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

This is the original version of the thesis prepared by candidate Renato Lui Geh, as submitted to the Examining Committee.

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.

A crescente popularidade de modelos gerativos, assim como o aumento da demanda por modelos que produzam inferência exata e de forma flexível vêm motivando a comunidade de aprendizado de máquina a procurar por modelos probabilísticos que sejam tanto expressivos quanto tratáveis. Circuitos probabilísticos (PC, do inglês *probabilistic circuit*) são uma família de modelos probabilísticos tratáveis capazes de responder uma vasta gama de consultas de forma exata e em tempo polinomial. Sua sintaxe operacional concretizada por um grafo computacional junto a sua semântica probabilística possibilitam que seus parâmetros sejam estimados pelas eficientes e altamente escaláveis técnicas utilizadas em aprendizado profundo. Notavelmente, tratabilidade está fortemente ligada às restrições impostas no grafo subjacente: ao impor certas restrições gráficas, consultas como probabilidade marginal, *maximum a posteriori* ou entropia tornam-se computáveis em tempo linear, ao mesmo tempo retendo alta expressividade. Enquanto que inferência é, de forma geral, descomplicado, a tarefa de aprender PCs de forma que os circuitos tanto observem as restrições estruturais necessárias quanto explorem sua expressividade tem se provado um desafio. O atual estado-da-arte para algoritmos de aprendizado estrutural de PCs pode ser grosseiramente dividido em três categorias principais. A maior parte dos algoritmos de aprendizado buscam gerar um circuito em formato de árvore através de decomposições recursivas nos dados, na maior parte das vezes através de algoritmos de *clustering* e custosos testes de independência estatística, o que pode tornar o processo inviável em altas dimensões. Alternativamente, outras técnicas envolvem construir uma complexa rede por meio de métodos incrementais iterativos que preservem uma certa estrutura do grafo. Além desta técnica depender de um circuito inicial suficientemente expressivo, tais métodos podem demorar vários minutos por iteração, e muitas iterações até que haja uma melhora visível. Por último, outras alternativas envolvem gerar aleatoriamente um circuito probabilístico através de algum critério. Apesar desta técnica normalmente gerar modelos menos performativos quando comparados com outros métodos, PCs aleatórios são ordens de grandeza mais eficiente em relação a tempo de execução. Com isso em mente, esta dissertação busca propor algoritmos de aprendizado estrutural de PCs que sejam rápidos e escaláveis através de duas lentes distintas: de um ponto de vista lógico, buscamos construir um PC sob variáveis binárias altamente estruturado que tome conhecimento certo na forma de restrições lógicas, e traduza-as em um circuito probabilístico de forma escalável; por meio da ótica de busca por estruturas guiada por dados, nós propomos construir PCs de forma hierárquica por meio de hiperplanos aleatórios. Nós mostramos, de forma empírica, que ambas são competitivas comparadas ao estado-da-arte, e que podemos melhorar sua performance por meio de estratégias simples de *ensembles*.

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

List of Lists

List of Symbols

X, Y, Z, ...	Random variables or propositional variables
x, y, z, ...	Assignments of random or propositional variables
$\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$, ...	Sets of variables
$\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, ...	Sets of assignments
$\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, ...	Sample space of random variables
$\perp\!\!\!\perp$	Statistical independence
$\langle f \rangle$	Semantics of Boolean formula f
$f \equiv g$	Equivalence between Boolean formulae f and g (i.e. $\langle f \rangle = \langle g \rangle$)
$[a..b]$	Integer set $\{a, a+1, \dots, b\} \subset \mathbb{Z}$ for $b \geq a$
$[b]$	Integer set $\{1, 2, \dots, b\} \subset \mathbb{Z}$ for $b > 0$
$[\![\phi]\!]$	Iverson bracket (i.e. 1 if ϕ is true, 0 otherwise)
N, S, P, L	Graph nodes
N, **S**, **P**, **L**	Sets of nodes
Ch(N)	Set of all children of node N
Pa(N)	Set of all parents of node N
Desc(N)	Set of all descendants of node N
Sc(N)	Scope of node N
Inputs($\mathcal{C}$)	Set of input nodes of circuit $\mathcal{C}$
$\mathcal{N}$	Gaussian distribution
$v^{\leftarrow}$, $v^{\rightarrow}$	Left and right children of vtree node v

List of Figures

List of Tables

List of Algorithms

List of Examples

List of Remarks

Contents

1 Introduction

Machine learning has become, without a doubt, ubiquitous in today's technology. From self-driving vehicles to bioinformatics, data-driven models have taken the center stage when it comes to state-of-the-art prediction and modeling of real-world tasks (Grigorescu *et al.*, 2020; Lan *et al.*, 2018; Li *et al.*, 2019; Khan and Yairi, 2018; Sezer *et al.*, 2020). In many of these scenarios, such as system diagnosis of automated power plants or real-time translation, timely decisions are crucial to the well-functioning of often critical systems (Enshaei and Naderkhani, 2019; Niehues *et al.*, 2018). In others, agents must be either precise or come with approximation guarantees of their estimation, as is the case of decision making in healthcare, finance or other sensitive data (Lou *et al.*, 2019). Even more crucially, these predictive models often need to perform complex reasoning tasks over hundreds if not thousands of variables, sometimes under known constraints dependent on domain (Xu *et al.*, 2018; Pogancic *et al.*, 2020; Wong *et al.*, 2012; Lu *et al.*, 2013); and yet prove expressive enough to learn from the possibly enormous quantity of data available.

Fulfilling these requirements are Probabilistic Circuits (PCs), a class of probabilistic models distinctly 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. More concretely, these structural and operational constraints lead to sufficient conditions for the polynomial time computation of complex exact queries, providing a powerful toolbox for probabilistic reasoning. 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. These specific families of PCs have been known throughout literature by different names: Arithmetic Circuits (ACs, Darwiche, 2003), Sum-Product Networks (SPNs, Poon and P. Domingos, 2011), Cutset Networks (CNets, Rahman, Kothalkar, *et al.*, 2014), Probabilistic Sentential Decision Diagrams (PSDDs, Kisa *et al.*, 2014), Probabilistic Decision Graphs (PDGs, Jaeger, 2004) and And/Or-Graphs (AOGs, Dechter and Mateescu, 2007) are some of the more well-known formalisms caught under the PC framework.

While inference in PCs is usually straightforward, learning their structure so that they both obey the needed structural restrictions and prove expressive enough for the task at hand has proven to be a challenging process. Even so, many techniques have

been proposed in the last decade, with encouraging results. These techniques however, often do not scale up well when faced with higher dimensional data, as they require either carefully handcrafted architectures (Poon and P. Domingos, 2011; Cheng *et al.*, 2014; Nath and P. M. Domingos, 2016) or usually involve 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). 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, Vergari, *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, 2021b; Peharz, Lang, *et al.*, 2020). Apart from the scalability side of random structure generation, usual structure learning algorithms often require an extensive 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 time consuming and tedious, especially if the goal is to analyze and infer from large data, and not to achieve top tier performance on benchmark datasets.

