Scalable Learning of Probabilistic Circuits

Renato Lui Geh

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Master of Science

Program:	Computer Science
Advisor:	Professor Denis Deratani Mauá

This work was supported by CNPq grant #133787/2019-2,
CAPES grant #88887.339583/2019-00 and EPECLIN FM-USP.

São Paulo
November 1, 2021

Scalable Learning of Probabilistic Circuits

Renato Lui Geh

Acknowledgements

Abstract

Renato Lui Geh. **Scalable Learning of Probabilistic Circuits**. Thesis (Master's). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

The rising popularity of generative models together with the growing need for flexible and exact inferences has motivated the machine learning community to look for expressive yet tractable probabilistic models. Probabilistic circuits (PCs) are a family of tractable probabilistic models capable of answering a wide range of queries exactly and in polynomial time. Their operational syntax in the form of a computational graph and their principled probabilistic semantics allow their parameters to be estimated by the highly scalable and efficient optimization techniques used in deep learning. Importantly, tractability is tightly linked to constraints on their underlying graph: by enforcing certain structural assumptions, queries like marginals, *maximum a posteriori* or entropy become linear time computable while still retaining great expressivity. While inference is usually straightforward, learning PCs that both obey the needed structural restrictions and exploit their expressive power has proven a challenge. Current state-of-the-art structure learning algorithms for PCs can be roughly divided into three main categories. Most learning algorithms seek to generate a usually tree-shaped circuit from recursive decompositions on data, often through clustering and costly statistical (in)dependence tests, which can become prohibitive in higher dimensional data. Alternatively, other approaches involve constructing an intricate network by growing an initial circuit through structural preserving iterative methods. Besides depending on a sufficiently expressive initial structure, these can possibly take several minutes per iteration and many iterations until visible improvement. Lastly, other approaches involve randomly generating a probabilistic circuit by some criterion. Although usually less performant compared to other methods, random PCs are orders of magnitude more time efficient. With this in mind, this dissertation aims to propose fast and scalable random structure learning algorithms for PCs from two different standpoints: from a logical point of view, we efficiently construct a highly structured binary PC that takes certain knowledge in the form of logical constraints and scalably translate them into a probabilistic circuit; from the viewpoint of data guided structure search, we propose hierarchically building PCs from random hyperplanes. We empirically show that either approach is competitive against state-of-the-art methods of the same class, and that their performance can be further boosted by simple ensemble strategies.

Keywords: Probabilistic circuits. Machine learning. Probabilistic models.

Resumo

Renato Lui Geh. **Aprendizado Escalável de Circuitos Probabilísticos**. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Palavras-chave: Circuitos probabilísticos. Aprendizado de máquina. Modelos probabilísticos.

Nomenclature

List of Symbols

σ	Sigmoid function

List of Figures

List of Algorithms

List of Examples

List of Remarks

Notation

We use the following notation throughout the work. Random variables are written in upper case (e.g. X, Y) and their values in lower case (e.g. x, y). We write $\mathcal{X}$ as the sample space and for independence we use $\perp\!\!\!\perp$ to indicate that two variables X and Y are statistically independent, i.e. $X \perp\!\!\!\perp Y$. We identify propositional variables with 0/1-valued random variables, and use them interchangeably. Sets of variables and their joint values are written in boldface (e.g. $\mathbf{X}$, $\mathbf{x}$). Given a Boolean formula f, we write $\langle f \rangle$ to denote its semantics, i.e. the Boolean function represented by f. For Boolean formulas f and g, we write $f \equiv g$ if they are logically equivalent, that is, if $\langle f \rangle = \langle g \rangle$; we abuse notation and write $\phi \equiv f$ to indicate that $\phi = \langle f \rangle$ for a Boolean function ϕ. We use the notation $[a..b]$, with $b \geq a$ to denote the integer set $\{a, a+1, \ldots, b\} \subset \mathbb{Z}$. Similarly, we use $[b]$ as an equivalent for $[1..b]$. We denote the Iverson bracket as $[\![\phi]\!]$, i.e. a function that returns 1 if ϕ is true and 0 otherwise. In the context of graph theory, we use sans serif letters for graph nodes (e.g. N, S, P, L) and bold variants for sets of nodes (e.g. $\boldsymbol{\mathsf{N}}$, $\boldsymbol{\mathsf{S}}$, $\boldsymbol{\mathsf{P}}$, $\boldsymbol{\mathsf{L}}$). We call $\mathrm{Ch}(\mathsf{N})$ the set of all children of a node N, $\mathrm{Pa}(\mathsf{N})$ as the set of all parents, and $\mathrm{Desc}(\mathsf{N})$ the set of all descendants.

Contents

Annexes

1 Introduction

When reasoning about the world, rarely can we find a realistic model that perfectly subsumes all of the needed relationships for flawless prediction. As such, the presence of a reliable uncertainty quantifier in intelligent systems is essential in developing performant yet diagnostible agents. This is made explicitly clear in the case of high-risk settings, such as autonomous vehicles or automated power plant systems, where a wrong prediction could cause disastrous consequences. A well known example in the case of the former is obstacle avoidance: while the agent should be capable of accurately identifying obstructions in its way during normal conditions, so should it be able to identify its own lack of confidence in high uncertainty situations like ones brought by environmental factors, such as severe blizzard or heavy rain. In these situations where predictions are highly unreliable, the safest option might be for the agent to first identify its uncertainty, and second to reach out for human help. Other interesting applications of uncertainty quantification include out-of-distribution detection, which in our previous example could be visualized as the agent identifying a human's driving as abnormally irregular (due to inebriation, infirmity, etc.) and appropriately taking control of the vehicle before a potential accident takes place.

One popular approach to quantifying uncertainty is through probability theory. By abstracting the world as a probability distribution with a finite number of observable random variables that encode a possibly incomplete knowledge of the environment, we are, in theory, able to answer a diverse set of complex queries as long as we have access to the (approximate) true data distribution. Practice is far different from theory, however, as most machine learning models either lie within a very limited range in the tractability spectrum in terms of inference (Y. Choi, Vergari, and Van den Broeck, 2020) or are too simplistic for complex real-world problems. Besides, although the majority of recent advances in deep learning claim some probabilistic meaning from the model's output, they are often uncalibrated distributions, a result of focusing on maximizing predictive accuracy at the expense of predictive uncertainty (Guo *et al.*, 2017; Ovadia *et al.*, 2019; Chernikova *et al.*, 2019), ultimately producing overconfident and peculiar results (Szegedy *et al.*, 2013; Wei *et al.*, 2018; Su *et al.*, 2019; Chernikova *et al.*, 2019). Crucially, mainstream deep models (i.e. standard neural networks) usually optimize a conditional distribution over the to-be-predicted random variables – and are thus often called *discriminative* models – and do not model the actual joint distribution of the data, limiting inference capabilities and uncertainty estimation.

In contrast, *generative* models seek to extract information from the joint (in varying capacities), and have lately seen a sharp rise in interest within deep learning. Despite this, most popular models do not admit either exact or tractable querying of key inference scenarios. For instance, although Generative Adversarial Networks (GANs) allow for efficient sampling (Goodfellow *et al.*, 2014), basic queries such as likelihood or marginals are outside of their capabilities. Similarly, Normalizing Flows (NF) also permit access to efficient sampling, with the added feature of computing likelihoods, but are severely limited by the base distribution when it comes to discrete data (Rezende and Mohamed, 2015; Papamakarios *et al.*, 2021) albeit recent works on discretizing NFs have shown empirically good results (Lippe and Gavves, 2021; Ziegler and Rush, 2019). Variational Auto-Encoders (VAEs) are (under certain conditions) a generalization of NFs (Yu, 2020; Gritsenko *et al.*, 2019) with known extensions for categorical data (Rolfe, 2017; Vahdat, Macready, *et al.*, 2018; Vahdat, Andriyash, *et al.*, 2018), but only permit access to sampling and an upper-bound on the likelihood, with the latter available only after solving a complex optimization task (Kingma and Welling, 2014).

Despite the impressive achievements of the aforementioned generative models on realistically producing samples consistent with evidence, in none of the previous models are complex queries like structured prediction under partial observations, *maximum a posteriori* (MAP), conditional or marginal probabilities tractable. An obvious alternative would be Probabilistic Graphical Models (PGMs), although they too suffer from intractability when dealing with high treewidth networks (R. Dechter, 1998; Koller and Friedman, 2009), severely limiting expressivity. Instead, we draw our attention to an expressive class of models that subsumes several families of probabilistic models with tractable inference capabilities.

Probabilistic Circuits (PCs) define a superclass of probabilistic models distincly specified by recursive compositions of distributions through graphical formalisms. Vaguely speaking, PCs are computational graphs akin to neural networks, but whose network structure and computational units abide by special constraints. Within these specific conditions span a wide range of subclasses, each establishing a distinct set of restrictions on their structure in order to enable different segments within the tractability spectrum. As an example, Sum-Product Networks (SPNs, Poon and P. Domingos, 2011) are usually loosely defined over a couple of constraints: namely *smoothness* and *decomposability*, which in turn enables likelihood, marginal and conditional computations. Arithmetic Circuits (ACs, Darwiche, 2003) add *determinism* to the mix, allowing for tractable computation of MAP probabilities. Similarly, Cutset Networks (CNets, Rahman, Kothalkar, *et al.*, 2014) employ the same constraints as ACs, but accept more expressive distributions as part of their computational units. Probabilistic Sentential Decision Diagrams (PSDDs, Kisa *et al.*, 2014), Probabilistic Decision Graphs (PDGs, Jaeger, 2004) and And/Or-Graphs (AOGs, Rina Dechter and Mateescu, 2007) all require *smoothness* and *determinism*, but also call for a stronger version of *decomposability*, permitting all queries previously mentioned as well as computation of the Kullback-Leibler divergence and expectation between two circuits (Y. Choi, Vergari, and Broeck, 2020). Usually, PCs represent the joint distribution of the data, although they are sufficiently expressive for generative *and* discriminative modeling (Khosravi *et al.*, 2019; Rashwan *et al.*, 2018; Rooshenas and Lowd, 2016; Gens and P. Domingos, 2012; Shao *et al.*, 2020). In this dissertation though, we shall focus on the generative side of PCs.

While inference is usually straightforward, as we shall see in Section 2.2, learning the structure of PCs so that they obey the needed structural restrictions requires either careful handcrafted architectures (Poon and P. Domingos, 2011; Cheng *et al.*, 2014; Nath and P. M. Domingos, 2016) or usually involves running costly (in)dependence tests over most (if not all) variables (Gens and P. Domingos, 2013; Jaini, Ghose, *et al.*, 2018; Vergari, Mauro, *et al.*, 2015; Di Mauro *et al.*, 2017), which can become prohibitive in higher dimension data. Alternatively, some learning algorithms resort to structure preserving iterative methods to grow a PC that already initially satisfies desired constraints, adding complexity to the underlying distribution at each iteration (Liang, Bekker, *et al.*, 2017; Dang *et al.*, 2020). However, these can take several iterations until visible improvement and often take several minutes for each iteration when the circuit is big. Common techniques used in deep learning for generating scalable architectures for neural network also pose a problem, as the nature of the needed structural constraints make for sparse computational graphs. To circumvent these issues, work on scaling PCs to higher dimensions has focused mainly on random architectures, with competitive results (Peharz, Vergari, *et al.*, 2020; Mauro *et al.*, 2021; R. L. Geh and Denis Deratani Mauá, 2021; Peharz, Lang, *et al.*, 2020). Apart from the scalability side of random structure generation, usual structure learning algorithms often require grid-search for hyperparameter tuning to achieve top quality performance, which is usually not the case for random algorithms. For the usual data scientist or machine learning practicioner, hyperparameter tuning can become exhaustive, especially if the goal is to analyze and infer from large data, and not to achieve top tier performance on benchmark datasets.

In this dissertation, we propose two scalable structural learning algorithms for probabilistic circuits that are especially suited for large data and fast deployment. They both take advantage of random network generation to quickly construct PCs with little to no need for hyperparameters. The first is effective for constructing PCs from binary data with a highly constrained structure, and thus appropriate when complex querying is needed. The second builds less constrained random PCs, but supports both discrete and continuous data.

1.1 Contributions and Dissertation Outline

We organize this dissertation as follows. We begin Chapter 2 by formally defining probabilistic circuits, conducting a review of some of the structural constraints that we might impose on PCs, as well as what we may gain from them in terms of tractability. We then list existing formalisms that may be viewed as instances of PCs, and what their structure entail in terms of inference power. In Chapter 3, we address existing PC structure learning algorithms, and which guarantees in terms of tractability each give. We cover the two new structure learners in Chapter 4, providing empirical results on their performance. The final chapter is dedicated to summarizing our research contributions and pointing to potential future work in learning PCs.

Our contributions in this dissertation address the following research topics.

Scalably learning PCs directly from background knowledge

In R. L. Geh and Denis Deratani Mauá (2021), we provide a learning algorithm for PSDDs that learns a PC directly from background knowledge in the form of logical constraints. The algorithm samples a structure from a distribution of possible PSDDs that are weakly consistent with the logical formula. How weak consistency is depends on a parameter that trades permission of false statements as non zero probability events with circuit complexity. We provide the algorithm and empirical results in Section 4.??.

Using ensembles to strengthen consistency

The PC sampler given by R. L. Geh and Denis Deratani Mauá (2021) produces competitive probabilistic models (in terms of likelihood), albeit weak logical models in the sense that it possibly assigns non-zero probability to false variable assignments – as we discuss in Section 4.??, it never assigns zero probability to true statements. By producing many weak models, we not only gain in terms of data fitness, but also consistency: if any one component in the ensemble returns an assignment to be impossible, the whole model should return false.

Random projections to efficiently learn PCs

Usual methods often employ clustering algorithms for constructing convex combinations of computational units. These can take many iterations to converge or require space quadratic in the number of data points. Instead, in Section 4.?? we present linear alternatives based on random projections (Freund *et al.*, 2008; Dasgupta and Freund, 2008).

2 Probabilistic Circuits

As we briefly mentioned in the last chapter, Probabilistic Circuits (PCs) are conceptualized as computational graphs under special conditions. In this chapter, Section 2.1 to be more precise, we formally define PCs and give an intuition on their syntax, viewing other probabilistic models through the lenses of the PC framework. In Section 2.2, we describe the special structural constraints that give PCs their inference power over other generative models and state which queries (as far as we know) are enabled from each constraint.

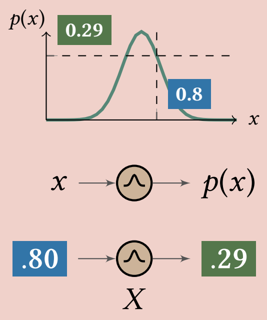
2.1 Distributions as Computational Graphs

Probabilistic circuits are directed acyclic graphs usually recursively defined in terms of their computational units. In its simplest form a PC is a single unit with no outgoing edges whose value corresponds to the result of a function. These are often called *input* nodes, and can take any form as long as its value is tractably computable. More concretely, input nodes typically represent probability density (or mass) functions, although they also support inputs as joint probability density functions of complex non-parametric models as well. To simplify notation, from here on out we shall use the term distribution and probability density (resp. mass) function interchangeably and argue that the input node represents a probability distribution.

Example 2.1.1: Gaussians as probabilistic circuits

Let L_p an input node and $p(X) = \mathcal{N}(X; \mu, \sigma^2)$ a univariate Gaussian distribution. Computing any query on L_p is straightforward: any query on L_p directly translates to p. As an example, suppose $\mu = 0$ and $\sigma^2 = 1$ and we wish to compute $\mathsf{L}_p(x = 0.8)$. The probability of this input shall then be

$$\mathsf{L}_p(x = 0.8) = \mathcal{N}(x = 0.8; \mu = 0, \sigma^2 = 1) = 0.29.$$



Let L_p a PC input node and denote p as its inherent probability distribution. By definition, any query $f : \mathcal{X} \to \mathcal{Y}$ which is tractable on p is tractable on L_p. We shall denote $\mathsf{L}_p(\mathbf{X}) = p(\mathbf{X})$, and often omit p when its explicit form is not needed. Evidently, a single input node lacks the expressivity for modeling complex models, otherwise we would have just used the input distribution as a standalone model. The expressiveness of PCs comes from recursively combining distributions into complex functions. This can be done through computational units that either compute convex combinations or products of their children. Let us first look at convex combinations, known in the literature as *sum* nodes.

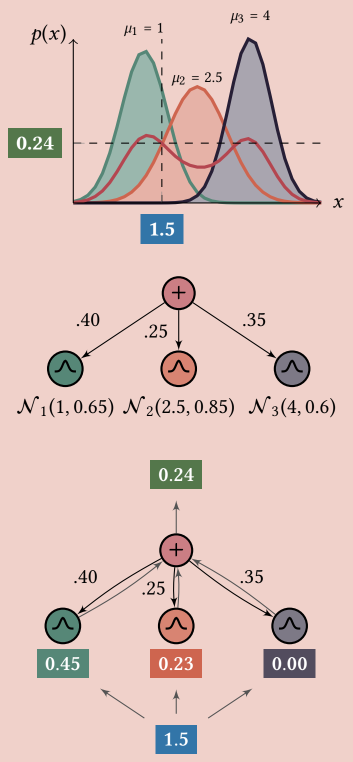
Let S be a PC sum node, and denote by $\mathrm{Ch}(\mathsf{S})$ the children nodes of S. For every edge $\overrightarrow{\mathsf{SC}}$ coming out of S and going to C, we attribute a weight $w_{\mathsf{S},\mathsf{C}} > 0$, such that $\sum_{\mathsf{C}\in\mathrm{Ch}(\mathsf{S})} w_{\mathsf{S},\mathsf{C}} = 1$. A sum node semantically defines a mixture model over its children, essentially acting as a latent variable over the component distributions (Poon and P. Domingos, 2011; Peharz, Gens, Pernkopf, *et al.*, 2016). Its value is the weighted sum of its children $p_{\mathsf{S}}(\mathbf{X} = \mathbf{x}) = \sum_{\mathsf{C}\in\mathrm{Ch}(\mathsf{S})} w_{\mathsf{S},\mathsf{C}} \cdot p_{\mathsf{C}}(\mathbf{X} = \mathbf{x})$. To simplify notation, we shall use $\mathsf{N}(\mathbf{X} = \mathbf{x})$ as an alias for $p_{\mathsf{N}}(\mathbf{X} = \mathbf{x})$, that is, the probability function given by N's induced distribution. We often omit the variable assignment $\mathbf{X} = \mathbf{x}$ to $\mathbf{x}$ if the notation is unambiguous.

Example 2.1.2: Gaussian mixture models as probabilistic circuits

A Gaussian Mixture Model (GMM) defines a mixture over Gaussian components. Say we wish to compute the probability of $X = x$ for a GMM $\mathcal{G}$ with three components $\mathcal{N}_1(\mu_1 = 1, \sigma_1^2 = 0.65)$, $\mathcal{N}_2(\mu_2 = 2.5, \sigma_1^2 = 0.85)$ and $\mathcal{N}_3(\mu_3 = 4, \sigma_3^2 = 0.6)$, and suppose we have weights set to $\phi = (0.4, 0.25, 0.35)$. Computing the probability of G amounts to the weighted summation

$$\begin{aligned}\mathcal{G}(X = x) =&0.4 \cdot \mathcal{N}_1(x; \mu_1, \sigma_2^2) + 0.25 \cdot \mathcal{N}_2(x; \mu_2, \sigma_2^2) + \\ &0.35 \cdot \mathcal{N}_3(x; \mu_3, \sigma_3^2),\end{aligned}$$

which is equivalent to a computational graph (i.e. a PC) with a sum node whose weights are set to ϕ and children are the components of the mixture. The figure on the right shows $\mathcal{G}$ (top) and its corresponding PC (middle). Given $x = 1.5$ (in blue), input nodes are computed following the inference flow (bottom, gray edges) up to the root sum node (in red), where a weighted summation is computed to output the probability (in green).

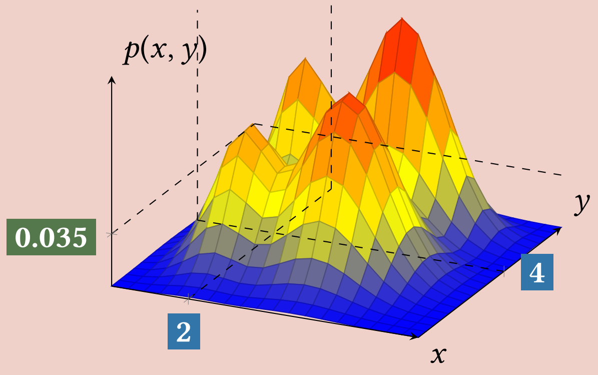


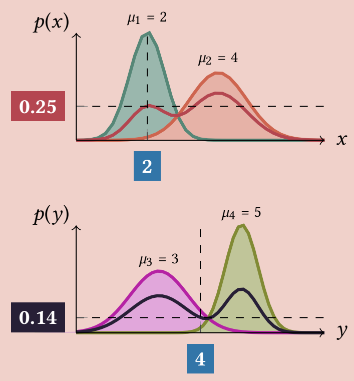
So far, the only nonlinearities present in PCs come from the internal computations of input nodes. In fact, a PC that only contains sums inputs can always be reduced to a sum node rooted PC with a single layer, i.e. a mixture model (see Theorem A.1.1). Adding *product nodes* as another form of nonlinearity increases expressivity sufficiently for PCs to be capable of representing any (discrete) probability distribution (DARWICHE, 2003; MARTENS and MEDABALIMI, 2014; PEHARZ, TSCHIATSCHEK, *et al.*, 2015). More importantly, products semantically act as factorizations of their children, indicating an independence relationship between variables from different children. In practice, product nodes are simply products of their childrens' distribution: if P is a PC product node, then its value is given by $\mathrm{P}(\mathbf{X}=\mathbf{x}) = \prod_{\mathrm{C}\in\mathrm{Ch}(\mathrm{P})} \mathrm{C}(\mathbf{X}=\mathbf{x})$.

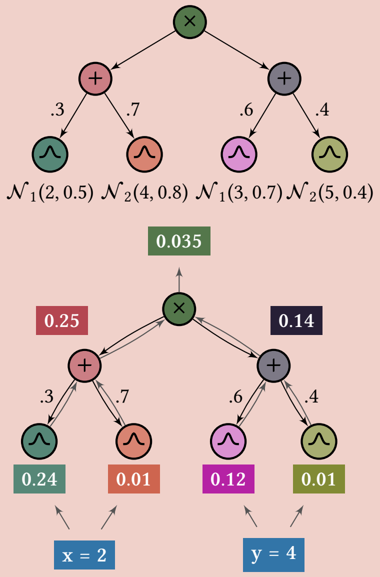
Example 2.1.3: Factors as probabilistic circuits

Say we have two GMMs $\mathcal{G}_1$ and $\mathcal{G}_2$. The first is a mixture model over variable X, with component weights $\phi_1 = (0.3, 0.7)$ and gaussians $\mathcal{N}_1(\mu_1 = 2, \sigma_1 = 0.5)$ and $\mathcal{N}_2(\mu_2 = 4, \sigma_2 = 0.8)$. The second is composed of $\mathcal{N}_3(\mu_3 = 3, \sigma_2 = 0.7)$ and $\mathcal{N}_4(\mu_4 = 5, \sigma_2 = 0.4)$, both distributions over variable Y and with weights $\phi_2 = (0.6, 0.4)$.

Suppose $X \perp\!\!\!\perp Y$, yet we wish to compute the joint probability of both x and y. If $X \perp\!\!\!\perp Y$, then $p(x, y) = p(x)p(y) = \mathcal{G}_1(x)\mathcal{G}_2(y)$, which corresponds to a factoring of mixtures. This is represented as a product node (in green) over the two mixture models (in red and purple). The resulting joint of this circuit is shown below.







.3
.7
.6
.4
$\mathcal{N}_1(2, 0.5)$ $\mathcal{N}_2(4, 0.8)$ $\mathcal{N}_1(3, 0.7)$ $\mathcal{N}_2(5, 0.4)$
0.035
0.25
0.14
0.24
0.01
0.12
0.01
x = 2
y = 4

Now that we have introduced the three most important computational units in PCs, we are finally ready to formally define probabilistic circuits.

