) = \mathcal{S}(b_1, \ldots, b_n, 1 - b_1, \ldots, 1 - b_n)/z$ where z is the partition function. The *partition function* is the sum of the evaluation of all complete evidences e, that is, when $\mathsf{I} = [n]$.

Definition 2.2.5 (Validity)**.** *An SPN $\mathcal{S}$ is* valid *iff $\mathcal{S}(\mathsf{e})/z = \Pr_{\mathcal{S}}(\mathsf{e})$ for all evidence e.*

Completeness and consistency are not necessary, although sufficient, conditions for validity. However, they are necessary for a stronger property that every subnetwork of an SPN to be valid.

Theorem 2.2.1. *An SPN is* valid *if it is complete and consistent.*

A proof for Theorem 2.2.1 can be found in [PD11].

Gens and Domingos [GD13] later redefined SPN as follows:

Definition 2.2.6.

1. *A tractable univariate distribution is an SPN.*
2. *A product of SPNs with disjoint scopes is an SPN.*
3. *A weighted sum of SPNs with the same scope is an SPN, provided all weights are positive*
4. *Nothing else is an SPN.*

The definition above (called *generalized* SPNs by Peharz et al. in [PTPD15]) gives us a broader intuition of what an SPN is, although it implies decomposability. The Definitions 2.2.2 and 2.2.4 are quite important to us because part of our work relies on the SPN to Bayesian Network transformation proposed by Zhao et al. in [ZMP15]. Their reformulation algorithm admits as input an SPN said to be in the *normal form*, whose definition depends primarily on the concepts of *completeness*, *decomposability* and *normalization*. In order to simplify and relate our work with the last mentioned, we present the concept of *normalization* and define the *normal form* of an SPN.

A *normalized* SPN $\mathcal{S}$ is such that, for each sum node in $\mathcal{S}$, the weight of the arcs emanating from it are nonnegative and sum up to 1. Under these conditions, we have that $\mathcal{S}$ represents a normalized probabilistic distribution, that is, $\sum_{\mathsf{e}} \mathcal{S}(\mathsf{e}) = 1$, for all complete evidence e.

Definition 2.2.7 (Normal Form)**.** *An SPN is said to be* normal *if*

1. *It is complete and decomposable.*
2. *It is normalized.*
3. *Every leaf node in the SPN is a univariate distribution over a Boolean variable and the size of the scope of a sum node is at least 2 (sum nodes whose scope is of size 1 are reduced to terminal nodes).*

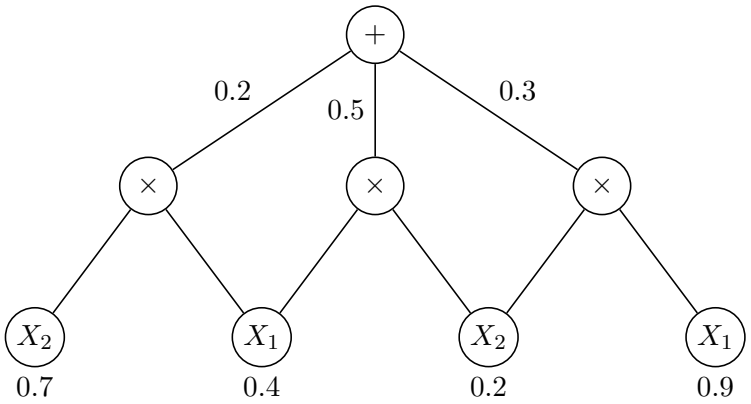


Figure 2.2: A normal SPN based on the one shown in Figure 2.1. We might see that since the original SPN was already normalized, the only difference between them is the representation of the leaf nodes, which previously represented indicator variables of X_i and now represent the variable X_i itself. Below the leaf nodes, only the probabilities $\Pr(X_i = 1)$ are shown.

As we might see, the SPN described in Definition 2.2.7 is contained in the one described in Definition 2.2.6: i) a leaf node is a tractable univariate distribution, ii) a product node is a product of SPNs with disjoint scopes and iii) a sum node is a weighted sum of SPNs with the same scope. For an SPN $\mathcal{S}$ and a vector of random variables $\mathbf{X} = (X_1, \ldots, X_n)$, we denote $\mathcal{S}(\mathbf{X}) = \mathcal{S}(\lambda_{X_1}, \ldots, \lambda_{X_n}, \bar{\lambda}_{X_1}, \ldots, \bar{\lambda}_{X_n})$, where $\lambda_{X_i} = X_i$ and $\bar{\lambda}_{X_i} = 1 - X_i$, for $i \in [n]$.

Theorem 2.2.2. *For any complete and consistent SPN $\mathcal{S}$, there exists a normal SPN $\mathcal{S}'$ such that $\Pr_{\mathcal{S}}(\cdot) = \Pr_{\mathcal{S}'}(\cdot)$ and $|\mathcal{S}'| = O(|\mathcal{S}|^2)$.*

The key points of Theorem 2.2.2 are two:

1. $\mathcal{S}$ and $\mathcal{S}'$ represent the same probability distribution (beside the fact that $\mathcal{S}$ might not be normalized), and

2. $\mathcal{S}'$ can be built within a polynomial increase of size from $\mathcal{S}$.

There is no formal proof of Theorem 2.2.2 in [ZMP15], and we understand that it is not under the scope of this work to show its correctness. However, we assume that the SPN used as input for our algorithm in Chapter 4 is normal, since it is provided that any SPN has its normal form representation.

2.3 Decision Diagrams

Boolean functions are considerably hard to be expressed in computational terms because they yield an exponential number of outputs on the number of inputs. In a broader sense, Decision Diagrams handle such problem because are a compact representation of not only Boolean functions but functions with Boolean domain through Directed Acyclic Graphs (DAGs). In this section, we discuss how Decision Diagrams evolved, firstly introducing Binary Decision Diagrams (BDDs) [Bry86], then Algebraic Decision Diagrams (ADDs) [BFG+97] and finally Parameterized Algebraic Decision Diagrams (PADDs) [DSDB11].

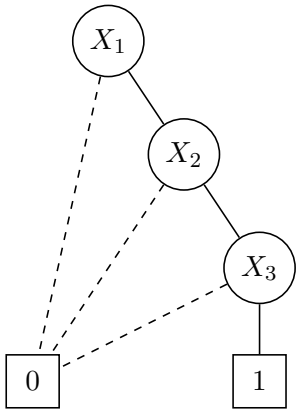
In fact, they share many properties since each of them was proposed successively as a conceptual expansion of the previous. In this work, we are interested in the memory gain and the tractable arithmetic operations provided by such structure, once we might deal with a large number of variables. In addition, all Decision Diagrams mentioned in Chapter 4 are considered PADDs once their expanded structure covers the concepts of BDDs and ADDs.

2.3.1 Binary Decision Diagrams

The basic idea behind BDDs' structure is the Boole's Expansion Theorem[1] [Boo54] (or sometimes referred as Shannon Expansion, since it was part of Shannon's work such as in [Sha38] and in [Sha49]). A first glimpse of what was to become a BDD was the Binary Decision-Programs presented by Chang-Yeong Lee in [Lee59], but the breakthrough came when Randal Bryant published his article about "Graph-Based Algorithms for Boolean Function Manipulation" [Bry86]. There, we find how Boolean functions can be represented by a DAG with some restrictions of how to order the variables. He argued that, although in the worst case a function requires a graph of exponential size in the number of arguments, many of the functions found in common applications have a reasonable representation.

Formally, we might present BDDs as follows:

Definition 2.3.1. *A* Binary Decision Diagram (BDD) *is a graphical representation of a Boolean function* $f : \{0,1\}^n \to \{0,1\}$*, where the graph is a rooted DAG. There are two kinds of nodes in a BDD. Leaf nodes, whose outdegree is 0, are associated with Boolean values. Internal nodes, whose out-degree is 2, are associated with Boolean variables* X_i*,* $i \in [n]$*. For each internal node* X_i*, the dashed out-arc is labeled with* $X_i = 0$ *and the straight out-arc is labeled with* $X_i = 1$*.*



X_1	X_2	X_3	$X_1X_2X_3$
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0
0	0	1	0
1	0	1	0
0	1	1	0
1	1	1	1

Figure 2.3: The Boolean function $f(X_1, X_2, X_3) = X_1X_2X_3$ represented as a BDD. It clearly shows that any variable assuming the 0 (or `FALSE`) value leads the function to return the value 0. It is also possible to see how it avoids redundant information representation in comparison with a tabular representation.

It is shown in Figure 2.3 the representation of a 3-input `and` function as a BDD. We might as well see in literature and software libraries a different representation of BDDs where the `FALSE` and `TRUE` are respectively represented as the left and right out-arc, instead the dashed

[1] For a Boolean funcion f, we have $f(X,Y) = f(1,Y)X + f(0,Y)(1-X)$

and straight lines emanating from the internal nodes. We rather use the latter since it becomes easier to differ the `FALSE` and `TRUE` choice for each variable.

At this point, it is important to mention that the variable layers ordering may change considerably the complexity of a BDD. Figure 2.4 shows how the representation of a BDD depends on the order in which the variables appear. Hence, we might note that the property of canonicity is directly associated with the variable layer ordering. A Boolean function f can be represented by two distinct BDDs (necessarily with distinct layer ordering), which have two distinct canonical forms.

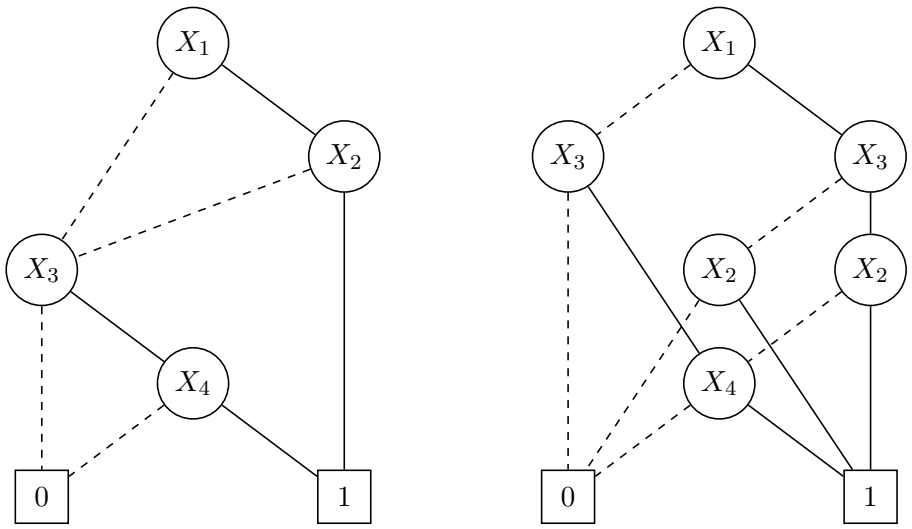


Figure 2.4: Both BDDs represent the function $f(X_1, X_2, X_3, X_4) = X_1X_2 + X_3X_4$. On the left, we can see that the layers of variables (from top to bottom) were defined according to the order they appear in the expression. On the right, we can see that a simple scramble in the order increases considerably the complexity of representation

In Chapter 4 we will be interested in performing some specific operations between PADDs. Since they inherit most of BDDs properties, we might explain these details by now. Together with the BDD structure, Bryant also proposed polynomial time procedures for BDD manipulation [Bry86]: `apply`, `reduce` and `restrict`. For complexity matters, we assume that $\mathcal{B}$ is a BDD representing a Boolean function f through the DAG $\mathcal{G}$ in the following explanations in this subsection.

- `apply`

  The procedure `apply` allows binary operations between two BDDs, providing the basic method for creating the representation of a function according to the operators in a Boolean expression or logic gate network. It takes DAGs representing functions f_1 and f_2, a binary operator `<op>` (that is, any Boolean function of 2 arguments) and produces a DAG representing the function f_1 `<op>` f_2 defined as

  $$[f_1 \texttt{<op>} f_2](X_1, \ldots, X_n) = f_1(X_1, \ldots, X_n) \texttt{<op>} f_2(X_1, \ldots, X_n)\,.$$

  The algorithm starts from the roots of the two DAGs downward, creating vertices in the

resultant DAG at the branching points of the two DAGs.

In other words, let us interpret BDDs $\mathcal{B}_1$ and $\mathcal{B}_2$ as representations of Boolean functions $f_1(X_1, X_3) = \bar{X}_1\bar{X}_3$ and $f_2(X_2, X_3) = X_2X_3$, respectively, as shown in Figure 2.5. If a new BDD $\mathcal{B}_3$ was to represent, let us say, $f_3 = f_1 \vee f_2$, the `<op>` placeholder would turn into the logical `or` Boolean operator. As result, $\mathcal{B}_3$ would represent the Boolean function $f_3(X_1, X_2, X_3) = \overline{X_1\bar{X}_2X_3}$, as shown in Figure 2.6.

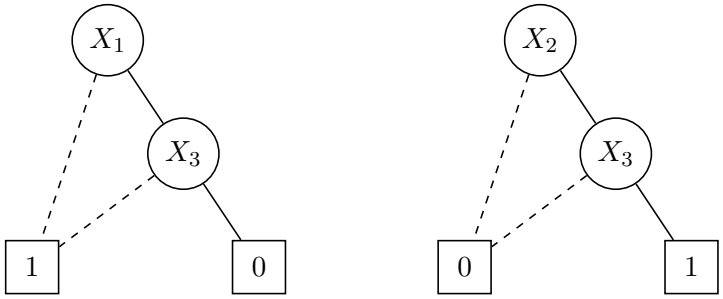


Figure 2.5: BDDs $\mathcal{B}_1$ and $\mathcal{B}_2$ representing of Boolean functions $f_1(X_1, X_3) = \bar{X}_1\bar{X}_3$ and $f_2(X_2, X_3) = X_2X_3$, respectively.

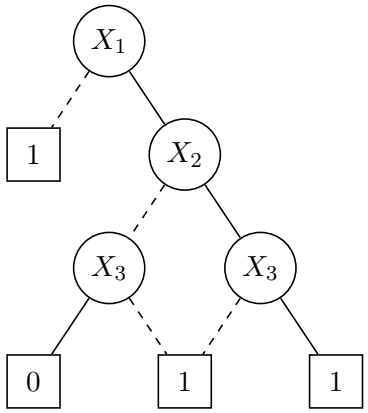


Figure 2.6: BDD $\mathcal{B}_3$ representing the Boolean function $f_3(X_1, X_2, X_3) = \overline{X_1\bar{X}_2X_3}$

However, as we can see from Figure 2.6, the result of the `apply` procedure left the DAG's structure redundant. For that reason, Bryant included, in the end of the `apply`, another procedure called `reduce`, which is explained next.

We might note that it could be quite complex to compute such Boolean operations between Boolean functions if we look over the distributive property, since it depends on the number of terms in each Boolean function. Bryant took advantage of the compact DAG structure to simplify the `apply` procedure, whose polynomial complexity is, including the complexity of the `reduce` in the end, $O(|\mathcal{G}_1| \cdot |\mathcal{G}_2|)$.

- `reduce`

  The `reduce` procedure transforms an arbitrary function DAG into a reduced DAG denoting

the same function, that is, it simplifies the BDD structure to its canonical form. This procedure is essential because many isomorphic BDDs' DAGs can express the same Boolean function, but it is interesting to keep the most compact one. It is implicit that all BDDs are in their canonical form. Figure 2.7 shows how the BDD from Figure 2.6 becomes after the `reduce` procedure.

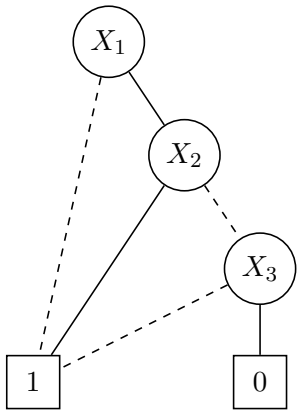


Figure 2.7: BDD $\mathcal{B}_3$ from Figure 2.6 after the `reduce` procedure, yielding a canonical BDD $\mathcal{B}'_3$ that represents the Boolean function $f_3(X_1, X_2, X_3) = \overline{X_1 \bar{X}_2 X_3}$

The complexity of this procedure is $O(|\mathcal{G}| \cdot log(|\mathcal{G}|))$.

- `restrict`

  The `restrict` procedure transforms the DAG of a function $f(X_1, \ldots, X_i, \ldots, X_n)$ into one representing the function $f|_{X_i=a} = f(X_1, \ldots, X_i = a, \ldots, X_n)$ for specified values of $i \in [n]$ and $a \in \{0, 1\}$. It removes the layer where all nodes reference variable X_i and adjust the BDD's DAG connecting dangling nodes which had their parent removed to the nodes that lost a child.

  If the `restrict` procedure was to be used on the BDD shown in Figure 2.7, let us say, by restricting $f_3|_{X_2=0}$, we would have the following BDD shown in Figure 2.8.