The objectives of this dissertation are two-fold. First, we seek to provide a concise literature review on structure learning algorithms for probabilistic circuits, describing a few of the most popular techniques for learning PCs. This particular contribution comes from a need for a systematic comparison of the large body of techniques that have been developed so far, each with different trade-offs of computational cost, accuracy and structural properties. To this end, we compare and categorize structure learning algorithms with respect to time and memory requirements, efficient queries enabled by the learned model and the overall pros and cons brought by each.

Second, we present two new structural learning algorithms for PCs whose focus are on efficiently learning from complex domains; we show that our approaches are scalable and competitive against the state-of-the-art. They both take advantage of random circuit generation to quickly construct PCs with little to no need for hyperparameter tuning. The first approaches the problem of circuit construction through the lenses of symbolic machine learning, effective for constructing PCs from highly structured binary data where prior knowledge of the domain can be embedded into the model as a means to define its support. We describe a randomized algorithm capable of taking background knowledge in the form of logical constraints and produce PC samples whose support are relaxations of the given logical formula. By taking advantage of known expert knowledge, the resulting PC attributes mass to the more relevant portions of the sample space, resulting in a more performant model, especially in data scarce regimes with abundant certain knowledge. We show how this approach scales well to complex formulae and large amounts of data. From a pure data point of view, we present a simple yet effective way to learn the structure of probabilistic circuits by exploiting a recently discovered connection between PCs and

random forests (Correia *et al.*, 2020). We revisit a well-known inductive method of hierarchically stacking oblique projections to learn decision trees (Freund *et al.*, 2008; Dasgupta and Freund, 2008) and transplant them into the context of probabilistic circuits, adapting one of the more popular techniques for constructing PCs and proposing a simpler and faster random version based on random oblique projections. We found that our approach produced fairly competitive PC structures in a fraction of the time.

1.1 Dissertation Outline

We begin Chapter 2 by formally defining probabilistic circuits (Section 2.1), 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. This is followed with a description on how to algorithmically compute inference queries in PCs. We then list existing formalisms that may be viewed as instances of PCs, and what their structure entail in terms of inference power. We finish the chapter by looking at a particular class of PCs which allow the embedding of logical formulae within their support.

In Chapter 3, we address existing PC structure learning algorithms dividing them into three classes: divide-and-conquer learning (Section 3.1), incremental learning (Section 3.2) and random learning (Section 3.3). For each, we give a brief analysis on their complexity and discuss their advantages and disadvantages.

In Chapter 4 and Chapter 5, we propose the two scalable structural learning algorithms for probabilistic circuits that are especially suited for large data and fast deployment. The final chapter (Chapter 6) is dedicated to summarizing our research contributions and pointing to potential future work in learning PCs.

1.2 Contributions

Overall, our contributions throughout this dissertation address the following research topics.

A concise review of literature on structure learning of PCs

Chapter 3 is dedicated to an extensive review of some of the existing techniques for learning the structure of probabilistic circuits. We categorize them into three different classes and analyze each in terms of their time requirements. We describe them in detail and list insights on what seems to work and what could potentially be improved. Perhaps more importantly, we provide a birds-eye view of what each learning algorithm guarantees in terms of structural constraints, time requirement and number of hyperparameters needed during learning.

Scalably learning PCs directly from background knowledge

In R. L. Geh and Denis Deratani Mauá (2021b), 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 Chapter 4.

Using ensembles to strengthen consistency

The PC sampler given by R. L. Geh and Denis Deratani Mauá (2021b) 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 Chapter 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 Chapter 5 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, formalizing the many structural constraints that give sufficient conditions for tractable inference. In Section 2.3, we address PCs as knowledge bases, representing logic formulae through circuits.

Before we formally introduce probabilistic circuits, we start with a brief preliminary on notation and nomenclature. Call $\mathbf{X} = \{X_1, X_2, \dots, X_m\}$ a set of random variables (RVs); we denote by $\mathbf{x} = \{x_1, x_2, \dots, x_m\}$ the *assignment* of each value x_i to RV X_i. Let p be a probability distribution over variables $\mathbf{X}$; we use the notation $p(\mathbf{X} = \mathbf{x})$ to represent the probability of $\mathbf{X}$ taking values $\mathbf{x}$ according to p. On that note, if $\mathbf{X}$ is the set of all RVs of probability distribution p, then we say that $\mathbf{X}$ is the *scope* of p, here denoted by the functional $\mathrm{Sc}(p)$. When an assignment $\mathbf{y}$ over RVs $\mathbf{Y}$ covers only a portion of $\mathrm{Sc}(p) = \mathbf{X}$, or in other words $\mathbf{Y} \subset \mathbf{X}$, then $\mathbf{y}$ is called a *partial assignment*; otherwise, if it captures the entire scope, then it is said to be a *complete assignment*.

We borrow a few concepts from graph theory and say that, for a graph $\mathcal{G} = (\mathbf{N}, \mathbf{E})$, where $\mathbf{N}$ is the set of nodes in $\mathcal{G}$ and $\mathbf{E}$ the set of edges, the function $\mathrm{Ch}(\mathsf{N})$ maps a node $\mathsf{N} \in \mathbf{N}$ to the set of all children of N, that is, all nodes which have an edge *coming from* N. Similarly, $\mathrm{Pa}(\mathsf{N})$ maps N to its parents: the set of nodes which have an edge *going to* N. We assume that edge connections are unique, meaning that for two connected nodes N_1 and N_2, there can only exist a single edge connecting them, denoting $\overrightarrow{\mathsf{N}_1\,\mathsf{N}_2}$ the edge coming from N_1 to N_2.

2.1 Distributions as Computational Graphs

Probabilistic circuits are computational graphs usually recursively defined in terms of their computational units. We start with a broad definition of probabilistic circuits.

Definition 2.1.1 (Probabilistic circuit)**.** *A* probabilistic circuit $\mathcal{C}$ *is a rooted connected DAG whose nodes describe non-negative functions: a* sum node S *represents a weighted summation over its children* $\mathsf{S}(\mathbf{x}) = \sum_{\mathsf{C} \in \mathrm{Ch}(\mathsf{S})} w_{\mathsf{S},\mathsf{C}} \cdot \mathsf{C}(\mathbf{x})$*, a* product node P *multiplies all of its children*