Definition 2.1.1 (Probabilistic circuit)**.** *A probabilistic circuits C is a rooted connected DAG whose nodes compute any tractable operation of their children, usually either convex combinations, known as* sum *nodes, or* products*. Nodes with no outgoing edges, i.e.* input *nodes, are tractable nonnegative functions whose integrals exist and equal to one. Computing a value from C amounts to a bottom-up feedforward pass from input nodes to root.*

While we assume that *tractable* operation or function is acceptable, we are usually interested in $\mathcal{O}(1)$ time computable operations, and often assume the same of input functions to simplify analysis. Further, in this dissertation we are only interested in convex combinations and products, and as such only these operations are considered. When a probabilistic circuit $\mathcal{C}$ contains no consecutive sums or products (i.e. for every sum all of its children are either inputs or products and respectively for products) then it is said to be a *standard* form circuit. Any PC can be transformed into a *standardized* circuit, a process we call *standardization* (see Theorem A.1.2).

Remark 2.1.1: On operators and tractability

Throughout this work we consider only products and convex combinations (apart from the implicit operations contained within input nodes) as potential computational units. The question of whether any other operator could be used to gain expressivity without loss of tractability is without a doubt an interesting research question, and one that is actively being pursued. However, this is certainly out of the scope of this dissertation, and so we restrict discussion on this topic and only give a brief comment on operator tractability here, pointing to existing literature in this area of research.

A. Friesen and P. Domingos (2016) formalize the notion of replacing sums and products in PCs with any pair of operators in a commutative semiring, giving results on the conditions for marginalization to be tractable. They provide examples of common semirings and to which known formalisms they correspond to. One such example are PCs under the Boolean semiring $(\{0,1\}, \vee, \wedge, 0, 1)$ for logical inference, which are equivalent to Negation Normal Form (NNF, Barwise, 1982) and constitute an instance of Logic Circuits (LCs), of which Sentential Decision Diagrams (SDDs, Darwiche, 2011) and Binary Decision Diagrams (BDDs, Akers, 1978) are a part of. Another less common semiring in PCs is the real min-sum semiring $(\mathbb{R}_\infty, \min, +, \infty, 0)$ for nonconvex optimization (Abram L. Friesen and P. Domingos, 2015).

Recently, Vergari, Y. Choi, *et al.* (2021) extensively covered tractability conditions and complexity bounds for convex combinations, products, exp (and more generally powers in both naturals and reals), quotients and logarithms, even giving results for complex information-theoretic queries, such as entropies and divergences. Notably, they analyze whether structural constraints (and thus, in a sense, tractability) under these conditions are preserved.

Up to now, we have only considered summations as nonnegative weighted sums. Indeed, in most literature the sum node is defined as a convex combination. However, negative weights have appeared in Logistic Circuits (Liang and Van den Broeck, 2019) for discriminative modeling; and in Probabilistic Generating Circuits (Zhang *et al.*, 2021), a class of tractable probabilistic models that subsume PCs. Denis D. Mauá *et al.* (2017) extend (nonnegative) weights in sum nodes with probability intervals, effectively inducing a credal set (Fabio G. Cozman, 2000) for measuring imprecision.

Other works include PCs with quotients (Sharir and Shashua, 2018), transformations (Pevný *et al.*, 2020), max (Melibari, Poupart, and Doshi, 2016), and einsum (Peharz, Lang, *et al.*, 2020) operations.

Before we address the key components that make PCs interesting tractable probabilistic models, we must first discuss some important concepts that often come up in PC literature.

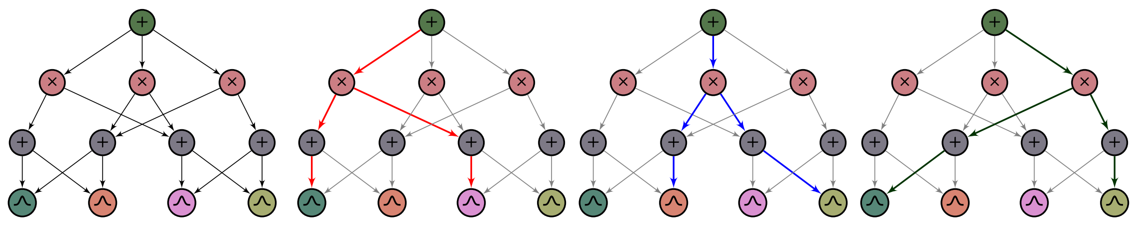


Figure 2.1: *A probabilistic circuit (a) and 3 of the 12 possible induced subcircuits (b).*

Mainly, we are interested in defining two notions here: the scope of a unit and induced subcircuits.

In simple terms, the scope of a computational unit N of a PC is merely the set of all variables that appear in the descendants of N. More formally, denote by $\mathrm{Sc}(\mathsf{N})$ the set of all variables that appear in N. We inductively compute the scope of circuit by a bottom-up approach: the scope of an input node L_p is the set of variables that appear in p's distribution[1], and the scope of any other node is the union of all of its childrens' scopes. The notion of scope is essential to the structural constraints seen in Section 2.2.

As an example, take the circuit from Example 2.1.3. The scope of input nodes ● and ● are Sc(●) = Sc(●) = $\{X\}$, while Sc(●) = Sc(●) = $\{Y\}$. Consequentially, their parent sum nodes will have the same scope as their children Sc(⊕) = $\{X\}$ and Sc(⊕) = $\{Y\}$, yet the root node's scope is Sc(⊗) = $\{X, Y\}$, since its childrens' scopes are distinct. The size of a probabilistic circuit is the number of nodes and edges of its computational graph. We use $|\mathcal{C}|$ to denote the size of a probabilistic circuit $\mathcal{C}$.

Let $\mathcal{C}$ a probabilistic circuit and node $\mathsf{N} \in \mathcal{C}$. We say that $\mathcal{S}_\mathsf{N}$ is a subcircuit of $\mathcal{C}$ rooted at N if $\mathcal{S}_\mathsf{N}$'s root is N, all nodes and edges in $\mathcal{S}_\mathsf{N}$ are also in $\mathcal{C}$ and $\mathcal{S}$ is also a probabilistic circuit. We now introduce the concept of induced subcircuits (Chan and Darwiche, 2006; Dennis and Ventura, 2015; Peharz, Gens, and P. Domingos, 2014).

Definition 2.1.2 (Induced subcircuit)**.** *Let $\mathcal{C}$ a probabilistic circuit. An induced subcircuit $\mathcal{S}$ of $\mathcal{C}$ is a subcircuit of $\mathcal{C}$ rooted at $\mathcal{C}$'s root such that all edges coming out of product nodes in $\mathcal{C}$ are also in $\mathcal{S}$, and of all edges coming out of sum nodes in $\mathcal{C}$, only one is in $\mathcal{S}$.*

Examples of induced subcircuits are visualized in Figure 2.1. When the induced subcircuit is a tree, as is the case in Figure 2.1, they are referred to as induced tree (H. Zhao, Melibari, *et al.*, 2015; H. Zhao, Adel, *et al.*, 2016).

So far, by Definition 2.1.1 a PC does not yet necessarily represent a probability distribution with tractable (marginal) inference. In the next section, we formally define sufficient conditions for tractability in the form of structural constraints that we have only previously mentioned in passing.

[1] Although we previously defined input nodes as mere functions, here we are explicitly associating a random variable to a *probability* function. Indeed, if we are being rigorous, we should define input nodes as a pair of random variable and function. To save space we instead assume, as previously stated, that the function is seen as both the probability (density) function as well as the distribution itself, and thus its scope is the scope of its distribution, i.e. the random variables that come into play in a probability distribution.

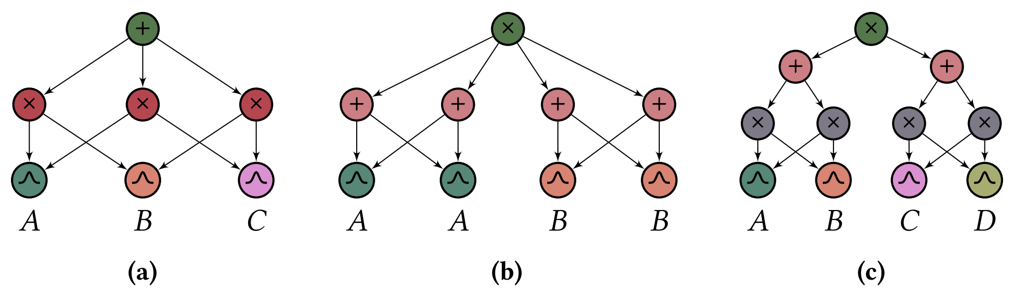


Figure 2.2: *Decomposable but non-smooth (a), smooth but non-decomposable (b), and smooth and decomposable (c) circuits.*

2.2 Deciding What to Constraint

In this section we are interested in studying how the structural constraints in PCs enable different inference tasks. We shall first cover the more basic queries, namely *probability of evidence* (EVI), *marginal probability* (MAR), *conditional probability* (CON) and *maximum a posteriori probability* (MAP). After that we briefly address more complex queries such as mutual information, entropies and *expectation* (EXP).

2.2.1 Basic Queries

The most basic inference task we are interested in computing is the probability of evidence. To unlock this, we must introduce two structural constraints known as *smoothness* and *decomposability*.

Definition 2.2.1 (Smoothness). *A probabilistic circuit $\mathcal{C}$ is said to be* smooth *if for every sum node S in $\mathcal{C}$, $\mathrm{Sc}(\mathcal{C}_1) = \mathrm{Sc}(\mathcal{C}_2)$ for $\mathcal{C}_1, \mathcal{C}_2 \in \mathrm{Ch}(S)$.*

Definition 2.2.2 (Decomposability). *A probabilistic circuit $\mathcal{C}$ is said to be* decomposable *if for every product node P in $\mathcal{C}$, $\mathrm{Sc}(\mathcal{C}_1) \cap \mathrm{Sc}(\mathcal{C}_2) = \emptyset$ for $\mathcal{C}_1, \mathcal{C}_2 \in \mathrm{Ch}(P)$.*

When a PC is *decomposable*, all of its induced subcircuits are induced trees. For any *smooth* and *decomposable* PC, computing EVI is done in linear time in the number of edges. In fact this is true for MAR and CON as well. To compute marginals, it is sufficient to compute the corresponding marginals with respect to each input node and proceed to propagate values bottom-up. For conditionals, we simply compute two passes: one where we marginalize the conditional variables and the other any other variables that are not present in our query. These procedures are formalized in the theorem below.

Theorem 2.2.1 (Poon and P. Domingos, 2011; Y. Choi, Vergari, and Van den Broeck, 2020; Vergari, Y. Choi, *et al.*, 2021). *Let $\mathcal{C}$ a* smooth *and* decomposable *PC. Any one of EVI, MAR or CON can be computed in linear time (in the number of edges of $\mathcal{C}$).*

Importantly, EVI and CON are both special cases of MAR in the sense that they are marginalizations over different intervals (see page 43). More specifically, EVI is a marginalization over an empty set and CON is simply the quotient between two marginalization passes. Algorithmically, this means that the only distinction between these three queries is on what to do on the input nodes. We say that a variable assignment $\mathbf{x}$ for circuit $\mathcal{C}$ is *complete* if for every variable $X \in \mathrm{Sc}(\mathcal{C})$, $\mathbf{x}$ assigns a value to X; otherwise it is said to be a

Algorithm 1 EVI

Input A probabilistic circuit $\mathcal{C}$ and complete assignment $\mathbf{x}$
Output Probability $\mathcal{C}(\mathbf{x})$

```
1: Let $v$ a hash function mapping a node to its probability
2: for each N in reverse topological order do
3:     if N is an input then $v_\mathsf{N} \leftarrow \mathsf{N}(\mathbf{x})$
4:     else if N is a sum then $v_\mathsf{N} \leftarrow \sum_{\mathsf{C}\in\mathrm{Ch}(\mathsf{N})} w_{\mathsf{N},\mathsf{C}} v_\mathsf{C}$
5:     else if N is a product then $v_\mathsf{N} \leftarrow \prod_{\mathsf{C}\in\mathrm{Ch}(\mathsf{N})} v_\mathsf{C}$
6: return $v_R$, where R is $\mathcal{C}$'s root
```

Algorithm 2 MAR

Input A probabilistic circuit $\mathcal{C}$ and partial assignment $\mathbf{x}$
Output Probability $\int \mathcal{C}(\mathbf{x}, \mathrm{y})\, \mathrm{dy}$

```
1: Let $v$ a hash function mapping a node to its probability
2: for each N in reverse topological order do
3:     if N is an input then $v_\mathsf{N} \leftarrow \int \mathsf{N}(\mathbf{x}, \mathrm{y})\, \mathrm{dy}$
4:     else if N is a sum then $v_\mathsf{N} \leftarrow \sum_{\mathsf{C}\in\mathrm{Ch}(\mathsf{N})} w_{\mathsf{N},\mathsf{C}} v_\mathsf{C}$
5:     else if N is a product then $v_\mathsf{N} \leftarrow \prod_{\mathsf{C}\in\mathrm{Ch}(\mathsf{N})} v_\mathsf{C}$
6: return $v_R$, where R is $\mathcal{C}$'s root
```

partial assignment. Algorithm 1 and Algorithm 2 show the exact algorithmic procedures to extract EVI and MAR queries from $\mathcal{C}$. For CON, it suffices to run two MARs.

Because EVI, MAR and CON are the most basic forms of querying in PCs, we shall refer to these as *base queries*. Although smoothness and decomposability are sufficient conditions for tractable base queries, they are not necessary conditions. As a matter of fact, decomposability can be replaced with a third weaker constraint known as *consistency*. Denote by Desc(N) the set of all descendants of N.

Definition 2.2.3 (Consistency). *A probabilistic circuit $\mathcal{C}$ is said to be* consistent *if for any product node P in $\mathcal{C}$, it holds that for every two children $C_1, C_2 \in \mathrm{Ch}(P)$ there does not exist two input nodes $L_p^1 \in \mathrm{Desc}(C_1)$ and $L_q^2 \in \mathrm{Desc}(C_2)$ whose scope is the same and $p(\mathbf{x}) \neq q(\mathbf{x})$ for any $\mathbf{x} \in \boldsymbol{\mathcal{X}}$.*

In Peharz, Tschiatschek, *et al.* (2015), the authors show that although smooth and consistent PCs are more succinct (Darwiche and Marquis, 2002) compared to smooth and decomposable circuits, the gain is only mild, further proving that any smooth and consistent PC can be polynomially translated to a decomposable equivalent. In practice, because constructing decomposable circuits (and verifying decomposability) is much easier compared to doing the same with consistency, we instead focus on smooth and decomposable PCs.

Suppose we wish to compute the most probable assignment of a variable, say for classification or image reconstruction. To do so, we must compute the conditional query

$$\max_{\mathrm{y}} p(\mathrm{y}|\mathbf{x}) = \max_{\mathrm{y}} \frac{p(\mathrm{y}, \mathbf{x})}{p(\mathbf{x})} = \frac{\max_{\mathrm{y}} p(\mathrm{y}, \mathbf{x})}{p(\mathbf{x})}, \tag{2.1}$$

Algorithm 3 MAP

```
Input  A probabilistic circuit $\mathcal{C}$ and evidence assignment $\mathbf{x}$
Output  Probability $\max_{\mathbf{y}} \mathcal{C}(\mathbf{y}|\mathbf{x})$
1: Let $v$ a hash function mapping a node to its probability
2: for each $\mathsf{N}$ in reverse topological order do
3:     if $\mathsf{N}$ is an input then $v_{\mathsf{N}} \longleftarrow \max_{\mathbf{y}} \mathsf{N}(\mathbf{y},\mathbf{x})$
4:     else if $\mathsf{N}$ is a sum then $v_{\mathsf{N}} \longleftarrow \max_{\mathsf{C}\in\mathsf{Ch}(\mathsf{N})} w_{\mathsf{N},\mathsf{C}} v_{\mathsf{C}}$
5:     else if $\mathsf{N}$ is a product then $v_{\mathsf{N}} \longleftarrow \prod_{\mathsf{C}\in\mathsf{Ch}(\mathsf{N})} v_{\mathsf{C}}$
6: return $v_R/\mathcal{C}(\mathbf{x})$, where $\mathsf{R}$ is $\mathcal{C}$'s root
```

Algorithm 4 ARGMAP

```
Input  A probabilistic circuit $\mathcal{C}$ and evidence assignment $\mathbf{x}$
Output  The most probable state $\arg\max_{\mathbf{y}} \mathcal{C}(\mathbf{y}|\mathbf{x})$
1: Compute $\max_{\mathbf{y}} \mathcal{C}(\mathbf{y}|\mathbf{x})$ and store values in $v$
2: $\mathbf{N} \longleftarrow \textsc{MaxInducedTree}(\mathcal{C}, v)$
3: Call $\mathbf{y}$ the set of initially empty arg max states
4: for each input node $\mathsf{L} \in \mathbf{N}$ do
5:     $\mathbf{y} \longleftarrow \mathbf{y} \cup \arg\max_{\mathbf{z}} \mathsf{N}(\mathbf{z},\mathbf{x})$
6: return $\mathbf{y}$
```

often called the *maximum a posteriori* (MAP) probability. For this dissertation, we shall only consider the case of full MAP, as computing partial MAP is provably hard in most PCs (Peharz, Gens, Pernkopf, *et al.*, 2016; De Campos, 2011). Unless explicitly stated, MAP shall mean full MAP, i.e. when $\mathbf{x} \cup \mathbf{y}$ forms a complete assignment of the scope. Full MAP also goes by the name of *most probable explanation* (MPE) in literature. Although at first it seems like MAP is no harder than computing a CON, it turns out that for smooth and decomposable PCs, MAP is unfortunately intractable (Conaty *et al.*, 2017; Mei *et al.*, 2018). To unlock access to MAP we must make the circuit *deterministic.*

Definition 2.2.4 (Determinism). *A probabilistic circuit $\mathcal{C}$ is said to be* deterministic *if for every sum node $\mathsf{S} \in \mathcal{C}$ only one child of S has nonnegative value at a time for any complete assignment.*

At this point, we must introduce a graphical notation for *indicator* nodes. An indicator node is an input node whose function is the characteristic function $f(x) = [\![x = k]\!]$, i.e. a degenerate function with all of its mass on k and zero anywhere else. A special case is when X is binary and $k = 1$, in which case we say the input node is a literal node, denoting by the usual propositional notation X for when $k = 1$ and $\neg X$ for $k = 0$. Graphically, we shall use ◎ for indicators and the textual propositional notation for literals.

With determinism, we now have access to MAP.

Theorem 2.2.2 (Peharz, Gens, Pernkopf, *et al.*, 2016)**.** *Let $\mathcal{C}$ a smooth, decomposable and* deterministic *PC. MAP is computable in linear time (in the number of edges of $\mathcal{C}$).*

When a PC is smooth, decomposable and deterministic, the MAP is easily computable by simply replacing sum nodes with a max operation and performing a feedforward

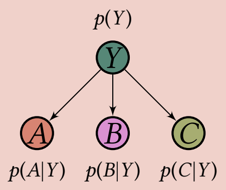
EVI pass. This is commonly called the Max-Product algorithm, shown more formally in Algorithm 3. To find the states that maximize Equation (2.1) in a given circuit $\mathcal{C}$, we first compute the MAP probabilities through the usual bottom-up pass, and then find the maximum (in terms of probability) induced tree $\mathcal{M}$ rooted at $\mathcal{C}$. This maximum induced tree can be retrieved by a top-down pass selecting the most probable sum child nodes according to the probabilities set by MAP. Since $\mathcal{C}$ is decomposable, there cannot exist a node in $\mathcal{M}$ with more than one parent, meaning it is by construction a tree whose leaves are input nodes with scopes whose union is the scope of $\mathcal{C}$. This reduces the problem to a divide-and-conquer approach where each input node is individually maximized (see Algorithm 4 and page 44 for a formal proof on its validity).

Example 2.2.1: Naïve Bayes as probabilistic circuits

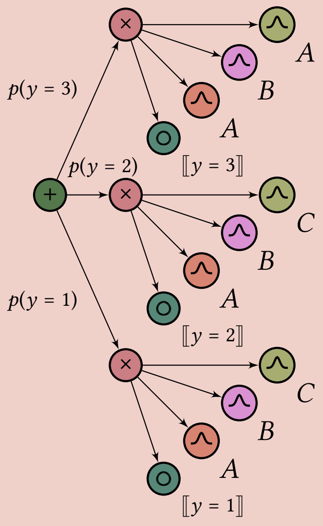
Suppose we have samples of per capita census measurements on three different features, say age A, body mass index B and average amount of cheese consumed daily C from three different cities Y. Assuming A, B and C are independent, given a sample $x = (a, b, c)$ we can use Gaussian naïve Bayes to predict x's class

$$p(y|a, b, c) = p(y)p(a|y)p(b|y)p(c|y).$$

In PC terms, $p(y)$ are the prior probabilities, i.e. sum weights, for each class and $p(z|y)$ are Gaussian input nodes corresponding to the distributions of each feature in each city. To make sure that these are in fact conditional distributions, we introduce indicator variables "selecting" Y's state. Since the PC resulting is deterministic, we can compute the MAP for classification in linear time by simply replacing the root node with a max, which is exactly equivalent to finding the highest likelihood of x for each city y.



Gaussian naïve Bayes



Equivalent PC

2.2.2 Complex Queries

Although base queries cover most of the needs of the usual data scientist, more complex tasks involving information-theoretic measures, logical queries or distributional divergences are also (tractably) computable under the right conditions. Particularly, we are interested in a key component for tractability in all these tasks: the notion of *vtrees* and *structure decomposability*, a stronger variant of decomposability where variable partitionings on product nodes follow a hierarchy. This hierarchy is easily visualized through a *vtree* (variable tree), a data structure that defines a (partial) ordering of variables.

Definition 2.2.5 (Vtree). *A variable tree $\mathcal{V}$, or* vtree, *over a set of variables* $\mathbf{X}$ *is a binary*

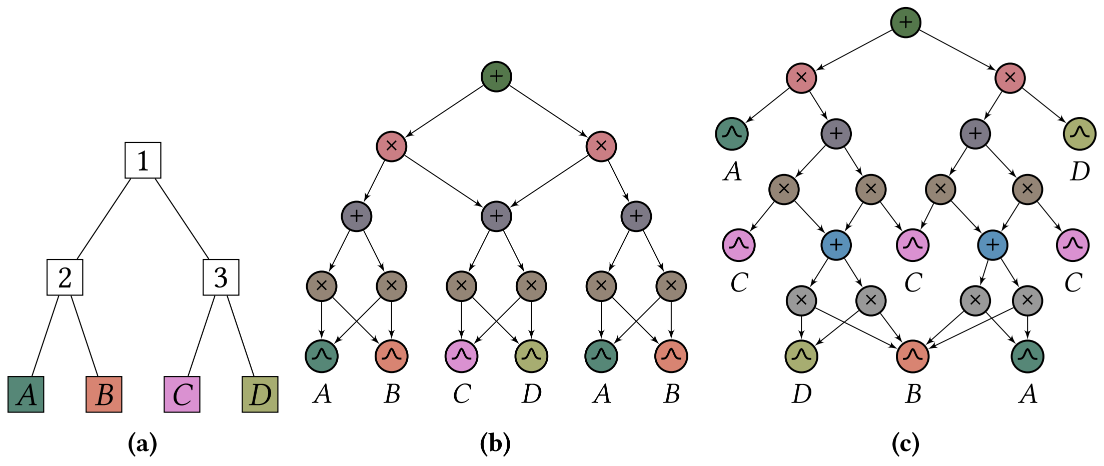


Figure 2.3: *A vtree (a) defining an order (A, B, C, D), a 2-standard structure decomposable probabilistic circuit that respects the vtree (b), and a 2-standard decomposable probabilistic circuit that does not (c).*

tree whose leaf nodes have a one-to-one and onto mapping $\phi_{\mathcal{V}}$ with $\mathbf{X}$.

We shall adopt the same scope definition and notation $\text{Sc}(\cdot)$ for vtrees as in PCs. Let v a vtree node from a vtree $\mathcal{V}$. If v is a leaf node, its scope is $\phi_{\mathcal{V}}(v)$, i.e. the leaf's assigned variable; otherwise its scope is the union of the scope of its children. For an inner node v, we shall call its left child $v^{\leftarrow}$ and right child $v^{\rightarrow}$. Every inner node v of a vtree $\mathcal{V}$ defines a variable *partitioning* of the scope $(\text{Sc}(v^{\leftarrow}), \text{Sc}(v^{\rightarrow}))$, while the leaves of $\mathcal{V}$ define a partial ordering of $\text{Sc}(\mathcal{V})$. We are especially interested in the scope partitioning aspect of vtrees.