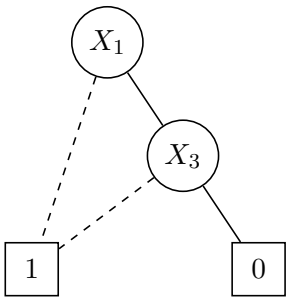


Figure 2.8: BDD $\mathcal{B}_4$ representing the Boolean function $f_4 = f_3(X_1, 0, X_3) = \bar{X}_1 \bar{X}_3$. Note that $\mathcal{B}_4 = \mathcal{B}_1$, hence $f_4 = f_1$, from Figure 2.5.

The computation required for each vertex is constant, and hence its complexity is $O(|\mathcal{G}| \cdot log(|\mathcal{G}|))$. We might note that this procedure could simultaneously restrict several of the

function variables without changing the complexity.

2.3.2 Algebraic Decision Diagrams

In order to create a broader concept for BDDs, Bahar et al. proposed ADDs as an extension of BDDs [BFG+97], allowing not only the representation of Boolean functions but also functions with Boolean domain and real codomain $f : \{0,1\}^n \to \mathbb{R}$. We might see a range of applications for ADDs, such as matrix multiplication in semirings, stochastic planning and SPN to Bayesian Network conversion algorithm.

Formally:

Definition 2.3.2. *An* Algebraic Decision Diagram (ADD) *is a graphical representation of a real function with Boolean input variables $f : \{0,1\}^n \to \mathbb{R}$, where the graph is a rooted DAG. There are two kinds of nodes in an ADD. Leaf nodes, whose outdegree is 0, are associated with real values. Internal nodes, whose out-degree is 2, are associated with Boolean variables X_i, $i \in [n]$. For each internal node X_i, the dashed out-arc is labeled with $X_i = 0$ and the straight out-arc is labeled with $X_i = 1$.*

We might note that the only difference between BDDs and ADDs is that their leaf nodes have values in distinct domains: the first contains leaf nodes with Boolean values, the last, leaf nodes with real values. Figure 2.9 shows an example of an ADD $\mathcal{A}$ that represents the function

$$\mathcal{A}(X_1, X_2) = (3.14)\bar{X}_1\bar{X}_2 + (7)\bar{X}_1X_2 + (7)X_1\bar{X}_2 + (-0.8)X_1X_2\,,$$

where $\bar{X}_i = 1 - X_i$.

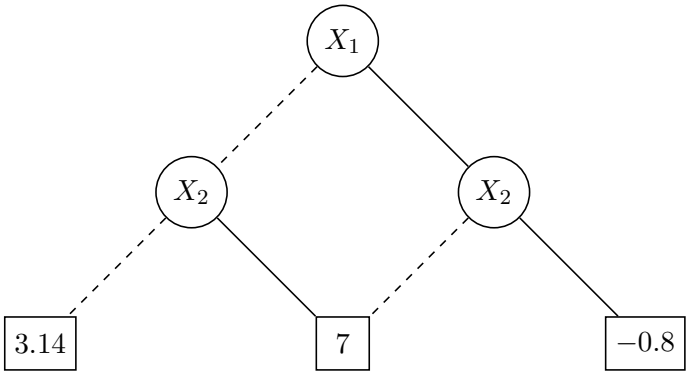


Figure 2.9: An ADD that represents the expression $(3.14)\bar{X}_1\bar{X}_2 + (7)\bar{X}_1X_2 + (7)X_1\bar{X}_2 + (-0.8)X_1X_2$. We might see that its whole structure is similar to the BDD structure with the exception that the values in the leaf nodes have different domains.

For instance, if we evaluate the function it represents assuming that $X_1 = 1$ and $X_2 = 0$, we

would have

$$\begin{aligned}\mathcal{A}(1,0) &= (3.14)(0)(1) + (7)(0)(0) + (7)(1)(1) + (-0.8)(1)(0)\\ &= 0 + 0 + 7 + 0\\ &= 7\,.\end{aligned}$$

ADDs inherited the BDDs' DAG structure, properties of canonicity and variable layers ordering complexity, and the procedures `apply`, `reduce` and `restrict`, but allowing real values in its leaf nodes. However, it is interesting to note that ADDs `apply` procedure requires algebraic operations, such as *sum* and *product*, differently from BDDs, which required operations such as `and` and `or`. Also, the reduce procedure in BDDs worked by means of equality comparison between leaf nodes values, which is quite simple when such nodes contain Boolean values. In the ADDs' context, it depends on a computational comparison between floating point numbers, so it is convenient to set an acceptable precision to determine whether two constants are equal or not.

2.3.3 Parameterized Algebraic Decision Diagrams

To our knowledge, PADDs were firstly introduced by Delgado, Sanner, and de Barros to perform symbolic reasoning with Markov Decision Processes with Imprecise Transition Probabilities (MDP-IP) [DSDB11]. Such problems require the manipulation of multilinear expressions, not supported by traditional ADDs.

Since their goal differs from ours, they presented a slightly different definition from the one that follows:

Definition 2.3.3. *A* Parameterized Algebraic Decision Diagram (PADD) *is a graphical representation of a function with Boolean domain and* $f : \{0,1\}^n \to \mathbb{E}$*, where* $\mathbb{E}$ *is the space of multilinear expressions and the graph is a rooted DAG. There are two kinds of nodes in a PADD. Leaf nodes, whose outdegree is 0, are associated with multilinear expressions. Internal nodes, whose out-degree is 2, are associated with Boolean variables* X_i*,* $i \in [n]$*. For each internal node* X_i*, the dashed out-arc is labeled with* $X_i = 0$ *and the straight out-arc is labeled with* $X_i = 1$*.*

We might see from this definition that its scope is restricted for our purposes, since we want to manipulate multilinear expressions as well. PADDs rely on the structure of Decision Diagrams inherited from ADDs (and hence, BDDs), however, in a much broader sense, they allow the representation of functions whose codomain is any space, including multilinear expressions. As a result, we might note that our definition of PADD encloses the previously given definition of ADD, once a single real constant can be considered a multilinear expression with only one term and no variables.

PADDs' DAG complexity is also sensitive to the variable ordering. Moreover, they inherited the `apply`, `reduce` and `restrict` procedures, which work similarly as in BDDs and ADDs on the matter of structural changes on the DAG.

Still, it is important to mention that PADDs' `apply` is more related to ADDs' `apply`, since multilinear expressions are made over the *sum* and *product* operations, than BDDs' `apply`, that

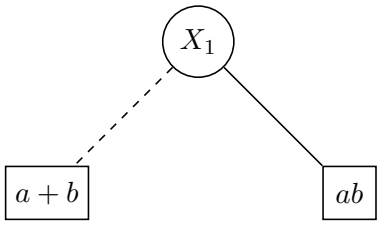


Figure 2.10: A PADD that represents the function $(a+b)\bar{X}_1 + (ab)X_1$. Instead of having values in the leaf nodes, as do BDDs and ADDs, PADDs have symbolic expressions. It is important to note that the variables a and b in the leaf nodes are not decision variables.

is made over, for example, `and` and `or` operations. Delgado, Sanner and de Barros define the *sum* and *product* of PADDs in MDP-IP problems by the operators $\oplus$ and $\otimes$ respectively. Considering that their PADDs have multilinear expressions in the leaf nodes, we might see that $\oplus$ is a closed operation while $\otimes$ is not. This is because the product between two expressions with the same variables yields an expression where such variables carry an exponent, hence not multilinear, while, on the other hand, *product* is closed for polynomials. Their claim is that, in the context of MDP-IP problem, they would never perform any *product* operation between two multilinear expressions that share a same variable, which means that the result would also be a multilinear expression, which also applies for our work.

For the `reduce` procedure, it is important to define the relationship of equality between two objects in the leaf nodes. This is because reducing a PADD would imply to seek two or more leaf nodes that contain the same object to merge them into one only leaf node, for later adjusting properly the out-arcs from some internal nodes. Then, a *reduced PADD*, for a given variable ordering, is such that its size is minimal. The proof of existence and uniqueness is mostly based on BDDs concepts, besides the fact that it is necessary to ensure that there exists a way to identify two leaf nodes with the same object. Since we are dealing with multilinear expressions as objects in the leaf nodes, this could be done by:

1. sorting the previously indexed variables in each expression term
2. factoring out and summing constants in identical expression terms
3. sorting the list of terms according to the lowest variable index and number of parameters (as a lexicographical order by indexes)

These three steps were presented in [DSDB11] where PADDs contain only multilinear expressions. However, it is not hard to see that it could work as well for expressions such as polynomials, as long as terms where the same variable has different exponents, such as X^3 and X^7, have a unique representation. And similarly to ADDs, it is necessary to define an acceptable precision to determine whether two real constants are equal or not.

Finally, once $\oplus$ is well defined, the `restrict` procedure can be used to compute the sum of

the `FALSE` and `TRUE` restricted functions for a variable ($f|_{X_i=0} \oplus f|_{X_i=1}$) eliminating the variable X_i from the PADD. This unary operation is called *marginalization*, which is closed since $\oplus$ is closed for PADDs. *Marginalization* plays an important role in the algorithm proposed here, as it will be explained in Subsection 4.2.1.

2.4 Domain Graphs

The *domain graph* of a set of functions is a graph whose nodes are the variables appearing in any of these functions, and such that two nodes are connected by an edge if and only if the two corresponding variables appear in the same function.

For instance, let us consider the following set of functions:

$$\mathsf{S} = \{f_1(X_1, X_2), f_2(X_1, X_2, X_3), f_3(X_3, X_4, X_5)\}\,.$$

Figure 2.11 shows how graph $\mathcal{G}$ is built over the variables that appear as argument in the functions of set S.

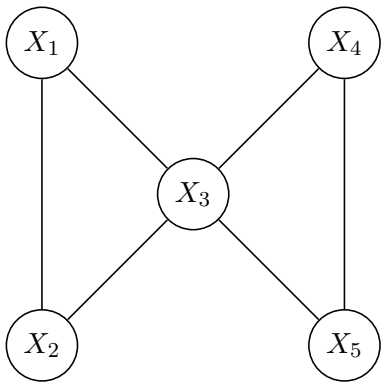


Figure 2.11: The domain graph $\mathcal{G}$ over the variables X_1, X_2, X_3, X_4 and X_5, which are used as arguments of functions f_1, f_2 and f_3 in set S. We can see, for example, that X_1 and X_4 have not appeared together as arguments of the same function.

We might note that all variables that appear as argument for a specific function will appear connected in the domain graph, hence, generating a clique in $\mathcal{G}$.

2.5 Path Decompositions

A *path decomposition* of a graph $\mathcal{G}$ is a new graph $\mathcal{P}$ whose nodes are cliques of $\mathcal{G}$ and such that it satisfies the following properties:

1. Every node of $\mathcal{P}$ has at most two neighbors.

2. For any variable/node in $\mathcal{G}$, the subgraph obtained by considering only nodes of $\mathcal{P}$ that contain that variable is connected. Which is also known as the running intersection property.

3. The union of all variables/nodes associated to some node of $\mathcal{P}$ is the set of variables/nodes in $\mathcal{G}$.

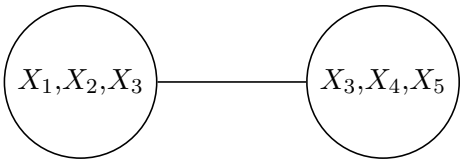


Figure 2.12: The path decomposition for the domain graph $\mathcal{G}$ shown in Figure 2.11. We might see that the set $\{X_1, X_2, X_3\}$ composes a clique in $\mathcal{G}$ as well as $\{X_3, X_4, X_5\}$, and hence, they both become nodes in $\mathcal{P}$.

A path decomposition is so-called because it organizes the variables in cliques of $\mathcal{G}$ in a path. We might note that the path decomposition can be applied to any graph, however we will be interested to apply it specifically over the domain graphs mentioned in section 2.4, since it will be part of the algorithm described in Chapter 4. Figure 2.12 shows the path decomposition of the domain graph shown in Figure 2.11.

While obtaining a path decomposition of small size (given by the size of the largest node in $\mathcal{P}$) is NP-hard, there are efficient heuristics, such as shown in Bodlaender et al. [BGHK95] and Catell et al. [CDF96].

2.6 Mixed-Integer Linear Programming

Mixed-integer linear programming (MILP) is a method used to find the maximum or minimum value of a function under a set of constraints. Differently from *linear programming* and *integer programming*, where all variables are assumed to be continuous and integers, respectively, MILP allow us to optimize functions with variables from both domains: continuous and integers.

A MILP program can be formulated as following:

$$\begin{aligned} \text{maximize } & \mathsf{C}^T\mathbf{X} \\ \text{subject to } & \mathsf{A}_1\mathbf{X} \leq \mathsf{B}_1 \\ & \mathsf{A}_2\mathbf{X} = \mathsf{B}_2 \\ & \mathsf{A}_3\mathbf{X} \geq \mathsf{B}_3 \\ & \mathbf{X} \in \mathbb{Z}^n \times \mathbb{R}^p\,, \end{aligned}$$

where C and B_i are vectors of known coefficients used in the objective function and constraints, respectively; A_i is a matrix of known coefficients used in the set of constraints; and $\mathbf{X}$ is the vector of n integer variables and p continuous variables.

In this work, we will be focus on MILP programs where the integer variables are, in fact, restricted to the Boolean domain. Hence, the last constraint shown in the formulation above would actually be

$$\mathbf{X} \in \{0,1\}^n \times \mathbb{R}^p\,.$$

Chapter 3

Current Algorithms for MAP Inference

In this chapter, we present a brief explanation of the current solutions for MAP inference in SPNs: *Max-Product* and *Argmax-Product*, which are approximate algorithms; and the *MaxSearch*, a branch-and-bound exact solver.

Some of these algorithms focus on approximate solutions, which we understand as considerably more efficient than a brute force algorithm, that demand exponential effort to compute all solutions. The branch-and-bound approach for the exact solution is more related to our work, since it is also an anytime algorithm. Although not necessary for the understanding of the algorithm proposed in this work itself, these are solutions for the same problem, and mentioning them would clarify the context of MAP inference in SPNs and contrast with the algorithm developed in this research, which has as goal an exact solution for MAP inference in SPNs through a MILP reformulation approach.

Let us recall Equation 2.2 for an evidence e and assume $\mathbf{X} = (X_1, \ldots, X_n)$, where X_i is a random Boolean variable, $i \in [n]$. We denote $\mathsf{X} \sim \mathsf{e}$ when X is a value of $\mathbf{X}$ consistent to e (that is, the projection of X onto $\mathsf{I} \subseteq [n]$ is e). Finally, we present a formal definition of the problem we aim to solve as follows:

Definition 3.0.1 (MAP Inference)**.** *Given an SPN $\mathcal{S}$ and an evidence e, find X^* such that $\mathcal{S}(\mathsf{X}^*) = \max_{\mathsf{X} \sim \mathsf{e}} \mathcal{S}(\mathsf{X})$.*

We might understand it as a problem where we want to maximize $\mathcal{S}$ only for variables X_j where $j \notin \mathsf{I}$, since X_i, $i \in \mathsf{I}$, is part of evidence e and already has an assignment.

3.1 Max-Product

The first approximation for MAP inference in SPNs is the *Max-Product* algorithm, which takes linear time in the size of the network. It was proposed by Poon and Domingos in the article where they introduced SPNs as a new PGM [PD11].

Their approximation can be computed by replacing the sum nodes by max nodes. The first step is to sweep the SPN from the bottom to the top, when the max nodes output the maximum

weighted value among its children instead of their weighted sum, as did a sum node. We might note that the new value of the SPN, that is, the value of the root node, does not correspond to a any probability value.

After reaching the root node, the last step is to trace back the variables configuration that originated the maximized value of the root node. It starts from the root node and recursively selects the highest-valued child of a max node, or any highest-valued child if there are more than one, and all children of a product node. According to Definition 2.2.4, if an SPN is decomposable, each variable will be chosen only once by the end of the algorithm.

The following figures show how the algorithm proceeds through the normal SPN shown in Figure 2.2.

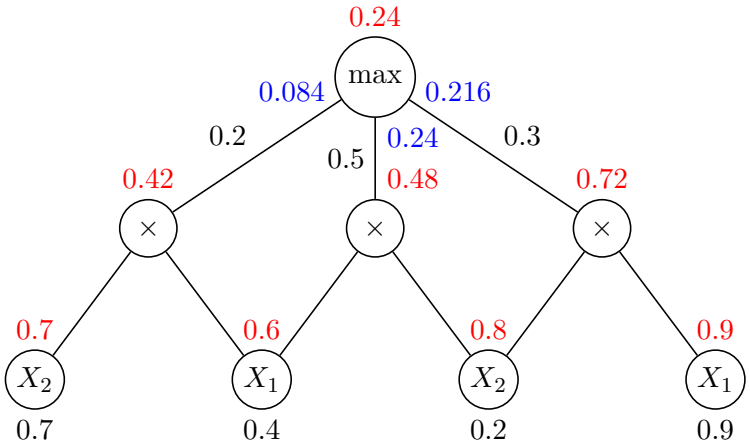


Figure 3.1: The upward pass of the Max-Product. The values of each node of the SPN are marked in red above them. The values beside and below the max node are marked in blue and represent the weighted values from its children. We might note that the max node value is 0.24 since it is the value of its highest input, and also that the center leaf nodes X_1 and X_2 yielded the values 0.6 and 0.8, respectively, since the assignments $X_1 = 0$ and $X_2 = 0$ maximize their output.