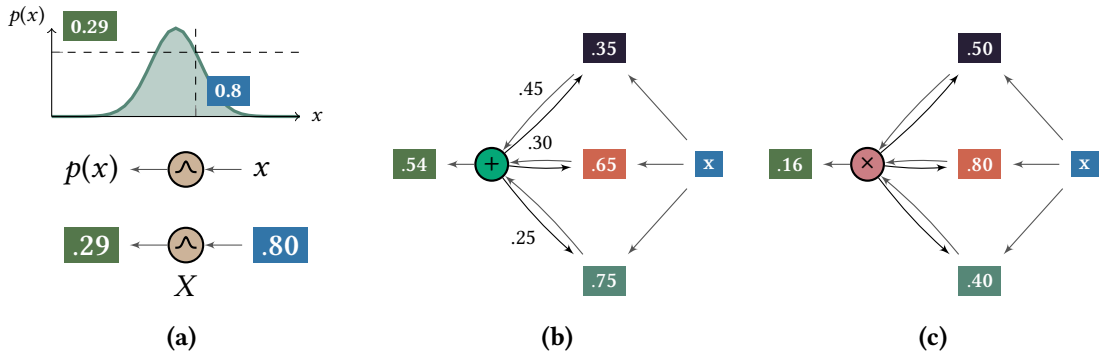


Figure 2.1: *An input node as a Gaussian distribution* **(a)***, a sum node* **(b)***, and a product node* **(c)***, the last two with three children each. Arrows in black signal (possibly weighted) edges in the computational graph, while gray edges indicate the computational flow: given an assignment* x*, the computation flows the opposite direction, starting on inputs and going up to the root, resulting in the final value in* green*.*

$\mathrm{P}(\mathbf{x}) = \prod_{\mathrm{C} \in \mathrm{Ch}(\mathrm{P})} \mathrm{C}(\mathbf{x})$*, and* input nodes*, i.e. nodes with no outgoing edges, are defined as univariate probability density functions. The* size *of $\mathcal{C}$ probabilistic circuit is the number of nodes and edges of its graph, denoted by $|\mathcal{C}|$.*

In its simplest form, a PC is a single input node annotated with an unnormalized probability distribution over a single variable[1]. In practice, however, inputs are typically portrayed as normalized parametric density (or mass) functions, and we shall assume them as so throughout this dissertation unless explicitly stated otherwise. We also assume, for purposes of simplifying analysis, that any query on an input node is computed in $\mathcal{O}(1)$ on the size of the input. To simplify notation, from here on out we shall use the term distribution to mean a probability density (or mass) function and argue that input nodes represent probability distributions. The semantics of this shall be clear given the context.

Let L be a PC input node representing a distribution p whose scope is over variable X. For an assignment x of variable X, we use the notation $\mathrm{L}(X = x)$ to mean $p(X = x)$, often omitting X when meaning is clear from context. We say that the *value* of input node L with respect to some query is the value of p with respect to that same query. As an example, Figure 2.1a shows the case when L represents a Gaussian. If we were to compute the probability of $x = 0.29$ according to L, then we would have to perform that same query on L's distribution p, in this case $p(x = 0.29) = 0.80$. Similarly, the values of inner nodes with respect to a query correspond to the resulting values of their functions given an assignment (and the query). Figure 2.1b shows a sum node S taking value $\mathrm{S}(\mathbf{x}) = 0.45 \cdot 0.35 + 0.30 \cdot 0.65 + 0.25 \cdot 0.75 = 0.54$, while Figure 2.1c shows the same, but for a product $\mathrm{P}(\mathbf{x}) = 0.50 \cdot 0.80 \cdot 0.40 = 0.16$. This concept of value of a node shall become clearer when we talk about reasoning in Section 2.2 and how to compute queries in PCs. For now, we assume that this value is simply the result of the computation of the function it represents given an assignment and a query.

[1] Although we define inputs as univariate, weaker definitions of PCs often permit inputs to be multivariate as well, provided mild conditions are met. In general, there is not much loss of expressivity or generality in assuming only univariate inputs, as multivariate inputs can often be represented as a PC instead.

As previously mentioned, an inner node indicates an operation to be computed from the value of its children. Although Definition 2.1.1 only references sums and products, the subject of more general operations and their benefits in terms of inference is an interesting question which we briefly touch in Remark 2.1, but otherwise remains out of the scope of this work. In this dissertation, we restrict ourselves to the study of sums and products as inner nodes. More precisely on the subject of sum nodes, we are interested in sums whose weights are non-negative and sum to one, or in other words, nodes which correspond to convex combinations of their children

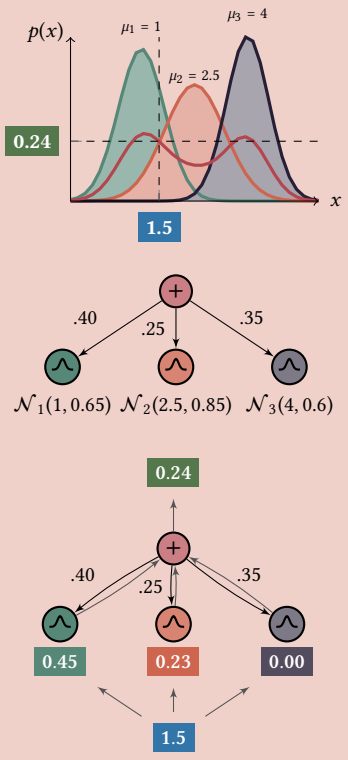
Semantically, the root of a PC represents an unnormalized probability distribution composed out of the functions of its descendant computational units. When all sums in a PC are convex combinations and every one of its inputs are normalized probability distributions, then the encompassing PC is also normalized, meaning that the distribution it represents is normalized. **Throughout this dissertation, we shall assume sums to be convex combinations and inputs to be normalized**[2]**.** Locally, sum nodes have the semantic interpretation of 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). Example 2.1 shows a case of a Gaussian mixture model as a PC, that is, a single sum node over Gaussian distributions as inputs.

Example 2.1: 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 $\mathcal{N}_1(\mu_1 = 1, \sigma_1 = 0.65)$, $\mathcal{N}_2(\mu_2 = 2.5, \sigma_1 = 0.85)$ 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 x according to G amounts to the weighted summation

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

which is equivalent to a computational graph (i.e. a PC) with a sum node as root and whose weights are set to ϕ and children to 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 computation flow (bottom, gray edges) up to the root sum node (in red), where a weighted summation is computed to output the probability (in green).



[2] This assumption incurs in no loss of generality, as Peharz, Tschiatschek, *et al.*, 2015 show.

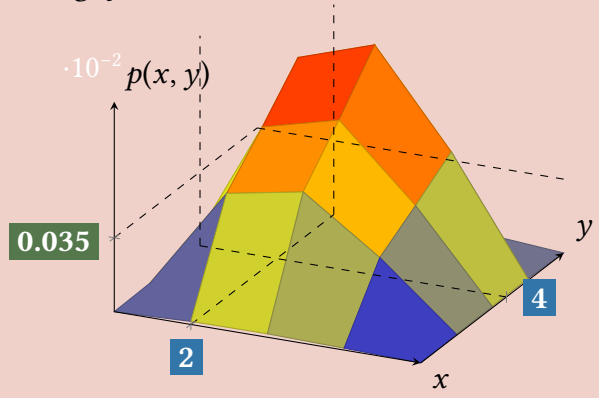
With the understanding that a PC determines a probability distribution, we may now extend the notion of *scope* to PCs, discoverable in an inductive fashion: the scope of an input node is the scope of its distribution, while the scope of an inner node N is the union of the scopes of its children $\text{Sc}(\text{N}) = \bigcup_{\text{C} \in \text{Ch}(\text{N})} \text{Sc}(\text{C})$. As an example, take the circuit from Example 2.2. The scope of input nodes and are $\text{Sc}() = \text{Sc}() = \{X\}$, while $\text{Sc}() = \text{Sc}() = \{Y\}$. Consequentially, their parent sum nodes will have the same scope as their children $\text{Sc}() = \{X\}$ and $\text{Sc}() = \{Y\}$, yet the root node's scope is $\text{Sc}() = \{X, Y\}$, since its childrens' scopes are distinct. The notion of scope is essential to many structural properties in PCs, many of which provide sufficient conditions for tractably computing a wide range of inference queries on probabilistic circuits as feedforward passes on the computational graph, as we detail in Section 2.2.