Definition 2.2.6. *A product node P* respects *a vtree node v if P contains only two children* $\text{Ch}(P) = \{C_1, C_2\}$*, and* $\text{Sc}(C_1) = \text{Sc}(v^{\leftarrow})$ *and* $\text{Sc}(C_2) = \text{Sc}(v^{\rightarrow})$.

Obviously, the above definition is vague with regards to which child (i.e. graphically, left or right) of P should respect the scope of which v child. We therefore assume a fixed order for P's children and say that the (graphically) left child is called the *prime* and (graphically) right child the *sub*, and refer to P as an *element*. This ultimately means that the scope of the prime (resp. sub) of P must be the same as the scope of the left (resp. right) child of v. Although the graphical concept of left and right is needed for easily visualizing the scope partitioning of a product with respect to a vtree node, we do not use it strictly. In fact, when the situation is unambiguous, we compactly represent the computational graph without adhering to the left-right convention in favor of readability.

We say that a vtree is linear, if either it is left-linear or right-linear. A left- (resp. right) linear vtree is a vtree whose inner nodes all have leaf nodes on their right (resp. left) child. Similarly, a vtree is said to be left- (resp. right) leaning if the number of leaf nodes as right (resp. left) children is much higher then left (resp. right) children. Otherwise, it is a balanced vtree. The variable order of a vtree is the sequence of leaf nodes (i.e. variables) read from left to right. Figure 2.3a shows a balanced vtree with order (A, B, C, D).

Now that we understand what a vtree is, we can properly introduce *structure decomposability*, a stronger variant of decomposability. We say that a PC is 2-standard if it is standard and all of its product nodes have exactly two children.

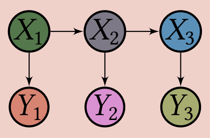
Definition 2.2.7 (Structure decomposability)**.** *Let $\mathcal{C}$ a 2-standard probabilistic circuit and $\mathcal{V}$ a vtree with same scope as $\mathcal{C}$. $\mathcal{C}$ is said to be* structure decomposable *if every i-th product layer of $\mathcal{C}$ respects every i-th inner node layer of $\mathcal{V}$.*

Example 2.2.2: Hidden Markov models as probabilistic circuits

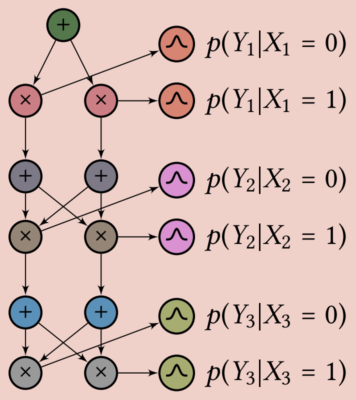
Say we wish to model a temporal structured dependence between three latent binary variables, for example the presence of a subject X_1, verb X_2 and object X_3 in a natural language phrase. Each observation Y_i is a fragment (of X_i) taken from a complete sentence $\mathbf{y} = (y_1, y_2, y_3)$. The first-order Hidden Markov Model (HMM) (on the right) models the joint probability of sentences

$$p(X_{1..3}, Y_{1..3}) = p(X_1) \prod_{i=2}^{3} p(X_i|X_{i-1}) \prod_{i=1}^{3} p(Y_i|X_i).$$

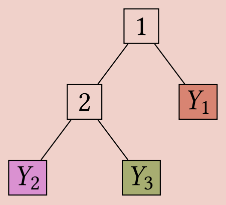
This is computationally equivalent to the PC on the right. Each input node $p(Y_i|X_i)$ is a conditional distribution model (possibly another PC) for each assignment (here two) of X_i, meaning that if $p(Y_i|X_i = 0) > 0$, then $p(Y_i|X_i = 1) = 0$ and vice-versa. Root weights are exactly $p(X_1)$, and each $p(X_i|X_{i-1})$ translates into the other matched color sum weights. Further, every product (except for the redundant ⊗ nodes) follows the partitionings imposed by the vtree. This means that this PC is not only smooth, but structure decomposable and deterministic.



Hidden Markov Model



Equivalent PC



Vtree

Although we assume a *2-standard* PC in Definition 2.2.7, its assumption was only for convenience, and does not imply in a loss of expressivity. As a matter of fact, any PC can be 2-standardized, i.e. to standardize the circuit such that every product node only has two children (see Theorem A.1.3). Intuitively, structure decomposability merely states that every two product nodes whose scopes are the same must partition their scopes (between their two children) exactly the same (and according to their corresponding vtree node). Semantically speaking, a vtree's inner node v defines an independence relationship between $\mathrm{Sc}(v^{\leftarrow})$ and $\mathrm{Sc}(v^{\rightarrow})$ under the distribution encoded by its PC.

Figure 2.3 shows a vtree $\mathcal{V}$ and two probabilistic circuits, say $\mathcal{C}_1$ for the one in the middle and $\mathcal{C}_2$ for the one on the right. Notice how $\mathcal{C}_1$ respects $\mathcal{V}$, as each ⊗ respects the split at vtree node 1 (namely $\{A, B\}$). The primes are then ⊕ whose scopes are $\{A, B\}$, while the sub is the one with two parents and scope $\{C, D\}$. For each of these, their children ⊗ also respect $\mathcal{V}$: they either encode he same split as 2 or as 3, depending on whether they are descendants from the sub or prime of 1. Although $\mathcal{C}_2$ is decomposable, it does *not* respect $\mathcal{V}$, as ⊗ encode different variable partionings $((\{A\}, \{B, C, D\})$ and $(\{A, B, C\}, \{D\}))$. In fact, it is not structure decomposable, as it does not respect any vtree.

Despite our structure decomposability definition relying on a vtree, there is at least one alternative definition that defines it in terms of *circuit compatibility*. Essentially, a circuit $\mathcal{C}_1$ is *compatible* with $\mathcal{C}_2$ if they can be 2-standardized and rearranged such that any two products with same scope, one from $\mathcal{C}_1$ and the other $\mathcal{C}_2$, partition the scope the same way (Vergari, Y. Choi, *et al.*, 2021). A structure decomposable PC is then defined as a PC that is compatible with a copy of itself. In summary, the two definitions of structure decomposability are equivalent, except compatibility implicitly assumes an arrangement of product scopes that is analogous to a vtree.

The notion of a vtree (or compatibility for that matter) is key to more complex queries. For instance, given two probabilistic circuits $\mathcal{C}_1$ and $\mathcal{C}_2$, computing cross entropy between the two is $\mathcal{O}(|\mathcal{C}_1||\mathcal{C}_2|)$ as long as both have the same vtree and at least the circuit that needs to be log-computable is also deterministic. Likewise, computing the Kullback-Leibler (KL) divergence between $\mathcal{C}_1$ and $\mathcal{C}_2$ requires that the two share the same vtree and both be deterministic. Mutual Information (MI), in turn, calls for the circuit to be smooth, structure decomposable and an even stronger version of determinism known as *marginal determinism* where sums can only have one nonnegative valued child for any *partial* assignment at a time. In fact, when a PC is smooth, decomposable and marginal deterministic, marginal MAP, i.e. MAP over partial assignments becomes linear time computable. For a more detailed insight on the tractability of these (and other) queries, as well as proofs on these results, we point to the comprehensive study of Vergari, Y. Choi, *et al.* (2021).

A particularly interesting class of queries that becomes tractable when circuits are structure decomposable is expectation (EXP). One notable example from this class is computing the probability of logical events. This leads us to logic circuits, a parallel version of probabilistic circuits for logical reasoning.

2.3 Probabilistic Circuits as Knowledge Bases

We superficially mentioned in Remark 2.1.1 that PCs under a Boolean semiring with conjunctions and disjunctions as operators are known as Logic Circuits (LCs). In this section, we formally yet briefly define LCs and more precisely show the connection between PCs and LCs.

2.3.1 From Certainty...

Logic circuits are computational graphs just like PCs, but whose input are always Booleans (and as such the scope is over propositional variables) and computational units define either a conjunction, disjunction or literal of their inputs. While the computational graph in PCs encodes uncertainty as a probability distribution, in LCs their computational graph encodes certain knowledge as a propositional language. Similar to PCs, computing the satisfiability of an assignment is done by a bottom-up feedforward evaluation of the circuit. In terms of notation, we shall use ⌂ for disjunction nodes, ◻ for conjunction nodes, and X and $\neg X$ for literal nodes.

Definition 2.3.1 (Logic circuit). *A logic circuit $\mathcal{L}$ is a rooted connected DAG whose nodes compute either a conjunction or a disjunction of their children. Nodes with no outgoing*

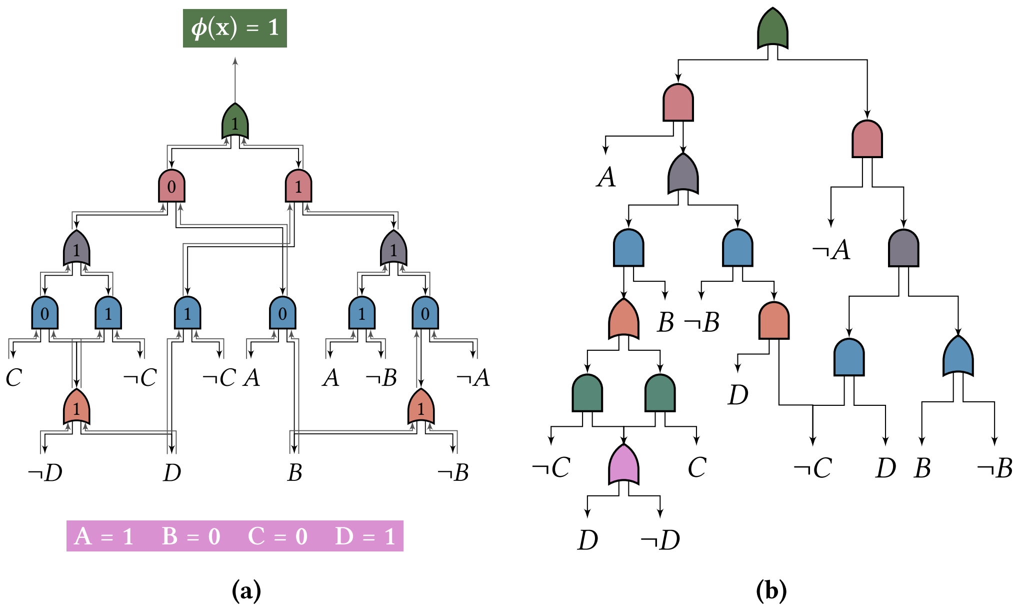


Figure 2.4: *Two smooth, structure decomposable and deterministic logic circuits encoding the same logic constraint $\phi \equiv (A \wedge B) \vee (\neg C \wedge D)$ for a balanced (a) and a right-linear (b) vtree. In (a), a circuit evaluation for an assignment, with each node value in the bottom-up evaluation pass shown inside nodes.*

edges, i.e. literal *nodes, are indicator functions of either a positive (true) or negative (false) assignment. Computing the satisfiability of $\mathcal{L}$ amounts to a bottom-up feedforward pass from literal nodes to root.*

Evidently, logic circuits are closely related to probabilistic circuits. The strikingly similar definitions are not coincidence: much of the literature on PCs have their origins on LCs. In fact, most structural constraints in PCs are the exact same (up to even their names[2]) as their LC analogues. In this dissertation, we are particularly interested in the specific subset of smooth, structure decomposable and deterministic LCs, known as Sentential Decision Diagrams (SDDs, Darwiche, 2011). Figure 2.4 shows two SDDs encoding the same knowledge base but under different vtrees. The one on the left respects a balanced vtree and the one on the right a right-linear vtree.

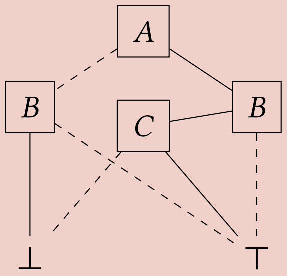
Logic circuits have appeared in literature under the name of Negated Normal Form (NNF) (Darwiche, 2001; Darwiche, 1999), a superset of other propositional compilation languages such as Binary Decision Diagrams (BDDs, Bryant, 1986), Propositional DAGs (PDAGs, Wachter and Haenni, 2006), Sentential Decision Diagrams (SDDs, Darwiche, 2011), DNFs and CNFs. Although no structural constraint is required for correctly evaluating satistifiability in logic circuits, the same properties defined in past sections have come up in LCs to enable other more complex queries like equivalence, implicates, sentential entailment and model counting, as well as transformations such as closed conditionings, forgetting, conjunctions and disjunctions (Darwiche and Marquis, 2002). The succinctness (i.e. expressive efficiency) of LCs are also impacted by these structural restrictions

[2] There are some exceptions. Smoothness is sometimes referred to as *completeness* in PCs, while determinism has the alternative name of *selectivity*.

(Gogic *et al.*, 1995; Papadimitriou, 1994; Darwiche and Marquis, 2002). See Darwiche and Marquis (2002) and Darwiche (2020) for more on logic circuits.

Example 2.3.1: BDDs as logic circuits

A Binary Decision Diagram (BDD, Bryant, 1986) defines a Boolean function over binary variables as a rooted DAG. BDDs are a subset of smooth, structure decomposable and deterministic LCs (i.e. SDDs) whose vtrees are always right-linear.



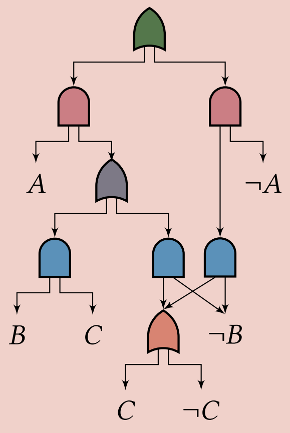
In their usual notation, inner nodes are variables and leaves are constants $\bot$ and $\top$. Evaluating an assignment $\mathbf{x} \subseteq \{0,1\}^n$ is equivalent to a path from the root to a leaf where each variable X determines a decision to go through the dashed line when $x = 0$ or solid line when $x = 1$. If the path ends at a $\top$, the function returns 1, otherwise it must end at a $\bot$ and therefore returns 0. The BDD above encodes the following logic formula

$$\phi(A, B, C) = (A \lor \neg B) \land (\neg B \lor C),$$

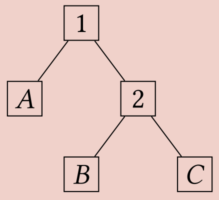
also shown as a truth table on the right, together with a logic circuit that encodes the same truth table and its vtree.

A	B	C	$\phi(\mathbf{x})$
0	0	0	1
1	0	0	1
0	1	0	0
1	1	0	0
0	0	1	1
1	0	1	1
0	1	1	0
1	1	1	1

Truth Table



Equivalent LC



Vtree

2.3.2 ...to Uncertainty

Logic circuits are easily extensible to probabilistic circuits. In fact, if we think of an LC as the support of a PC the connections between the two come naturally. Suppose a 2-standard smooth, structure decomposable and deterministic probabilistic circuit $\mathcal{C}$ over binary variables. We can construct an identically structured logic circuit (up to input nodes) $\mathcal{L}$ with same vtree as $\mathcal{C}$ whose underlying Boolean function encodes $\phi(\mathbf{x}) = [\![\mathcal{C}(\mathbf{x}) > 0]\!]$. Since sums act exactly like disjunctions and products like conjunctions under the Boolean semiring, products in $\mathcal{L}$ are replaced with conjunctions in $\mathcal{L}$, and sums with disjunctions. Input nodes from $\mathcal{C}$ are replaced with a literal node if the function is degenerate, or with a disjunction over positive and negative literals otherwise. This makes sure $\mathcal{L}$ acts as the support of $\mathcal{C}$, as each disjunction node $\mathsf{S}_{\mathcal{L}}$ of $\mathcal{L}$ defines

$$\mathsf{S}_{\mathcal{L}}(\mathbf{x}) = \bigvee_{\mathsf{C}\in\mathrm{Ch}_{\mathcal{C}}(\mathsf{S}_{\mathcal{L}})} [\![\mathsf{C}_p(\mathbf{x}) > 0]\!] \wedge [\![\mathsf{C}_s(\mathbf{x}) > 0]\!],$$

where $\mathrm{Ch}_{\mathcal{C}}(\mathsf{S})$ retrieves the children of S's corresponding sum node in $\mathcal{C}$, with $\mathsf{C}_p(\mathbf{x})$ and $\mathsf{C}_s(\mathbf{x})$ the probabilities of C's prime and sub respectively. The corresponding sum node $\mathsf{S}_{\mathcal{C}}$ in $\mathcal{C}$ then only attributes a weight (i.e. probability) to each positive element as usual

$$\mathsf{S}_{\mathcal{C}}(\mathbf{x}) = \sum_{\mathsf{C}\in\mathrm{Ch}(\mathsf{S}_{\mathcal{C}})} w_{\mathsf{S}_{\mathcal{C}},\mathsf{C}} \cdot \mathsf{C}_p(\mathbf{x}) \cdot \mathsf{C}_s(\mathbf{x}).$$

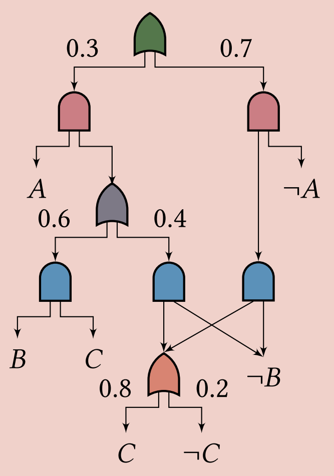
Example 2.3.2: Embedding certain knowledge in probabilistic circuits

Recall the logic circuit $\mathcal{L}$ from Example 2.3.1. Imagine we wish to model the uncertainty coming from all assignments where $\mathcal{L}(\mathbf{x}) = 1$. In other words, we want to assign a positive probability to all true entries in the previous example's truth table, turning it into a probability table. The table on the right shows the chosen probabilities for each instance. Naturally, they all sum to one, with logically impossible assignments set to zero.

Compiling an LC into a PC is straightforward: replace conjunctions with product nodes and disjunctions with sum nodes. Input nodes are left untouched, as literal nodes are just degenerate probability distributions. Sum weights are what ultimately define the probabilities in the probability table. The PC on the right is the result of the compilation of $\mathcal{L}$ into a probabilistic circuit whose distribution is defined by the probability table above it. When we mean to say that a PC has its support defined by its underlying LC, then we use the logic gate notation with the added weights on edges coming out from nodes.

A	B	C	$\phi(\mathbf{x})$	$p(\mathbf{x})$
0	0	0	1	0.140
1	0	0	1	0.024
0	1	0	0	0.000
1	1	0	0	0.000
0	0	1	1	0.560
1	0	1	1	0.096
0	1	1	0	0.000
1	1	1	1	0.180

Probability Table



Probabilistic Circuit

This compatibility between logic and probabilistic circuits allows certain knowledge to be embedded into an uncertain model by constructing a computational graph whose underlying logic circuit correctly attributes positive values only to the support of the distribution. When such a computational graph is also smooth, structure decomposable and deterministic, then it belongs to a special subclass of PCs called Probabilistic Sentential Decision Diagrams (PSDDs, Kisa *et al.*, 2014). An alternative use case for logic circuits within the context of probabilistic reasoning is querying for the probability of logical events, i.e. the expectation of a logic query with respect to a distribution. This kind of

Algorithm 5 EXP

```
Input  A smooth, structure decomposable PC $\mathcal{C}$ and LC $\mathcal{L}$ both with vtree $\mathcal{V}$
Output  The expectation $\mathbb{E}_{\mathcal{C}}[\mathcal{L}] = \int \mathcal{C}(\mathbf{x})\mathcal{L}(\mathbf{x})\,d\mathbf{x}$
1: Let $v$ a hash function mapping a pair of nodes to an expectation
2: for each pair of nodes $(\mathsf{N}_{\mathcal{C}}, \mathsf{N}_{\mathcal{L}})$ in reverse topological order do
3:     if $\mathsf{N}_{\mathcal{C}}$ is an input then $v_{\mathsf{N}_{\mathcal{C}},\mathsf{N}_{\mathcal{L}}} \leftarrow \mathbb{E}_{\mathsf{N}_{\mathcal{C}}}[\mathsf{N}_{\mathcal{L}}]$
4:     else if $\mathsf{N}_{\mathcal{C}}$ is a product then $v_{\mathsf{N}_{\mathcal{C}},\mathsf{N}_{\mathcal{L}}} \leftarrow \prod_{\mathsf{C}\in\mathrm{Ch}(\mathsf{N}_{\mathcal{C}})} v_{\mathsf{C}_{\mathcal{C}},\mathsf{C}_{\mathcal{L}}}$
5:     else if $\mathsf{N}_{\mathcal{C}}$ is a sum then $v_{\mathsf{N}_{\mathcal{C}},\mathsf{N}_{\mathcal{L}}} \leftarrow \sum_{\mathsf{C}'\in\mathrm{Ch}(\mathsf{N}_{\mathcal{C}})} \sum_{\mathsf{C}''\in\mathrm{Ch}(\mathsf{N}_{\mathcal{L}})} w_{\mathsf{S},\mathsf{C}'} \cdot v_{\mathsf{C}',\mathsf{C}''}$
6: return $v_{\mathsf{R}}$, where R is $\mathcal{C}$'s root
```

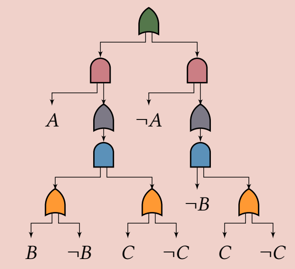
query is enabled by the following result.

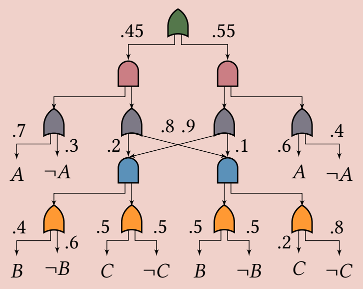
Theorem 2.3.1 (Y. Choi, Vergari, and Broeck, 2020). *If $\mathcal{C}$ is a smooth and structure decomposable probabilistic circuit with vtree $\mathcal{V}$, and $\mathcal{L}$ a structure decomposable logic circuit also respecting $\mathcal{V}$, then $\mathbb{E}_{\mathcal{C}}[\mathcal{L}]$ is polynomial time computable (in the number of edges).*

Let $\mathcal{C}$ a smooth and structure decomposable circuit with vtree $\mathcal{V}$, and $\mathcal{L}$ a logic circuit representing a logical query whose vtree is also $\mathcal{V}$. Computing the probability of $\mathcal{L}$ with respect to the distribution encoded by $\mathcal{C}$ is done by a bottom-up evaluation over both circuits at the same time. Algorithm 5 shows the procedure algorithmically. Importantly, the procedure relies on evaluating the expectation for each node in a reverse topological fashion according to line 2. This paired reverse topological sorting is done by a recursive assignment. For product nodes, primes are paired with primes, and subs with subs; for sums, each combination of PC child with LC child is paired up.

Example 2.3.3: Computing the probability of logical events

Say we have a PC encoding the distribution shown in the table on the right. Suppose we wish to compute the probability of a logical event, say $\phi \equiv A \vee \neg B$. A naïve approach would be to go over each assignment $\mathbf{x}$ where $\phi(\mathbf{x}) = 1$, and compute their sum. This is obviously exponential on the number of variables. Instead, we may compile ϕ into the logic circuit above and run the EXP algorithm to (in polynomial time) compute this otherwise intractable marginalization. Running the EXP gives us a probability of $0.6415 = 1.0 - (0.1365 + 0.2220)$, which is exactly the desired probability.





A	B	C	$\phi(\mathbf{x})$	$p(\mathbf{x})$
0	0	0	1	.1000
1	0	0	1	.0580
0	1	0	0	.1365
1	1	0	1	.0805
0	0	1	1	.1860
1	0	1	1	.0970
0	1	1	0	.2220
1	1	1	1	.1195

Remark 2.3.1: On applications of probabilistic circuits

So far, we have not yet addressed real-world applications of probabilistic circuits. Although their usage has not yet gained popularity among the data science crowd, they have been successfuly employed in a multitude of interdisciplinary tasks. Here, we give a brief survey on the different use cases of probabilistic circuits present in literature.