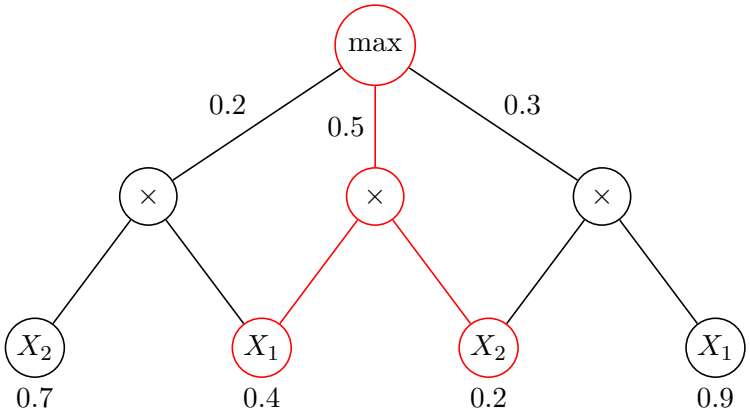


Figure 3.2: The downward pass of the Max-Product. The red nodes are the result after i) selecting the highest value input into a max node, which came from the center product node; and ii) selecting all children of a product node.

In fact, the output value 0.24 of the max node is not the final Max-Product approximation. As we might note from Figure 3.2, the selected variables X_1 and X_2, which can be seen as sum nodes, recalling the first SPN representation from Figure 2.1, output 0.6 and 0.8 from the assignments $X_1 = 0$ and $X_2 = 0$, respectively. Hence, if we assume these assignments for X_1 and X_2, we finally obtain 0.3 in the original SPN value, which is not the exact solution for the MAP inference, as it will be shown later in Chapter 4

3.2 Argmax-Product

The *Argmax-Product* algorithm was proposed by Conaty, Mauá and de Campos [CMdC17] as a variant of the Max-Product. In their paper, they were concerned about proving the NP-hardness of the MAP inference in SPNs, arguing that it can be solved efficiently in SPNs of height less or equal to one, but it is NP-hard to approximate when the height is strictly greater than one. Their proof is based on the reduction of MAP inference problem from the NP-hard problem Maximal Independent Set (MIS) in graphs, which relates to deciding whether there is a subset of vertices of a certain size such that no two vertices in the set are connected. From that, they showed how to build an SPN of height two that encodes the MIS problem. As an advantage of this result, they obtained a corollary of the non-approximability of MAP inference within a sublinear factor in SPNs of height two.

By applying simple modifications in the Max-Product, they proved that the Argmax-Product returns i) a solution at least as good as the one returned by the Max-Product for any instance; and ii) solutions exponentially better than the Max-Product for some instances. Tests were made with synthetic SPNs, which encode instances of the MIS problem described previously, and SPNs learned from real-world data to compare the performance of the Argmax-Product with the Max-Product in several structured prediction tasks. Although Argmax-Product complexity is quadratic in the size of the SPN, their empirical results have shown that it finds significantly better solutions than the Max-Product with a small increase in runtime. This improvement is less pronounced in networks learned from real data, but it still is a viable and improved alternative.

After all, the algorithm is presented as it follows, assuming an SPN $\mathcal{S}$ and an evidence $\mathtt{e}$, where `amap` stands as a short for *Argmax-Product*. Later, we offer a more detailed explanation concerning how the algorithm proceeds, using again the normal SPN shown in Figure 2.2.

- If $\mathcal{S}$ is a sum node with children $\mathcal{S}_1, \ldots, \mathcal{S}_t$, then compute

$$\text{amap}(\mathcal{S}, \mathtt{e}) = \arg \max_{\mathsf{X} \in \{\mathsf{X}_1, \ldots, \mathsf{X}_t\}} \sum_{i=1}^{t} w_i \cdot \mathcal{S}_i(\mathsf{X})$$

  where $\mathsf{X}_k = \text{amap}(\mathcal{S}_k, \mathtt{e})$, that is, X_k is the solution of the MAP inference problem obtained by Argmax-Product for network $\mathcal{S}_k$ (it runs from the bottom to the top)

- Else if $\mathcal{S}$ is a product node with children $\mathcal{S}_1, \ldots, \mathcal{S}_t$, then $\text{amap}(\mathcal{S}, \mathtt{e})$ is the concatenation of $\text{amap}(\mathcal{S}_1, \mathtt{e}), \ldots, \text{amap}(\mathcal{S}_t, \mathtt{e})$

- Else, $\mathcal{S}$ is a leaf node, and $\text{amap}(\mathcal{S}, \mathtt{e}) = \arg\max_{\mathsf{X} \sim \mathtt{e}} \mathcal{S}(\mathsf{X})$

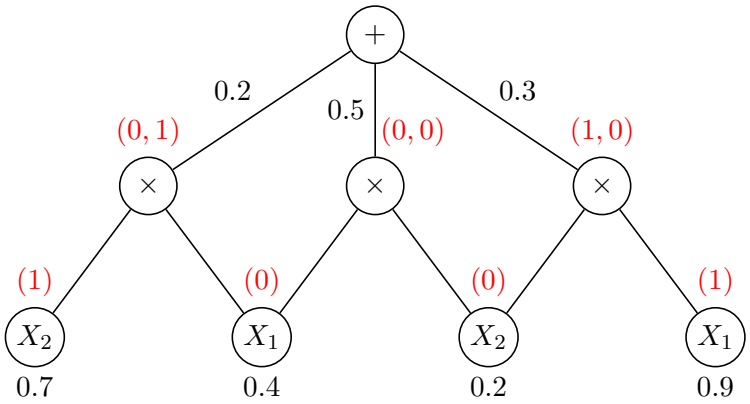


Figure 3.3: The status of the Argmax-Product algorithm after completing two layers of nodes. The tuples marked in red above each node represent the argument returned from `amap` evaluation. We might note that the tuples above the leaf nodes have only one value, since leaf nodes have only themselves as scope. The tuples above the product nodes are concatenations of the results from their children and are sorted according to the variables indexes.

As we can see, Figure 3.3 shows how the algorithm proceeds for leaf and product nodes. Since the sum node requires more complex calculations, we decided to present its last iteration by writing all expressions computed until its conclusion.

Then, we can assume that $\mathcal{S}$ is the last sum node, which gives us $t = 3$, and let us say that $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ are the product nodes from left to right, in order. From this indexation, we have $(\mathsf{X}_1, \mathsf{X}_2, \mathsf{X}_3) = ((0,1),(0,0),(1,0))$ and $(w_1, w_2, w_3) = (0.2, 0.5, 0.3)$. The goal is to maximize $\sum_{i=1}^{3} w_i \cdot \mathcal{S}_i(\mathsf{X})$ by evaluating the options of $\mathsf{X} \in \{\mathsf{X}_1, \mathsf{X}_2, \mathsf{X}_3\}$.

$$\mathsf{X} = \mathsf{X}_1 \Longrightarrow \begin{Bmatrix} \mathcal{S}_1(\mathsf{X}_1) = 0.42 \\ \mathcal{S}_2(\mathsf{X}_1) = 0.12 \\ \mathcal{S}_3(\mathsf{X}_1) = 0.02 \end{Bmatrix} \Longrightarrow \begin{Bmatrix} w_1 \cdot \mathcal{S}_1(\mathsf{X}_1) = 0.084 \\ w_2 \cdot \mathcal{S}_2(\mathsf{X}_1) = 0.060 \\ w_3 \cdot \mathcal{S}_3(\mathsf{X}_1) = 0.006 \end{Bmatrix} \Longrightarrow \sum_{i=1}^{3} w_i \cdot \mathcal{S}_i(\mathsf{X}_1) = 0.15 \tag{3.1}$$

$$\mathsf{X} = \mathsf{X}_2 \Longrightarrow \begin{Bmatrix} \mathcal{S}_1(\mathsf{X}_2) = 0.18 \\ \mathcal{S}_2(\mathsf{X}_2) = 0.48 \\ \mathcal{S}_3(\mathsf{X}_2) = 0.08 \end{Bmatrix} \Longrightarrow \begin{Bmatrix} w_1 \cdot \mathcal{S}_1(\mathsf{X}_2) = 0.036 \\ w_2 \cdot \mathcal{S}_2(\mathsf{X}_2) = 0.240 \\ w_3 \cdot \mathcal{S}_3(\mathsf{X}_2) = 0.024 \end{Bmatrix} \Longrightarrow \sum_{i=1}^{3} w_i \cdot \mathcal{S}_i(\mathsf{X}_2) = 0.30 \tag{3.2}$$

$$\mathsf{X} = \mathsf{X}_3 \Longrightarrow \begin{Bmatrix} \mathcal{S}_1(\mathsf{X}_3) = 0.12 \\ \mathcal{S}_2(\mathsf{X}_3) = 0.32 \\ \mathcal{S}_3(\mathsf{X}_3) = 0.72 \end{Bmatrix} \Longrightarrow \begin{Bmatrix} w_1 \cdot \mathcal{S}_1(\mathsf{X}_3) = 0.024 \\ w_2 \cdot \mathcal{S}_2(\mathsf{X}_3) = 0.160 \\ w_3 \cdot \mathcal{S}_3(\mathsf{X}_3) = 0.216 \end{Bmatrix} \Longrightarrow \sum_{i=1}^{3} w_i \cdot \mathcal{S}_i(\mathsf{X}_3) = 0.40 \tag{3.3}$$

We can confirm from Equations 3.1, 3.2 and 3.3 that the Argmax-Product approximation for the SPN in Figure 2.2 returns $\mathsf{X}_3 = (1, 0)$, with $\mathcal{S}(\mathsf{X}_3) = 0.4$. In fact, this SPN was convenient since the Argmax-Product solved the MAP inference problem exactly, returning the actual probabilities for $\mathcal{S}(\mathsf{X}_1)$, $\mathcal{S}(\mathsf{X}_2)$ and $\mathcal{S}(\mathsf{X}_3)$. As discussed previously, a improved solution if compared to the one given by Max-Product.

3.3 MaxSearch

Mei, Jiang and Tu discuss in [MJT18] some theoretical results about the MAP inference in SPNs, proving its approximation is NP-hard with a similar (simpler, but weaker) result as Conaty, Mauá and de Campos did in [CMdC17]. Also, they propose some approximation algorithms as solutions for the problem, and among them a exact branch and bound solver, which they claim to run reasonably fast and it could handle SPNs with up to 1000 variables and 150k arcs.

Since MAP inference is NP-hard in SPNs, there is no efficient solution for MAP inference in SPNs, although they say that a combination of pruning, heuristic and optimization might allow us to have a reasonably fast exact solver. Their technique is based on a systemic search within the configuration space of the MAP inference, using marginal checking for pruning such space and then setting an upper-bound value for the solution. As a standard branch-and-bound algorithm, it is necessary to start from an initial valid solution, which, in this case, could be simply obtained by assuming random values for each variable, or even by using the previous algorithms to provide a solution closer value to the MAP inference solution (if not the MAP inference solution itself).

To better explain their algorithm, we use their notation $\text{val}(X)$ to represent the set with all possible values that a random variable X can assume, and also, for a variable $X \in \mathbf{X}$, $\mathsf{X}[X]$ as the projection of X onto X. Hence, for a $\mathbf{Y} \subseteq \mathbf{X}$, $\mathsf{X}[\mathbf{Y}]$ is the projection of X onto $\mathbf{Y}$. For a initial solution X and the set $\mathsf{S} = \text{val}(\mathbf{X})$, where $\text{val}(\mathbf{X}) \coloneqq \times_{i \in [n]} \text{val}(X)$, the following scheme shows how the procedure `search`(S, X) works.

1. Take for instance a variable X where $|\mathsf{S}[X]| > 1$
2. If there is no such variable, return the only element in S
3. For all elements $x \in \mathsf{S}[X]$, assume $\mathsf{S}' = \{x\} \times \mathsf{S}[\mathbf{X} \setminus \{X\}]$ and call

$$\mathsf{S}' \leftarrow \text{checking}(\mathsf{S}', \mathsf{X})$$

   - If $\mathsf{S}' = \varnothing$, call

$$\mathsf{X} \leftarrow \text{search}(\mathsf{S}', \mathsf{X})$$

4. Return X

The `checking` procedure can be seen with more details at Mei, Jiang and Tu paper [MJT18]. In fact, they describe two different checking procedures: the `marginal checking` and `forward`

`checking`. The `marginal checking` compares the best solution so far during the algorithm with the configuration space already pruned, however marginalized according to the remaining values for each variable. The `forward checking` uses the first derivative on the configuration space under search. The important part is that both of them return a smaller configuration space S' than the previous one S. However, it is not clear which criteria they use to call one specific `checking` over the other. Also, they do not clearly explain their heuristic when choosing one variable over another, or how to order them on `for all` $X \in \mathbf{X}$ loop, which could change the performance of their algorithm.

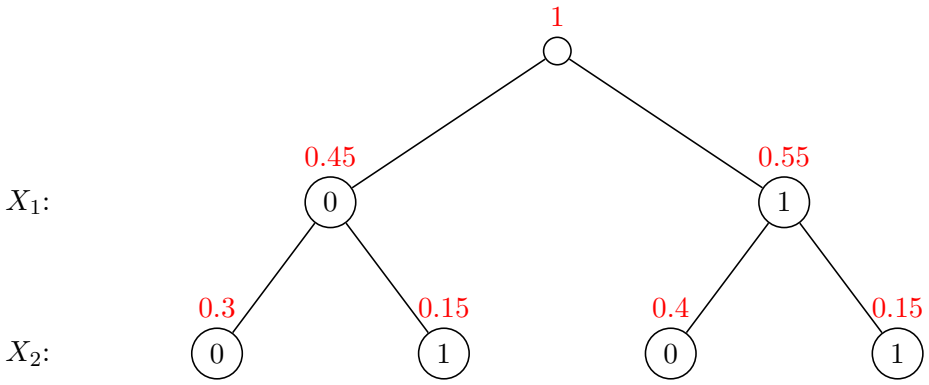


Figure 3.4: The configuration space for the SPN $\mathcal{S}$ in Figure 2.2 with layers ordered from X_1 to X_2. Each node has a value inside that represents the assignment for the respective variable in its layer. The values in red above each node represent the evaluation of such decision in $\mathcal{S}$. Note that a variable that has not been assigned yet is considered to be marginalized, hence the topmost node has value 1, since it represents the marginalization of both variables X_1 and X_2.

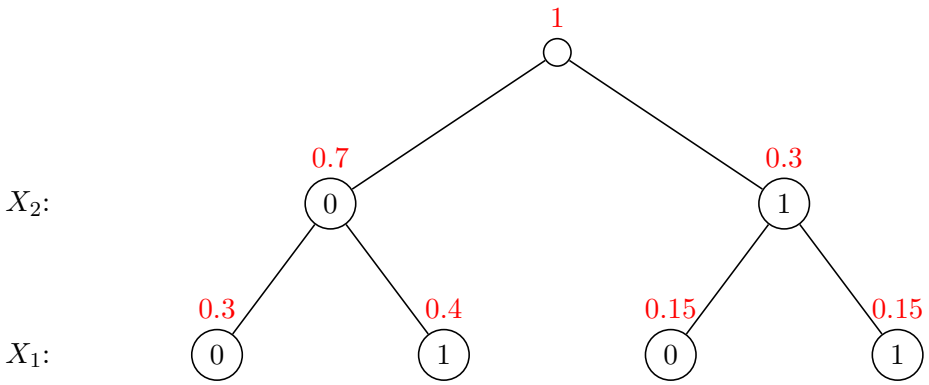


Figure 3.5: The configuration space for the SPN $\mathcal{S}$ in Figure 2.2 with layers ordered from X_2 to X_1. Each node has a value inside that represents the assignment for the respective variable in its layer. The values in red above each node represent the evaluation of such decision in $\mathcal{S}$. Note that a variable that has not been assigned yet is considered to be marginalized, hence the topmost node has value 1, since it represents the marginalization of both variables X_2 and X_1.

To illustrate at least how the `marginal checking` procedure works, Figures 3.4 and 3.5 show the behavior of the SPN from Figure 2.2 when variables X_1 and X_2, respectively, are

marginalized. We might note that the values above the nodes add up to 1 for each layer, since the bottom layer explores all the configurations for variables X_1 and X_2 and the middle layer explores the all configurations of X_1 when X_2 is marginalized, and vice-versa.