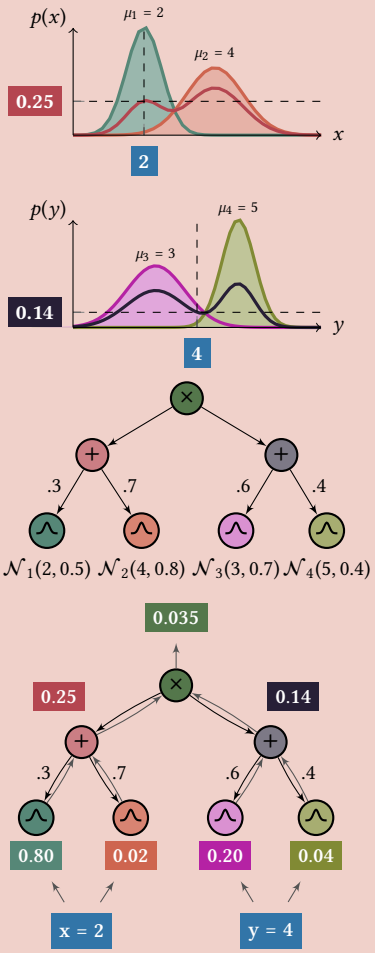
When computing the value of a node N, the domain of the function it computes $\text{N}(\mathbf{X} = \mathbf{x})$ is restricted to the node's scope $\text{Sc}(\text{N}) = \mathbf{X}$. For instance, in the case of Example 2.2, the domain of is over X and Y, while and are only over X and Y respectively. Having said that, for simplification purposes we often abuse notation and assume that, given an assignment $\mathbf{x}$ of RVs $\mathbf{X}$, the application of $\text{N}(\mathbf{x})$ is strictly over the variables in $\text{Sc}(\text{N})$, meaning that the assignments of any RVs outside of N's scope are simply ignored.

Example 2.2: 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 $X \perp\!\!\!\perp Y$, and 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 over the two mixture models and . The resulting joint of this circuit is shown below.





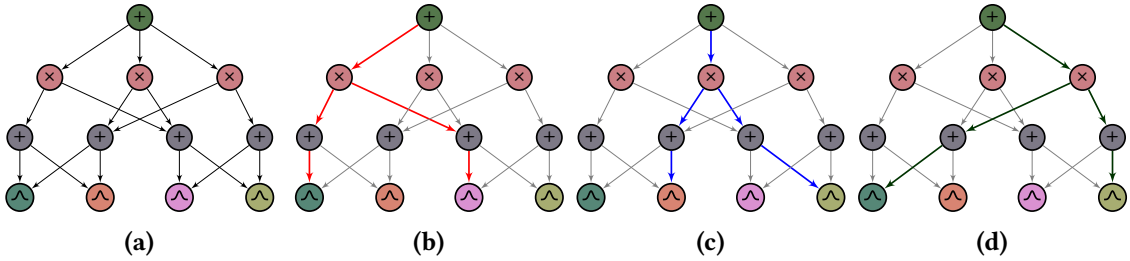


Figure 2.2: *A probabilistic circuit* **(a)** *and 3 of its 12 possible induced subcircuits* **(b-d)**.

Before we address reasoning in probabilistic circuits, we first describe indicator nodes, subcircuits, circuit size, induced subcircuits, and standard circuits, all of which are basic concepts of PCs that we shall need later throughout the text. We start with a special case of an input node. An *indicator node* is an input node whose distribution is the indicator function $f(x) = [\![x = k]\!]$, i.e. a degenerate distribution 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 either ◎ or just the textual Iverson bracket $[\![\cdot]\!]$ representation for indicators, while literals will be denoted by their textual propositional notation.

Let $\mathcal{C}$ be a probabilistic circuit and consider a node $\mathsf{N} \in \mathcal{C}$. We say that $\mathsf{N}_{\mathcal{C}}$ is a subcircuit of $\mathcal{C}$ rooted at N if $\mathsf{N}_{\mathcal{C}}$'s root is N, all nodes and edges in $\mathcal{C}_{\mathsf{N}}$ are also in $\mathcal{C}$ and $\mathcal{C}_{\mathsf{N}}$ is also a probabilistic circuit.

Definition 2.1.2 (Induced subcircuit)**.** *Let $\mathcal{C}$ be 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 for each sum node, out of all edges coming out of it, only one is in $\mathcal{S}$.*

Examples of induced subcircuits are visualized in Figure 2.2. When the induced subcircuit is tree-shaped, as is the case in Figure 2.2, they are referred to as induced trees (H. Zhao, Melibari, *et al.*, 2015; H. Zhao, Adel, *et al.*, 2016).

A probabilistic circuit $\mathcal{C}$ that contains no consecutive sums or products (i.e. for every sum all of its children are either inputs or products, and for products their children are either inputs or sums) is said to be in *standard* form circuit. Any PC can be efficiently transformed into a *standard* circuit in a process we call *standardization* (see Theorem A.1.2).

Remark 2.1: 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 subject 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) and Mattei, Antonucci, *et al.* (2020a) 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.

2.2 Reasoning with Probabilistic Circuits

Up to now, we have only vaguely touched on the issue of computing the value of a probabilistic circuit. Throughout this section, we define some of the possible inference tasks available to PCs, describe sufficient conditions for enabling these queries within this framework, and algorithmically show how to tractably compute them in the computational graph when these conditions are met. We start defining the four most basic probabilistic queries in PCs: probability of evidence, marginal probability, conditional probability and maximum a posteriori probability.