Computer vision is perhaps the most popular application for deep learning, and this could not be different for probabilistic circuits. PCs have been used for image classification (Gens and P. Domingos, 2012; Sguerra and F. G. Cozman, 2016; Llerena and Deratani Mauá, 2017; R. Geh and D. Mauá, 2019; Peharz, Vergari, *et al.*, 2020), image reconstruction and sampling (Poon and P. Domingos, 2011; Dennis and Ventura, 2017; Peharz, Lang, *et al.*, 2020; Cory J Butz *et al.*, 2019), image segmentation and scene understanding (Abram L Friesen and P. Domingos, 2017; Yuan *et al.*, 2016; Abram L Friesen and P. M. Domingos, 2018; Rathke *et al.*, 2017), and activity recognition (J. Wang and G. Wang, 2018; Amer and Todorovic, 2012; Nourani *et al.*, 2020; Amer and Todorovic, 2016).

Probabilistic circuits have also been used for sequential data (Melibari, Poupart, Doshi, and Trimponias, 2016), such as speech recognition and reconstruction (Peharz, Kapeller, *et al.*, 2014; Ratajczak *et al.*, 2014; Ratajczak *et al.*, 2018) and natural language processing (Cheng *et al.*, 2014).

Remarkably, probabilistic circuits have seen a recent boom in hardware-aware research (Shah, Isabel Galindez Olascoaga, *et al.*, 2019; Olascoaga *et al.*, 2019) and dedicated hardware for PCs in embedded systems (Sommer *et al.*, 2018; Shah, Olascoaga, Meert, *et al.*, 2020; Shah, Olascoaga, S. Zhao, *et al.*, 2021).

Some other applications include robotics (Sguerra and F. G. Cozman, 2016; R. Geh and D. Mauá, 2019; Zheng *et al.*, 2018; Pronobis *et al.*, 2017), biology (Cory J. Butz, Santos, *et al.*, 2018; Abram L. Friesen and P. Domingos, 2015), probabilistic programming (Stuhlm"uller and Goodman, 2012; Saad *et al.*, 2021), and fault localization (Nath and P. M. Domingos, 2016).

Now that we have addressed the (arguably) most important aspects of probabilistic circuits, we are now ready to understand how to build them in a principled way.

3 Learning Probabilistic Circuits

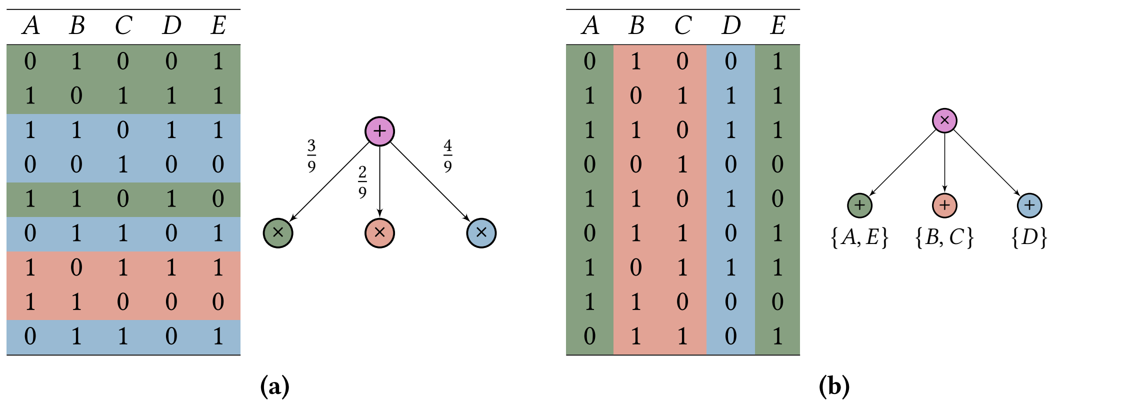
As we have seen in Chapter 2, inference in probabilistic circuits is, for the most part, straightforward. This is not so much the case when *learning* PCs. Despite the uncomplicated syntax, learning sufficiently expressive PCs in a principled way is comparatively harder to, say the usual neural network. For a start, we are usually required to comply with smoothness and decomposability to ensure marginalization at the least. This restriction excludes the possibility of adopting any of the most popular neural network patterns or architectures used in deep learning today. To make matters worse, constructing a PC graph more often than not involves costly statistical tests that make learning their structure a challenge for high dimensional data.

In this chapter, we review the most popular PC structure learning algorithms, their pros and cons, and more importantly, what can we learn from them to efficiently build scalable probabilistic circuits. We broadly divide existing structure learners into three main categories: divide-and-conquer (`DIV`, Section 3.1), incremental methods (`INCR`, Section 3.2) and random approaches (`RAND`, Section 3.3).

3.1 Divide-and-Conquer Learning

Arguably the most popular approach to learning the structure of probabilistic circuits are algorithms that follow a *divide-and-conquer* scheme. This class of PC learning algorithms, which here we denote by `DIV`, are characterized by recursive calls over (usually mutually exclusive) subsets of data in true divide-and-conquer fashion. This kind of procedure is more clearly visualized by LearnSPN, the first, most well-known, and perhaps most archetypal of its class.

Before we start however, we must first address how we denote data. Data is commonly represented as a matrix where rows are assignments (of all variables), and columns are the values that each variable takes at each assignment. Let $\mathbf{D} \in \mathbb{R}^{m \times n}$ a matrix with m rows and n columns. We use $\mathbf{D}_{i,j}$ to access an element of $\mathbf{D}$ at the i-th row, j-th column of matrix $\mathbf{D}$. We denote by $\mathbf{D}_{\mathbf{i},\mathbf{j}}$, where $\mathbf{i} \subseteq [1..n]$ and $\mathbf{j} \subseteq [1..m]$ are sets of indices, a submatrix from the extraction of the $\mathbf{i}$ rows and $\mathbf{j}$ columns of $\mathbf{D}$. We use a colon as a shorthand for selecting all rows or columns, e.g. $\mathbf{D}_{:,:} = \mathbf{D}$, $\mathbf{D}_{:,j}$ is the j-th column and $\mathbf{D}_{i,:}$ is the i-th row.



A	B	C	D	E
0	1	0	0	1
1	0	1	1	1
1	1	0	1	1
0	0	1	0	0
1	1	0	1	0
0	1	1	0	1
1	0	1	1	1
1	1	0	0	0
0	1	1	0	1

Figure 3.1: *LearnSPN assigns either rows (a) or columns (b) for sum and product nodes respectively. For sums, their edge weights are set proportionally to the assignments. For product children, scopes are defined by which columns are assigned to them.*

Algorithm 6 LearnSPN

Input Data **D**, whose columns are indexed by variables **X**
Output A smooth and decomposable probabilistic circuit learned from **D**

```
1: if $|\mathbf{X}| = 1$ then return an input node learned from $\mathbf{D}$
2: else
3:     Find scope partitions $\mathbf{X}_1, \dots, \mathbf{X}_t \subseteq \mathbf{X}$ st $\mathbf{X}_i \perp\!\!\!\perp \mathbf{X}_j$ for $i \neq j$
4:     if $k > 1$ then return $\prod_{j=1}^{t} \text{LearnSPN}(\mathbf{D}_{:,\mathbf{X}_j}, \mathbf{X}_j)$
5:     else
6:         Find subsets of data $\mathbf{x}_1, \dots, \mathbf{x}_k \subseteq \mathbf{D}$ st all assignments within $\mathbf{x}_i$ are all similar
7:         return $\sum_{i=1}^{k} \frac{|\mathbf{x}_i|}{|\mathbf{D}|} \cdot \text{LearnSPN}(\mathbf{x}_i, \mathbf{X})$
```

3.1.1 LearnSPN

Recall the semantics of sum and product nodes in a smooth and decomposable probabilistic circuit. A sum is a mixture of distributions $p(\mathbf{X}) = \sum_{i=1}^{m} w_i \cdot p_i(\mathbf{X})$ whose children scopes are all the same. A product is a factorization $p(\mathbf{X}_1, \dots, \mathbf{X}_n) = \prod_{i=1}^{n} p(\mathbf{X}_i)$, implying that $\mathbf{X}_i \perp\!\!\!\perp \mathbf{X}_j$ for $i, j \in [n]$ and $i \neq j$. LearnSPN (Gens and P. Domingos, 2013) exploits these semantics in an intuitive and uncomplicated manner: sum children are defined by sub-PCs learned from similar (by some arbitrary metric) assignments, and product children are sub-PCs learned from data conditioned on the variables defined by their scope. In practice, this means that, for a dataset $\mathbf{D} \in \mathbb{R}^{m \times n}$, sums assign rows to their children, while product children are assigned columns. This procedures continues recursively until data are reduced to a $k \times 1$ matrix, in which case a univariate distribution acting as input node is learned from it. This recursive procedure is shown more formally in Algorithm 6.

Notably, (Gens and P. Domingos, 2013) purposely does not strictly specify which techniques should be used for assigning rows and columns, although they do provide empirical results on a particular form of LearnSPN where row assignments are computed through EM clustering and products by pairwise G-testing. Instead, they call the algorithm a *schema* that incorporates several actual learning algorithms whose concrete form depends on the choice of how to split data.

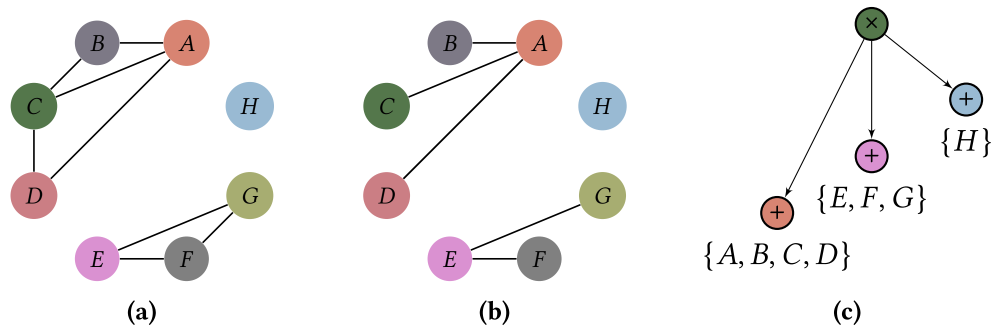


Figure 3.2: *The pairwise (in)dependence graph where each node is a variable. In (a) we show the full graph, computing independence tests for each pair of variables in $\mathcal{O}(m^2)$. However, it suffices to compute for only the connected components (b), saving up pairwise computation time for reachable nodes. The resulting product node and scope partitioning is shown in (c).*

Complexity

To be able to analyze the complexity of LearnSPN, we assume a common implementation where sums are learned from k-means clustering, and products through pairwise G-testing. We know learning sums is efficient: k-means takes $\mathcal{O}(n \cdot k \cdot m \cdot c)$ time, where k is the number of clusters and c the number of iterations to be run. Products, on the other hand, are much more costly. The naïve approach would be to compute whether $X_i \perp\!\!\!\perp X_j$ for every possible combination. This is clearly quadratic on the number of variables $\mathcal{O}\left(\binom{m}{2} = \frac{m!}{2(m-2)!}\right)$ assuming an $\mathcal{O}(1)$ oracle for independence testing. In reality, G-test takes $\mathcal{O}(n \cdot m)$ time, as we must compute a ratio of observed versus expected values for each cell in the contingency table. This brings the total runtime for products to a whopping $\mathcal{O}\left(n \cdot m^3\right)$, prohibitive to any reasonably large dataset. In terms of space, independence tests most commonly used require either a correlation (for continuous data) or contingency (for discrete data) matrix that takes up $\mathcal{O}(m^2)$ space, another barrier for scaling up to high dimensional data.

Alternatively, instead of computing the G-test for every possible combination of variables, (Gens and P. Domingos, 2013) constructs an independence graph $\mathcal{G}$ whose nodes are variables and edges indicate whether two variables are statistically dependent. Within this context, the variable partitions we attribute to product children are exactly the connected components of $\mathcal{G}$, meaning it suffices testing only some combinations. Even so, this heuristic is still quadratic worst case. Figure 3.2 shows $\mathcal{G}$, the spanning forest resulted from the connected component heuristic, and the equivalent product node from this decomposition.

Pros and cons

Pros. Perhaps the main factor for LearnSPN's popularity is how easily implementable, intuitive and modular it is. Even more remarkably, it is an empirically competitive PC learning algorithm despite its age, serving as a baseline for most subsequent works in PC literature. Lastly, the fact that each recursive call from LearnSPN is completely independent from each the other makes it an attractive candidate for CPU parallelization.

Algorithm 7 ExtendID

Input Data $\mathbf{D}$, whose columns are indexed by variables $\mathbf{X}$, and memoization function $\mathcal{M}$
Output A smooth and decomposable probabilistic circuit learned from $\mathbf{D}$

```
1:  Find scope partitions $\mathbf{X}_1, \dots, \mathbf{X}_t \subseteq \mathbf{X}$ st
2:  if $k > 1$ then
3:      for each $j \in [t]$ do
4:          $\mathsf{N}_j \leftarrow$ LearnMarkov($\mathbf{D}_{:,\mathbf{X}_j}, \mathbf{X}_j$)
5:          Associate $\mathcal{M}(\mathsf{N}_j)$ with $\mathbf{D}_{:,\mathbf{X}_j}$ and $\mathbf{X}_j$
6:      return $\prod_{j=1}^{t} \mathsf{N}_j$
7:  else
8:      Find subsets of data $\mathbf{x}_1, \dots, \mathbf{x}_k \subseteq \mathbf{D}$ st all assignments within $\mathbf{x}_i$ are all similar
9:      for each $i \in [k]$ do
10:         $\mathsf{N}_i \leftarrow$ LearnMarkov($\mathbf{x}_i, \mathbf{X}$)
11:         Associate $\mathcal{M}(\mathsf{N}_i)$ with $\mathbf{x}_i$ and $\mathbf{X}$
12:     return $\sum_{i=1}^{k} \frac{|\mathbf{x}_i|}{|\mathbf{D}|} \cdot \mathsf{N}_i$
```

Cons. Debatably, one of the key weakness of LearnSPN is its tree-shaped computational graph, meaning that they are strictly less succint compared to non-tree DAG PCs (Martens and Medabalimi, 2014). In terms of runtime efficiency, the algorithm struggles on high dimensional data due to the complexity involved in computing costly statistical tests. Despite Algorithm 6 giving the impression that no hyperparameter tuning is needed for LearnSPN, in practice the modules for learning sums and products often take many parameters, most of which (if not all) are exactly the same for every recursive call. This can have a negative impact on the algorithm's performance, since the same parameters are repeatedly used even under completely different data.

3.1.2 ID-SPN

A subtle yet effective way of improving the performance of LearnSPN is to consider tractable probabilistic models over many variables as input nodes instead of univariate distributions. ID-SPN (Rooshenas and Lowd, 2014) does so by assuming that input nodes are Markov networks. Further, instead of blindly applying the recursion over subsequent sub-data, it attempts to compute some metric of quality from each node. The worst scored node is then replaced with a LearnSPN-like tree. This is repeated until no significant increase in likelihood is observed. Algorithm 8 shows the ID-SPN pipeline, where ExtendID is used in line 7 to grow the circuit in a divide-and-conquer fashion. The name ID-SPN comes from *direct* variable interactions, meaning the relationships modeled through the Markov networks as input nodes; and *indirect* interactions brought from the latent variable interpretation of sum nodes.

With respect to its implementation, ID-SPN is as modular as LearnSPN in the sense that the data partitioning is left as a subroutine. Indeed, even the choice of input distributions is customizable: although Rooshenas and Lowd recommend Markov networks, any tractable distribution will do. Despite this seemingly small change compared to the original LearnSPN algorithm, ID-SPN seems to perform better compared to its counterpart most

Algorithm 8 ID-SPN

```
Input  Data $\mathbf{D}$, whose columns are indexed by variables $\mathbf{X}$
Output  A smooth and decomposable probabilistic circuit learned from $\mathbf{D}$
1: Create a single-node PC: $\mathcal{C} \leftarrow \textsc{LearnMarkov}(\mathbf{D}, \mathbf{X})$
2: Let $\mathcal{M}$ a memoization function associating a node with a dataset and scope
3: Call $\mathcal{C}'$ a copy of $\mathcal{C}$
4: while improving $\mathcal{C}$ yields better likelihood do
5:     Pick worse node $\mathsf{N}$ from $\mathcal{C}'$
6:     Extract sub-data $\mathbf{D}'$ and sub-scope $\mathbf{X}'$ from $\mathcal{M}(\mathsf{N})$
7:     Replace $\mathsf{N}$ with $\textsc{ExtendID}(\mathbf{D}', \mathbf{X}', \mathcal{M})$
8:     if $\mathcal{C}'$ has better likelihood than $\mathcal{C}$ then $\mathcal{C} \leftarrow \mathcal{C}$
9: return $\mathcal{C}$
```

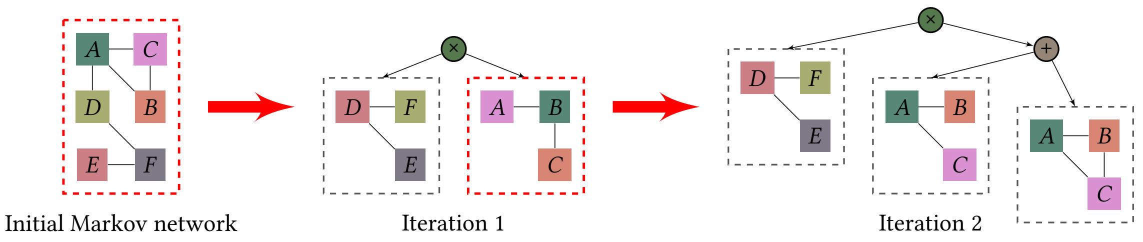


Figure 3.3: *Two iterations of ID-SPN, where the contents inside the dashed line are Markov networks. The red color indicates that a node has been chosen as the best candidate for an extension with* ExtendID. *Although here we only extend input nodes, inner nodes can in fact be extended as well.*

of the time (Rooshenas and Lowd, 2014; Jaini, Ghose, *et al.*, 2018), although at a cost to learning speed. Further, because of the enormous parameter space brought by having to learn Markov networks as inputs *and* perform the optimizations from sums and products, grid search hyperparameter tuning is infeasible. (Rooshenas and Lowd, 2014) recommend random search (Bergstra and Bengio, 2012) as an alternative.

Complexity

As ID-SPN is a special case of LearnSPN, the analysis for the sums and products subroutines holds. The only difference is on the runtime complexity for learning input nodes and the convergence rate for ID-SPN. Assuming input nodes are learned from the method suggested by Rooshenas and Lowd (2014), which involves learning a probabilistic circuit from a Markov network (Lowd and Rooshenas, 2013), then each "input" node takes time $\mathcal{O}(i \cdot c(k \cdot n + m))$, where i is the number of iterations to run, c is the size of the generated PC, and constant k is a bound on the number of candidate improvements to the circuit, which can grow exponentially for multi-valued variables. Importantly, opposite from LearnSPN where we only learn input nodes once per call *if* data is univariate, ID-SPN requires learning multiple multivariate inputs for *every* ExtendID call.

Pros and Cons

Pros. If we assume any multivariate distribution in place of Markov networks, PCs learned from ID-SPN are strictly more expressive than ones learned from LearnSPN, as

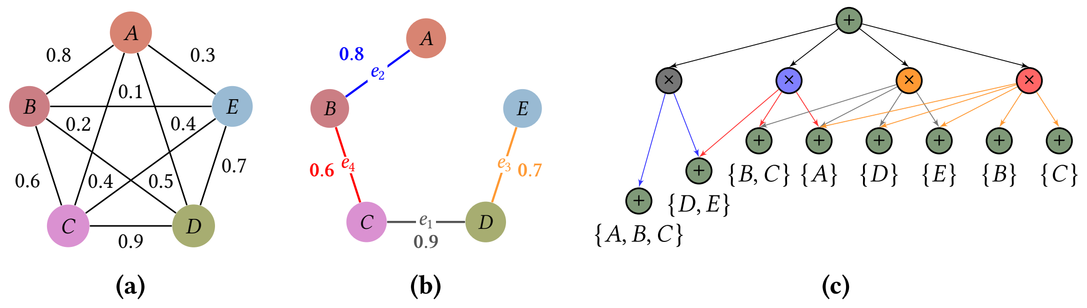


Figure 3.4: *The fully connected correlation graph (a) with weights as the pairwise correlation measurements for each pair of variables; the maximum spanning tree for determining decompositions (b); and the mixture of decompositions (c). Colors in (b) match their partitionings in (c).*

input nodes could potentially be replaced with LearnSPN distributions. Additionally, the modularity inherited from LearnSPN allows ID-SPN to adapt to data according to expert knowledge, bringing some flexibility to the algorithm.

Cons. Unfortunately, most of the disadvantages from LearnSPN also apply to ID-SPN. Just like LearnSPN, independence tests are more often than not a bottleneck for most executions with resonably large number of variables. However, ID-SPN relies on a likelihood improvement for the computational graph to be extended, which ends up curbing the easy parallelization aspect of LearnSPN. Besides, the complexity involved in learning Markov networks (or any other complex multivariate distribution as input node) carries a heavy weight during learning. This, coupled with the fact that hyperparameter tuning in the huge parameter space of ID-SPN must be done by a random search method, can take a heavy price in terms of learning time.

3.1.3 Prometheus

So far, we have only considered structure learning algorithms that produce tree-shaped circuits. Even though ID-SPN *might* produce non-tree graphs at the input nodes depending on the choice of families of multivariate distributions, it does not do so as a rule. We now turn our attention to a PC learner that *does* generate non-tree computational graphs in a divide-and-conquer manner.

Recall that in both LearnSPN and ID-SPN the scope partitioning is done greedily; we define a graph encoding the pairwise (in)dependencies of variables and greedily search for connected components by comparing independence test results with some correlation threshold, adding an edge if the correlation is sufficiently high. The choice of this threshold is often arbitrary and subject to hyperparameter tuning during learning, which is especially worrying when dealing with high dimensional data. In this section we review Prometheus (Jaini, Ghose, *et al.*, 2018), a divide-and-conquer LearnSPN-like PC learning algorithm with two main features that stand out compared to the last two methods we have seen so far: (1) it requires no hyperparameter tuning for variable partitionings, and (2) accepts a more scalable alternative to computing all pairwise correlations.

Let $\mathcal{G}$ the independence graph for scope $\mathbf{X} = \{X_1, X_2, \dots, X_m\}$. Remember that $\mathcal{G}$'s vertices are $\mathbf{X}$ and each (undirected) edge $\overline{X_i X_j}$ coming from X_i to X_j means that $X_i \perp\!\!\!\perp X_j$.