For example, let us imagine the first iteration of the algorithm, assuming that the initial solution is $\mathsf{X} = (X_1, X_2) = (1, 1)$, thus $\mathcal{S}(\mathsf{X}) = 0.15$, and where $\mathsf{S} = \{(0,0), (1,0), (0,1), (1,1)\}$. Let us take X_1, once $|\mathsf{S}[X_1]| = |\{0,1\}| > 1$. Then, for all $x \in \mathsf{S}[X]$:

- $x = 0 \rightarrow \mathsf{S}' = \{(0,0), (0,1)\}$
- $x = 1 \rightarrow \mathsf{S}' = \{(1,0), (1,1)\}$

Let us suppose the `marginal checking` call with $x = 0$. It then compares if $\mathcal{S}(\mathsf{S}') > \mathcal{S}(\mathsf{X})$. Since $\mathcal{S}(\mathsf{S}') = 0.45$, as we can see from Figure 3.4, and $\mathcal{S}(\mathsf{X}) = 0.15$, the next call on the `search` procedure will already happen with a pruned configuration space S'.

Chapter 4

MILP Reformulation Algorithm to MAP Inference in SPN

In this chapter, we present the algorithm for reformulation of the Maximum A Posteriori inference problem in Sum-Product Networks as a Mixed-Integer Linear Programming optimization problem.

Firstly, we present an explanation through the means of an example, but later on, we also provide a formal description of the algorithm as shown in Section 4.3. We believe that both approaches are important, allowing a practical and theoretical perspective of our work.

In practical terms, the algorithm is divided in two major parts:

1. In Section 4.1, we show the first part of algorithm, which generates a set of PADDs based on the normal SPN used as input. It is based on part of the algorithm proposed in [ZMP15] that converts SPNs into Bayesian Networks;

2. In Section 4.2, we show the second part of the algorithm, explaining how to use of the set of PADDs generated in Section 4.1 to formulate the MILP program. As we understand, it counts as the main contribution of this work.

Shown once more for the purpose of convenience, we use the SPN in Figure 2.2 as input, and the explanation throughout this chapter explains how each procedure of the algorithm operates the SPN until we have a MILP program at the end of Section 4.2.

We are interested in finding values for variables X_1 and X_2 which maximizes the value of the root node, that is, the value of the SPN $\mathcal{S}$ shown in Figure 2.2. Let us remember that the value of the root node represents the probability of occurrence of such configuration under the probability distribution represented by a normalized SPN.

Since $\mathcal{S}$ is a fairly small SPN, we could simply list the probability for all configurations of

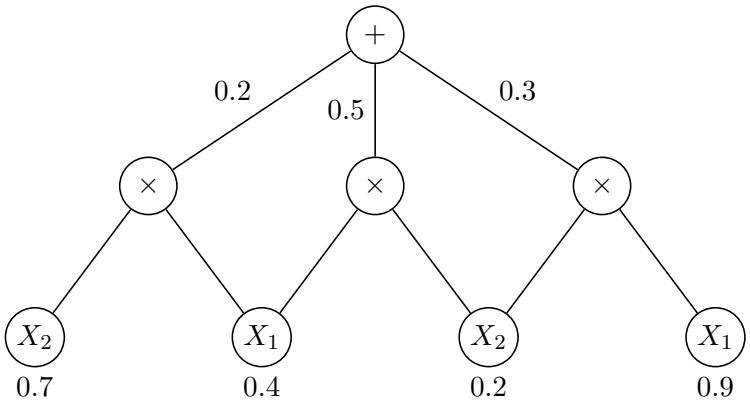


Figure 2.2: The normal SPN showed in Section 2.2. However, in order to use this SPN as input, we need to rearrange the topmost sum node so that it has only 2 children. This procedure will be explained in Section 4.1.1 (repeated from page 12)

X_1 and X_2 to find the solution for the MAP inference.

$$\begin{aligned} \mathcal{S}(0,0) &= 0.3 \\ \mathcal{S}(1,0) &= 0.4 \\ \mathcal{S}(0,1) &= 0.15 \\ \mathcal{S}(1,1) &= 0.15 \end{aligned} \tag{4.1}$$

We can see that the configuration that maximizes the value of $\mathcal{S}$ is $(X_1, X_2) = (1, 0)$. This is actually the solution for MAP inference in this network, but listing all configurations is an intractable procedure, as its processing runtime increases exponentially over the linear increase of variables.

4.1 SPN to PADDs

As we said previously, this is the part of our algorithm which is based on the work of Zhao, Melibari and Poupart, where they developed a procedure for efficiently extracting from an SPN a set of ADDs such that, when combined, represent the same distribution with only a polynomial increase of size [ZMP15].

Although the procedures presented in this section are slightly different from those presented in their article, it is not hard to see that the principles behind them are the same. However, as mentioned in Section 2.3, the space of functions with Boolean domain and multilinear expressions codomain actually contains the space of functions with Boolean domain and real codomain, and thus we formally say our algorithm generates PADDs instead of ADDs, since we deal with multilinear expressions in the end.

4.1.1 Sum nodes binarization

As we might see in [ZMP15], Zhao, Melibari and Poupart extended the definition of ADD so that its nodes could not only represent binary decisions, thus making their proofs and algorithm simplified. Since we intend to rely on established software packages, we believe to be more intuitive to slightly change the structure of the SPN used as input to fit them into the standard definition of ADD, whose nodes represent only binary decisions. Since standard ADDs (and PADDs as well, by extension) have internal nodes with only 2 children (representing the Boolean decisions `FALSE` and `TRUE`), we must have a normal SPN where each sum node has only 2 children. A sum node with k children can be replaced by a binary tree with $k-1$ sum nodes, where the arc weights of the original sum node are to be associated with the leaf nodes of this subtree. The same could be done with the product nodes with more than 2 children if necessary.

Then, let us start recalling the SPN from Figure 2.2 again. It will serve as input example for the explanation of the algorithm as a whole. We might change the SPN (but keeping the same probability distribution) by adding a new sum node appended as child of the topmost sum node. In this case, we chose to append a sum node by replacing the arcs whose weights are 0.2 and 0.5, but in any case it should not matter where to append a child sum node. Then, we transfer the arcs weights of the parent sum node consistently to the arcs of the child sum node, and we set as 1 the weight between the parent and the child sum node.

It is important to note that this transformation keeps the properties of *completeness* (Definition 2.2.2), once the children of the sum node in focus have all the same scope, and *decomposability* (Definition 2.2.4), since we did not changed any product node. However, we might note that now we have an *unnormalized* SPN. We can find in [ZMP15] a bottom-up propagation procedure to normalize any SPN. Hence, we assume that processing it back to the normal form is not a major issue, and it is shown in Figure 4.2 what the input SPN looks like at this point.

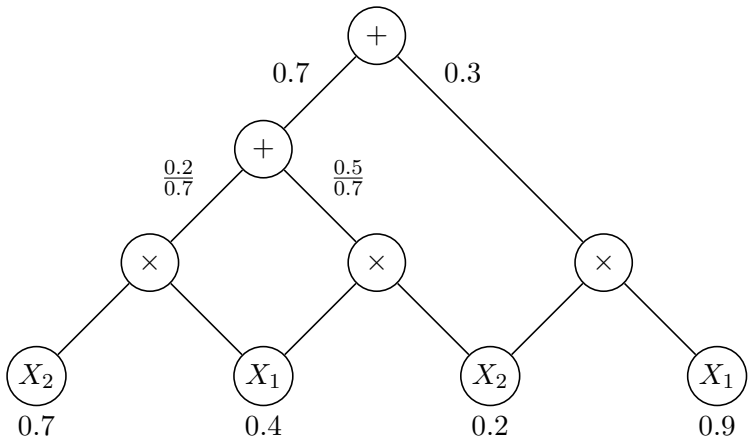


Figure 4.2: The SPN showed in Section 2.2 after the sum nodes binarization procedure and normalized again. We might note that the topmost sum node has now only 2 children.

4.1.2 Adding auxiliary variable Y_i

This part consists in associating a random Boolean variable to each sum node in the SPN. If there are m sum nodes, we must add, let us say, variables $Y_1, ..., Y_m$, associated with each sum node. It is important to note that these variables do not represent anything related to the original variables X_i, they just represent the decision of which path from the leaf nodes to the root node our algorithm will choose to compute.

There is no need to change any arc weight in this procedure, since no sum nodes were added, and those which had any changes were just dragged around and appended in a new product node. Also, product nodes are the only new nodes added to the structure, and it is implicit that their arc weights are 1.

As Zhao, Melibari and Poupart would describe, this procedure might be understood as the association of a hidden variable to each sum node. Let us consider a sum node $s \in [m]$, for instance. Since every sum node in an SPN have only two children, as result of the procedure described in Subsection 4.1.1, s also has only two children. In addition, since we are dealing with normal SPNs, we have $\sum_{i=1}^{2} w_{s_i} = 1$ and $w_{s_i} \geq 0$, where each w_{s_i} is a weight of s for $i \in [2]$.

This naturally suggests that each sum node s in an SPN can be associated to a hidden random variable Y_s with binomial distribution $\mathrm{P}(Y_s = i) = w_{s_i}$. Thus, an SPN might be thought as defining a joint probability distribution over $X_1, \ldots, X_n$ as observable variables and $Y_1, \ldots, Y_m$ as hidden variables. As for inference in such SPN, its computation can be done as described in Section 2.2, with all the hidden variables Y_s summed out, that is, $\lambda_{Y_s} = \bar{\lambda}_{Y_s} = 1$.

Structurally, we could follow the next sequence of steps for each sum node:

1. Insert a intermediate product node between the sum node s and each one of its children. We might note that the sum node still has two children, both product nodes, and now each product node is the parent of one child of the original sum node.

2. Append the indicator λ_{Y_s} to one of the product nodes and the indicator $\bar{\lambda}_{Y_s}$ to the other. These indicators are both related with the random Boolean variable Y_s, which is associated with the original sum node.

It is relevant to mention that we may lose the property of completeness of the SPN, since sum nodes might not have the same scope. The size of our structure will increase by four times the number of sum nodes. Also, this procedure might generate redundant pairs of product nodes. They can be taken out without loss of representativity, either at this procedure or at next explained in Section 4.1.3, although it might be more efficient to do it now. However, for understanding purposes, to explicitly illustrate the inclusion of extra nodes related with such auxiliary variables, we left the redundant pairs of product nodes in Figure 4.3, which shows the state of our structure at this point.

4.1.3 Restricting SPN by variable X_i

Once we have added the auxiliary variables into the SPN, we can start restricting our input structure. The idea is to generate, for each variable X_i, a restricted SPN discarding every subnetwork not containing X_i on its scope, or, in other words, we want a restricted SPN where

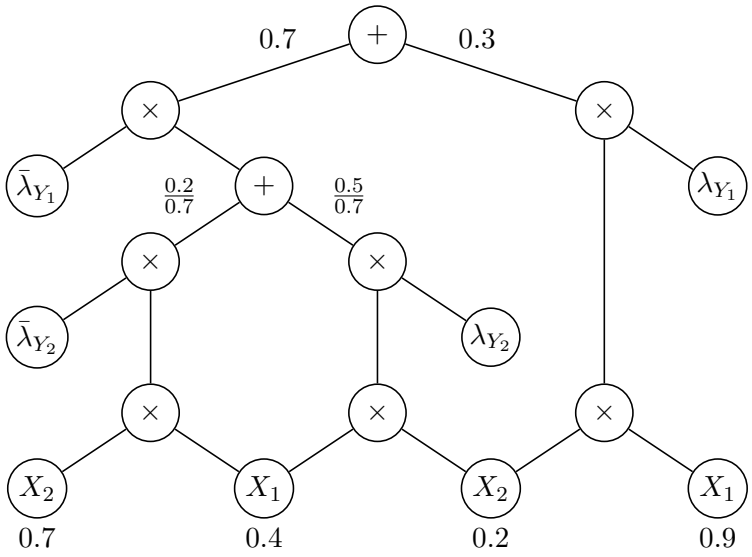


Figure 4.3: The SPN after adding nodes related to the variables Y_1 and Y_2. It is possible to see that its complexity has increased, as well as its height. The completeness property was lost, once the topmost sum node has different scopes for each child: $\{X_1, X_2, Y_1, Y_2\}$ for the one whose arc weight is 0.7, and $\{X_1, X_2, Y_1\}$ for one whose arc weight is 0.3

all the other variables X_j, $j \in [n] \backslash \{i\}$, are out. Also, let us say, without loss of generality, that the set $\{Y_1, ..., Y_k\}$ contains are all the variables associated with sum nodes that appear on a path from a leaf node with scope X_i up to the root node. These variables must appear in the restricted SPN as well.

One way of doing that is to select such variable X_i, to look at all leaf nodes containing this variable, and thus start tracing all the possible paths up from them to the root node. This could be done with a simple depth-first search (DFS) algorithm. If, along the way, the path contains a product node that has as a child a node that references some auxiliary variable $Y_{j \in [k]}$, it must be taken as part of the subnetwork as well. Our goal with this procedure is to obtain a subnetwork for each $X_{i \in [n]}$.

Since we did not remove those redundant product nodes which appeared in the last procedure described in Section 4.1.2, we do it now without losing representativity. Figure 4.4 shows the restricted SPNs for variables X_1 and X_2 based on our example.

4.1.4 Building PADDs

One of the reasons why we want to convert SPNs into PADDs is the fact that, as mentioned in Sections 2.2 and 2.3, the operations between two SPNs are not tractable, while operations between PADDs are. Since we intend to multiply each PADD generated from the original SPN restricted by variables X_i among themselves, we understand that this conversion is necessary.

From a theoretical perspective, the PADDs built in this subsection suggest the individual behavior of each variable X_i inside the SPN. Naturally, we may not compute the inference separately for each observable variable, and that is one of the reasons why the auxiliary (or hidden) variables Y were associated to each sum node in the original SPN. These auxiliary

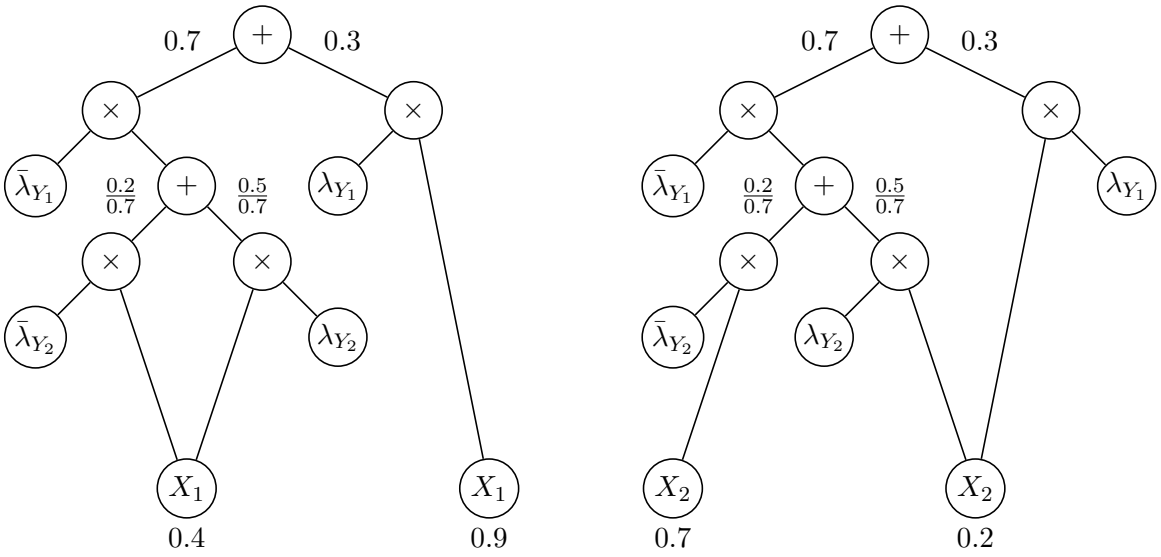


Figure 4.4: On the left, the SPN restricted for variable X_1. On the left, the SPN restricted for variable X_2. Eventual redundancies caused by the concatenation of two product nodes in sequence were suppressed, and a single product node was put in place.

variables create a binding between the PADDs generated for each variable X_i and allow us to represent how they influence each other probabilistically.

4.1.4.1 PADD $\mathcal{A}_i$ from SPN restricted by variable X_i

Once we have a restricted SPN for each variable X_i, it is possible to build specific PADDs for each of them. Here, we take advantage of the fact that both PADDs and SPNs can represent expressions of Boolean variables. By now, our restricted SPN by variable X_1, as shown in Figure 4.4, encodes the expression

$$0.7(2/7X_1(\bar{\lambda}_{Y_2}) + 5/7X_1(\lambda_{Y_2}))(\bar{\lambda}_{Y_1}) + 0.3X_1(\lambda_{Y_1})\,,$$

and we want to generate a PADD which encodes it as well. Above, we tried to focus on the indicators of both variables Y_1 and Y_2 to highlight that they are related with the selection of what part of the expression should be calculated.

The process of converting a restricted SPN to a PADD can be made recursively starting from the root node and searching all ways down to leaf nodes. When building the PADD, each sum and product operation represented by restricted SPN nodes can be reproduced by using the operation `apply`, such as mentioned in Section 2.3. For example, Figure 4.5 shows the PADDs built from the restricted SPN by variable X_1 and X_2.

We might note that PADD $\mathcal{A}_1$ does not include the weights of the sum nodes associated with variables Y_1 and Y_2. The reason is that we will represent such weights in external PADDs, so that we can still recover the probability distribution of the original SPN. This will be explained in the Subsection 4.1.4.2 right ahead. Thus, variable X_1 does not depend on the value of the auxiliary variable Y_2, which is not represented in PADD $\mathcal{A}_1$.