Consider a normalized probability distribution p whose scope is $\mathrm{Sc}(p) = \mathbf{X}$ and let $\mathbf{x}$ be a complete assignment of $\mathbf{X}$. We define the *probability of evidence*, shortened to EVI for convenience, as the query $p(\mathbf{x})$, i.e. the probability of $\mathbf{X}$ taking values $\mathbf{x}$ according to p, with $\mathbf{X}$ being called the *query variables*. In a somewhat similar vein, given a set of variables $\mathbf{Y}$ such that $\mathbf{Y} \subset \mathbf{X}$, and calling $\mathbf{y}$ a partial assignment of it, we say that the

Algorithm 1 EVI

Input A PC $\mathcal{C}$ and complete assignment $\mathbf{x}$
Output Value $\mathcal{C}(\mathbf{x})$

Let v be a hash function mapping a node to its value
for each N in reverse topological order **do**
    if N is an input **then** $v(\mathsf{N}) \leftarrow \mathsf{N}(\mathbf{x})$
    else if N is a sum **then** $v(\mathsf{N}) \leftarrow \sum_{\mathsf{C} \in \mathrm{Ch}(\mathsf{N})} w_{\mathsf{N},\mathsf{C}}\, v_{\mathsf{C}}$
    else if N is a product **then** $v(\mathsf{N}) \leftarrow \prod_{\mathsf{C} \in \mathrm{Ch}(\mathsf{N})} v_{\mathsf{C}}$
return $v(R)$, where R is $\mathcal{C}$'s root

query $p(\mathbf{y})$ is the *marginal probability*, here denoted by MAR, of query variables $\mathbf{Y}$ with the remaining variables $\mathbf{Z} = \mathbf{X} \cap \mathbf{Y}$ marginalized. This corresponds to the summing out $\sum_{\mathbf{z}} p(\mathbf{y}, \mathbf{z})$ when in the discrete, and the integral $\int_{\mathbf{z}} p(\mathbf{y}, \mathbf{z})\, d\mathbf{z}$ in the continuous. Aside from EVI and MAR, prediction tasks such as classification often require computing the *conditional probability* $p(\mathbf{y}|\mathbf{z}) = \frac{p(\mathbf{y},\mathbf{z})}{p(\mathbf{z})}$, which we shall call by the shorthand CON, of query variables $\mathbf{Y}$ given evidence variables $\mathbf{Z}$ in order to understand, in the case of image classification, the probability of a certain label given pixel values. A related yet more difficult task is to compute the *maximum a posteriori* probability (MAP), which involves finding the most probable assignment of a set of RVs $\mathbf{Y}$, said to be the query variables, conditioned on evidence variables $\mathbf{X}$, say for image reconstruction. To do so, we must compute the most probable assignment $\mathbf{y}$ (for instance, the missing pixels to be reconstructed) conditioned on evidence $\mathbf{x}$ (for example, values of the known pixels), or more formally

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

For this dissertation, we shall only consider the case of *full* MAP, i.e. when $\mathbf{x} \cup \mathbf{y}$ forms a complete assignment of the scope, since computing the *partial* MAP, i.e. when $\mathbf{x} \cup \mathbf{y}$ forms a partial assignment, is hard in most PCs (Peharz, Gens, Pernkopf, *et al.*, 2016; De Campos, 2011). Unless explicitly stated, MAP shall mean full MAP. Full MAP also goes by the name of *most probable explanation* (MPE, Darwiche, 2009) in literature. Although at first it may seem like MAP is no harder than computing a CON, it turns out that for smooth and decomposable PCs MAP is NP-hard (Conaty *et al.*, 2017; Mei *et al.*, 2018). Now that we have properly defined the queries we are interested in, we begin listing sufficient conditions for their tractability in PCs and the algorithms that compute them.

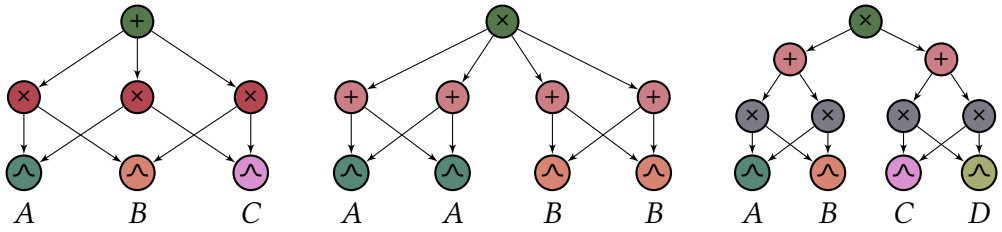
As we have already (informally) seen in the previous section, computing the probability of evidence for an assignment $\mathbf{x}$ according to a PC $\mathcal{C}$ amounts to the computation of the value of $\mathcal{C}$'s root by a bottom-up computation of node values. Starting with inputs, we compute their value by querying their distribution for their probability of evidence; this is then followed by simply computing inner node values normally. Algorithm 1 shows this procedure algorithmically, computing the EVI in linear time to the size of the circuit. Of note is the fact that, if the PC is not normalized, then the result of this procedure must be normalized by a constant, extractable in a similar way; however, since we assume all PCs to be normalized, the value returned by Algorithm 1 is already normalized.

By our definition of PCs, computing the EVI of a PC is always tractable. Computing

Algorithm 2 MAR

Input A smooth and decomposable PC $\mathcal{C}$ and partial assignment $\mathbf{x}$
Output Value $\int_{\mathbf{y}} \mathcal{C}(\mathbf{x}, \mathbf{y})\, d\mathbf{y}$ ▷ Call $\mathbf{Y}$ the remaining variables not in $\mathbf{X}$

Let v be a hash function mapping a node to its value
for each N in reverse topological order **do**
  if N is an input **then** $v(\mathsf{N}) \leftarrow \int_{\mathbf{y}} \mathsf{N}(\mathbf{x}, \mathbf{y})\, d\mathbf{y}$
  else if N is a sum **then** $v(\mathsf{N}) \leftarrow \sum_{\mathsf{C} \in \mathrm{Ch}(\mathsf{N})} w_{\mathsf{N},\mathsf{C}}\, v_{\mathsf{C}}$
  else if N is a product **then** $v(\mathsf{N}) \leftarrow \prod_{\mathsf{C} \in \mathrm{Ch}(\mathsf{N})} v_{\mathsf{C}}$
return $v(R)$, where R is $\mathcal{C}$'s root



(a) **(b)** **(c)**

Figure 2.3: *Decomposable but non-smooth* **(a)**, *smooth but non-decomposable* **(b)**, *and smooth and decomposable* **(c)** *circuits. Labels below inputs indicate their scope.*

the MAR, on the other hand, does not always come for free. We now introduce the first two structural constraints for probabilistic circuits which, together, enable tractable MAR in PCs.

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

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

When a probabilistic circuit $\mathcal{C}$ is both *smooth* and *decomposable*, any marginal is linear time computable in $\mathcal{C}$ (Poon and P. Domingos, 2011; Peharz, Tschiatschek, *et al.*, 2015). Although smoothness and decomposability are sufficient for tractably computing marginals, they are not necessary. In fact, *consistency* is a weaker constraint on products that, coupled with smoothness, confers efficient MAR (Poon and P. Domingos, 2011). Figure 2.3 shows examples of smooth and decomposable circuits. Although there exist PCs that are neither smooth nor decomposable (or consistent) and also have tractable MAR, as is the case of Example 2.3, these two properties are often adopted during learning due to their intuitive and uncomplicated syntax. In fact, **all PCs shown throughout this dissertation shall be *at least* smooth and decomposable unless explicitly stated otherwise.**