Algorithm 9 Prometheus

```
Input  Data $\mathbf{D}$, whose columns are indexed by variables $\mathbf{X}$
Output  A smooth and decomposable probabilistic circuit learned from $\mathbf{D}$
 1: if $|\mathbf{X}|$ is sufficiently small then return an input node learned from $\mathbf{D}$
 2: else
 3:     Find subsets of data $\mathbf{x}_1, \dots, \mathbf{x}_k \subseteq \mathbf{D}$ st all assignments within $\mathbf{x}_i$ are all similar
 4:     Create a sum node $\mathsf{S}$ with initially no children and uniform weights
 5:     for each $\mathbf{x}_i$ do
 6:         $\mathcal{T} \longleftarrow$ CorrelationMST$(\mathbf{x}_i, \mathbf{X})$
 7:         for each weighted edge $e_j$ in $\mathcal{T}$ in decreasing order do
 8:             Remove edge $e_i$ from $\mathcal{T}$
 9:             Call $\mathrm{S}_1, \dots, \mathrm{S}_t$ the scopes of each component in $\mathcal{T}$
10:             Create product node $\mathsf{P}_j$ and associate it with $\mathrm{S}_1, \dots, \mathrm{S}_t$
11:             Associate $\mathsf{P}_j$ with dataset $\mathbf{x}_i$
12:             Add $\mathsf{P}_j$ as a child of $\mathsf{S}$
13:     Let $\mathcal{H}$ a hash table (initially empty) associating scopes to sum nodes
14:     for each $\mathsf{P} \in \mathrm{Ch}(\mathsf{S})$ do
15:         for each scope $\mathrm{S}$ associated with $\mathsf{P}$ do
16:             if $\mathrm{S} \notin \mathcal{H}$ then
17:                 Let $\mathbf{x}$ the dataset associated with $\mathsf{P}$
18:                 $\mathsf{N} \longleftarrow$ Prometheus$(\mathbf{x}_{:,\mathrm{S}}, \mathrm{S})$
19:                 Add $\mathsf{N}$ as a child of $\mathsf{P}$
20:                 $\mathcal{H}_\mathrm{S} \longleftarrow \mathsf{N}$
21:             else
22:                 Add $\mathcal{H}_\mathrm{S}$ as a child of $\mathsf{P}$
23:     return $\mathsf{S}$
```

Previously, we constructed $\mathcal{G}$ by comparing the output of an independence test (such as the G-test) against a threshold (e.g. a sufficiently low p-value). Instead, suppose $\mathcal{G}$ is fully connected and that we attribute weights corresponding to a correlation metric of X_i against X_j for each edge (e.g. Pearson's correlation coefficient). The *maximum spanning tree* (MST) of $\mathcal{G}$, here denoted by $\mathcal{T}$, defines a graph where the removal of any edge in $\mathcal{T}$ partitions the component into two subcomponents. Let e_i the i-th lowest (weight) valued edge; Prometheus obtains a set of decompositions by iteratively removing edges from e_1 to $e_{|\mathbf{X}|-1}$. In other words, the algorithm constructs a product node for each decomposition, assigning the scope of each child as the scope of each component at each edge removal. These products are then joined together by a parent sum node that acts as a mixture of decompositions. Figure 3.4 shows an example of $\mathcal{T}$, the subsequent decompositions, and the resulting mixture of decompositions.

Sum nodes are learned by clustering data into similar instances, just like in previous cases. Since the previously mentioned procedure involving products creates a mixture of decompositions (and thus a sum node), we can simply collapse the consecutive sum layers into a single sum node. Algorithm 9 shows the algorithm in its entirety. CorrelationMST computes the (fully connected) correlation graph, returning its MST. It is worth mentioning that Prometheus makes sure each recursive call shares subcircuits whenever scopes are the same (this is when the hash table $\mathcal{H}$ in Algorithm 9 comes into play). This avoids an exponential growth from the $k \cdot (|\mathbf{X}| - 1)$ potential recursive calls.

Complexity

Up to now, the computation of decompositions is done by a $\mathcal{O}(m^2)$ construction of a fully connected correlation graph. This gives Prometheus no asymptotic advantage over neither LearnSPN nor ID-SPN. To change this, Jaini, Ghose, *et al.* propose a more scalable alternative: in place of constructing the entire correlation graph, sample $m \log m$ variables and construct a correlation graph where only $\log m$ edges are added for each of these sampled variables instead, bringing down complexity to $\mathcal{O}(m\,(\log m)^2)$.

The analysis of sum nodes is exactly the same as LearnSPN if we assume the same clustering method. If Prometheus is implemented with the same multivariate distributions as ID-SPN at the input nodes, the analysis for those also holds.

Pros and Cons

Pros. The notable achievements of Prometheus are evidently the absence of parameters for computing scope partitionings, reducing the dimension of hyperparameters to tune; a scalable alternative to partitionings that runs in sub-quadratic time; and (more debatably) the fact that the algorithm produces non-tree shaped computational graphs. Further, since product nodes are learned through correlation metrics, Prometheus is easily adaptable to continuous data. To some extent, Prometheus also inherits the modularity of LearnSPN, as the choice of how to cluster and what input nodes to use is open to the the user.

Cons. Although the construction of the correlation graph in Prometheus is not done greedily (at least in the quadratic version), selecting the decompositions (i.e. partitioning the graph into maximal components) is; of course, this is not exactly a drawback but a compromise, as graph partitioning is a known NP-hard problem (Feldmann and Foschini, 2015). Because Prometheus accounts for all decompositions yielded from components after the removal of each edge from the MST, the circuit can grow considerably, even if we reuse subcircuits at each recursive call. An alternative would be to globally reuse subcircuits (i.e. share $\mathcal{H}$ among different recursive calls) throughout learning, although this curbs expressivity somewhat, as these subcircuits are learned from possibly (completely) different data. Another option would be to bound the number of decompositions, or in other words remove only a bounded number of edges from the MST.

Remark 3.1.1: On variations of divide-and-conquer learning

Because of LearnSPN's simplicity and modularity, there is a lot of room for improvement. This is reflected in the many works in literature on refining LearnSPN to specific data, choosing the right parameters, producing non-tree shaped circuits, and choice of input nodes. In this remark segment, we briefly discuss other advances in divide-and-conquer PC learning.

As we have previously mentioned, one of the drawbacks of LearnSPN is the sheer volume of hyperparameters involved. Vergari, Mauro, *et al.* (2015) suggests simplifying clustering to only binary row splits, while Y. Liu and Luo (2019) proposes clustering methods that automatically decide the number of clusters from data. Together with Prometheus, the space of hyperparameters to tune is greatly reduced.

We again go back to the issue of reducing the cost of learning variable partitions. Apart from Prometheus, Di Mauro *et al.* (2017) also investigate more efficient decompositions, proposing two approximate sub-quadratic methods to producing variable splits: one by randomly sampling pairs of variables and running G-test, and the other by a linear time entropy criterion.

Vergari, Mauro, *et al.* (2015) proposes the use of Chow-Liu Trees as input nodes instead of univariate distributions, while Molina, Natarajan, *et al.* (2017) recommend Poisson distributions for modeling negative dependence. Bueff *et al.* (2018) combines LearnSPN with weighted model integration by learning polynomials as input nodes for continuous variables and counts for discrete data. Molina, Vergari, *et al.* (2018) adapts LearnSPN to hybrid domains by employing the randomized dependence coefficient for both clustering and variable partitioning, with pairwise polynomial approximations for input nodes.

Other contributions include adapting LearnSPN to relational data (Nath and P. Domingos, 2015), an empirical study comparing different techniques for clustering and partitioning in LearnSPN (Cory J. Butz, Oliveira, *et al.*, 2018), and LearnSPN post-processing strategies for deriving non-tree graphs (Rahman and Gogate, 2016).

3.2 Incremental Learning

Learning algorithms from the `DIV` class heavily rely on recursively constructing a probabilistic circuit in a top-down fashion. This facilitates learning, as we need only to greedily optimize at a local level. We now draw our attention to incremental algorithms that iteratively grow an initial circuit. These usually require a search over possible candidate nodes to be extended, and as such involve evaluating the entire circuit to determine best scores. In this section, we look at two examples of `INCR` class learning algorithms: LearnPSDD and Strudel.

3.2.1 LearnPSDD

As the name suggests, LearnPSDD (Liang, Bekker, *et al.*, 2017) learns a smooth, structure decomposable and deterministic probabilistic circuit (see Section 2.3.2), meaning its computational graph must respect a vtree. We therefore must address the issue of learning the vtree before we turn to the PC learning algorithm *per se.*

Recall that for a vtree $\mathcal{V}$, every inner node $v \in \mathcal{V}$ with $\mathbf{X} = \text{Sc}(v^{\leftarrow})$ and $\mathbf{Y} = \text{Sc}(v^{\rightarrow})$ determines that $\mathbf{X}$ and $\mathbf{Y}$ are independent, i.e. $p_{\mathcal{C}}(\mathbf{X}, \mathbf{Y}) = p_{\mathcal{C}}(\mathbf{X})p_{\mathcal{C}}(\mathbf{Y})$ for a PC $\mathcal{C}$. This means that a PC's vtree is pivotal in embedding the independencies of the circuit's distribution.

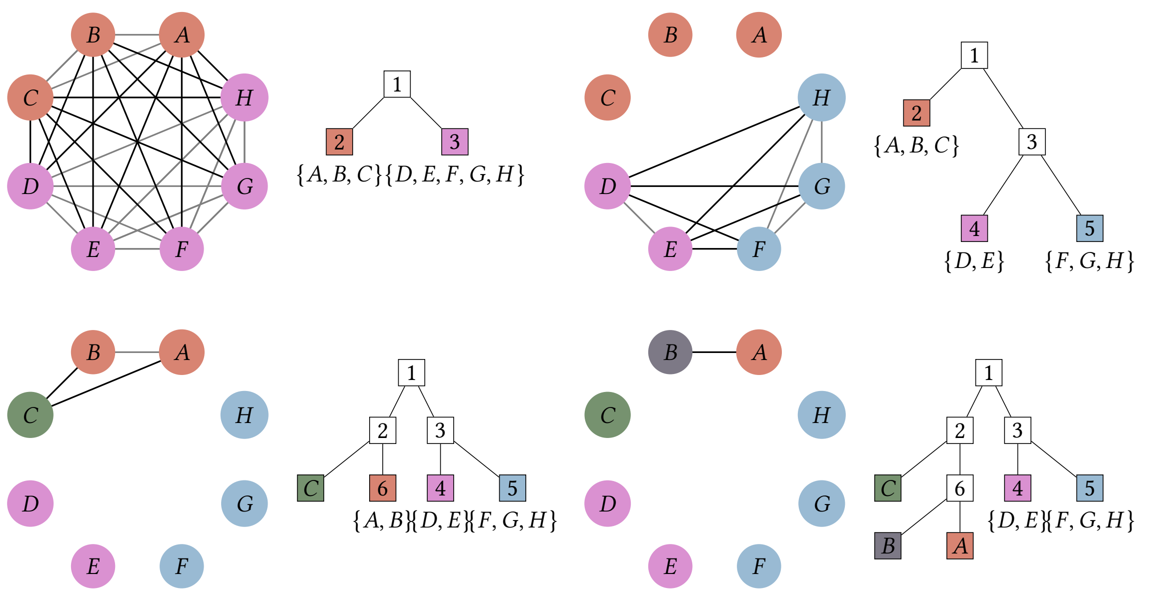


Figure 3.5: *Snapshots of four iterations from running the vtree top-down learning strategy with pairwise mutual information. Each iteration shows a variable partitioning, the cut-set that minimizes the average pairwise mutual information as black edges, and the subsequent (partial) vtree. The algorithm finishes when all partitions are singletons.*

With this in mind, Liang, Bekker, *et al.* (2017) propose two approaches to inducing vtrees from data, both of which use mutual information

$$\text{MI}(\mathbf{X}, \mathbf{Y}) = \sum_{\mathbf{X}=\mathbf{x}} \sum_{\mathbf{Y}=\mathbf{y}} p(\mathbf{x}, \mathbf{y}) \log \frac{p(\mathbf{x}, \mathbf{y})}{p(\mathbf{x})p(\mathbf{y})}$$

for deciding independence. To avoid computing an exponential number of MI terms, an approximation based on the average pairwise MI is computed instead

$$\text{pMI}(\mathbf{X}, \mathbf{Y}) = \frac{1}{|\mathbf{X}||\mathbf{Y}|} \cdot \sum_{X \in \mathbf{X}} \sum_{Y \in \mathbf{Y}} \text{MI}(X, Y).$$

The first approach learns vtrees in a top-down fashion, starting with a full scope and recursively partitioning down to the unit set. The second learns bottom-up, starting with singletons and joining sets of variables up to full scope.

Top-down vtree learning. Let $\mathcal{G}$ a fully connected weighted graph where variables are nodes. For each edge $\overrightarrow{XY}$, attribute its weight as $\text{MI}(X, Y)$. Learning the vtree top-down amounts to partitioning $\mathcal{G}$ such that the cut-set that divides the two partitions $\mathbf{X}$ and $\mathbf{Y}$ is minimal with respect to pMI. Liang, Bekker, *et al.* (2017) further argue that balanced vtrees produce smaller PCs, and so they reduce learning to a balanced min-cut bipartition problem. Although this is known to be NP-complete (Garey and Johnson, 1990), optimized solvers are able to produce high quality bipartitions efficiently (Karypis and Kumar, 1998). In a nutshell, the vtree construction goes as follows: find a balanced min-cut bipartition $(\mathbf{X}, \mathbf{Y})$ in $\mathcal{G}$ minimizing the pMI of the edges; add a vtree inner node representing this bipartition and connect it to the two vtrees produced by the recursive calls over $\mathbf{X}$ and

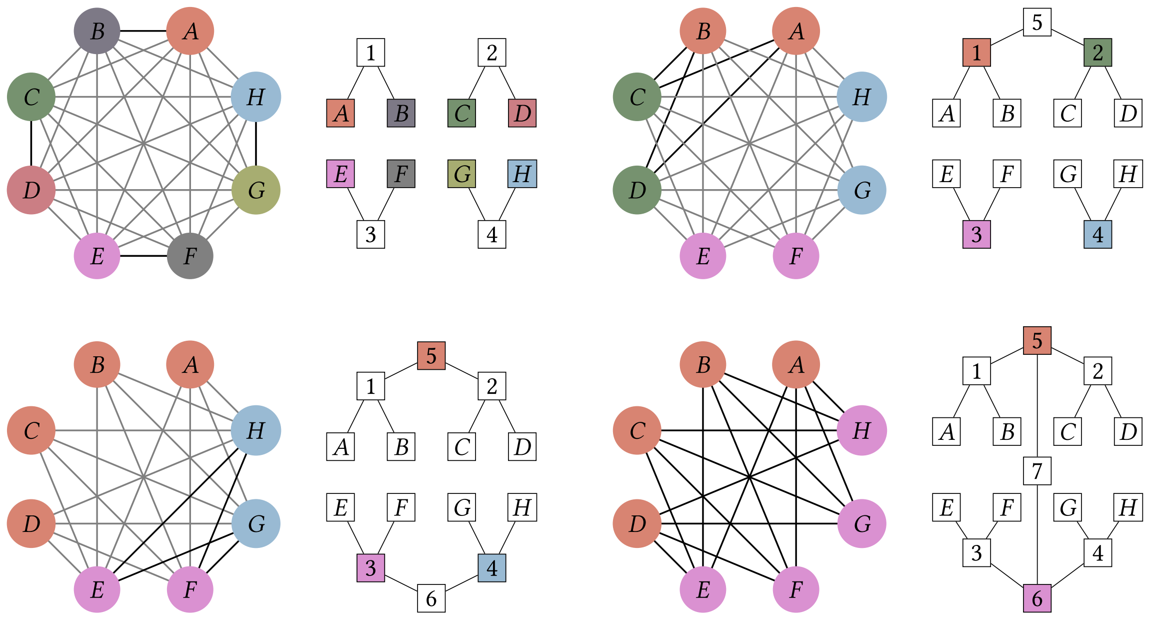


Figure 3.6: *Snapshots from running the vtree bottom-up learning strategy with pairwise mutual information. Snapshots show pairings of two vtrees, with edges between partitions joined into a single edge whose weight is the average pairwise mutual information of all collapsed edges. In black are edges that correspond to the matchings that maximize the average pairwise mutual information. The algorithm finishes when all vtrees have been joined together into a single tree.*

Y; if $\mathbf{X} = \{X\}$ (resp. $|\mathbf{Y}| = \{Y\}$), produce a leaf node X (resp. Y). Figure 3.5 shows four iterations of this procedure.

Bottom-up vtree learning. Again, take $\mathcal{G}$ as the fully connected weighted graph from computing the pairwise mutual information of variables. Now consider that every node of $\mathcal{G}$ is a vtree whose only node is the variable itself. To learn a vtree bottom-up is to find pairings of vtrees such that the mutual information between them is high, meaning that the partitionings at higher levels are minimized (and so determine the "true" independence relationships between subsets of variables). To produce balanced vtrees, the algorithm attempts to join vtrees of same height whose pMI is maximal; this is equivalent to min-cost perfect matching, which can be solved, in our case, in $\mathcal{O}(m^4)$, where m is the number of variables (Edmonds, 1965; Kolmogorov, 2009).

LearnPSDD is an incremental learning algorithm. This means that it takes an existing PC and incrementally grows the circuit by some criterion, preserving the structural constraints from the PC in the process. Once a vtree $\mathcal{V}$ has been learned from data, we use it to construct an initial circuit that respects $\mathcal{V}$. The choice of circuit initialization is dependent on our task. For example, within the context of PSDDs, we are mostly interested in starting out with a PC induced from an LC encoding a certain knowledge base (see Section 2.3); this is usually done in a case-by-case basis, where LCs are compiled for a particular task and then promoted to PCs (see Remark 3.2.1). However, if one does not require specifying the distribution's support, any PC will do.

How and *where* the circuit is grown – once we have acquired a vtree and an initial circuit – are the main topics of interest now. We first address the matter of *how*, i.e. how

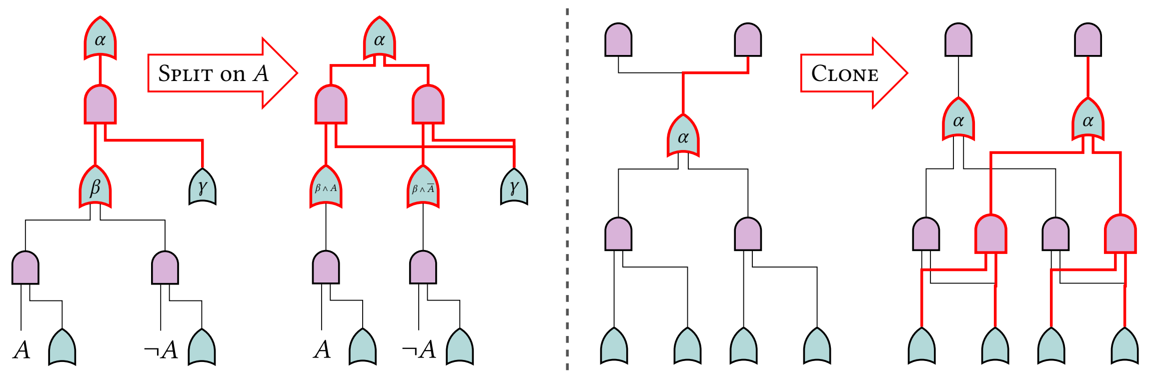


Figure 3.7: *SPLIT (left) and CLONE (right) operations for growing a circuit when $m = 1$. Nodes and edges highlighted in red show the modified structure. In both cases smoothness, (structure) decomposability and determinism are evidently preserved.*

can we increase a PC's expressivity such that we preserve a desired set of structural constraints; and later of *where*, i.e. which portions of the circuit are eligible for growth and how do we know they are good candidates.

LIANG, BEKKER, *et al.* propose two local transformations for growing a circuit $\mathcal{C}$: SPLIT and CLONE. The first acts by multiplying a sum node's product child P into $\mathsf{P}_1, \dots, \mathsf{P}_k$ products such that $\pi_1, \dots, \pi_k$ (primes of $\mathsf{P}_1, \dots, \mathsf{P}_k$ respectively) are mutually exclusive. This is done by attributing all possible values of a variable in $\mathrm{Sc}(\mathsf{P})$, say A, to each prime, meaning that π_i will contain the assignment $A = i$ for every $i \in [k]$. This attribution is done by partially copying $\mathcal{C}_{\mathsf{P}}$ into k circuits $\mathcal{C}_{\mathsf{P}}^{(1)}, \dots, \mathcal{C}_{\mathsf{P}}^{(k)}$ up to some depth m and then conditioning $\mathcal{C}_{\mathsf{P}}^{(i)}$ on $[\![A = i]\!]$. This is straightforward for the discrete case: at the appropriate vtree node (i.e. one that contains A as a leaf), replace the input node whose scope is A into an indicator node, setting it to the appropriate assignment of A. Although LIANG, BEKKER, *et al.* (2017) only considers the binary case, the transformation can be extended to the continuous if we consider k piecewise distributions whose support is over only a set interval. Naturally, input nodes must then have their support truncated to the appropriate i-th interval, which is no easy feat in the general case. The left side of Figure 3.7 shows SPLIT for the binary case.

The other proposed transformation, CLONE, does something similar for sum nodes. Pick a sum node S whose children are $\mathsf{C}_1, \dots, \mathsf{C}_k$ and parents P_1 and P_2; double S and $\mathsf{C}_1, \dots, \mathsf{C}_k$, producing clones S' and $\mathsf{C}'_1, \dots, \mathsf{C}'_k$. Disconnect the edge coming from P_2 to S and instead connect it to S'. Connect all $\mathsf{C}'_1, \dots, \mathsf{C}'_k$ to the same children as their original counterparts. This operation is visualized on the right side of Figure 3.7. One can further extend CLONE to apply this operation cloning nodes up to some depth m and then joining the last remaining deepest nodes similar to what was described for $\mathsf{C}'_1, \dots, \mathsf{C}'_k$.

It is easy to see that, in both cases, smoothness, structure decomposability and determinism are preserved. In fact, if the original circuit encodes a particular support (i.e. a knowledge base), the PC resulting from applying any of the two transformations must also encode the same support, since we have only made the underlying logic circuit more redundant. Probabilistically though, this "redudancy" only increases the parameterization space and as such increases the expressiveness of the PC. However, not all applications of

Algorithm 10 LearnPSDD

```
Input  Data $\mathbf{D}$, vtree $\mathcal{V}$, initial PC $\mathcal{C}$, max depth $m$, scope $\mathbf{X}$
Output  A smooth, structure decomposable and deterministic PC learned from $\mathbf{D}$
 1: while there is score improvement or has not reached the iteration/time limit do
 2:     $s_S \leftarrow -\infty$
 3:     Let $(S^*, P^*)$ the best Split candidate seen so far, initially empty
 4:     for each candidate $(S, P)$ of all possible Split candidates do
 5:         $\mathcal{C}' \leftarrow \text{Split}(\mathcal{C}, S, P, \mathcal{V}, m)$
 6:         $s' \leftarrow \text{Score}(\mathbf{D}, \mathcal{C}, \mathcal{C}')$
 7:         if $s' > s_S$ then $s_S \leftarrow s'$ and $S^*, P^* \leftarrow S, P$
 8:     $s_C \leftarrow -\infty$
 9:     Let $C^*$ the best Clone candidate seen so far, initially empty
10:     for each candidate C of all possible Clone candidates do
11:         $\mathcal{C}' \leftarrow \text{Clone}(\mathcal{C}, C, \mathcal{V}, m)$
12:         $s' \leftarrow \text{Score}(\mathbf{D}, \mathcal{C}, \mathcal{C}')$
13:         if $s' > s_C$ then $s_C \leftarrow s'$ and $C^* \leftarrow C$
14:     if $s_S > s_C$ then $\mathcal{C} \leftarrow \text{Split}(\mathcal{C}, S^*, P^*, \mathcal{V}, m)$
15:     else $\mathcal{C} \leftarrow \text{Clone}(\mathcal{C}, C^*, \mathcal{V}, m)$
16: return $\mathcal{C}$
```

Split or Clone are equal in terms of performance. While it is true that the application of Split to any product node or Clone to any sum node strictly increases expressivity, it is more meaningful to choose candidates whose growth carries a bigger impact on the overall fit relative to the training data. LearnPSDD searches for reasonable candidates by computing

$$\text{Score}(\mathbf{D}, \mathcal{C}, \mathcal{C}') = \frac{\log \mathcal{C}'(\mathbf{D}) - \log \mathcal{C}(\mathbf{D})}{|\mathcal{C}'| - |\mathcal{C}|},$$

where $\mathcal{C}$ and $\mathcal{C}'$ are, respectively, the PCs before and after the application of any of the two operations. In other words, the algorithm randomly evaluates applying Split and/or Clone and ultimately chooses the one candidate that maximizes the log-likelihood of training data penalized by the size of the resulting PC, iteratively growing the circuit until there is no more improvement or reaches an iteration step or time limit, as Algorithm 10 shows.

Complexity

Although learning the vtree top-down reduces to an NP-complete min-cut graph partitioning problem, there are approximate algorithms that provide high quality partitionings in $\mathcal{O}(|\mathbf{X}|^2)$ (Karypis and Kumar, 1998). Learning bottom-up is reduced to min-cost perfect matching, which can be done in $\mathcal{O}(|\mathbf{X}|^4)$ via the Edmonds Blossom algorithm (Edmonds, 1965; Kolmogorov, 2009).