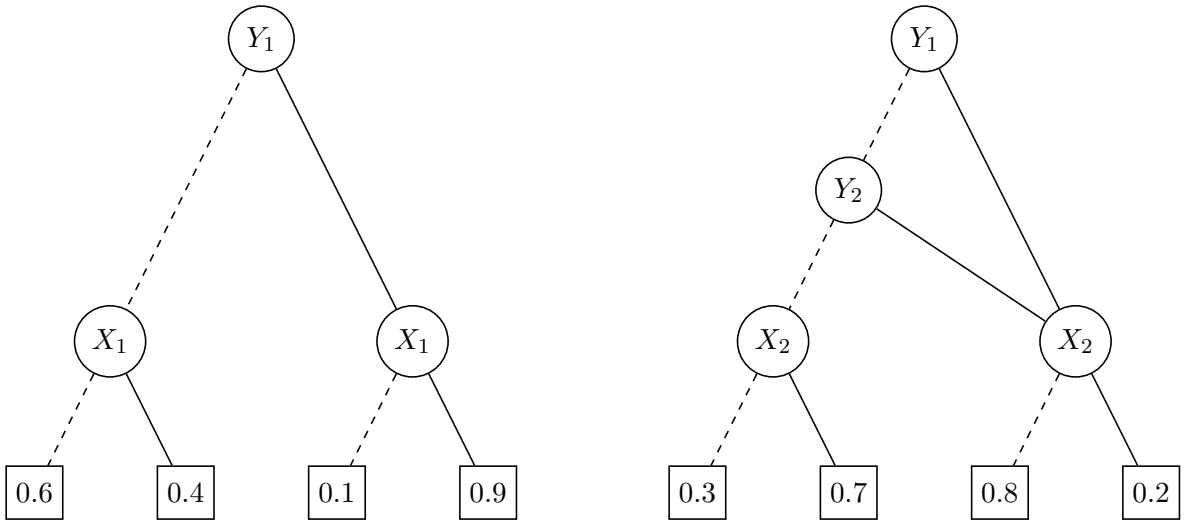


Figure 4.5: On the left, the PADD $\mathcal{A}_1$ built based on the SPN restricted by the variable X_1. Note that, since the weight values of the sum node associated with variable Y_2 will be represented in an external PADD, $\mathcal{A}_1$ is independent of such variable. On the left, the PADD $\mathcal{A}_2$ built based on the SPN restricted by the variable X_2.

4.1.4.2 PADD $\hat{\mathcal{A}}_i$ from variable Y_i

Since we can not explicitly keep the weights of the sum nodes in the PADDs generated in Subsection 4.1.4.1, we must have a simple external PADD $\hat{\mathcal{A}}_i$ representing the values of the weights for each variable Y_i as well. For example, Figure 4.6 shows the external PADDs representing the weights of the variables Y_1 and Y_2.

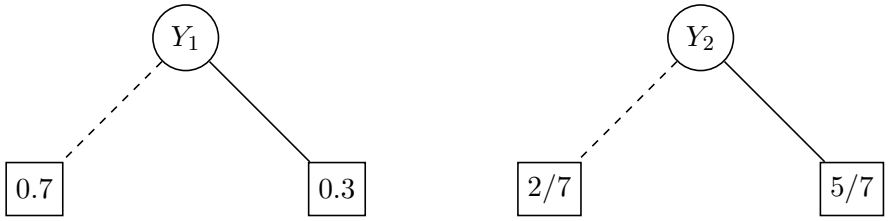


Figure 4.6: PADDs $\hat{\mathcal{A}}_1$ and $\hat{\mathcal{A}}_2$ representing the weights of the sum nodes associated with variables Y_1 and Y_2.

4.1.4.3 PADD $\mathcal{X}_i$ from variable X_i

It is important to note that the MILP problem must contain only variables X_i and no variables Y_i, so, besides the already generated PADDs, we must have a structure that represents the symbolic expression for each X_i. For that reason, we must then use a PADD, let us say, $\mathcal{X}_i$, for each variable X_i which contains as leaf nodes the symbolic expressions $\bar{x}_i$ and x_i. It might be an obvious structure, but necessary since we do not want these variables to assume any value. For our example, Figure 4.7 shows how $\mathcal{X}_1$ and $\mathcal{X}_2$ should look like.

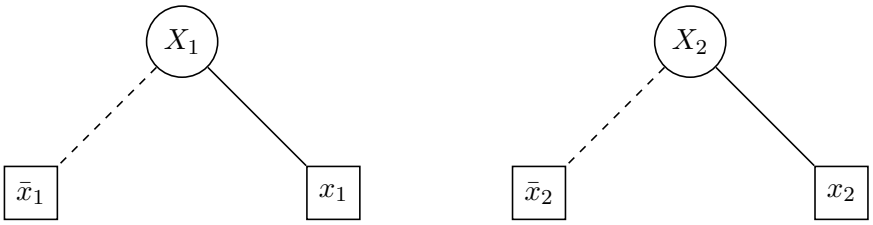


Figure 4.7: PADDs $\mathcal{X}_1$ and $\mathcal{X}_2$ representing the symbolic values $\bar{x}_1$ and x_1 for variable X_1, and $\bar{x}_2$ and x_2 for variable X_2. Generically they represent the expression $(\bar{x}_i)\bar{X}_i + (x_i)X_i$ for any variable X_i.

We can see that when $X_i = 0$, we will have the expression $(\bar{x}_i)1 + (x_i)0 = \bar{x}_i$, and when $X_i = 1$ we will have $(\bar{x}_i)0 + (x_i)1 = x_i$.

4.1.5 Recovering the original SPN distribution

The content of this subsection is not intrinsically related to the algorithm itself, but it shows that the PADDs generated previously can be used to recover the probability distribution represented by the original SPN.

At this point we have generated a set of PADDs $\mathcal{A}_i$ and $\mathcal{X}_i$ from each variable X_i and a set of PADDs $\hat{\mathcal{A}}_i$ from each variable Y_i. By construction, we might note that $\mathcal{X}_i$ and $\hat{\mathcal{A}}_i$ depends only on X_i and Y_i respectively, while $\mathcal{A}_i$ depends on X_i and a subset of $\{Y_1, ..., Y_m\}$. Also, we might note below that the PADDs $\mathcal{X}_i$ are not necessary for recovering the original SPN distribution, but only for building the MILP program.

We can build back the probability distribution of the original SPN by calling the procedure `apply` and `restrict`. Firstly, let us consider the following PADD $\mathcal{A}$, which is the multiplication of all generated PADDs:

$$\mathcal{A} = \Big(\prod_{i=1}^{n} \mathcal{A}_i\Big)\Big(\prod_{j=1}^{m} \hat{\mathcal{A}}_j\Big). \tag{4.2}$$

If we use the procedure `restrict` over all the possible combinations for variables Y_i over $\mathcal{A}$, and use the procedure `apply` with the sum operation, we would have the equation

$$\sum_{i_1=0}^{1} ... \sum_{i_m=0}^{1} \mathcal{A}|_{Y_1=i_1,...,Y_m=i_m} \tag{4.3}$$

that builds back the original SPN distribution.

Following our example and considering Equation 4.2, we would have

$$\mathcal{A} = \mathcal{A}_1\mathcal{A}_2\hat{\mathcal{A}}_1\hat{\mathcal{A}}_2$$

Then, summing all $\mathcal{A}$ restricted by each possible combination of Y_i as described in Equation 4.3,

$$\mathcal{A}|_{Y_1=0,Y_2=0} + \mathcal{A}|_{Y_1=0,Y_2=1} + \mathcal{A}|_{Y_1=1,Y_2=0} + \mathcal{A}|_{Y_1=1,Y_2=1}$$

would lead to the PADD shown in Figure 4.8. As we might see, it represents the set of equations

shown in 4.1.

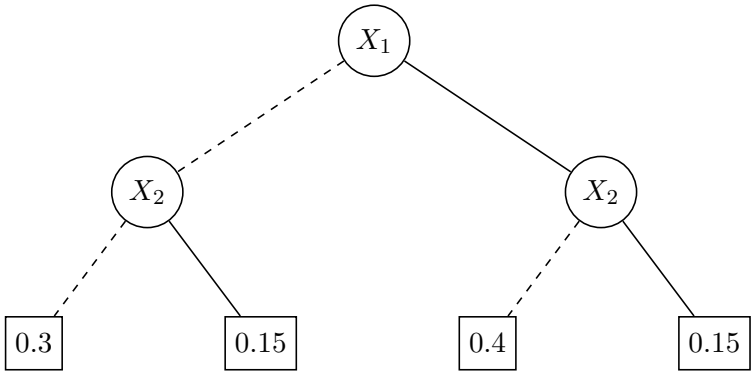


Figure 4.8: The PADD whose leaf nodes are values equivalent to the ones of the original SPN under a specific configuration. In our example in Section 2.2, $S(X_1 = 1, X_2 = 0) = 0.4$, such as it is shown above.

We might confirm the correctness of such construction in a probabilistic point of view. We could see each PADD $\mathcal{A}_i$ as the probability function $\mathrm{P}(X_i|Y_1, ..., Y_k)$ and each PADD $\hat{\mathcal{A}}_i$ as $\mathrm{P}(Y_i)$. The PADD $\mathcal{A}$ from our example could then be interpreted as

$$\mathrm{P}(X_1, X_2, Y_1, Y_2) = \mathrm{P}(X_1|Y_1)\mathrm{P}(X_2|Y_1, Y_2)\mathrm{P}(Y_1)\mathrm{P}(Y_2)\,.$$

For instance, if we want to evaluate a specific configuration, let us say, $X_1 = 1$ and $X_2 = 0$, we would have

$$\begin{aligned}
\mathrm{P}(X_1 = 1, X_2 = 0) &= \sum_{Y_1}\sum_{Y_2} \mathrm{P}(X_1 = 1|Y_1)\mathrm{P}(X_2 = 0|Y_1, Y_2)\mathrm{P}(Y_1)\mathrm{P}(Y_2) \\
&= \mathrm{P}(X_1 = 1|Y_1 = 0)\mathrm{P}(X_2 = 0|Y_1 = 0, Y_2 = 0)\mathrm{P}(Y_1 = 0)\mathrm{P}(Y_2 = 0) \\
&+ \mathrm{P}(X_1 = 1|Y_1 = 1)\mathrm{P}(X_2 = 0|Y_1 = 1, Y_2 = 0)\mathrm{P}(Y_1 = 1)\mathrm{P}(Y_2 = 0) \\
&+ \mathrm{P}(X_1 = 1|Y_1 = 0)\mathrm{P}(X_2 = 0|Y_1 = 0, Y_2 = 1)\mathrm{P}(Y_1 = 0)\mathrm{P}(Y_2 = 1) \\
&+ \mathrm{P}(X_1 = 1|Y_1 = 1)\mathrm{P}(X_2 = 0|Y_1 = 1, Y_2 = 1)\mathrm{P}(Y_1 = 1)\mathrm{P}(Y_2 = 1)\,.
\end{aligned}$$

Based on the Figure 4.5, we have the values for $\mathrm{P}(X_1|Y_1)$ and $\mathrm{P}(X_2|Y_1, Y_2)$, and based on Figure 4.6, we have the values for $\mathrm{P}(Y_1)$ and $\mathrm{P}(Y_2)$.

$$\begin{aligned}
\mathrm{P}(X_1 = 1, X_2 = 0) &= (0.4)(0.3)(0.7)(2/7) \\
&+ (0.9)(0.8)(0.3)(2/7) \\
&+ (0.4)(0.8)(0.7)(5/7) \\
&+ (0.9)(0.8)(0.3)(5/7) = 0.4
\end{aligned}$$

As showed previously, the evaluation of $X_1 = 1$ and $X_2 = 0$ is 0.4 both in the original SPN representation and in the PADD representation shown in Figure 4.8. Hence, we might see that the generated PADDs can be seen as probability functions over the variables X_i and Y_i.

4.2 PADDs to MILP

In this section, we explain our contribution by proposing an algorithm for MAP inference. At this point, we will stop following the algorithm proposed in [ZMP15], since their goal was to obtain a Bayesian Network from the original SPN, and ours is to offer a solution for MAP inference.

Our MILP reformulation approach consists in modifying Zhao, Melibari and Poupart's approach to replace the Variable Elimination step with a symbolic variant that represents PADDs as sets of multilinear constraints. In addition to the PADDs created by the approach, we might also include distributions $\mathcal{X}_i(X_i)$, that represent the "choice" of a configuration of variable X_i in the MAP inference solution. To ensure that only bilinear constraints are generated, we perform the symbolic variable elimination using the same ordering as the one defined by the path decomposition of the domain graph of the PADDs, that is, we can perform symbolic message passing in the path decomposition. The effectiveness of this approach depends on the size of the generated MILP program.

4.2.1 Variable Elimination

Once we have a set of PADDs, as described in Subsection 4.1.4, we might now focus on finding a *path decomposition* for the *domain graph* of the set of functions represented by such.

As mentioned in Section 2.4, the *domain graph* of a set of functions (represented in any form; in our case, PADDs) is a graph whose nodes are the variables appearing in any of these functions, and such that two nodes are connected by an edge it and only if the two corresponding variables appear in the same functions. Figure 4.9 shows how the variables X_i and Y_i are organized based on the set of PADDs generated previously.

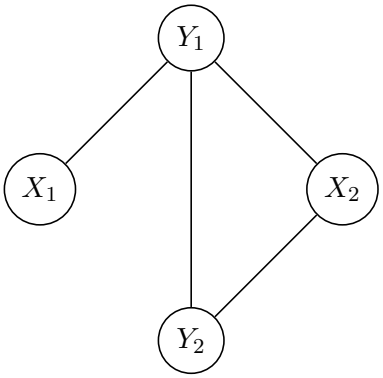


Figure 4.9: The domain graph over the variables X_1, X_2, Y_1 and Y_2, based on their appearances on the functions described by the PADDs. We can see that X_1 and X_2 have not appeared together as argument of the same function.

We understand that such graph is not hard to obtain since it is possible to keep track of the variables used in each restricted PADD. We might note two facts about its structure: i) since the restricted SPNs are based on variables X_i, we never have a domain graph where two variables X_i and X_j, with $i \neq j$, are connected, and ii) each PADD built from this process generates a clique in the domain graph. From now on, since we want to relate each PADD as a function, we

will write them followed by their arguments, such as $\mathcal{A}_1(X_1, Y_1)$, $\hat{\mathcal{A}}_2(Y_2)$ or $\mathcal{X}_i(X_i)$.

Once we have obtained the domain graph, the following step consists in finding it a *path decomposition*, as explained in Section 2.5.

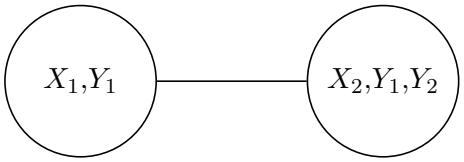


Figure 4.10: The path decomposition for the domain graph shown in Figure 4.9. We might see that X_1 and Y_1 compose a clique in the domain graph as well as X_2, Y_1 and Y_2, and hence, they both became nodes. The functions associated with each node are displayed above them.

In fact, instead of finding an optimal path decomposition to perform variable elimination, which can be related to a future work within this research, we use an established order according to the degree of each variable X_i in the domain graph. To better understand our procedure, let us assume, without loss of generality, that variables $X_1, \ldots, X_n \in \mathsf{X}$ are sorted according to their degree in the domain graph $\mathcal{G}$, that is, for whatever i and j such that $i < j$, $\deg(X_i) \leq \deg(X_j)$, where $\deg(X_i)$ is the number of edges that are incident to node X_i in $\mathcal{G}$. Also, without loss of generality, let us assume that $Y_1, \ldots, Y_m \in \mathsf{Y}$ are sorted in the topological order according to the original SPN, that is, for whatever i and j such that $i < j$, Y_i is closer or at most at the same depth of Y_j from the root of the SPN. Then, the variable elimination order is $X_1, \ldots, X_n, Y_m, \ldots, Y_1$.

Since we want to fit this ordering in terms of path decomposition, we would have it as described by Figure 4.11.

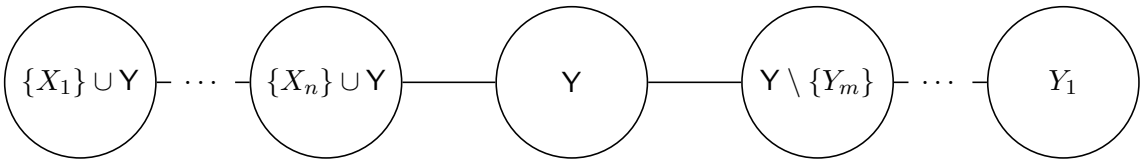


Figure 4.11: The generic path decomposition used to perform variable elimination in this research.