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

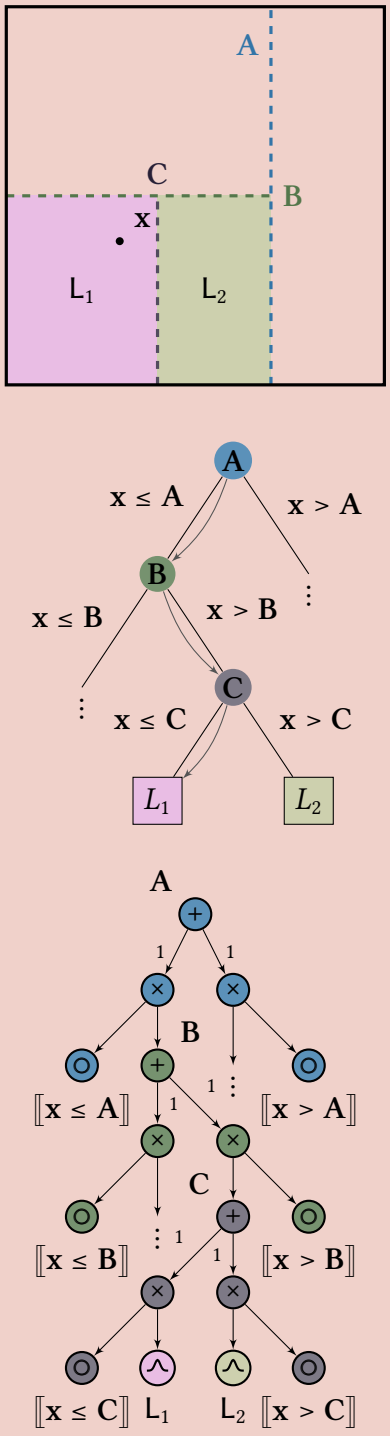
Example 2.3: Density estimation trees as probabilistic circuits

A density estimation tree (DET) is a decision tree for the task of density estimation (Ram and Gray, 2011). Briefly, a decision tree $\mathcal{T}$ partitions the data space (usually $\mathbb{R}^d$) into cells by laying out hyperplanes usually orthogonal to the axes, creating a latent variable for each node that essentially determines in which region an observation should fall into. The density function of a tree $\mathcal{T}$ given a dataset $\mathbf{D}$ of dimension d is defined as the piecewise function

$$p_{\mathcal{T}}(\mathbf{x}) = \sum_{\mathsf{L} \in \text{Leaves}(\mathcal{T})} \frac{|\mathbf{D} \in \mathsf{L}|}{|\mathbf{D}|} \cdot \frac{[\![\mathbf{x} \in \mathsf{L}]\!]}{\text{Vol}(\mathsf{L})}, \tag{2.2}$$

where $|\mathbf{D} \in \mathsf{L}|$ indicates the number of assignments $\mathbf{x}$ in the training dataset $\mathbf{D}$ which fall inside the cell determined by leaf L, and $\text{Vol}(\mathsf{L})$ retuns the volume of the d-dimensional cell L. The top figure on the right shows a two-dimensional data space being partitioned, with the corresponding decision tree below it. Each node in the tree is a (hyper)plane partitioning data, with every edge determining which cell the observation falls into.

Equivalently, an unnormalized, smooth and nondecomposable PC whose sum nodes are followed by products which determine which side of the hyperplane an assignment goes configures the same semantics as a DET decision node. Inputs in this PC act as the leaf nodes in the equivalent DET. The PC on the right translates the DET on top of it, with matching colors in the PC showing the equivalent leaves and decision nodes in the DET. Because the resulting density is constant-piecewise, marginalization is tractable, highlighting the sufficiency but not necessity of smoothness and decomposability. An alternative smooth and decomposable formulation of DETs is given by Correia *et al.* (2020).



What Theorem 2.2.1's proof on page 89 tells us is that, algorithmically, the only difference between EVI, MAR and CON is what is done on the input nodes. This can be seen in Algorithm 2, which shows how to compute marginals in PCs. With MAR available to us, querying for CON reduces to a simple two-pass evaluation over the circuit: the

Algorithm 3 MaxProduct

Input A smooth, decomposable and deterministic PC $\mathcal{C}$ and evidence assignment $\mathbf{x}$
Output Value $\max_{\mathbf{y}} \mathcal{C}(\mathbf{y}|\mathbf{x})$

1: Let v be 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** $v(\mathsf{N}) \leftarrow \max_{\mathbf{y}} \mathsf{N}(\mathbf{y}, \mathbf{x})$
4: **else if** N is a sum **then** $v(\mathsf{N}) \leftarrow \max_{\mathsf{C} \in \mathrm{Ch}(\mathsf{N})} w_{\mathsf{N},\mathsf{C}}\, v_{\mathsf{C}}$
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** $v_R/\mathcal{C}(\mathbf{x})$, where R is $\mathcal{C}$'s root and $\mathcal{C}(\mathbf{x})$ is the EVI on $\mathbf{x}$

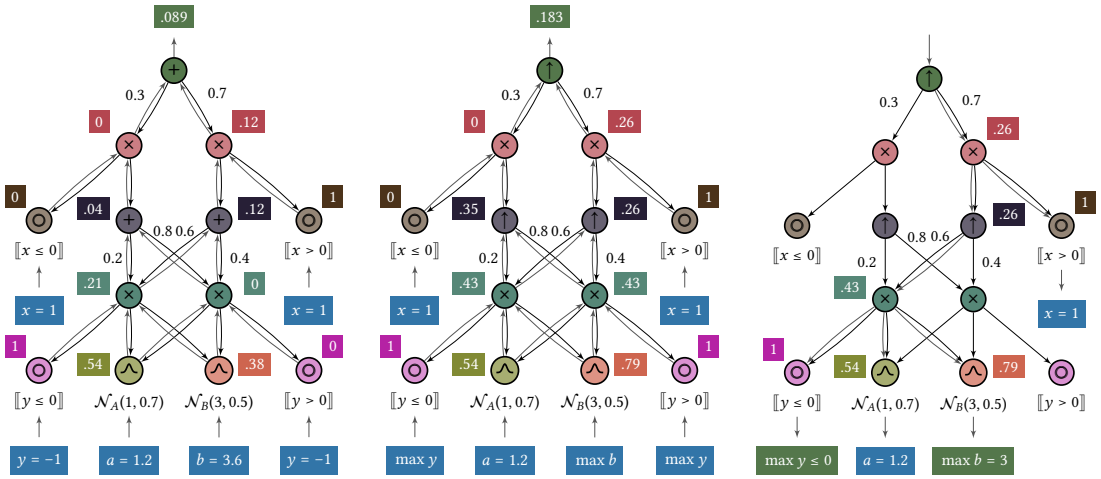


Figure 2.4: *The computation, on a smooth, decomposable and deterministic probabilistic circuit, of EVI* $p(a = 1.2, b = 3.6, x = 1, y = -1) \approx 0.089$ **(a)**, *MAP* $\max_{b,y} p(b, y|a = 1.2, x = 1) \approx 0.183$ *via* MaxProduct **(b)**, *and* $\arg\max_{b,y} p(b, y|a = 1.2, x = 1) = \{b = 3, y \leq 0\}$ *by backtracking the values set by* MaxProduct **(c)**. ↑ *nodes signal the replacement of sums with maximizations in* MaxProduct. *The backtracking in* **(c)** *is done from the root down, finding a max induced tree by propagating through all product children and only the highest valued child in* ↑ *nodes.*

first computes the numerator as the marginal $p(\mathbf{Y}, \mathbf{X})$, and the second the denominator marginal $p(\mathbf{X})$.