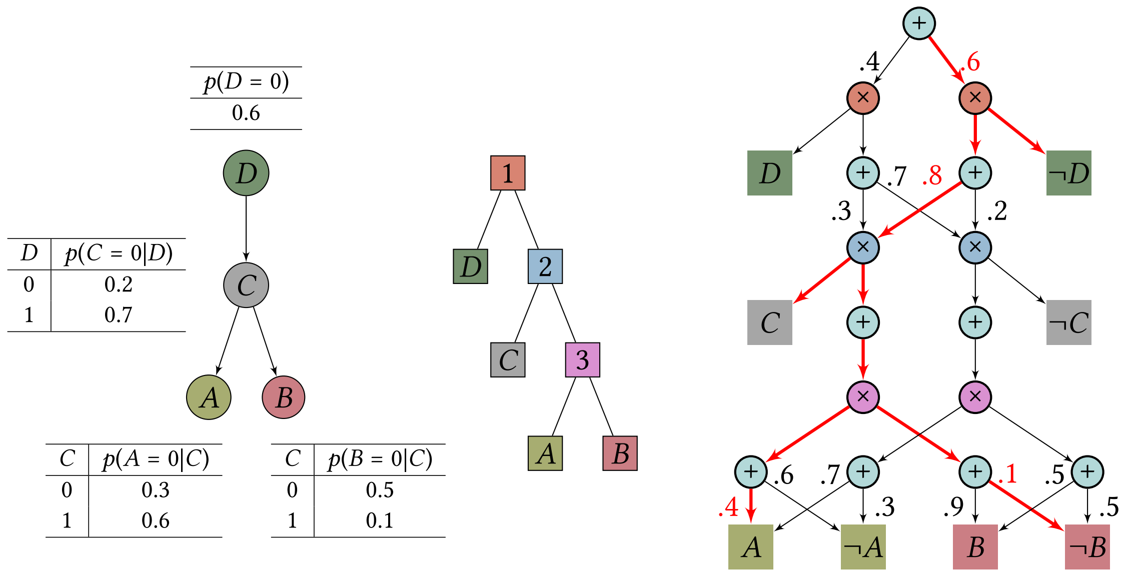
Split runs, for a given variable X, in $\mathcal{O}(v \cdot |\mathcal{C}|)$ if unbounded by m, where v is $|\text{Val}(X)|$, the number of possible assignments to X if X is discrete; or the number of intervals to fragment $\text{Val}(X)$ if X is continuous. Clone's runtime is $\mathcal{O}(|\mathcal{C}|)$ when m is unbounded, as it needs to produce an almost exact copy of the circuit. We say that a local transformation, such as Split or Clone, is *minimal* when the copy depth is $m = 0$. When Split and Clone are minimal and X is binary, then the transformation is done in constant time. In fact, any non-minimal transformation can be composed out of minimal transformations (Liang, Bekker, *et al.*, 2017).

Perhaps the most costly routine of LearnPSDD is its score function. Although log-likelihood is linear time computable on the number of edges of the circuit, $\mathcal{C}$ can grow substantially as transformations pile up. Each score evaluation requires four passes on the circuit: log-likelihoods and circuit sizes for both $\mathcal{C}$ and its updated circuit $\mathcal{C}'$. However, since transformations are local, log-likelihood and circuit sizes only change for the nodes affected in the transformation and their ancestors, allowing LearnPSDD to cache values. The overall complexity of LearnPSDD at each iteration is therefore $\mathcal{O}(|\mathcal{C}|^2)$ if we assume $m = 0$, with the first $|\mathcal{C}|$ coming from the search of all candidates in $\mathcal{C}$, and the second from the computation of Score. Each iteration further increases $|\mathcal{C}|$, slowing down the algorithm's runtime.

Pros and Cons

Pros. The fact that LearnPSDD preserves smoothness, structural decomposability, determinism *and* any logical semantic coming from its underlying LC is remarkable. On top of that, in theory and under minor modifications to Split and Clone, any PC is eligible as an initial circuit, even ones which do not respect any vtree. Besides, computing variable splits beforehand through a separate process of learning the vtree relieves the learning algorithm from having to compute costly statistical tests at each product node. Where LearnPSDD really shines (and perhaps more fittingly PSDDs in general) is when the support is explicitly defined through the initial circuit's LC; because the PC attributes non-zero probability only to events where the LC does not return false, the circuit wastes no mass on impossible events.

Cons. In practice, LearnPSDD is very slow even with caching; even worse, it may take several hours for only a minor (if any) improvement. (Liang, Bekker, *et al.*, 2017) suggests improving performance by producing ensembles of LearnPSDDs, although this negates determinism in the final model, denying the access to tractably computing queries like divergences, MI and entropies. Another issue is with the choice of the initial circuit. As previously mentioned, any circuit will do, however the performance (and efficiency) of LearnPSDD is highly dependent on it. Within the context of PSDDs and encoding their support, LearnPSDD requires that a separate algorithm compiles an LC for a specific task without looking at data. Although there are many ways of doing so, they are often not task agnostic (see Remark 3.2.1). More importantly, because the process of learning the circuit (from data) is decoupled from the task of encoding logical constraints imposed by a knowledge base, all variables that do not appear in the logic formula are compiled into a trivial form (e.g. fully factorized circuit). Lastly, although decoupling the process of learning the vtree from learning the PC helps with scalability, the ability of identifying the proper vtree for the most expressive PC given data is certainly desirable, and one which might be hindered by this separated process.



$p(D = 0)$
0.6

D	$p(C = 0\|D)$
0	0.2
1	0.7

C	$p(A = 0\|C)$
0	0.3
1	0.6

C	$p(B = 0\|C)$
0	0.5
1	0.1

Figure 3.8: *A vtree (middle) and probabilistic circuit (right) compiled from a Chow-Liu Tree (left). Each conditional probability $p(Y|X)$ is encoded as a (deterministic) sum node where each of the two children sets Y to 0 or 1. Colors in the CLT indicate the variables in the PC, while vtree inner node colors match with product nodes that respect them. Edges in red indicate the induced subcircuit activated on assignment* $\{A = 1, B = 0, C = 1, D = 0\}$.

3.2.2 Strudel

(Dang *et al.*, 2020) build upon the work of LearnPSDD and propose Strudel, which mainly improves LearnPSDD on two fronts: (1) by providing a simple algorithm for generating an initial circuit and vtree from data, and (2) proposing a heuristic for efficiently searching for good transformation candidates.

We first address how to construct the initial circuit from data. Dang *et al.* suggests doing so by compiling both a vtree and linear sized PC (in the number of variables) from a Chow-Liu Tree (CLT, Chow and C. Liu, 1968). Let $\mathcal{T}$ a CLT over variables $\mathbf{X} = \{X_1, \dots, X_m\}$. A vtree $\mathcal{V}$ is extracted from $\mathcal{T}$ by traversing $\mathcal{T}$ top-down. For each node $X_i \in \mathcal{T}$, if X_i is a leaf node in $\mathcal{T}$, then create a vtree leaf node of X_i; otherwise create an inner vtree node v, attach a vtree leaf node of X_i as $v^{\leftarrow}$ and assign $v^{\rightarrow}$ as a vtree built over all the vtrees coming from the children of X_i. The construction of $v^{\leftarrow}$ depends on how balanced one wishes the vtree to be: if we want a more right-leaning vtree, it suffices to construct a right-linear vtree connecting all vtrees from each child $X_j \in \text{Ch}(X_i)$. Likewise, a balanced vtree is built by balancing the vtree connecting the recursive vtree calls from each X_j. Note that this does not necessarily mean that $v^{\rightarrow}$ is completely right-linear or balanced, only that it is somewhat close to it, as the rest of the structure depends on the recursive calls of each CLT node.

Strudel compiles an initial circuit by looking at the vtree bottom-up and caching subcircuits. Let v a vtree node and $Y \in \mathcal{T}$ a CLT node with conditional probability $p(Y|X)$, where X is the parent of Y. If v is a leaf node in $\mathcal{V}$ and v's variable is also a leaf node in $\mathcal{T}$, two sum nodes S_0 and S_1 over literal nodes $\neg Y$ and Y are created, each with weights $w_{\mathrm{S}_0,\neg Y} = p(Y = 0|X = 0)$, $w_{\mathrm{S}_0,Y} = p(Y = 1|X = 0)$ and $w_{\mathrm{S}_1,\neg Y} = p(Y = 0|X = 1)$,

Algorithm 11 InitialStrudel

```
Input  Data $\mathbf{D}$, whose columns are indexed by variables $\mathbf{X}$
Output  A smooth, structure decomposable and deterministic initial PC and vtree
 1: $\mathcal{T} \leftarrow$ LearnCLT($\mathbf{D}, \mathbf{X}$)
 2: $\mathcal{V} \leftarrow$ CompileVtree($\mathcal{T}$)
 3: Let $\mathcal{M}$ a hash table for caching circuits, initially empty
 4: for each vtree node $v \in \mathcal{V}$ in reverse topological order do
 5:     if $v$ is a leaf node then
 6:         Let $X \in \mathcal{T}$ the variable represented by $v$, and $Y$ its parent
 7:         if $X$ is a leaf node in $\mathcal{T}$ then
 8:             $\mathsf{S}_j \leftarrow \sum_{i \in \text{Val}(X)} p(X=i|Y=j) \cdot [\![X=i]\!]$ for each $j \in \text{Val}(Y)$
 9:             $\mathcal{M}(v) \leftarrow \mathcal{M}(v) \cup \{\mathsf{S}_j | \forall j \in \text{Val}(Y)\}$
10:         else
11:             $\mathcal{M}(v) \leftarrow \mathcal{M}(v) \cup \{[\![X=i]\!] | i \in \text{Val}(X)\}$
12:     else
13:         Attribute $\mathbf{X} \leftarrow \text{Sc}(v^{\leftarrow})$ and $\mathbf{Y} \leftarrow \text{Sc}(v^{\rightarrow})$
14:         Let $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$
15:         Attribute $\mathsf{N}^{\leftarrow} \leftarrow \mathcal{M}(v^{\leftarrow})$ and $\mathsf{N}^{\rightarrow} \leftarrow \mathcal{M}(v^{\rightarrow})$
16:         $k \leftarrow |\text{Val}(X)|$
17:         Construct product nodes $\mathsf{P} = \{\mathsf{N}_i^{\leftarrow} \times \mathsf{N}_i^{\rightarrow} | \forall i \in [k]\}$
18:         if $\text{Pa}(X) = \text{Pa}(Y)$ then
19:             Create sum nodes $\mathsf{S}_i$ each with only a single child $\mathsf{P}_i \in \mathsf{P}$, for each $i \in [k]$
20:             $\mathcal{M}(v) \leftarrow \mathcal{M}(v) \cup \{\mathsf{S}_1, \dots, \mathsf{S}_k\}$
21:         else
22:             $\mathsf{S}_j \leftarrow \sum_{i \in \text{Val}(X)} p(\mathbf{Y}|X=j) \cdot [\![X=i]\!]$, for each $j \in \text{Val}(Y)$
23:             $\mathcal{M}(v) \leftarrow \mathcal{M}(v) \cup \{\mathsf{S}_j | \forall j \in \text{Val}(Y)\}$
24: return $\mathcal{M}(v_r)$, where $v_r$ is $\mathcal{V}$'s root node
```

$w_{\mathsf{S}_1,Y} = p(Y=0|X=1)$. The two sum nodes connecting B and $\neg B$ in the PC shown on the right of Figure 3.8 show this exact case. The left sum node encodes $p(B|C=1)$ and the right one $p(B|C=0)$. These circuits are then cached by associating them with v. When Y is not a leaf node in $\mathcal{T}$ but v is, we simply return literal nodes. If v is an inner node, we must define a scope partition, splitting $\mathbf{X} = \text{Sc}(v^{\leftarrow})$ and $\mathbf{Y} = \text{Sc}(v^{\rightarrow})$ into product nodes $\mathsf{P}_1, \dots, \mathsf{P}_k$, one for each value cached value in v. Each prime is set to the cached circuits from $v^{\leftarrow}$ and each sub the cached circuits from $v^{\rightarrow}$. Finally, if two variables $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$ are such that their parents are the same variable, say Z, then X and Y are independent when Z is given (because of a divergent connection in $\mathcal{T}$) and thus cannot be merged together into a single sum because of the context-specific independence set by Z (Boutilier *et al.*, 1996). This is visualized in the $\otimes$ nodes; in this situation, A and B are siblings coming from C, and so $A \perp\!\!\!\perp B|C$ (redundant sum nodes are added for standardization). When the prior situation is not true, then not only X is the only variable in $\mathbf{X}$, but X must also be the parent of Y and so we must model $p(\mathbf{Y}|X)$. This is the case for $\otimes$, where C is the parent of B and so we have to be join the two by sum nodes attributing the conditional probabilities $p(A,B|C=0)$ for the right-most $\otimes$ and $p(A,B|C=1)$ for the left-most sibling. This procedure is shown more formally in Algorithm 11.

Once we have an initial PC constructed from InitialStrudel, we are ready to discuss Strudel's second contribution. To do so, we must first understand the notion of *circuit flows* introduced in Dang *et al.* (2020). In short, the circuit flow of a deterministic probabilistic circuit $\mathcal{C}$ with respect to a variable assignment $\mathbf{x}$ is the induced tree (see Definition 2.1.2) whose edges are all non-zero when $\mathcal{C}$ is evaluated under $\mathbf{x}$. Such an induced tree is unique in deterministic PCs because every sum node admits only one non-zero valued child for $\mathbf{x}$ (or any assignment for that matter). Note how circuit flows are more specific in the sense they are intrinsically linked to an assignment, while induced subcircuits specify a deterministic subcircuit within its supercircuit. The circuit flow of deterministic PCs helps us understand how to efficiently compute inference in circuits of that nature. As we briefly mentioned before, for any assignment $\mathbf{x}$ in a smooth, decomposable and deterministic PC $\mathcal{C}$, there exists a unique circuit flow $\mathcal{F}$ that encodes the log-likelihood computation

$$\mathcal{C}(\mathbf{x}) = \mathcal{F}_{\mathcal{C}}(\mathbf{x}) = \prod_{(\mathrm{S},\mathrm{C})\in\mathrm{Edges}(\mathcal{F}_{\mathcal{C}})} w_{\mathrm{S},\mathrm{C}} \prod_{\mathrm{L}\in\mathrm{Inputs}(\mathcal{F}_{\mathcal{C}})} p_{\mathrm{L}}(\mathbf{x}),$$

where Inputs$(\cdot)$ returns the set of input nodes of a circuit. When inputs are all binary, then one might encode $\mathcal{F}_{\mathcal{C}}$ as a mapping $f_{\mathcal{C}} : \mathcal{X} \to \{0,1\}^{|\mathcal{W}_{\mathcal{C}}|}$, here $\mathcal{W}_{\mathcal{C}}$ denoting the set of all parameters (i.e. sum node weights) of $\mathcal{C}$, which "activates" edge $w \in \mathcal{W}_{\mathcal{C}}$ under assignment $\mathbf{x}$. With this, the above operation under log-space is reduced to a vector multiplication

$$\log \mathcal{C}(\mathbf{x}) = f_{\mathcal{C}}(\mathbf{x})^{\top} \cdot \log\left(\mathcal{W}_{\mathcal{C}}\right).$$

Importantly, by aggregating circuit flows through counting the number of activations of each parameter $w_{\mathrm{S},\mathrm{C}}$ in the entire training dataset $\mathbf{D}$, we get a sense of the number of samples $w_{\mathrm{S},\mathrm{C}}$ impacts over $\mathbf{D}$, and thus a sense of how meaningful is that edge on the fitness of data. As we shall see briefly, this aggregated circuit flow shall then be used as a score for a greedy search over the space of candidates for local transformations.

To overcome the scalability limitations of LearnPSDD, Strudel proposes using only Split to reduce the search space, looking at performing the search greedily instead of exhaustively and exploiting the efficiency of aggregate circuit flows as a fast heuristic in place of computing the whole likelihood. Searching is done by finding the edge to Split whose aggregate circuit flow is maximal

$$\mathrm{Score}_{\mathrm{eFlow}}(w_{\mathrm{S},\mathrm{C}}|\mathcal{C},\mathbf{D}) = \sum_{\mathbf{x}\in\mathbf{D}} f_{\mathcal{C}}(\mathbf{x})\left[w_{\mathrm{S},\mathrm{C}}\right],$$

while the choice of which variable to condition Split on is done by selecting the variable X that shares the most dependencies (and thus the higher pairwise mutual information) with other variables within the scope of that edge, estimated from the aggregate flows

$$\mathrm{Score}_{\mathrm{vMI}}(X, w_{\mathrm{S},\mathrm{C}}|\mathcal{C},\mathbf{D}) = \sum_{\substack{Y\in\mathrm{Sc}(\mathrm{S})\\ Y\neq X}} \mathrm{MI}(X,Y).$$

The entire algorithm for Strudel is showcased in Algorithm 12.

Algorithm 12 Strudel

```
Input  Data $\mathbf{D}$, max depth $m$, scope $\mathbf{X}$
Output  A smooth, structure decomposable and deterministic PC learned from $\mathbf{D}$
1: $\mathcal{C}, \mathcal{V} \leftarrow$ InitialStrudel($\mathbf{D}, \mathbf{X}$)
2: while there is score improvement of has not reached the iteration/time limit do
3:     Compute the aggregate flow over all edges
4:     $w^*_{\mathrm{S,C}} \leftarrow \arg\max_{w \in \mathcal{W}_C} \mathrm{Score}_{\mathrm{eFlow}}(w|\mathcal{C}, \mathbf{D})$
5:     $X^* \leftarrow \arg\max_{X \in \mathrm{Sc(S)}} \mathrm{Score}_{\mathrm{vMI}}(X, w^*_{\mathrm{S,C}}|\mathcal{C}, \mathbf{D})$
6:     $\mathcal{C} \leftarrow$ Split($\mathcal{C}$, S, C, $\mathcal{V}$, $m$)
7: return $\mathcal{C}$
```

Complexity

Learning the Chow-Liu Tree is done in $\mathcal{O}(|\mathbf{X}|^2 \cdot |\mathbf{D}|)$ through Chow-Liu's algorithm (Chow and C. Liu, 1968), while the vtree is compiled in time linear to the size of the CLT, i.e. $\mathcal{O}(|\mathbf{X}|)$ since the Bayesian network is a tree. Consequentially, InitialStrudel runs in $\mathcal{O}(|\mathbf{X}| \cdot |\mathrm{Val}(X)|)$, or linear on $|\mathbf{X}|$ if we assume binary variables as originally intended. The bulk of the computation falls under Strudel, which runs in $\mathcal{O}\left(|\mathbf{X}|^2 \cdot |\mathbf{D}| + i(|\mathcal{C}| \cdot |\mathbf{D}| + |\mathbf{X}|^2)\right)$ assuming a bounded max depth m and binary variables. Term $|\mathbf{X}|^2 \cdot |\mathbf{D}|$ corresponds to learning the CLT, $|\mathcal{C}| \cdot |\mathbf{D}|$ to the computation of the aggregate circuit flows, $|\mathbf{X}|^2$ to the computation of $\mathrm{Score}_{\mathrm{vMI}}$ which involves the pairwise mutual informations of $\mathbf{X}$, and i the number of iterations of Strudel.

Pros and Cons

Pros. Arguably, the most valuable contribution in Strudel is its scalability. Compared to LearnPSDD, Strudel can take orders of magnitude less time per iterationCompared to LearnPSDD, Strudel can take orders of magnitude less time per iteration, meaning that it can produce more Splits

Cons. **Remark 3.2.1: On the choice of initial circuits**

(A. Choi and Darwiche, 2013; Oztok and Darwiche, 2015; A. Choi, Shen, *et al.*, 2017; Shen *et al.*, 2017)

3.3 Random Learning

A Appendix

A.1 Proofs

Theorem A.1.1. *Let $\mathcal{C}$ a probabilistic circuit whose first l layers are composed solely of sum nodes. Call $\mathbf{N}$ the set of all nodes in layer $l+1$. $\mathcal{C}$ is equivalent to a PC $\mathcal{C}'$ whose root is a sum node with $\mathbf{N}$ as children.*

Proof. We adapt a similar proof due to Jaini, Poupart, *et al.* (2018). Every sum node is of the form

$$\mathrm{S}(\mathbf{x}) = \sum_{\mathrm{C}\in\mathrm{Ch}(\mathrm{S})} w_{\mathrm{S},\mathrm{C}} \cdot \mathrm{C}(\mathbf{x}).$$

Particularly, every child C in a sum node in layer $1 \leq i \leq l-1$, is a sum node, and so for the first layer we have that

$$\begin{aligned}\mathrm{S}(\mathbf{x}) &= \sum_{\mathrm{C}_1\in\mathrm{Ch}(\mathrm{S})} w_{\mathrm{S},\mathrm{C}_1} \sum_{\mathrm{C}_2\in\mathrm{Ch}(\mathrm{C}_1)} w_{\mathrm{C}_1,\mathrm{C}_2}\,\mathrm{C}_2(\mathbf{x})\\ &= \sum_{\mathrm{C}_1\in\mathrm{Ch}(\mathrm{S})}\sum_{\mathrm{C}_2\in\mathrm{Ch}(\mathrm{C}_1)} w_{\mathrm{S},\mathrm{C}_1} w_{\mathrm{C}_1,\mathrm{C}_2}\,\mathrm{C}_2(\mathbf{x}).\end{aligned}$$

Define a one-to-one mapping that takes a tuple $(\mathrm{C}_1, \mathrm{C}_2)$ where $\mathrm{C}_1 \in \mathrm{Ch}(\mathrm{S})$ and $\mathrm{C}_2 \in \mathrm{Ch}(\mathrm{C}_1)$ and returns a (unique) path from S to every grandchild C_2 of S. Call $\mathbf{K}$ the set of all paths, and $w_{\mathrm{S},\mathrm{C}_1}$ and $w_{\mathrm{C}_1,\mathrm{C}_2}$ the weights for one such path. We can merge these two weights into a single weight $w'_{\mathrm{S},\mathrm{C}_2} = w_{\mathrm{S},\mathrm{C}_1} \cdot w_{\mathrm{C}_1,\mathrm{C}_2}$, yielding

$$\mathrm{S}(\mathbf{x}) = \sum_{(w_{\mathrm{S},\mathrm{C}_1}, w_{\mathrm{C}_1,\mathrm{C}_2})\in\mathbf{K}} w'_{\mathrm{S},\mathrm{C}_2}\,\mathrm{C}_2(\mathbf{x}).$$

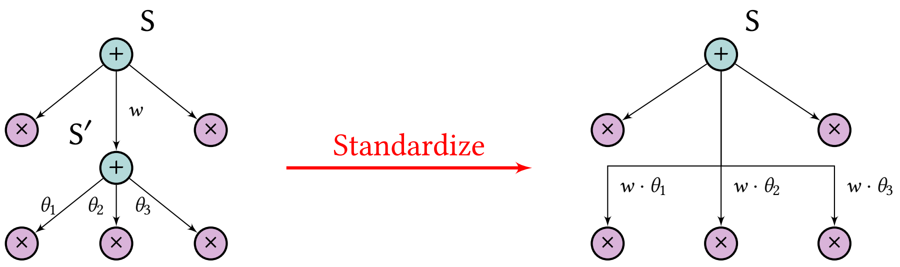
This ensures that two consecutive sum layers can be collapsed into a single layer. Particularly, for the first (root) and second layers, the above transformation generates a circuit with one fewer layer and whose root has $\mathcal{O}(nm)$ edges, where n and m are the number of edges coming from the original root and its children respectively. We can apply this procedure until there are no more consecutive sum nodes. This results in a PC of the form

$$\mathsf{S}(\mathbf{x}) = \sum_{\mathsf{C} \in \mathrm{Ch}(S)} w_{\mathsf{S},\mathsf{C}}\, \mathsf{N}(\mathbf{x}),$$

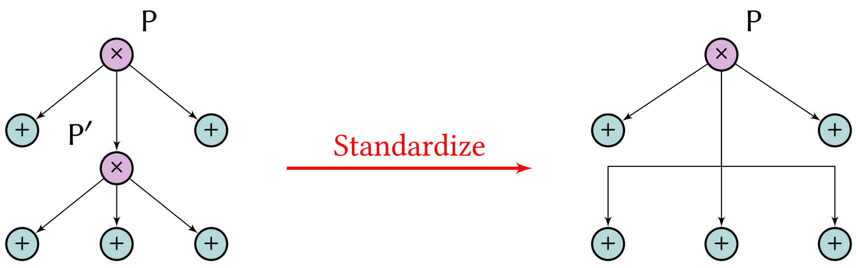
where $\mathsf{N} \in \mathbf{N}$. The number of children of the resulting root sum node will be exponential on the number of edges of its children. □

Theorem A.1.2 (Standardization). *Any probabilistic circuit C can be reduced to a circuit where every sum node contains only products or inputs and every product node contains only sums or inputs.*

Proof. If C is already standard we are done. Otherwise, there exists either (i) a sum node S with a sum S' as child; or (ii) a product node P with a product P' as child. We first address (i): let w be the weight of edge $\overrightarrow{\mathsf{S}\,\mathsf{S}'}$ and θ_i the weights from all edges coming out from S'.