Each node of the path decomposition will be related to a set of variables X_i and Y_i, which are the arguments for the functions (PADDs) we have generated before. We want to associate each function to a node in the path decomposition if a variable is both in the set of such node and in the argument of such function. For example, variables X_1 and Y_1 appear both in the first left node of the path decomposition and as arguments in $\mathcal{A}_1(X_1, Y_1)$. Then, we can associate $\mathcal{A}_1(X_1, Y_1)$ to the left node.

4.2.2 Building the MILP program

Now that we have a set of functions associated with each node in the path decomposition, we can start building the MILP program by choosing one of the ends in such path to begin the message passing. Let us start from the left node as in the example shown in Figure 4.12.

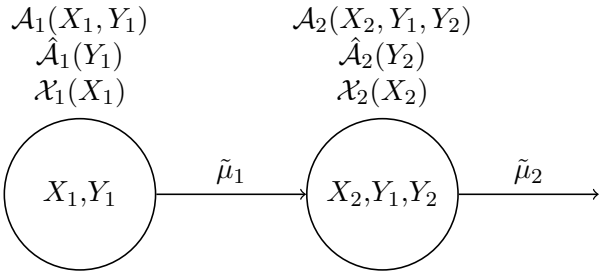


Figure 4.12: μ_1 is the first message, sent from the left node to the right node. $\mu_1(0)$ and $\mu_1(1)$ will both become constraints and μ_2 will become the objective function in the MILP program.

Each message transmitted along the path will originate the constraints in our MILP program. The messages will be represented by the symbol μ_i, and will be composed by the multiplication of all functions in the set of the node which is transmitting. The first is then $\mu_1(Y_1)$. In our example, we have

$$\mu_1(Y_1) = \sum_{X_1} \mathcal{A}_1(X_1, Y_1)\hat{\mathcal{A}}_1(Y_1)\mathcal{X}_1(X_1)$$

We can see that μ_1 is a PADD that can be interpreted as a function. We might just slightly change the notation for the purpose of simplification. Instead of writing the `restrict` operation as $\mu_1(Y_1)|_{Y_1=0}$, we might only write $\mu_1(0)$. Both $\mu_1(0)$ and $\mu_1(1)$ will become variables associated with constraints in the MILP problem. Then, to pass the message μ_1 along the path, we take a copy $\tilde{\mu}_1(Y_1)$ of PADD $\mu_1(Y_1)$ and correspondingly substitute the expression on each leaf for the actual symbolic variables $\mu_1(0)$ and $\mu_1(1)$, to be multiplied in the next step.

The same procedure can be done to obtain the second message and so on. The only difference is that besides multiplying all functions related to the second node of the path, we must also multiply the message coming from the previous node. Thus, in this case, we would have μ_2 as follows

$$\mu_2(Y_2, Y_1) = \tilde{\mu}_1(Y_1) \sum_{X_2} \mathcal{A}_2(X_1, Y_1, Y_2)\hat{\mathcal{A}}_2(Y_2)\mathcal{X}_2(X_2)$$

The last node in the path decomposition has no other node to pass the message, hence the last generated message μ_n will become the objective function of a mixed-integer bilinear program, which can then be linearized using standard techniques from the literature to generate a MILP program. In your case, once we have only 2 nodes in the path decomposition, μ_2 will become the objective function.

However, it would not mean anything for us to have Y_i as variables in our MILP program. For that reason, we should assume values for variables Y_i in order to eliminate them from the constraints and the objective function, such as shown in the expression below.

$$\mu_2 = \sum_{X_2} \sum_{Y_1} \sum_{Y_2} \mathcal{A}_2(X_2, Y_1, Y_2)\hat{\mathcal{A}}_2(Y_2)\tilde{\mu}_1(Y_1)\mathcal{X}_2(X_2)$$

Theoretically, our mixed-integer bilinear program would be the following:

$$
\begin{aligned}
&\text{maximize } \mu_2 \\
&\text{subject to } \mu_1(0) = \mathcal{A}_1(0,0)\hat{\mathcal{A}}_1(0)\mathcal{X}_1(0) + \mathcal{A}_1(1,0)\hat{\mathcal{A}}_1(0)\mathcal{X}_1(1) \\
&\qquad \mu_1(1) = \mathcal{A}_1(0,1)\hat{\mathcal{A}}_1(1)\mathcal{X}_1(0) + \mathcal{A}_1(1,1)\hat{\mathcal{A}}_1(1)\mathcal{X}_1(1) \\
&\qquad \mu_2 = \mathcal{A}_2(0,0,0)\hat{\mathcal{A}}_2(0)\mu_1(0)\mathcal{X}_2(0) + \mathcal{A}_2(0,0,1)\hat{\mathcal{A}}_2(1)\mu_1(0)\mathcal{X}_2(0) \\
&\qquad\quad + \mathcal{A}_2(0,1,0)\hat{\mathcal{A}}_2(0)\mu_1(1)\mathcal{X}_2(0) + \mathcal{A}_2(0,1,1)\hat{\mathcal{A}}_2(1)\mu_1(1)\mathcal{X}_2(0) \\
&\qquad\quad + \mathcal{A}_2(1,0,0)\hat{\mathcal{A}}_2(0)\mu_1(0)\mathcal{X}_2(1) + \mathcal{A}_2(1,0,1)\hat{\mathcal{A}}_2(1)\mu_1(0)\mathcal{X}_2(1) \\
&\qquad\quad + \mathcal{A}_2(1,1,0)\hat{\mathcal{A}}_2(0)\mu_1(1)\mathcal{X}_2(1) + \mathcal{A}_2(1,1,1)\hat{\mathcal{A}}_2(1)\mu_1(1)\mathcal{X}_2(1) \\
&\qquad \mathcal{X}_1(0) + \mathcal{X}_1(1) = 1 \\
&\qquad \mathcal{X}_2(0) + \mathcal{X}_2(1) = 1 \\
&\qquad \mu_1(0), \mu_1(1), \mu_2 \in [0,1] \\
&\qquad \mathcal{X}_1(0), \mathcal{X}_1(1), \mathcal{X}_2(0), \mathcal{X}_2(1) \in \{0,1\}
\end{aligned}
$$

It might look complex at first sight, but it will look simpler if we use values instead. We will use the values from Figures 4.5 and 4.6 for substitution.

$$
\begin{aligned}
&\text{maximize } \mu_2 \\
&\text{subject to } \mu_1(0) = 0.42\bar{x}_1 + 0.28x_1 \\
&\qquad \mu_1(1) = 0.03\bar{x}_1 + 0.27x_1 \\
&\qquad \mu_2 = (0.3(2/7) + 0.8(5/7))\mu_1(0)\bar{x}_2 + (0.8(2/7) + 0.8(5/7))\mu_1(1)\bar{x}_2 \\
&\qquad\quad + (0.7(2/7) + 0.2(5/7))\mu_1(0)x_2 + (0.2(2/7) + 0.2(5/7))\mu_1(1)x_2 \\
&\qquad \bar{x}_1 + x_1 = 1 \\
&\qquad \bar{x}_2 + x_2 = 1 \\
&\qquad \mu_1(0), \mu_1(1), \mu_2 \in [0,1] \\
&\qquad \bar{x}_1, x_1, \bar{x}_2, x_2 \in \{0,1\}
\end{aligned}
$$

We might note that, by construction, these bilinear expressions contain only products of nonnegative linear and integer variables, so it can be easily transformed into a mixed-integer linear program by including, for each product $\mu \cdot \mathcal{X}$, where μ is continuous and $\mathcal{X}$ is binary, a new continuous variable γ and constraints $\mathcal{X} - 1 + \mu \leq \gamma \leq \mu$ and $0 \leq \gamma \leq \mathcal{X}$. For example, the product of variables $\mu_1(0)\bar{x}_2$ can be transformed into the variable γ_1.

The MAP inference solution can be extracted by the value of the integer valued variables in the program (the variables $\mathcal{X}_1(0)$, $\mathcal{X}_1(1)$, $\mathcal{X}_2(0)$ and $\mathcal{X}_2(1)$, which are the same as $\bar{x}_1$, x_1, $\bar{x}_2$ and x_2 in our example). If we apply these rules over the already generated mixed-integer bilinear program, we can finally obtain our MILP program as described as follows:

$$
\begin{aligned}
&\text{maximize } \mu_2 \\
&\text{subject to } \mu_1(0) = 0.42\bar{x}_1 + 0.28x_1 \\
&\qquad \mu_1(1) = 0.03\bar{x}_1 + 0.27x_1 \\
&\qquad \mu_2 = (4.6/7)\gamma_1 + (5.6/7)\gamma_2 + (2.4/7)\gamma_3 + (1.4/7)\gamma_4 \\
&\qquad \bar{x}_2 - 1 + \mu_1(0) \leq \gamma_1 \leq \mu_1(0), \quad \bar{x}_2 - 1 + \mu_1(1) \leq \gamma_2 \leq \mu_1(1) \\
&\qquad x_2 - 1 + \mu_1(0) \leq \gamma_3 \leq \mu_1(0), \quad x_2 - 1 + \mu_1(1) \leq \gamma_4 \leq \mu_1(1) \\
&\qquad 0 \leq \gamma_1 \leq \bar{x}_2, \quad 0 \leq \gamma_2 \leq \bar{x}_2 \\
&\qquad 0 \leq \gamma_3 \leq x_2, \quad 0 \leq \gamma_4 \leq x_2 \\
&\qquad \bar{x}_1 + x_1 = 1, \quad \bar{x}_2 + x_2 = 1 \\
&\qquad \mu_1(0), \mu_1(1), \mu_2 \in [0, 1] \\
&\qquad \bar{x}_1, x_1, \bar{x}_2, x_2 \in \{0, 1\}
\end{aligned}
$$

This program was tested by using the software Gurobi Optimizer [GO16] under academic license. The optimal value for the objective function was 0.4 with the variables assuming the following values: $x_1 = 1$, $\bar{x}_1 = 0$, $x_2 = 0$, $\bar{x}_2 = 1$, $\mu_1(0) = 0.280000$, $\mu_1(1) = 0.269999$, $\mu_2 = 0.399999$, $\gamma_1 = 0.279999$, $\gamma_2 = 0.269999$, $\gamma_3 = 0.0$ and $\gamma_4 = 0.0$. We can see that it matches the result shown in the set of equations 4.1, since $(X_1, X_2) = (1, 0)$ is given as the optimal solution, with little numerical imprecision.

4.3 Formal Description

The following explanation can be found as well in [MRKA20], however, to better fit with the notation in this document and with the ideas shown through Sections 4.1 and 4.2, we present it here with minor modifications.

Formally, the proposed algorithm works as follows. For simplification, let us consider an SPN $\mathcal{S}$ over variables X_i where each sum node has two children, skipping the part of the algorithm described in Subsection 4.1.1. Associate with every sum node i a new binary random variable Y_i. It is important to note here that only sum nodes are associated to latent variables. Interpret $Y_i = 0$ and $Y_i = 1$ as selecting the left and right child of node i, respectively, and let $\mathbf{Y}$ denote the set of all such variables.

As in the work of Zhao et al. [ZMP15], construct a PADD $\hat{\mathcal{A}}_i$ for each $Y_i \in \mathbf{Y}$ that represents the function $f(Y_i) = (1 - Y_i)w_l + Y_i w_r$, where w_l and w_r are the weights of the outgoing arcs of i in $\mathcal{S}$. However, differently from the construction proposed by Zhao et al. [ZMP15], for each variable X_i in the scope of $\mathcal{S}$, obtain a PADD $\mathcal{A}_i$ as follows:

1. Construct the restriction $\mathcal{S}|_{X_i}$ of $\mathcal{S}$ from X_i by removing any node whose scope does not contain X_i
2. Replace sum nodes with the corresponding variables Y_i

Furthermore, for each variable X_i, construct a PADD $\mathcal{X}_i$ that represents the function $f(X_i) = (1 - X_i)\bar{x}_i + X_i x_i$, where $\bar{x}_i$ and x_i are symbolic constants that will respectively become the variables in the MILP program codifying the indicator variables $\bar{\lambda}_{X_i}$ and λ_{X_i}.

As mentioned previously in Section 2.4, the domain graph of a collection of functions, or their representation as PADDs, is the graph whose nodes are the variables in the domain and two variables are connected if and only if they cooccur in some function. As well in Section 2.5, a path decomposition of an undirected graph $\mathcal{G} = (\mathsf{V}, \mathsf{E})$ is a collection of subsets $\mathsf{Z}_1, \ldots, \mathsf{Z}_m \subseteq \mathsf{V}$ called clusters or paths, such that the endpoints of any edge in E co-occur in some $\mathsf{Z_i}$, and for any $\mathsf{Z}_i, \mathsf{Z}_j, \mathsf{Z}_k$, $\mathsf{Z}_i \cap \mathsf{Z}_k \subseteq \mathsf{Z}_j$.

The MILP program is generated by processing the generated PADDs using a variable elimination procedure that operates according to a path decomposition of the domain graph of the PADDs. The use of a path decomposition ensures that at most one PADD obtained by restricting a variable X_i is combined at each step, which means that the generated constraints can be cast into the mixed-integer linear form. To this end, associate to each cluster of the path decomposition a collection of PADDs whose scope is contained in that cluster, with at most one PADD generated by a restriction of a variable X_i per cluster. Assume that clusters are numbered from left to right in the path decomposition. Let $\mathcal{C}_i$ denote the collection of PADDs associated with cluster i, and $\mathsf{Z}_i \subseteq (\mathsf{X} \cup \mathsf{Y})$ denote the variables in that cluster, where X is the set of the original variables X_i from $\mathcal{S}$. We obtain a new PADD μ_i by multiplying all PADDs in $\mathcal{C}_i$ and marginalizing the variables in the set $\mathsf{Z}_{i+1} \setminus \mathsf{Z}_i$:

$$\mu_i(\mathsf{Z}_i \cap \mathsf{Z}_{i+1}) = \sum_{\mathsf{Z}_{i+1} \setminus \mathsf{Z}_i} \left(\prod_{\mathcal{A}_j, \hat{\mathcal{A}}_k, \mathcal{X}_l \in \mathcal{C}_i} \mathcal{A}_j \hat{\mathcal{A}}_k \mathcal{X}_l \right) \tag{4.4}$$

Create a new PADD $\tilde{\mu}_i$ by replacing every leaf of μ_i by a new fresh variable u_k. Insert $\tilde{\mu}_i$ in the collection $\mathcal{C}_{i+1}$ and repeat the operation. When the rightmost cluster m is reached, generate a PADD μ_m by eliminating all variables Z_m. The MILP program's constraints are such that

$$\tilde{\mu}_i(\phi) = \mu_i(\phi) \text{ for each leaf } \phi \text{ in } \mu_i, i = 1, \ldots, m, \tag{4.5}$$

where $\tilde{\mu}_i(\phi)$ is the new variable at leaf ϕ associated with the multilinear expression $\mu_i(\phi)$. Recall that a leaf in a PADD corresponds to the set of configurations of its scope which map to the same value. In particular, the PADD μ_m contains a single node, as all variables have been marginalized, with an expression that is used as the objective of the program.

4.4 High Level Implementation

Since we have detailed each step of the algorithm, we present a succinct sequence of steps to convert the MAP inference problem in a normal SPN to a MILP optimization problem.

Algorithm 1 MILP Reformulation Algorithm to MAP Inference in SPN

Input: Normal SPN $\mathcal{S}$
Output: MILP program $\mathcal{M}$

$\mathcal{S}' \leftarrow$ binarized $\mathcal{S}$
for each sum node i in $\mathcal{S}'$ **do**
    Add a latent variable Y_i in $\mathcal{S}'$
    $\hat{\mathcal{A}}_i \leftarrow$ PADD built from weights of i
for each variable X_i in $\mathcal{S}'$ **do**
    $\mathcal{S}'_i \leftarrow \mathcal{S}'$ restricted by X_i
    $\mathcal{A}_i \leftarrow$ PADD built from $\mathcal{S}'_i$
    $\mathcal{X}_i \leftarrow$ PADD built from X_i
$\mathcal{G} \leftarrow$ domain graph over every $\mathcal{A}_i$ ▷ We may see each $\mathcal{A}_i$ as a function
$\mathcal{P} \leftarrow$ variable elimination ordering from $\mathcal{G}$
$\mathcal{M} \leftarrow$ MILP program built on message passing through $\mathcal{P}$ over $\mathcal{A}_i$, $\hat{\mathcal{A}}_i$ and $\mathcal{X}_i$
return $\mathcal{M}$

Chapter 5

Empirical Analysis

This chapter contains the results achieved so far. Firstly, we tested the algorithm developed in this work by creating SPNs whose MAP inference corresponds to the solution of Maximal Independent Set problem in Section 5.1, which works as a sanity check as we understand. However, these tests do not correspond to any real problem involving SPNs or its learning process as they are merely a benchmark.

Lately, we tested our algorithm on some SPNs obtained through learning from real data. These tests were made by using the package `pyspn` [PU20b] developed by the Probabilistic Artificial Intelligence Lab of University of São Paulo (ProbAI-USP), also written in Python, which contains several functionalities for SPNs such as learning, classification and inference. Among them, they provide the algorithms mentioned in Chapter 3, Max-Product (Section 3.1), Argmax-Product (Section 3.2) and MaxSearch (Section 3.3), used to solve MAP inference in SPNs through a different approach than the one presented in this work. They also offer a repository of learned SPNs [PU20a], where raw data for training and testing SPNs, as well as already learned SPNs through LearnSPN algorithm, presented by Poon and Domingos in [PD11], are available.