Besides smoothness and decomposability, another structural constraint of interest is *determinism*. Together with the first two, determinism provides sufficient conditions for tractably computing the MAP.

Definition 2.2.3 (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 for any complete assignment.*

Theorem 2.2.2 (Peharz, Gens, Pernkopf, *et al.*, 2016). *Let $\mathcal{C}$ be a smooth, decomposable and* deterministic *PC.* MaxProduct *computes the MAP in $\mathcal{C}$ in linear time (on the size of $\mathcal{C}$).*

As Theorem 2.2.2's proof on page 90 suggests, the MAP on a smooth, decomposable and deterministic PC can be easily computed by simply replacing sum nodes with a max operation and performing a bottom-up EVI pass. This is commonly called the Max-Product

algorithm, shown more formally in Algorithm 3. To find the assignment $\mathbf{y}$ that maximizes 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 C by backtracking the graph to the most probable assignments. 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.

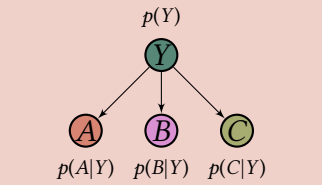
Many known probabilistic models can be subsumed as smooth, decomposable and deterministic PCs, such Markov networks (Lowd and Rooshenas, 2013), naïve Bayes, thin junction trees (Bach and Jordan, 2001), and more generally low-treewidth Bayesian networks (Darwiche, 2003). Example 2.4 shows a Gaussian naïve Bayes model as a probabilistic circuit.

Example 2.4: Naïve Bayes as probabilistic circuits

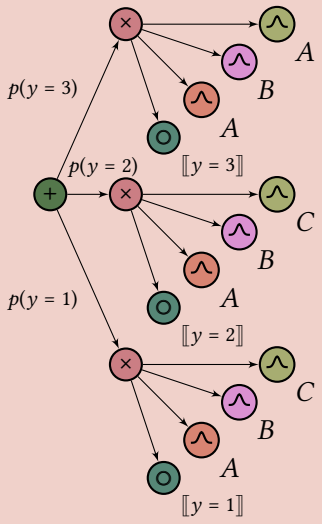
Suppose we have samples of per capita census measurements on three different features, say age A, body mass index B and average amount of cheese consumed daily C from three different cities Y. Assuming A, B and C are independent, given a sample $x = (a, b, c)$ we can use Gaussian naïve Bayes to predict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

$$p(y|a, b, c) = p(y)p(a|y)p(b|y)p(c|y). \qquad (2.3)$$

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 value of x for each city y.



Gaussian naïve Bayes



Equivalent PC

Although we have only covered the most basic queries so far, more complex tasks involving information-theoretic measures, logical queries or distributional divergences are also (tractably) computable in PCs 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*

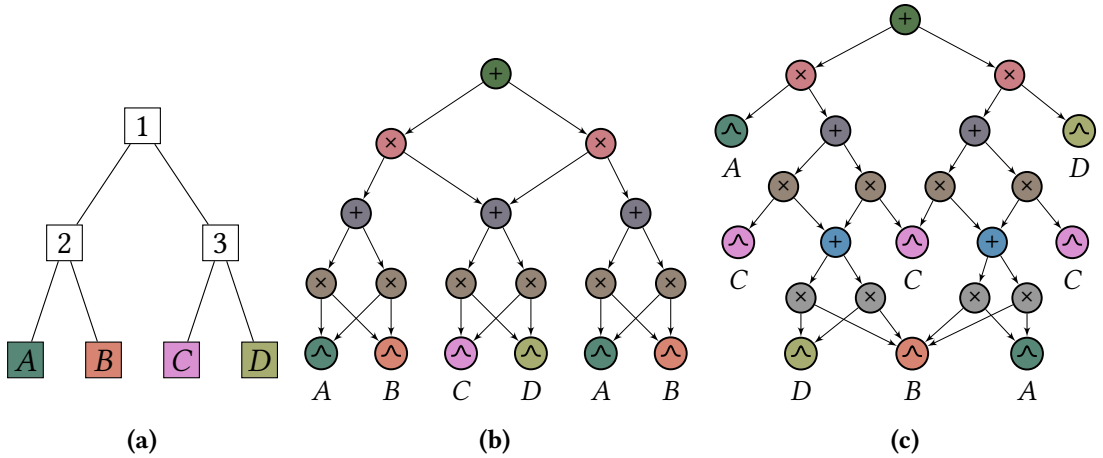


Figure 2.5: *A vtree* **(a)** *defining an order* (A, B, C, D)*, a 2-standard structure decomposable probabilistic circuit that respects the vtree* **(b)***, and a 2-standard decomposable probabilistic circuit that does not* **(c)**.

(variable tree), a data structure that defines a (partial) ordering of variables.

Definition 2.2.4 (Vtree)**.** *A variable tree* $\mathcal{V} = (\mathcal{B}, \phi)$*, or* vtree*, over a set of variables* $\mathbf{X}$ *is a pair made out of a binary tree* $\mathcal{B}$ *whose number of leaf nodes is* $|\mathbf{X}|$*, and a one-to-one and onto mapping* ϕ *of leaves of* $\mathcal{B}$ *with variables* $\mathbf{X}$*.*

We shall adopt the same scope definition and notation $\mathrm{Sc}(\cdot)$ for vtrees as in PCs. Let v be a vtree node from a vtree $\mathcal{V}$. If v is a leaf node of $\mathcal{V}$, 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 $(\mathrm{Sc}(v^{\leftarrow}), \mathrm{Sc}(v^{\rightarrow}))$, while the leaves of $\mathcal{V}$ define a partial ordering of $\mathrm{Sc}(\mathcal{V})$. We are especially interested in the scope partitioning aspect of vtrees.

Definition 2.2.5. *A product node* P respects *a vtree node* v *if* P *contains only two children* $\mathrm{Ch}(\mathsf{P}) = \{\mathsf{C}_1, \mathsf{C}_2\}$*, and* $\mathrm{Sc}(\mathsf{C}_1) = \mathrm{Sc}(v^{\leftarrow})$ *and* $\mathrm{Sc}(\mathsf{C}_2) = \mathrm{Sc}(v^{\rightarrow})$*.*

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 2.5a 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. Further, we call the i-th layer of a PC or a vtree as the set of all nodes that are at depth i (i.e. the shortest connected path from the root to the node has size i).

Definition 2.2.6 (Structure decomposability)**.** *Let $\mathcal{C}$ be 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 $\mathcal{V}$.*

Although we assume a 2-standard PC in Definition 2.2.6, this 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 (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 a context-specific independence relationship between $\mathrm{Sc}(v^{\leftarrow})$ and $\mathrm{Sc}(v^{\rightarrow})$ under the distribution encoded by its PC.