Connect S with every child of S', assigning as weight $w \cdot \theta_i$ for each child i. Delete S' and all edges coming out from it. The resulting circuit is computationally equivalent but now without a consecutive pair of sums. This transformation is visualized by the figure above. We do a similar procedure in (ii), but now instead remove P' and connect all children of P' to P, as we show below.

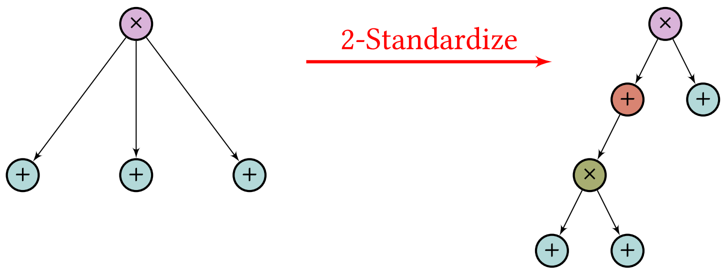


□

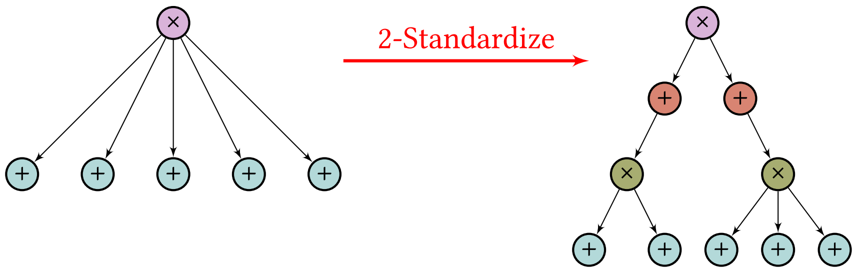
Theorem A.1.3 (2-Standardization). *Any probabilistic circuit C can be transformed into a circuit where every sum node contains only products or inputs and every product node contains only* two *sums or inputs.*

Proof. For sums, apply the same standardization procedure as theorem A.1.2. Let P a product and call $n = |\mathrm{Ch}(\mathsf{P})|$. If $n = 1$ and $\mathrm{Ch}(\mathsf{P})$ is a product, then remove P and connect all previous parents of P with its child. If $n = 1$ and Ch(P) is not a product, remove P and apply the standardization procedure for sums on all of Pa(P).

For $n > 2$, we simply need to split into 2-products recursively. We prove this by induction. The base case is when $n = 2$, which is already done, or $n = 3$, in which case we need apply the transformation below.



Where ⊕ and ⊗ are newly introduced nodes. When $n > 3$, we create two products P_1 and P_2, each connected with a sum and product, and with $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ potential children. By the induction hypothesis, we can recursively binarize the subsequent grandchildren products.



As an example, we have $n = 5$ in the figure above. We introduce the sums ⊕ and products ⊗ and then recursively apply the transformation again on the ⊗s.

When Ch(P) are product nodes we do the same procedure as before, but with the added post-process addition of a sum node connecting ⊗ to every Ch(P). □

Theorem 2.2.1 (Poon and P. Domingos, 2011; Y. Choi, Vergari, and Van den Broeck, 2020; Vergari, Y. Choi, *et al.*, 2021)**.** *Let C a* smooth *and* decomposable *PC. Any one of EVI, MAR or CON can be computed in linear time (in the number of edges of C).*

Proof. For a sum node S, we have the following marginalization query

$$\begin{aligned}\int S(\mathbf{x}, \mathbf{y})\, d\mathbf{y} &= \int \sum_{C \in Ch(S)} w_{S,C}\, C(\mathbf{x}, \mathbf{y})\, d\mathbf{y} \\ &= \sum_{C \in Ch(S)} w_{\Sigma,C} \int C(\mathbf{x}, \mathbf{y})\, d\mathbf{y}.\end{aligned}$$

Analogously, for a product node

$$\int \mathrm{P}(\mathbf{x}, \mathrm{y})\, d\mathrm{y} = \int \prod_{\mathrm{C} \in \mathrm{Ch(P)}} \mathrm{C}(\mathbf{x}, \mathrm{y})\, d\mathrm{y}$$
$$= \prod_{\mathrm{C} \in \mathrm{Ch(P)}} \int \mathrm{C}(\mathbf{x}, \mathrm{y})\, d\mathrm{y}.$$

This ensures that marginals are pushed down to children. This can be done recursively until C is an input node L_p, in which case we marginalize $\mathbf{y}$ according to p, which by definition should be tractable and here we assume can be done in $\mathcal{O}(1)$. We have proved the case for MAR. For EVI, we simply assign $\mathbf{y} = \emptyset$ with input nodes acting as probability density functions. Conditionals can easily be computed by an EVI or MAR followed by a second pass marginalizing the conditional variables $p(\mathbf{x}|\mathrm{y}) = \frac{p(\mathbf{x},\mathrm{y})}{p(\mathrm{y})}$ which are both done in linear time as we have seen here. □

Theorem 2.2.2 (Peharz, Gens, Pernkopf, *et al.*, 2016)**.** *Let* $\mathcal{C}$ *a smooth, decomposable and* deterministic *PC. MAP is computable in linear time (in the number of edges of* $\mathcal{C}$*).*

Proof. For a sum node S, we want to compute the following query

$$\max_{\mathrm{y}} \mathrm{S}(\mathrm{y}|\mathbf{x}) = \frac{1}{\mathrm{S}(\mathbf{x})} \max_{\mathrm{y}} \mathrm{S}(\mathrm{y}, \mathbf{x}) = \frac{1}{\mathrm{S}(\mathbf{x})} \max_{\mathrm{y}} \sum_{\mathrm{C} \in \mathrm{Ch(S)}} w_{\mathrm{S,C}}\, \mathrm{C}(\mathrm{y}, \mathbf{x}),$$

yet notice that for any assignment of $\mathbf{x}$ and y only one $\mathrm{C} \in \mathrm{Ch(S)}$ must have a nonnegative value by the definition of determinism, so we may replace the summation with a maximization over the children, giving

$$\max_{\mathrm{y}} \mathrm{S}(\mathrm{y}|\mathbf{x}) = \frac{1}{\mathrm{S}(\mathbf{x})} \max_{\mathrm{y}} \max_{\mathrm{C} \in \mathrm{Ch(S)}} w_{\mathrm{S,C}}\, \mathrm{C}(\mathrm{y}, \mathbf{x}) = \frac{1}{\mathrm{S}(\mathbf{x})} \max_{\mathrm{C} \in \mathrm{Ch(S)}} \max_{\mathrm{y}} w_{\mathrm{S,C}}\, \mathrm{C}(\mathrm{y}, \mathbf{x}).$$

For a product node P, we compute

$$\max_{\mathrm{y}} \mathrm{P}(\mathrm{y}|\mathbf{x}) = \frac{1}{\mathrm{P}(\mathbf{x})} \max_{\mathrm{y}} \mathrm{P}(\mathrm{y}, \mathbf{x}) = \frac{1}{\mathrm{P}(\mathbf{x})} \max_{\mathrm{y}} \prod_{\mathrm{C} \in \mathrm{Ch(P)}} \mathrm{C}(\mathrm{y}, \mathbf{x}) = \frac{1}{\mathrm{P}(\mathbf{x})} \prod_{\mathrm{C} \in \mathrm{Ch(P)}} \max_{\mathrm{y}} \mathrm{C}(\mathrm{y}, \mathbf{x}).$$

This is equivalent to an inductive top-down pass where we maximize instead of sum until we reach all input nodes, in which case we simply maximize the supposedly tractable functions. Once these are computed, we unroll the induction, maximizing over all values. □

Theorem 2.3.1 (Y. Choi, Vergari, and Broeck, 2020)**.** *If* $\mathcal{C}$ *is a smooth and structure decomposable probabilistic circuit with vtree* $\mathcal{V}$*, and* $\mathcal{L}$ *a structure decomposable logic circuit also respecting* $\mathcal{V}$*, then* $\mathbb{E}_{\mathcal{C}}[\mathcal{L}]$ *is polynomial time computable (in the number of edges).*

Proof. For completeness, we show the proof of this claim as stated in Y. Choi, Vergari, and Broeck, 2020. We assume, without loss of generality, that the layers of both $\mathcal{C}$ and $\mathcal{L}$ are compatible, i.e. they both have the same number of layers and if the i-th layer of $\mathcal{C}$ is made out of sums (resp. products), then the i-th layer of $\mathcal{L}$ is made out of disjunctions (resp. conjunctions). The expectation $\mathbb{E}_{\mathcal{C}}\left[\mathcal{L}\right]$ has the following form when the root of $\mathcal{C}$ is a product

$$\begin{aligned}\mathbb{E}_{\mathcal{C}}\left[\mathcal{L}\right] &= \int \mathcal{C}(\mathrm{x})\mathcal{L}(\mathrm{x})\,\mathrm{dx} = \int \left(\mathcal{C}_p(\mathrm{x})\mathcal{C}_s(\mathrm{x})\right)\left(\mathcal{L}_p(\mathrm{x})\mathcal{L}_s(\mathrm{x})\right)\mathrm{dx} \\ &= \int \left(\mathcal{C}_p(\mathrm{x})\mathcal{L}_p(\mathrm{x})\right)\left(\mathcal{C}_s(\mathrm{x})\mathcal{L}_s(\mathrm{x})\right)\mathrm{dx} = \int \left(\mathcal{C}_p(\mathrm{x})\mathcal{L}_p(\mathrm{x})\right)\mathrm{dx}\int \left(\mathcal{C}_s(\mathrm{x})\mathcal{L}_s(\mathrm{x})\right)\mathrm{dx} \\ &= \mathbb{E}_{\mathcal{C}_p}\left[\mathcal{L}_p\right]\cdot\mathbb{E}_{\mathcal{C}_s}\left[\mathcal{L}_s\right],\end{aligned}$$

where the subscript p and s indicate the prime and sub of a node. When the root is a sum

$$\begin{aligned}\mathbb{E}_{\mathcal{C}}\left[\mathcal{L}\right] &= \int \mathcal{C}(\mathrm{x})\mathcal{L}(\mathrm{x})\,\mathrm{dx} = \int \left(\sum_{\mathrm{C}'\in\mathrm{Ch}(\mathcal{C})} w_{\mathcal{C},\mathrm{C}'}\,\mathrm{C}'(\mathrm{x})\right)\left(\sum_{\mathrm{C}''\in\mathrm{Ch}(\mathcal{L})}\mathrm{C}''(\mathrm{x})\right)\mathrm{dx} \\ &= \int \sum_{\mathrm{C}'\in\mathrm{Ch}(\mathcal{C})}\sum_{\mathrm{C}''\in\mathrm{Ch}(\mathcal{L})} w_{\mathcal{C},\mathrm{C}'}\cdot\mathrm{C}'(\mathrm{x})\cdot\mathrm{C}''(\mathrm{x})\,\mathrm{dx} = \sum_{\mathrm{C}'\in\mathrm{Ch}(\mathcal{C})}\sum_{\mathrm{C}''\in\mathrm{Ch}(\mathcal{L})} w_{\mathcal{C},\mathrm{C}'}\int \mathrm{C}'(\mathrm{x})\,\mathrm{C}''(\mathrm{x})\,\mathrm{dx} \\ &= \sum_{\mathrm{C}'\in\mathrm{Ch}(\mathcal{C})}\sum_{\mathrm{C}''\in\mathrm{Ch}(\mathcal{L})} w_{\mathcal{C},\mathrm{C}'}\cdot\mathbb{E}_{\mathrm{C}'}\left[\mathrm{C}''\right].\end{aligned}$$

Therefore, if expectation is tractable for input nodes, then expectation is tractable for the whole circuit. □

References

[Akers 1978] S. B. Akers. “Binary decision diagrams”. In: *IEEE Transactions on Computers* 27.6 (1978), pp. 509–516 (cit. on p. 8).

[Amer and Todorovic 2012] Mohamed R. Amer and Sinisa Todorovic. “Sum-product networks for modeling activities with stochastic structure”. In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012, pp. 1314–1321. doi: 10.1109/CVPR.2012.6247816 (cit. on p. 21).

[Amer and Todorovic 2016] Mohamed R. Amer and Sinisa Todorovic. “Sum product networks for activity recognition”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.4 (2016), pp. 800–813. doi: 10.1109/TPAMI.2015.2465955 (cit. on p. 21).

[Barwise 1982] Jon Barwise. *Handbook of Mathematical Logic*. 1982 (cit. on p. 8).

[Bergstra and Bengio 2012] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization”. In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305. url: http://jmlr.org/papers/v13/bergstra12a.html (cit. on p. 27).

[Boutilier *et al.* 1996] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. “Context-specific independence in bayesian networks”. In: *Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence*. UAI’96. Portland, OR: Morgan Kaufmann Publishers Inc., 1996, pp. 115–123. isbn: 155860412X (cit. on p. 38).

[Bryant 1986] Randal E. Bryant. “Graph-based algorithms for boolean function manipulation”. In: *IEEE Transactions on Computers* 35.8 (1986), pp. 677–691 (cit. on pp. 17, 18).

[Bueff *et al.* 2018] Andreas Bueff, Stefanie Speichert, and Vaishak Belle. “Tractable querying and learning in hybrid domains via sum-product networks”. In: *CoRR* abs/1807.05464 (2018). arXiv: 1807.05464. url: http://arxiv.org/abs/1807.05464 (cit. on p. 31).

[Cory J Butz *et al.* 2019] Cory J Butz, Jhonatan S Oliveira, André E dos Santos, and André L Teixeira. “Deep convolutional sum-product networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 3248–3255 (cit. on p. 21).

[Cory J. Butz, Oliveira, *et al.* 2018] Cory J. Butz, Jhonatan S. Oliveira, *et al.* "An empirical study of methods for spn learning and inference". In: *Proceedings of the Ninth International Conference on Probabilistic Graphical Models.* Ed. by Václav Kratochvíl and Milan Studený. Vol. 72. Proceedings of Machine Learning Research. PMLR, Nov. 2018, pp. 49–60. URL: https://proceedings.mlr.press/v72/butz18a.html (cit. on p. 31).

[Cory J. Butz, Santos, *et al.* 2018] Cory J. Butz, André E. dos Santos, Jhonatan S. Oliveira, and John Stavrinides. "Efficient examination of soil bacteria using probabilistic graphical models". In: *Recent Trends and Future Technology in Applied Intelligence.* Ed. by Malek Mouhoub, Samira Sadaoui, Otmane Ait Mohamed, and Moonis Ali. Cham: Springer International Publishing, 2018, pp. 315–326. ISBN: 978-3-319-92058-0 (cit. on p. 21).

[Chan and Darwiche 2006] Hei Chan and Adnan Darwiche. "On the robustness of most probable explanations". In: *Proceedings of the 22nd Conference in Uncertainty in Artificial Intelligence.* 2006 (cit. on p. 9).

[Cheng *et al.* 2014] Wei-Chen Cheng, Stanley Kok, Hoai Vu Pham, Hai Leong Chieu, and Kian Ming A. Chai. "Language modeling with sum-product networks". In: *Fifteenth Annual Conference of the International Speech Communication Association.* 2014 (cit. on pp. 3, 21).

[Chernikova *et al.* 2019] A. Chernikova, A. Oprea, C. Nita-Rotaru, and B. Kim. "Are self-driving cars secure? evasion attacks against deep neural networks for steering angle prediction". In: *2019 IEEE Security and Privacy Workshops.* 2019, pp. 132–137 (cit. on p. 1).

[A. Choi and Darwiche 2013] Arthur Choi and Adnan Darwiche. "Dynamic minimization of sentential decision diagrams". In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence.* 2013, pp. 187–194 (cit. on p. 40).

[A. Choi, Shen, *et al.* 2017] Arthur Choi, Yujia Shen, and Adnan Darwiche. "Tractability in structured probability spaces". In: *Advances in Neural Information Processing Systems.* Vol. 30. 2017, pp. 3477–3485 (cit. on p. 40).

[Y. Choi, Vergari, and Broeck 2020] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. "Probabilistic circuits: a unifying framework for tractable probabilistic models". In: (2020). In preparation (cit. on pp. 2, 20, 44).

[Y. Choi, Vergari, and Van den Broeck 2020] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. "Lecture notes: probabilistic circuits: representation and inference". In: (Feb. 2020). URL: http://starai.cs.ucla.edu/papers/LecNoAAAI20.pdf (cit. on pp. 1, 10, 43).

[Chow and C. Liu 1968] C. Chow and C. Liu. "Approximating discrete probability distributions with dependence trees". In: *IEEE Transactions on Information Theory* 14.3 (1968), pp. 462–467. DOI: 10.1109/TIT.1968.1054142 (cit. on pp. 37, 40).

[Conaty *et al.* 2017] Diarmaid Conaty, Cassio Polpo de Campos, and Denis Deratani Mauá. “Approximation complexity of maximum A posteriori inference in sum-product networks”. In: *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence.* 2017 (cit. on p. 12).

[Fabio G. Cozman 2000] Fabio G. Cozman. “Credal networks”. In: *Artificial Intelligence* 120.2 (2000), pp. 199–233. issn: 0004-3702. doi: https://doi.org/10.1016/S0004-3702(00)00029-1. url: https://www.sciencedirect.com/science/article/pii/S0004370200000291 (cit. on p. 8).

[Dang *et al.* 2020] Meihua Dang, Antonio Vergari, and Guy Van den Broeck. “Strudel: learning structured-decomposable probabilistic circuits”. In: *Proceedings of the 10th International Conference on Probabilistic Graphical Models.* PGM. 2020 (cit. on pp. 3, 37, 39).

[Darwiche 1999] Adnan Darwiche. “Compiling knowledge into decomposable negation normal form”. In: *IJCAI.* Vol. 99. 1999, pp. 284–289 (cit. on p. 17).

[Darwiche 2001] Adnan Darwiche. “Decomposable negation normal form”. In: *J. ACM* 48.4 (July 2001), pp. 608–647. issn: 0004-5411. doi: 10.1145/502090.502091. url: https://doi.org/10.1145/502090.502091 (cit. on p. 17).

[Darwiche 2003] Adnan Darwiche. “A differential approach to inference in bayesian networks”. In: *Journal of the ACM* 50.3 (2003), pp. 280–305 (cit. on pp. 2, 7).

[Darwiche 2011] Adnan Darwiche. “SDD: a new canonical representation of propositional knowledge bases”. In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence.* 2011, pp. 819–826 (cit. on pp. 8, 17).

[Darwiche 2020] Adnan Darwiche. “Three modern roles for logic in ai”. In: *Proceedings of the 39th Symposium on Principles of Database Systems (PODS).* 2020 (cit. on p. 18).

[Darwiche and Marquis 2002] Adnan Darwiche and Pierre Marquis. “A knowledge compilation map”. In: *J. Artif. Int. Res.* 17.1 (Sept. 2002), pp. 229–264. issn: 1076-9757 (cit. on pp. 11, 17, 18).

[Dasgupta and Freund 2008] Sanjoy Dasgupta and Yoav Freund. “Random projection trees and low dimensional manifolds”. In: *Proceedings of the fortieth annual ACM symposium on Theory of computing.* STOC. 2008, pp. 537–546 (cit. on p. 4).

[De Campos 2011] Cassio P. De Campos. “New complexity results for map in bayesian networks”. In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three.* IJCAI’11. Barcelona, Catalonia, Spain, 2011, pp. 2100–2106. isbn: 9781577355151 (cit. on p. 12).

[R. Dechter 1998] R. Dechter. “Bucket elimination: a unifying framework for probabilistic inference”. In: *Learning in Graphical Models*. Ed. by Michael I. Jordan. Dordrecht: Springer Netherlands, 1998, pp. 75–104. isbn: 978-94-011-5014-9. doi: 10.1007/978-94-011-5014-9_4. url: https://doi.org/10.1007/978-94-011-5014-9_4 (cit. on p. 2).

[Rina Dechter and Mateescu 2007] Rina Dechter and Robert Mateescu. “And/or search spaces for graphical models”. In: *Artificial Intelligence* 171.2 (2007), pp. 73–106. issn: 0004-3702. doi: https://doi.org/10.1016/j.artint.2006.11.003. url: https://www.sciencedirect.com/science/article/pii/S000437020600138X (cit. on p. 2).

[Dennis and Ventura 2015] Aaron Dennis and Dan Ventura. “Greedy structure search for sum-product networks”. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. 2015, pp. 932–938 (cit. on p. 9).

[Dennis and Ventura 2017] Aaron Dennis and Dan Ventura. “Autoencoder-enhanced sum-product networks”. In: *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2017, pp. 1041–1044. doi: 10.1109/ICMLA.2017.00-13 (cit. on p. 21).

[Di Mauro *et al.* 2017] Nicola Di Mauro, Floriana Esposito, Fabrizio G. Ventola, and Antonio Vergari. “Alternative variable splitting methods to learn sum-product networks”. In: *AI*IA 2017 Advances in Artificial Intelligence*. Ed. by Floriana Esposito, Roberto Basili, Stefano Ferilli, and Francesca A. Lisi. Cham: Springer International Publishing, 2017, pp. 334–346. isbn: 978-3-319-70169-1 (cit. on pp. 3, 31).

[Edmonds 1965] Jack Edmonds. “Paths, trees, and flowers”. In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467. doi: 10.4153/CJM-1965-045-4 (cit. on pp. 33, 35).

[Feldmann and Foschini 2015] Andreas Emil Feldmann and Luca Foschini. “Balanced partitions of trees and applications”. In: *Algorithmica* 71.2 (Feb. 2015), pp. 354–376. issn: 0178-4617. doi: 10.1007/s00453-013-9802-3. url: https://doi.org/10.1007/s00453-013-9802-3 (cit. on p. 30).

[Freund *et al.* 2008] Yoav Freund, Sanjoy Dasgupta, Mayank Kabra, and Nakul Verma. “Learning the structure of manifolds using random projections”. In: *Advances in Neural Information Processing Systems*. Vol. 20. NeurIPS. 2008 (cit. on p. 4).

[A. Friesen and P. Domingos 2016] Abram Friesen and Pedro Domingos. “The sum-product theorem: a foundation for learning tractable models”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 2016, pp. 1909–1918. url: https://proceedings.mlr.press/v48/friesen16.html (cit. on p. 8).

[Abram L Friesen and P. Domingos 2017] Abram L Friesen and Pedro Domingos. "Unifying sum-product networks and submodular fields". In: *Proceedings of the Workshop on Principled Approaches to Deep Learning at ICML*. 2017 (cit. on p. 21).

[Abram L Friesen and P. M. Domingos 2018] Abram L Friesen and Pedro M Domingos. "Submodular field grammars: representation, inference, and application to image parsing". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio *et al.* Vol. 31. Curran Associates, Inc., 2018. url: https://proceedings.neurips.cc/paper/2018/file/c5866e93cab1776890fe343c9e7063fb-Paper.pdf (cit. on p. 21).

[Abram L. Friesen and P. Domingos 2015] Abram L. Friesen and Pedro Domingos. "Recursive decomposition for nonconvex optimization". In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, 2015, pp. 253–259. isbn: 9781577357384 (cit. on pp. 8, 21).

[Garey and Johnson 1990] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman Co., 1990. isbn: 0716710455 (cit. on p. 32).

[R. Geh and D. Mauá 2019] Renato Geh and Denis Mauá. "End-to-end imitation learning of lane following policies using sum-product networks". In: *Anais do XVI Encontro Nacional de Inteligência Artificial e Computacional*. Salvador: SBC, 2019, pp. 297–308. doi: 10.5753/eniac.2019.9292. url: https://sol.sbc.org.br/index.php/eniac/article/view/9292 (cit. on p. 21).