We used Gurobi [GO16] as solver to optimize the MILP program in all tests.

5.1 Maximal Independent Set Tests

One of the used ways to check the assertiveness and effectiveness is of our approach was running it on cases where we can easily know the optimal solution.

Formally, for a given graph $\mathcal{G} = (\mathsf{V}, \mathsf{E})$, an *independent set* of vertices is a subset $\mathsf{W} \subseteq \mathsf{V}$ such that no two vertices of are linked. Then, we say that W is the *maximal independent set* (MIS) of graph $\mathcal{G}$ if there is no independent set $\mathsf{W}' \subseteq \mathsf{V}$ containing more elements than W.

There are several algorithms for solving the MIS problem. Still, a very simple approach consists in its modeling as a linear programming problem such as follows:

$$\begin{array}{lll} \text{maximize} & \sum_{v \in \mathsf{V}} x_v & \\ \text{subject to} & x_u + x_v \leq 1, & \forall uv \in \mathsf{E} \\ & x_v \geq 0, & \forall v \in \mathsf{V} \\ & x_v \leq 1, & \forall v \in \mathsf{V}\,, \end{array}$$

where each variable x_v indicates whether the corresponding vertex v is or is not in the independent set. Independence is enforced by the $x_u + x_v \leq 1$ constraint.

Conaty, Mauá and de Campos showed how to reformulate the MIS problem in graphs as the MAP inference problem in SPN [CMdC17], so it was chosen as a first test for the software we developed. The MIS problem has a very simple direct formulation as a linear optimization problem and, for the tested cases, the solution our approach output is correct and optimal as are the ones output by the direct formulation.

For the following tests, graphs were randomly generated based on the number of vertices as parameter. Initially, a graph is generated as complete, however each edge has 0.5 probability to be removed. For the same graphs, we ran experiments with the algorithms presented in Chapter 3 and we show their comparative performance in the tables and graph below.

Nodes	MAP	MILP	MP	AMP	MS MC
10	0.009615384615	1	0.25	0.25	1
11	0.005411255411	1	0.20	0.20	1
12	0.006443298969	1	0.20	0.20	1
13	0.001619170984	1	0.20	0.20	1
14	0.001474056604	1	0.20	0.20	1
15	0.001726519337	1	0.20	0.20	1
16	0.000423728813	1	0.14	0.14	1
17	0.000466417910	1	0.17	0.17	1
18	0.000494071146	1	0.20	0.20	1
19	0.000267551369	1	0.20	0.20	1
20	0.000231481481	1	0.20	0.20	1

Table 5.1: The "Nodes" column contains the number of nodes in the original graph. The numbers shown in the "MAP" column are the real MAP inference value for each SPN generated from a graph. The remaining numbers are the ratio between the MAP inference value that the algorithm in its column output over the actual MAP inference value for that instance.

Nodes	Building	Solver	MILP	MP	AMP	MS MC
10	0.44	0.09	0.53	0.00	0.00	0.59
11	0.57	0.11	0.68	0.00	0.00	1.28
12	0.79	0.15	0.94	0.00	0.00	2.20
13	0.91	0.23	1.14	0.00	0.00	5.95
14	1.16	0.36	1.52	0.00	0.00	10.08
15	1.42	0.44	1.87	0.00	0.00	17.89
16	1.71	0.51	2.21	0.01	0.00	51.19
17	2.13	0.64	2.76	0.01	0.00	80.25
18	2.61	0.59	3.19	0.01	0.00	150.19
19	2.96	1.12	4.08	0.01	0.01	322.35
20	3.39	1.00	4.39	0.01	0.01	748.80

Table 5.2: The "Nodes" column contains the number of nodes in the original graph. The remaining columns contain numbers that show the time spent to obtain the MAP inference value through the respective algorithm. Note that "Building" + "Solver" = "MILP" for each instance.

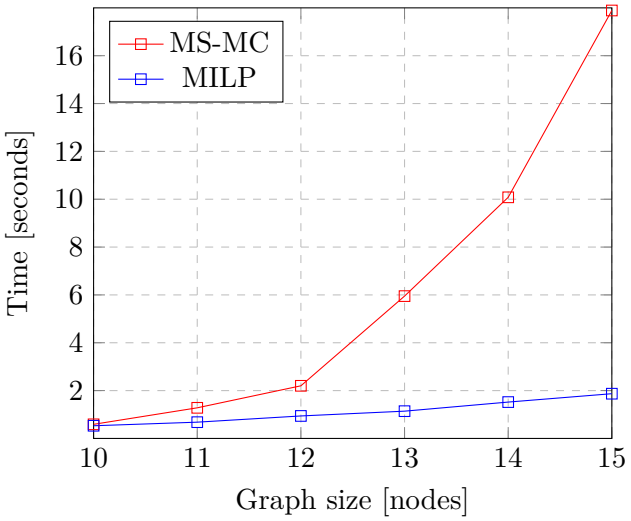


Figure 5.1: Performance comparison between MILP Reformulation and MaxSearch with Marginal Checking algorithms for MIS problem reformulated as MAP inference in SPN.

Although efficient, it is important to note that such SPNs derived from the MIS problem in graph are not related to any real problem, which would generate SPNs with completely different structures, certainly affecting the performance of our algorithm.

We also tested our algorithm with hard instances for the MIS problem. As described by Ke Xu in [Xu], finding challenging benchmarks for the MIS problem is not only of significance for experimentally evaluating the algorithms of solving this problem but also of interest to the theoretical computer science community. In fact, he proposed an algorithm to generate hard MIS problem instances by generating random graphs as follows:

1. Generate n disjoint cliques, each of which has n^a vertices (where a > 0 is a constant)
2. Randomly select two different cliques and then generate, without repetitions, pn^{2a} random edges between these two cliques (where $0 < p < 1$ is a constant)
3. Run step 2 (with repetitions) for another $rn\ln(n-1)$ times (where $r > 0$ is a constant).

It is easy to see that for the graph model above, the size of maximum independent sets is at most n. During these tests, we checked the answer of our algorithm, but also checked the validity of the selected variables in the original graph, confirming that the solution provided by the solver was a maximal independent set in fact. For larger graphs, with more than 50 vertices, we started to find failures in the optimal solution due to numerical issues. The MAP inference values for larger translated SPNs were below the minimum tolerance value allowed by Gurobi (10^{-6}) for verifying constraint feasibility and integrality.

5.2 Real Data Tests

Real datasets can be obtained throughout the web, and among them some are gathered at the ProbAI-USP GitLab repository, as described in the beginning of this chapter. Some tests with

real data were already made, but still they were somehow biased by how the SPNs were learned from the data. Since our goal was not related to SPN learning, we used the LearnSPN algorithm embedded in the package mentioned previously to generate SPNs through raw data. However, from the perspective of likelihood, these SPNs do not necessarily describe the best probability distribution that could be obtained from the dataset. Thus, is still one of our goals to find a good benchmark set of SPNs to properly understand the nuances of the algorithm developed in this research.

The experiments in this section were shown by Mauá et al. in [MRKA20]. We performed them with SPNs learned from a selected collection of datasets where all variables are discrete or have been discretized. We learn tree-shaped SPNs with our own implementation of the LearnSPN algorithm presented by Gens and Domingos in [GD13], selecting hyperparameters by grid search. We then modified the networks so that each node has at most two parents. Figure 5.3 contains the characteristics of the datasets and the SPNs we use, sorted by the number of indicators (which are related to the size of the space of configuration).

SPN	Variables	Nodes	Sum Nodes	Product Nodes	Indicator Nodes
NLTCS	16	3939	2315	1592	32
Mushrooms	112	8489	4587	3678	224
Molecular	58	10207	7515	2462	230
DNA	180	33465	17953	15152	360
US Census	68	162118	140722	20752	644
NIPS	500	13563	6876	5687	1000
Optdigits	65	23008	20739	1171	1098

Table 5.3: Details of the SPN learned from real data through LearnSPN algorithm [PD11].

For each SPN/dataset we generate MAP inference tasks as follows. First we randomly partition the variables into equally sized sets of optimized, marginalized and fixed (the partition is constant for each SPN). Then we select values for the fixed/evidence variables using the test set such that Max-Product is suboptimal. This procedure generated from 10 to 30 interesting instances per SPN/dataset (the maximum of 30 instances was imposed for computational convenience). We verify sub-optimality by obtaining a higher value solution from Argmax-Product. Thus, our results here approximate a worst-case performance for Max-Product rather than the expected performance. They also slightly favor Max-Product (as we might discard instances where Argmax-Product does not find an improving solution and some other competing method does.)

We compare the reformulation approaches against Max-Product (MP) presented by Poon and Domingos in [PD11], Argmax-Product followed by local search (AMP) presented by Conaty et al. in [CMdC17], and MaxSearch (MS) presented by Mei et al. in [MJT18]. We initialize MS with the solution found by MP, and use the forward checking heuristic. Except for MS, which performs branch-and-bound search, all other methods are approximate and run reasonably fast. The runtime limit of MS was set to 1 hour.

For the MILP reformulation, we obtain a path decomposition by eliminating variables according to the min-degree heuristic. We used Gurobi to optimize the corresponding MILP pro-

gram, multiplying the objective function and the constraints by some large constant (e.g., 10^6), to improve numerical stability. For the two smaller networks (NLTCS and Mushrooms), the translation to a MILP program was very fast, and the optimization finished within a couple of minutes with the guaranteed optimal solution. The translation was also very fast for the Molecular network, but Gurobi failed at finding an optimal solution due to numerical issues. The MAP inference values for that network ranged from 10^{-23} to 10^{-14}, below the minimum tolerance value allowed by Gurobi (10^{-6}) for verifying constraint feasibility and integrality. For the larger networks, the translation took more than 1 hour, at which point it was interrupted.

The results for the different approaches appear in Table 5.4. The numbers show the mean and standard deviation of the ratio between the value of a given solution and the value of the corresponding solution obtained by MP.

SPN	AMP	MS	MILP
NLTCS	2.35 ± 1.56	2.35 ± 1.56	2.35 ± 1.56
Mushrooms	1.44 ± 0.30	1.44 ± 0.29	1.44 ± 0.29
Molecular	3.01 ± 2.88	1.00 ± 0.00	•
DNA	1510.9 ± 1247.7	313.2 ± 363.9	$\star$
US Census	1.81 ± 0.63	1.81 ± 0.63	$\star$
NIPS	27325.3 ± 33348.6	1.00 ± 0.00	$\star$
Optdigits	80.19 ± 210.30	1.02 ± 0.13	$\star$

Table 5.4: The numbers show the mean and standard deviation of the ratio between the value of a given solution and the value of the corresponding solution obtained by Max-Product. For the MILP algorithm, entries marked with *bullet* had numerical issues during the solver processing, and entries marked with $\star$ passed the 1 hour runtime limit.

As we might see, our algorithm still can not handle larger networks. Thus, in a second round of tests with real data, we decided to select more manageable size of datasets, for which we generated the SPNs as shown in Table 5.5. This is so we could better analyze how the number of variables and nodes affect the complexity of the generated MILP program, further detailed in Subsections 5.2.1 and 5.2.2.

SPN	Variables	Nodes	Sum Nodes	Product Nodes	Indicator Nodes
Balance Scale	6	432	154	54	224
Haberman	7	861	293	98	470
EColi	8	787	266	79	442
Diabetes	9	3121	1054	333	1734
Glass	10	1056	355	91	610
Hepatitis	20	2282	761	133	1388

Table 5.5: Details of the SPNs obtained from real data through LearnSPN algorithm [PD11].

Next, we present the tables with details of the performance for each algorithm mentioned in this work, tested with the SPNs mentioned in Table 5.5. As we might see from the graphic in Figure 5.2, we have an intuition that there might exist some factors influencing the performance other than just the number of variables in the original SPN.

Algorithm	Time	Value	Variable Configuration
Max-Product	0.01024199	0.09099970	001000
Argmax-Product	0.00327550	0.09099970	001000
MaxSearch with MC	0.06409350	0.09099970	001000
MaxSearch with FC	0.09099970	0.09099970	001000
SPN-to-MILP	3.73637485	0.09099970	001000

Table 5.6: Performance details for the SPN "Balance Scale".

Algorithm	Time	Value	Variable Configuration
Max-Product	0.03250750	0.02582258	0001100
Argmax-Product	0.00646349	0.02582258	0001100
MaxSearch with MC	0.33531900	0.02582258	0001100
MaxSearch with FC	2.94009450	0.02582258	0001100
SPN-to-MILP	11.0407509	0.02582258	0001100

Table 5.7: Performance details for the SPN "Haberman".

Algorithm	Time	Value	Variable Configuration
Max-Product	0.03596050	0.10613223	11111000
Argmax-Product	0.00640099	0.10613223	11111000
MaxSearch with MC	0.28329500	0.10613223	11111000
MaxSearch with FC	2.00685500	0.10613223	11111000
SPN-to-MILP	7.32611107	0.10613223	11111000

Table 5.8: Performance details for the SPN "EColi".

Algorithm	Time	Value	Variable Configuration
Max-Product	0.13135750	0.02541521	1111111110
Argmax-Product	0.02774799	0.02541521	1111111110
MaxSearch with MC	1.49628350	0.02541521	1111111110
MaxSearch with FC	14.3683209	0.02541521	1111111110
SPN-to-MILP	242.578780	0.02541522	1111111110

Table 5.9: Performance details for the SPN "Diabetes".

Algorithm	Time	Value	Variable Configuration
Max-Product	0.03793299	0.06015176	0001110000
Argmax-Product	0.00901450	0.06015176	0001110000
MaxSearch with MC	0.63130249	0.06015176	0001110000
MaxSearch with FC	3.76724099	0.06015176	0001110000
SPN-to-MILP	11.1868999	0.06015178	0001110000

Table 5.10: Performance details for the SPN "Glass".

Algorithm	Time	Value	Variable Configuration
Max-Product	0.38725849	0.00195443	11100001000001110001
Argmax-Product	0.02245049	0.00195443	11100001000001110001
MaxSearch with MC	483.841924	0.00256849	11100001000001010001
MaxSearch with FC	607.224486	0.00256849	11100001000001010001
SPN-to-MILP	67.6663408	0.00256853	11100001000001010001

Table 5.11: Performance details for the SPN "Hepatitis".

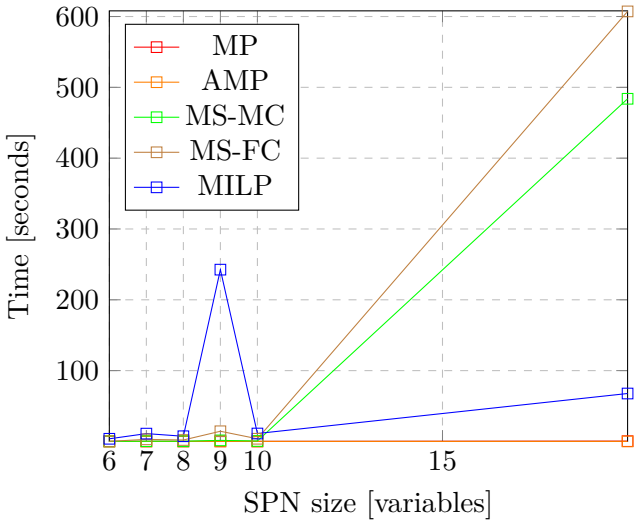


Figure 5.2: Performance comparison between MILP Reformulation and other algorithms for MAP inference in SPNs.

5.2.1 Generated MILP program size

To understand which are the causes correlated to the performance shown in Figure 5.2, we believe that one of the first places to look at is the size of the generated MILP program. For the next test, we used the set of SPNs as shown in Table 5.5.

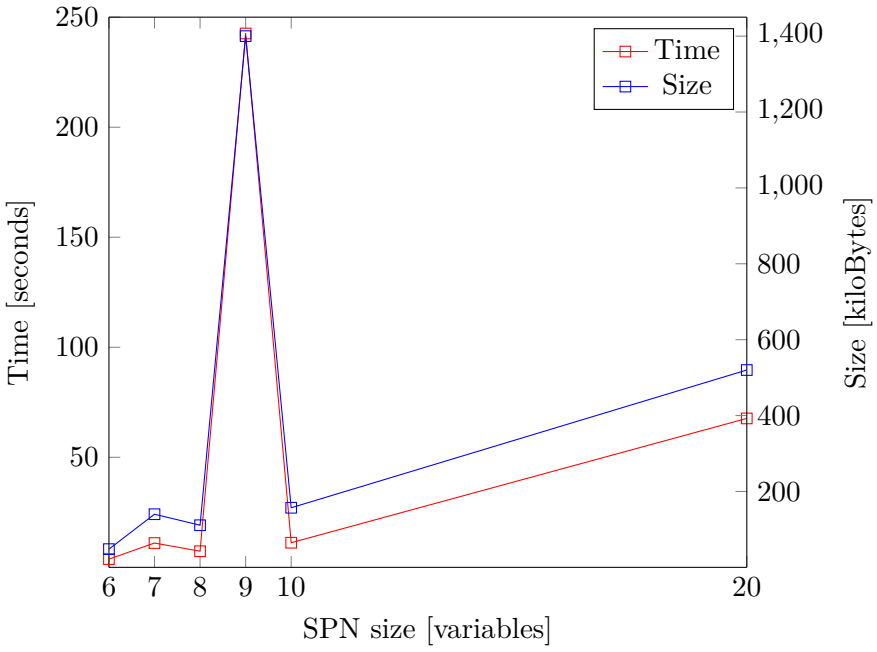


Figure 5.3: The relations between the time of execution for our algorithm and the size of the generated MILP program file `.lp`.