[R. L. Geh and Denis Deratani Mauá 2021] Renato Lui Geh and Denis Deratani Mauá. "Learning probabilistic sentential decision diagrams under logic constraints by sampling and averaging". In: *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. Ed. by Cassio de Campos and Marloes H. Maathuis. Vol. 161. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 2039–2049. url: https://proceedings.mlr.press/v161/geh21a.html (cit. on pp. 3, 4).

[Gens and P. Domingos 2012] Robert Gens and Pedro Domingos. "Discriminative learning of sum-product networks". In: *Advances in Neural Information Processing Systems 25*. NIPS, 2012, pp. 3239–3247 (cit. on pp. 3, 21).

[Gens and P. Domingos 2013] Robert Gens and Pedro Domingos. "Learning the structure of sum-product networks". In: *Proceedings of the 30th International Conference on Machine Learning*. ICML. 2013, pp. 873–880 (cit. on pp. 3, 24, 25).

[Gogic *et al.* 1995] Goran Gogic, Henry Kautz, Christos Papadimitriou, and Bart Selman. "The comparative linguistics of knowledge representation". In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'95. Montreal, Quebec, Canada, 1995, pp. 862–869. isbn: 1558603638 (cit. on p. 18).

[Goodfellow *et al.* 2014] Ian Goodfellow *et al.* “Generative adversarial nets”. In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., 2014, pp. 2672–2680. url: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf (cit. on p. 2).

[Gritsenko *et al.* 2019] Alexey A. Gritsenko, Jasper Snoek, and Tim Salimans. “On the relationship between normalising flows and variational- and denoising autoencoders”. In: *Deep Generative Models for Highly Structured Data, ICLR 2019 Workshop, New Orleans, Louisiana, United States, May 6, 2019*. OpenReview.net, 2019. url: https://openreview.net/forum?id=HklKEUUY%5C_E (cit. on p. 2).

[Guo *et al.* 2017] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. “On calibration of modern neural networks”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1321–1330 (cit. on p. 1).

[Jaeger 2004] Manfred Jaeger. “Probabilistic decision graphs-combining verification and ai techniques for probabilistic inference”. In: *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* 12.1 supp (Jan. 2004), pp. 19–42. issn: 0218-4885. doi: 10.1142/S0218488504002564. url: https://doi.org/10.1142/S0218488504002564 (cit. on p. 2).

[Jaini, Ghose, *et al.* 2018] Priyank Jaini, Amur Ghose, and Pascal Poupart. “Prometheus: directly learning acyclic directed graph structures for sum-product networks”. In: *International Conference on Probabilistic Graphical Models*. PGM. 2018, pp. 181–192 (cit. on pp. 3, 27, 28, 30).

[Jaini, Poupart, *et al.* 2018] Priyank Jaini, Pascal Poupart, and Yaoliang Yu. “Deep homogeneous mixture models: representation, separation, and approximation”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio *et al.* Vol. 31. Curran Associates, Inc., 2018. url: https://proceedings.neurips.cc/paper/2018/file/c5f5c23be1b71adb51ea9dc8e9d444a8-Paper.pdf (cit. on p. 41).

[Karypis and Kumar 1998] George Karypis and Vipin Kumar. “A fast and high quality multilevel scheme for partitioning irregular graphs”. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. doi: 10.1137/S1064827595287997 (cit. on pp. 32, 35).

[Khosravi *et al.* 2019] Pasha Khosravi, Yitao Liang, YooJung Choi, and Guy Van den Broeck. “What to expect of classifiers? reasoning about logistic regression with missing features”. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 2716–2724. doi: 10.24963/ijcai.2019/377. url: https://doi.org/10.24963/ijcai.2019/377 (cit. on p. 2).

[Kingma and Welling 2014] Diederik P. Kingma and Max Welling. “Auto-encoding variational bayes”. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings.* Ed. by Yoshua Bengio and Yann LeCun. 2014. url: http://arxiv.org/abs/1312.6114 (cit. on p. 2).

[Kisa *et al.* 2014] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. “Probabilistic sentential decision diagrams”. In: *Knowledge Representation and Reasoning Conference* (2014) (cit. on pp. 2, 19).

[Koller and Friedman 2009] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning.* The MIT Press, 2009. isbn: 0262013193 (cit. on p. 2).

[Kolmogorov 2009] Vladimir Kolmogorov. “Blossom v: a new implementation of a minimum cost perfect matching algorithm”. In: *Mathematical Programming Computation* 1.1 (July 2009), pp. 43–67. issn: 1867-2957. doi: 10.1007/s12532-009-0002-8. url: https://doi.org/10.1007/s12532-009-0002-8 (cit. on pp. 33, 35).

[Liang, Bekker, *et al.* 2017] Yitao Liang, Jessa Bekker, and Guy Van den Broeck. “Learning the structure of probabilistic sentential decision diagrams”. In: *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence.* 2017 (cit. on pp. 3, 31, 32, 34, 36).

[Liang and Van den Broeck 2019] Yitao Liang and Guy Van den Broeck. “Learning logistic circuits”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 4277–4286. doi: 10.1609/aaai.v33i01.33014277. url: https://ojs.aaai.org/index.php/AAAI/article/view/4336 (cit. on p. 8).

[Lippe and Gavves 2021] Phillip Lippe and Efstratios Gavves. “Categorical normalizing flows via continuous transformations”. In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.* OpenReview.net, 2021. url: https://openreview.net/forum?id=-GLNZeVDuik (cit. on p. 2).

[Y. Liu and Luo 2019] Yang Liu and Tiejian Luo. “The optimization of sum-product network structure learning”. In: *Journal of Visual Communication and Image Representation* 60 (2019), pp. 391–397. issn: 1047-3203. doi: https://doi.org/10.1016/j.jvcir.2019.02.012. url: https://www.sciencedirect.com/science/article/pii/S1047320319300653 (cit. on p. 31).

[Llerena and Deratani Mauá 2017] Julissa Villanueva Llerena and Denis Deratani Mauá. “On using sum-product networks for multi-label classification”. In: *2017 Brazilian Conference on Intelligent Systems (BRACIS).* 2017, pp. 25–30. doi: 10.1109/BRACIS.2017.34 (cit. on p. 21).

[Lowd and Rooshenas 2013] Daniel Lowd and Amirmohammad Rooshenas. "Learning markov networks with arithmetic circuits". In: *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Carlos M. Carvalho and Pradeep Ravikumar. Vol. 31. Proceedings of Machine Learning Research. Scottsdale, Arizona, USA: PMLR, Apr. 2013, pp. 406–414. url: https://proceedings.mlr.press/v31/lowd13a.html (cit. on p. 27).

[Martens and Medabalimi 2014] James Martens and Venkatesh Medabalimi. "On the expressive efficiency of sum product networks". In: *CoRR* abs/1411.7717 (2014). arXiv: 1411.7717. url: http://arxiv.org/abs/1411.7717 (cit. on pp. 7, 26).

[Denis D. Mauá *et al.* 2017] Denis D. Mauá, Fabio G. Cozman, Diarmaid Conaty, and Cassio P. Campos. "Credal sum-product networks". In: *Proceedings of the Tenth International Symposium on Imprecise Probability: Theories and Applications*. Ed. by Alessandro Antonucci, Giorgio Corani, Inés Couso, and Sébastien Destercke. Vol. 62. Proceedings of Machine Learning Research. PMLR, Oct. 2017, pp. 205–216. url: https://proceedings.mlr.press/v62/mau%C3%A117a.html (cit. on p. 8).

[Mauro *et al.* 2021] Nicola Di Mauro, Gennaro Gala, Marco Iannotta, and Teresa M. A. Basile. "Random probabilistic circuits". In: *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. 2021 (cit. on p. 3).

[Mei *et al.* 2018] Jun Mei, Yong Jiang, and Kewei Tu. "Maximum a posteriori inference in sum-product networks". In: *AAAI Conference on Artificial Intelligence*. 2018 (cit. on p. 12).

[Melibari, Poupart, and Doshi 2016] Mazen Melibari, Pascal Poupart, and Prashant Doshi. "Sum-product-max networks for tractable decision making". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. IJCAI'16. New York, New York, USA: AAAI Press, 2016, pp. 1846–1852. isbn: 9781577357704 (cit. on p. 8).

[Melibari, Poupart, Doshi, and Trimponias 2016] Mazen Melibari, Pascal Poupart, Prashant Doshi, and George Trimponias. "Dynamic sum product networks for tractable inference on sequence data". In: *Probabilistic Graphical Models*. Vol. 52. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 345–355 (cit. on p. 21).

[Molina, Natarajan, *et al.* 2017] Alejandro Molina, Sriraam Natarajan, and Kristian Kersting. "Poisson sum-product networks: a deep architecture for tractable multivariate poisson distributions". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31.1 (Feb. 2017). url: https://ojs.aaai.org/index.php/AAAI/article/view/10844 (cit. on p. 31).

[Molina, Vergari, *et al.* 2018] Alejandro Molina, Antonio Vergari, *et al.* "Mixed sum-product networks: a deep architecture for hybrid domains". In: (2018). url: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16865 (cit. on p. 31).

[Nath and P. Domingos 2015] Aniruddh Nath and Pedro Domingos. “Learning relational sum-product networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 29.1 (Feb. 2015). url: https://ojs.aaai.org/index.php/AAAI/article/view/9538 (cit. on p. 31).

[Nath and P. M. Domingos 2016] Aniruddh Nath and Pedro M Domingos. “Learning tractable probabilistic models for fault localization”. In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016 (cit. on pp. 3, 21).

[Nourani *et al.* 2020] Mahsan Nourani *et al. Don’t Explain without Verifying Veracity: An Evaluation of Explainable AI with Video Activity Recognition*. 2020. arXiv: 2005.02335 [cs.HC] (cit. on p. 21).

[Olascoaga *et al.* 2019] Laura Isabel Galindez Olascoaga, Wannes Meert, Nimish Shah, Marian Verhelst, and Guy Van den Broeck. “Towards hardware-aware tractable learning of probabilistic models”. In: *NeurIPS*. 2019, pp. 13726–13736 (cit. on p. 21).

[Ovadia *et al.* 2019] Yaniv Ovadia *et al.* “Can you trust your model’s uncertainty? evaluating predictive uncertainty under dataset shift”. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 2019 (cit. on p. 1).

[Oztok and Darwiche 2015] Umut Oztok and Adnan Darwiche. “A top-down compiler for sentential decision diagrams”. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. 2015, pp. 3141–3148 (cit. on p. 40).

[Papadimitriou 1994] C.H. Papadimitriou. *Computational Complexity*. Theoretical computer science. Addison-Wesley, 1994. isbn: 9780201530827 (cit. on p. 18).

[Papamakarios *et al.* 2021] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. “Normalizing flows for probabilistic modeling and inference”. In: *Journal of Machine Learning Research* 22.57 (2021), pp. 1–64. url: http://jmlr.org/papers/v22/19-1028.html (cit. on p. 2).

[Peharz, Gens, and P. Domingos 2014] Robert Peharz, Robert Gens, and Pedro Domingos. “Learning selective sum-product networks”. In: *Workshop on Learning Tractable Probabilistic Models*. 2014 (cit. on p. 9).

[Peharz, Gens, Pernkopf, *et al.* 2016] Robert Peharz, Robert Gens, Franz Pernkopf, and Pedro Domingos. “On the latent variable interpretation in sum-product networks”. In: *IEEE transactions on pattern analysis and machine intelligence* 39.10 (2016), pp. 2030–2044 (cit. on pp. 6, 12, 44).

[Peharz, Kapeller, *et al.* 2014] Robert Peharz, Georg Kapeller, Pejman Mowlaee, and Franz Pernkopf. “Modeling speech with sum-product networks: application to bandwidth extension”. In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2014, pp. 3699–3703 (cit. on p. 21).

[Peharz, Lang, *et al.* 2020] Robert Peharz, Steven Lang, *et al.* “Einsum networks: fast and scalable learning of tractable probabilistic circuits”. In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 7563–7574. url: https://proceedings.mlr.press/v119/peharz20a.html (cit. on pp. 3, 8, 21).

[Peharz, Tschiatschek, *et al.* 2015] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. “On theoretical properties of sum-product networks”. In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*. 2015, pp. 744–752 (cit. on pp. 7, 11).

[Peharz, Vergari, *et al.* 2020] Robert Peharz, Antonio Vergari, *et al.* “Random sum-product networks: a simple and effective approach to probabilistic deep learning”. In: *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*. Ed. by Ryan P. Adams and Vibhav Gogate. Vol. 115. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 334–344. url: https://proceedings.mlr.press/v115/peharz20a.html (cit. on pp. 3, 21).

[Pevný *et al.* 2020] Tomáš Pevný, Václav Smídl, Martin Trapp, Ondřej Poláček, and Tomáš Oberhuber. “Sum-product-transform networks: exploiting symmetries using invertible transformations”. In: *Proceedings of the 10th International Conference on Probabilistic Graphical Models*. Ed. by Manfred Jaeger and Thomas Dyhre Nielsen. Vol. 138. Proceedings of Machine Learning Research. PMLR, Sept. 2020, pp. 341–352. url: https://proceedings.mlr.press/v138/pevny20a.html (cit. on p. 8).

[Poon and P. Domingos 2011] Hoifung Poon and Pedro Domingos. “Sum-product networks: a new deep architecture”. In: *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*. 2011, pp. 337–346 (cit. on pp. 2, 3, 6, 10, 21, 43).

[Pronobis *et al.* 2017] Andrzej Pronobis, Francesco Riccio, and Rajesh PN Rao. “Deep spatial affordance hierarchy: spatial knowledge representation for planning in large-scale environments”. In: *ICAPS 2017 Workshop on Planning and Robotics*. 2017 (cit. on p. 21).

[Rahman and Gogate 2016] Tahrima Rahman and Vibhav Gogate. “Merging strategies for sum-product networks: from trees to graphs”. In: *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence*. UAI’16. Jersey City, New Jersey, USA: AUAI Press, 2016, pp. 617–626. isbn: 9780996643115 (cit. on p. 31).

[Rahman, Kothalkar, *et al.* 2014] Tahrima Rahman, Prasanna Kothalkar, and Vibhav Gogate. “Cutset networks: a simple, tractable, and scalable approach for improving the accuracy of chow-liu trees”. In: *Proceedings of the 2014th European Conference on Machine Learning and Knowledge Discovery in Databases*. 2014, pp. 630–645 (cit. on p. 2).

[Rashwan *et al.* 2018] Abdullah Rashwan, Pascal Poupart, and Chen Zhitang. "Discriminative training of sum-product networks by extended baum-welch". In: *Proceedings of the Ninth International Conference on Probabilistic Graphical Models*. Vol. 72. Proceedings of Machine Learning Research. 2018, pp. 356–367 (cit. on p. 3).

[Ratajczak *et al.* 2014] Martin Ratajczak, Sebastian Tschiatschek, and Franz Pernkopf. "Sum-product networks for structured prediction: context-specific deep conditional random fields". In: *International Conference on Machine Learning (ICML) Workshop on Learning Tractable Probabilistic Models Workshop*. 2014 (cit. on p. 21).

[Ratajczak *et al.* 2018] Martin Ratajczak, Sebastian Tschiatschek, and Franz Pernkopf. "Sum-product networks for sequence labeling". In: *arXiv preprint arXiv:1807.02324* (2018) (cit. on p. 21).

[Rathke *et al.* 2017] Fabian Rathke, Mattia Desana, and Christoph Schnörr. "Locally adaptive probabilistic models for global segmentation of pathological oct scans". In: *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer. 2017, pp. 177–184 (cit. on p. 21).

[Rezende and Mohamed 2015] Danilo Rezende and Shakir Mohamed. "Variational inference with normalizing flows". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 1530–1538. url: https://proceedings.mlr.press/v37/rezende15.html (cit. on p. 2).

[Rolfe 2017] Jason Tyler Rolfe. "Discrete variational autoencoders". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. url: https://openreview.net/forum?id=ryMxXPFex (cit. on p. 2).

[Rooshenas and Lowd 2014] Amirmohammad Rooshenas and Daniel Lowd. "Learning sum-product networks with direct and indirect variable interactions". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, June 2014, pp. 710–718. url: https://proceedings.mlr.press/v32/rooshenas14.html (cit. on pp. 26, 27).

[Rooshenas and Lowd 2016] Amirmohammad Rooshenas and Daniel Lowd. "Discriminative structure learning of arithmetic circuits". In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. Ed. by Arthur Gretton and Christian C. Robert. Vol. 51. Proceedings of Machine Learning Research. Cadiz, Spain: PMLR, Sept. 2016, pp. 1506–1514. url: https://proceedings.mlr.press/v51/rooshenas16.html (cit. on p. 3).

[Saad *et al.* 2021] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. “SPPL: probabilistic programming with fast exact symbolic inference”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Virtual, Canada: Association for Computing Machinery, 2021, pp. 804–819. isbn: 9781450383912. doi: 10.1145/3453483.3454078 (cit. on p. 21).

[Sguerra and F. G. Cozman 2016] B. M. Sguerra and F. G. Cozman. “Image classification using sum-product networks for autonomous flight of micro aerial vehicles”. In: *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*. 2016, pp. 139–144 (cit. on p. 21).

[Shah, Isabel Galindez Olascoaga, *et al.* 2019] Nimish Shah, Laura Isabel Galindez Olascoaga, Wannes Meert, and Marian Verhelst. “Problp: a framework for low-precision probabilistic inference”. In: *DAC 2019*. ACM, 2019, p. 190. doi: 10.1145/3316781.3317885. url: https://doi.org/10.1145/3316781.3317885 (cit. on p. 21).

[Shah, Olascoaga, Meert, *et al.* 2020] Nimish Shah, Laura Isabel Galindez Olascoaga, Wannes Meert, and Marian Verhelst. “Acceleration of probabilistic reasoning through custom processor architecture”. In: *DATE*. IEEE, 2020, pp. 322–325 (cit. on p. 21).

[Shah, Olascoaga, S. Zhao, *et al.* 2021] Nimish Shah, Laura Isabel Galindez Olascoaga, Shirui Zhao, Wannes Meert, and Marian Verhelst. “Piu: a 248gops/w stream-based processor for irregular probabilistic inference networks using precision-scalable posit arithmetic in 28nm”. In: *2021 IEEE International Solid- State Circuits Conference (ISSCC)*. Vol. 64. 2021, pp. 150–152. doi: 10.1109/ISSCC42613.2021.9366061 (cit. on p. 21).

[Shao *et al.* 2020] Xiaoting Shao *et al.* “Conditional sum-product networks: imposing structure on deep probabilistic architectures”. In: *Proceedings of the 10th International Conference on Probabilistic Graphical Models*. Ed. by Manfred Jaeger and Thomas Dyhre Nielsen. Vol. 138. Proceedings of Machine Learning Research. PMLR, Sept. 2020, pp. 401–412. url: https://proceedings.mlr.press/v138/shao20a.html (cit. on p. 3).

[Sharir and Shashua 2018] Or Sharir and Amnon Shashua. “Sum-product-quotient networks”. In: *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*. Ed. by Amos Storkey and Fernando Perez-Cruz. Vol. 84. Proceedings of Machine Learning Research. PMLR, Sept. 2018, pp. 529–537. url: https://proceedings.mlr.press/v84/sharir18a.html (cit. on p. 8).

[Shen *et al.* 2017] Yujia Shen, Arthur Choi, and Adnan Darwiche. “A tractable probabilistic model for subset selection”. In: *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence*. 2017 (cit. on p. 40).

[Sommer *et al.* 2018] Lukas Sommer *et al.* “Automatic mapping of the sum-product network inference problem to fpga-based accelerators”. In: *ICCD*. IEEE Computer Society, 2018, pp. 350–357 (cit. on p. 21).

[Stuhlm"uller and Goodman 2012] Andreas Stuhlm"uller and Noah D Goodman. “A dynamic programming algorithm for inference in recursive probabilistic programs”. In: *Workshop of Statistical and Relational AI (StarAI)* (2012) (cit. on p. 21).

[Su *et al.* 2019] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. “One pixel attack for fooling deep neural networks”. In: *IEEE Transactions on Evolutionary Computation* 23.5 (2019), pp. 828–841 (cit. on p. 1).

[Szegedy *et al.* 2013] Christian Szegedy *et al.* “Intriguing properties of neural networks”. In: *arXiv preprint arXiv:1312.6199* (2013) (cit. on p. 1).

[Vahdat, Andriyash, *et al.* 2018] Arash Vahdat, Evgeny Andriyash, and William Macready. “Dvae#: discrete variational autoencoders with relaxed boltzmann priors”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio *et al.* Vol. 31. Curran Associates, Inc., 2018. url: https://proceedings.neurips.cc/paper/2018/file/9f53d83ec0691550f7d2507d57f4f5a2-Paper.pdf (cit. on p. 2).

[Vahdat, Macready, *et al.* 2018] Arash Vahdat, William G. Macready, Zhengbing Bian, Amir Khoshaman, and Evgeny Andriyash. “DVAE++: discrete variational autoencoders with overlapping transformations”. In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 5042–5051. url: http://proceedings.mlr.press/v80/vahdat18a.html (cit. on p. 2).

[Vergari, Y. Choi, *et al.* 2021] Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. “A compositional atlas of tractable circuit operations: from simple transformations to complex information-theoretic queries”. In: *CoRR* abs/2102.06137 (2021). arXiv: 2102.06137 (cit. on pp. 8, 10, 16, 43).

[Vergari, Mauro, *et al.* 2015] Antonio Vergari, Nicola Di Mauro, and Floriana Esposito. “Simplifying, regularizing and strengthening sum-product network structure learning”. In: *ECML/PKDD*. 2015 (cit. on pp. 3, 31).

[Wachter and Haenni 2006] Michael Wachter and Rolf Haenni. “Propositional dags: a new graph-based language for representing boolean functions”. In: *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning*. KR’06. Lake District, UK, 2006, pp. 277–285. isbn: 9781577352716 (cit. on p. 17).

[J. Wang and G. Wang 2018] Jinghua Wang and Gang Wang. “Hierarchical spatial sum–product networks for action recognition in still images”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 28.1 (2018), pp. 90–100. doi: 10.1109/TCSVT.2016.2586853 (cit. on p. 21).

[Wei *et al.* 2018] Wenqi Wei *et al.* “Adversarial examples in deep learning: characterization and divergence”. In: *arXiv preprint arXiv:1807.00051* (2018) (cit. on p. 1).

[Yu 2020] Ronald Yu. *A Tutorial on VAEs: From Bayes’ Rule to Lossless Compression.* 2020. arXiv: 2006.10273 [cs.LG] (cit. on p. 2).

[Yuan *et al.* 2016] Zehuan Yuan *et al.* “Modeling spatial layout for scene image understanding via a novel multiscale sum-product network”. In: *Expert Syst. Appl.* 63.C (Nov. 2016), pp. 231–240. issn: 0957-4174. doi: 10.1016/j.eswa.2016.07.015. url: https://doi.org/10.1016/j.eswa.2016.07.015 (cit. on p. 21).

[Zhang *et al.* 2021] Honghua Zhang, Brendan Juba, and Guy Van Den Broeck. “Probabilistic generating circuits”. In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 12447–12457. url: https://proceedings.mlr.press/v139/zhang21i.html (cit. on p. 8).

[H. Zhao, Adel, *et al.* 2016] Han Zhao, Tameem Adel, Geoff Gordon, and Brandon Amos. “Collapsed variational inference for sum-product networks”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 2016, pp. 1310–1318. url: http://proceedings.mlr.press/v48/zhaoa16.html (cit. on p. 9).

[H. Zhao, Melibari, *et al.* 2015] Han Zhao, Mazen Melibari, and Pascal Poupart. “On the relationship between sum-product networks and Bayesian networks”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. Proceedings of Machine Learning Research. 2015, pp. 116–124 (cit. on p. 9).

[Zheng *et al.* 2018] Kaiyu Zheng, Andrzej Pronobis, and Rajesh PN Rao. “Learning graph-structured sum-product networks for probabilistic semantic maps”. In: *Thirty-Second AAAI Conference on Artificial Intelligence.* 2018 (cit. on p. 21).

[Ziegler and Rush 2019] Zachary M. Ziegler and Alexander M. Rush. “Latent normalizing flows for discrete sequences”. In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 7673–7682. url: http://proceedings.mlr.press/v97/ziegler19a.html (cit. on p. 2).