As we might see from Figure 5.3, there seems to be a close relation between the runtime of our algorithm and the size of MILP program files that it generates. Further on, we analyze what are the possible causes for such variance in the MILP program file size.

5.2.2 Relation between Variables and Constraints

One of the possible relations we can try to find is the one relating the number of variables X_i and the number of constraints in the MILP program file. The graph in Figure 5.4 shows such relation.

However, it shows basically what has been shown in Figure 5.3. But since the size of PADDs are mostly related to the number of variables Y_i as well, it would be interesting to check such relation but using the amount of sum nodes in the original SPNs.

As we might see, the performance of our algorithm seems to be very affected from the number of variables Y_i, that is, the number of sum nodes in the original SPN. Naturally, our tests show very little variation in the number of variables X_i, since they have a strong relation with the amount of sum nodes learned by an SPN. It makes sense to think that the more variables X_i there are in an SPN, more sum nodes are necessary to describe the relation between these variables.

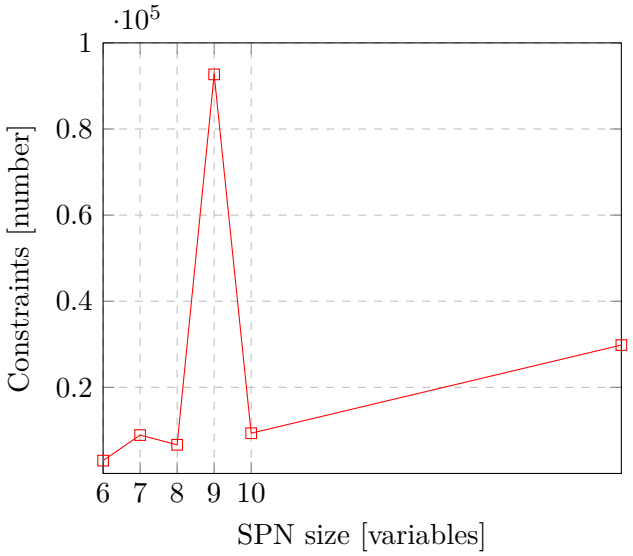


Figure 5.4: The number of constraints generated by the SPN-MILP algorithm according to the number of variables X_i in the original SPN.

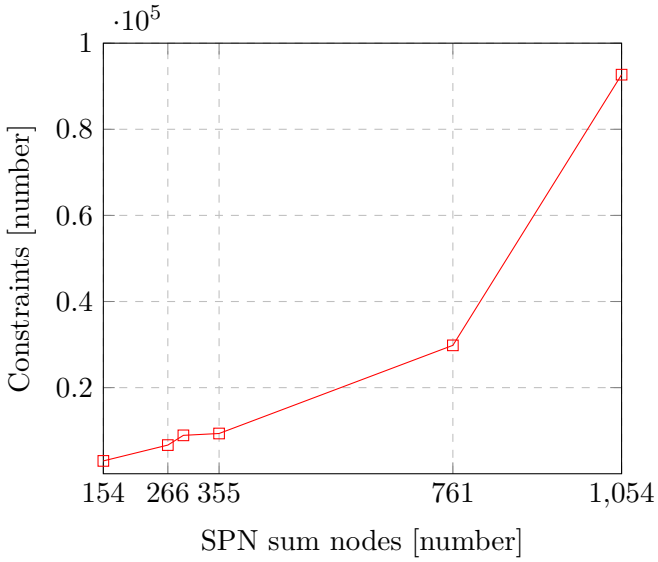


Figure 5.5: The number of constraints generated by the SPN-MILP algorithm according to the number of sum nodes in the original SPN.

Chapter 6

Discussion

In this document we present what has been studied and achieved until now within the field of study of MAP inference in SPNs as translated to a MILP optimization problem. We provided the concepts and details of implementation that we believe to be important for the understanding of what we developed.

Sum-Product Networks are often justified for their ability to deliver complex probabilistic queries in polynomial time. However, several successful examples of applications of SPNs require solving MAP inference, that is, maximizing over part of the variables given the values of some others. This is the case for example, when performing data imputation with SPNs, completing images, building multidimensional classifiers, and so on. Solving such a task is however NP-hard. As our main contribution, we presented a reformulation of the MAP problem as a Mixed-Integer Linear Program. This opens up the vast work on mathematical programming, as well as access to the very efficient commercial solvers available.

We understand that the presented research deals with several fields of study that were not related until now, and we believe our research to be a way of connecting them. We believe to have some meaningful contributions, such as i) the development of a PADDs support software package, ii) the development of the translation algorithm itself from MAP inference in SPN problem to a MILP optimization problem, iii) analysis related to how our proposed algorithm performs in comparison with other algorithms available and analysis of the performance of our algorithm depending on the input, and finally iv) the publication of an article whose content is partly based on this research.

6.1 Limitations

Naturally, we had limitations along this research that hindered us to obtain better results for the developed software. Some of them are listed below.

- The lack of adequate datasets/SPNs for an initial testing and analysis of the proposed algorithm.
- The numerical precision of the optimization solver. As we deal with small numbers that are multiplied in sequence.

- The experimental code and its suboptimal performance, developed in not such efficient programming language.

6.2 Future Work

In this section we suggest to where next research related to our work should point towards. Naturally, there are still many aspects to explore. For example, finding better path decomposition methods in order to perform variable elimination in the most efficient way possible, or finding better ways to develop the PADD software.

- Finding algorithms to provide better path decompositions in order to perform variable elimination more efficiently. Future work should evaluate different heuristics for obtaining path decomposition.
- Finding better datasets and SPNs, not only those learned through LearnSPN, to provide a more diverse and precise analysis of the current reformulation algorithm.

Appendix A

Implementation

In this appendix we present some implementation details about the software developed during this work, such as how the `pyddlib` was expanded and how we built PADDs $\mathcal{A}_i$ from Subsection 4.1.4.1. As for the source code, it was entirely written in Python 3.6.

A.1 Extending pyddlib

In order to work with PADDs, we extended the BDD and ADD Python package `pyddlib` [Bue]. To encode a PADD, we firstly had to find a way to change the type of the objects in the leaf nodes. Our purpose in using PADDs was for the manipulation of functions that output multilinear expressions, hence we focused on building a object class based on such expressions. It is possible to find our implementation on the ProbAI-USP GitLab repository [PU20b]. To use it, firstly we must import the PADD and the Multilinear Expression modules:

```
from pyddlib.mlexpr import MLExpr
from pyddlib.padd import PADD
```

To represent a multilinear expression we used the Python dictionary. Each of its elements is a pair (key, value) that represents a term in the original expression, hence, each term is implicit to be part of a sum. The key of an element is a tuple of numbers, where each number is the index of an specific variable. The tuple represents the multiplication of the variables related to each index, and the correspondent value is the coefficient of the term it represents.

For example, let us say that we are trying to encode the following multilinear expression:

$$3.4Y_2Y_4 - 0.3Y_1Y_2Y_3$$

We can associate each variable Y_i with the index i to create the dictionary that will represent the expression above. Remember that the dictionary is the whole expressions, while each of its elements are terms.

```
e1 = MLExpr({(2, 4): 3.4, (1, 2, 3): -0.3})
```

We assume, in the `MLExpr` class, that all the tuples in a dictionary are sorted, so as to avoid the case of having the last tuple as key, then avoiding inconsistent representation. Also,

this representation actually could handle not only multilinear expressions but also polynomial expressions, since we could have a tuple like `(1, 1, 2, 3)` as key, which could mean $Y_1^2 Y_2 Y_3$. Yet, we understand that it is not the most efficient one for such general purposes.

Once we had the structured the multilinear expressions objects, we just had to use them as elements for the terminal leaves, exchanging the usage of real numbers of the previous ADD already built in the pyddlib. And since we wanted to manipulate PADDs in terms of sums and products, we expanded such operations for multilinear expressions as well. The nature of our work allows us to handle the product of multilinear expressions because we assure that only multilinear expressions with disjoint scope are going to be multiplied, which means, for example, that X_i will never appear in both expressions, and hence we will never have to handle terms where variables carry exponents greater than 1. However, such rule is not coded in the package expansion, which means that the product of multilinear expressions that are not disjoint would yield an expression that is not multilinear.

A.2 Building PADDs

The PADDs building process is basically an execution of the DFS (Depth-First Search) algorithm, starting from the root node of the SPN. During its run-time, a stack is assembled based on what is the current taken path in every sum node, and every time the search reaches a leaf node, the stack is checked to assure what is the configuration of such a particular path between the root and a leaf node.

In addition, for every first visit of a sum node, a PADD $\hat{\mathcal{A}}_i$ is built, as described in Subsection 4.1.4.2, and for every visit of leaf node that references a variable X_i for the first time, the PADD $\mathcal{X}_i$ is created, according to Subsection 4.1.4.3.

At this stage, such path contains two important data:

- the variable X_i to which the leaf node is related
- the way taken in the SPN (or decision, in PADD terms) for each sum node, which will be used as reference to build the PADD A_i (since it is related to the variable X_i)

Let us say that the DFS reached a leaf node with variable X_i. If $\mathcal{X}_i$ is not created, than the algorithm creates it. Now that we have a complete path from the root down to one leaf containing X_i, the path stored in the stack is updated in $\mathcal{A}_i$. When the DFS is completed, all the paths between the root and leaf nodes with variables X_i are updated in $\mathcal{A}_i$.

At the end of the DFS, all the PADDs $\mathcal{A}_i$, $\hat{\mathcal{A}}_i$ and $\mathcal{X}_i$ are created.

Bibliography

[All01] Paul Allison. *Missing Data*. Quantitative Applications in the Social Sciences. Sage Publications, 2001. 2

[AT12] Mohamed Amer and Sinisa Todorovic. Sum-Product networks for modeling activities with stochastic structure. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1314–1321, 2012. 8

[AT16] Mohamed Amer and Sinisa Todorovic. Sum-Product networks for activity recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(4):800–813, 2016. 8

[BFG+97] Iris Bahar, Erica Frohm, Charles Gaona, Gary Hachtel, Enrico Macii, Abelardo Pardo and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2-3):171–206, 1997. 12, 17

[BGHK95] Hans L Bodlaender, John R Gilbert, Hjálmtýr Hafsteinsson and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18(2):238–255, 1995. 21

[Boo54] George Boole. *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities*. Dover Publications, 1854. 13

[Bry86] Randal Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986. 12, 13, 14

[Bue] Thiago Bueno. pyddlib: Python Decision Diagram Library. https://pypi.org/project/pyddlib. 59

[BVZ98] Yuri Boykov, Olga Veksler and Ramin Zabih. Markov random fields with efficient approximations. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 648–655, 1998. 4

[CDF96] Kevin Cattell, Michael J Dinneen and Michael R Fellows. A simple linear-time algorithm for finding path-decompositions of small width. *Information Processing Letters*, 57(4):197–203, 1996. 21

[CG08] Anton Chechetka and Carlos Guestrin. Efficient principled learning of thin junction trees. In *Advances in Neural Information Processing Systems*, pages 273–280, 2008. 10

[CKP+14] Wei-Chen Cheng, Stanley Kok, Hoai Vu Pham, Hai Leong Chieu and Kian Ming A Chai. Language modeling with Sum-Product networks. In *Proceedings of the 15th Annual Conference of the International Speech Communication Association*, 2014. 8

[CMdC17] Diarmaid Conaty, Denis Mauá and Cassio de Campos. Approximation complexity of maximum a posteriori inference in Sum-Product networks. *Proceedings of 33rd International Conference on Uncertainty in Artificial Intelligence*, pages 322–331, 2017. 8, 25, 27, 48, 50

[Col03] Michael Collins. Head-driven statistical models for natural language parsing. *Computational linguistics*, 29(4):589–637, 2003. 10

[Dar03] Adnan Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM (JACM)*, 50(3):280–305, 2003. 8, 10

[DSDB11] Karina Valdivia Delgado, Scott Sanner and Leliane Nunes De Barros. Efficient solutions to factored mdps with imprecise transition probabilities. *Artificial Intelligence*, 175(9-10):1498–1527, 2011. 12, 18, 19

[GD13] Robert Gens and Pedro Domingos. Learning the structure of Sum-Product networks. In *Proceedings of 30th International Conference on Machine Learning*, pages 873–880, 2013. 11, 50

[Geh] Renato Geh. GoSPN: A Sum-Product Network (SPN) library. https://github.com/RenatoGeh/gospn. 3

[GG84] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984. 4

[GO16] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016. 44, 47

[KRC+10] Terry Koo, Alexander Rush, Michael Collins, Tommi Jaakkola and David Sontag. Dual decomposition for parsing with non-projective head automata. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1288–1298, 2010. 4

[Lee59] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959. 13

[MF90] John McCarthy and Edward Feigenbaum. In memoriam - Arthur Samuel: Pioneer in machine learning. *AI Magazine*, 11(3):10–11, 1990. 1

[MJT18] Jun Mei, Yong Jiang and Kewei Tu. Maximum a posteriori inference in Sum-Product networks. In *32nd AAAI Conference on Artificial Intelligence*, 2018. 27, 50

[MRKA20] Denis Deratani Mauá, Heitor Ribeiro, Gustavo Katague and Alessandro Antonucci. Two reformulation approaches to maximum-a-posteriori inference in sum-product networks. In *Proceedings of the 10th International Conference on Probabilistic Graphical Models (PGM 2020)*, Proceedings of Machine Learning Research. PMLR, 2020. 5, 44, 50

[ND16] Aniruddh Nath and Pedro Domingos. Learning tractable probabilistic models for fault localization. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 1294–1301, 2016. 8

[PD11] Hoifung Poon and Pedro Domingos. Sum-Product networks: A new deep architecture. In *Proceedings of 27th Conference on Uncertainty in Artificial Intelligence*, pages 337–346, 2011. 1, 8, 9, 11, 23, 47, 50, 51

[Pea85] Judea Pearl. Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proceedings of the 7th Conference of the Cognitive Science Society*, pages 329–334, 1985. 1

[PKMP14] Robert Peharz, Georg Kapeller, Pejman Mowlaee and Franz Pernkopf. Modeling speech with Sum-Product networks: Application to bandwidth extension. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3699–3703, 2014. 4

[PTPD15] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf and Pedro Domingos. On theoretical properties of Sum-Product networks. In *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics*, pages 744–752, 2015. 10, 11

[PU20a] ProbAI-USP. learned-spns. https://gitlab.com/pgm-usp/learned-spns, 2020. 47

[PU20b] ProbAI-USP. pyspn. https://gitlab.com/pgm-usp/pyspn, 2020. 47, 59

[Rot96] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996. 4

[Sam59] Arthur Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959. 1

[Sha38] Claude Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12):713–723, 1938. 13

[Sha49] Claude Shannon. The synthesis of two-terminal switching circuits. *The Bell System Technical Journal*, 28(1):59–98, 1949. 13

[SS04] Małgorzata Steinder and Adarshpal Sethi. Probabilistic fault localization in communication systems using belief networks. *IEEE/ACM Transactions on Networking*, 12(5):809–822, 2004. 4

[SZS+08] Richard Szeliski, Ramin Zabih, Daniel Scharstein, Olga Veksler, Vladimir Kolmogorov, Aseem Agarwala, Marshall Tappen and Carsten Rother. A comparative study of energy minimization methods for Markov random fields with smoothness-based priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(6):1068–1080, 2008. 4

[Xu] Ke Xu. BHOSLIB: Benchmarks with hidden optimum solutions for graph problems (maximum clique, maximum independent set, minimum vertex cover and vertex coloring). http://sites.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm. 49

[Zha04] Nevin Zhang. Hierarchical latent class models for cluster analysis. *Journal of Machine Learning Research*, 5(6):697–723, 2004. 10

[ZMP15] Han Zhao, Mazen Melibari and Pascal Poupart. On the relationship between Sum-Product networks and Bayesian networks. In *International Conference on Machine Learning*, pages 116–124, 2015. 2, 6, 11, 12, 31, 32, 33, 40, 44

[ZNSS11] Cäcilia Zirn, Mathias Niepert, Heiner Stuckenschmidt and Michael Strube. Fine-grained sentiment analysis with structural features. In *Proceedings of 5th International Joint Conference on Natural Language Processing*, pages 336–344, 2011. 4