Figure 2.5 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. Example 2.5 shows a Hidden Markov model as a smooth, deterministic and structure decomposable PC whose sums describe the latent variables and products partition observable variables according to the 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 (in polynomial time) in such a way that any two products with same scope, one from $\mathcal{C}_1$ and the other $\mathcal{C}_2$, partition the scope into the same decompositions (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.

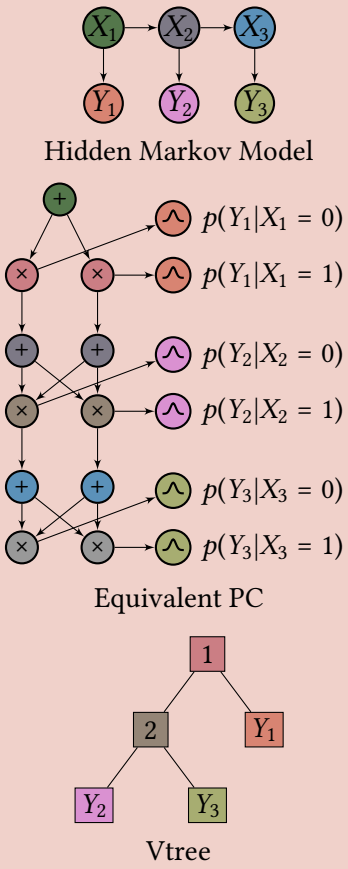
Probabilistic circuits appear in literature under many names. A PC which is both smooth and decomposable is often referred to as a Sum-Product Network (SPN, Poon and P. Domingos, 2011), although definitions vary around the presence of the two structural constraints as a requirement, with SPNs sometimes used as a synonym for probabilistic circuits. Smooth, decomposable and deterministic PCs often appear as Arithmetic Circuits (ACs, Darwiche, 2003) or Cutset Networks (CNets, Rahman, Kothalkar, *et al.*, 2014). Probabilistic Sentential Decision Diagrams (PSDDs, Kisa *et al.*, 2014), Probabilistic Decision Graphs (PDGs, Jaeger, 2004) and And/Or-Graphs (AOGs, Dechter and Mateescu, 2007) can all be described as smooth, deterministic and structure decomposable PCs.

Example 2.5: Hidden Markov models as probabilistic circuits

Say we wish to model a sequential 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

$$p(X_{1..3}, Y_{1..3}) = p(X_1) \prod_{i=2}^{3} p(X_i|X_{i-1}) \prod_{i=1}^{3} p(Y_i|X_i). \tag{2.4}$$

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 follows the partitionings imposed by the vtree, with ⊗ decomposing into $(\emptyset, \{Y_3\})$. This means that this PC is not only smooth, but structure decomposable and deterministic.



The notion of structure decomposability (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 the circuit that needs to come inside the log 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 the expectation (EXP) of a circuit with respect to another (Y. Choi, Vergari, and Broeck, 2020), defined as

$$\mathbb{E}_{\mathcal{C}}[\mathcal{S}] = \int_{\mathbf{x}} \mathcal{C}(\mathbf{x})\mathcal{S}(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{2.5}$$

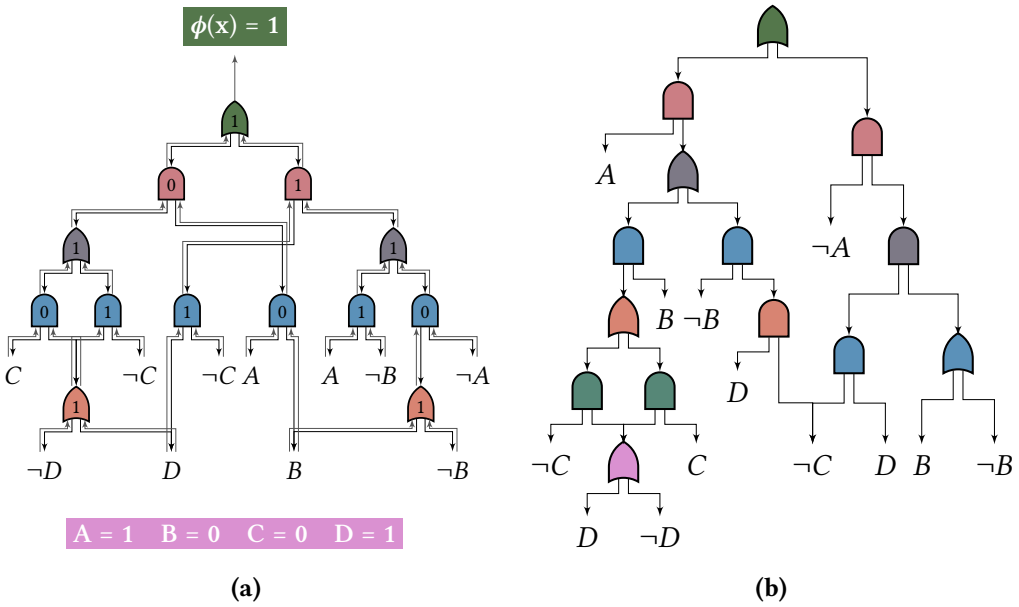


Figure 2.6: *Two smooth, structure decomposable and deterministic logic circuits encoding the same logic constraint $\phi \equiv (A \wedge B) \vee (\neg C \wedge D)$ for a balanced* **(a)** *and a right-linear* **(b)** *vtree. In* **(a)***, a circuit evaluation for an assignment, with each node value in the bottom-up evaluation pass shown inside nodes.*

One notable example from this class is computing the probability of logical events, which in terms of a EXP query amounts to computing the probability of C when circuit S represents a given logical query. This leads us to logic circuits, a parallel version of probabilistic circuits for logical reasoning which we shall see next.

2.3 Probabilistic Circuits as Knowledge Bases

We superficially mentioned in Remark 2.1 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 inner nodes compute either a conjunction or a disjunction of their children. Nodes with no outgoing*

edges, i.e. literal *nodes, are indicator functions of either a positive (true) or negative (false) assignment. Computing the satisfiability of a world* $\mathbf{x}$ *according to* $\mathcal{L}$ *amounts to a bottom-up pass where literal nodes are assigned values consistent with* $\mathbf{x}$ *and values are propagated up to the root.*

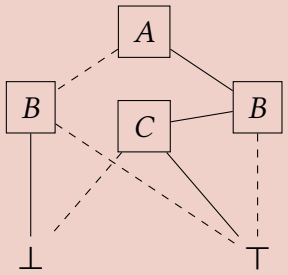
Evidently, logic circuits are closely related to probabilistic circuits. The strikingly similar definitions are not coincidence: much of the literature on PCs have their origins on LCs. In fact, most structural constraints in PCs are the exact same (up to even their names[3]) as their LC analogues. In 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 2.6 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 computer science literature under many names, more closely to the knowledge compilation community under the title 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. Example 2.6 shows a smooth, structure decomposable and deterministic logic circuit as a BDD. Although no structural constraint is required for model verification in logic circuits, the same properties defined in the past section have come up in LCs to enable other more complex queries like equivalence, implication, 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 (Gogic *et al.*, 1995; Papadimitriou, 1994; Darwiche and Marquis, 2002). See Darwiche and Marquis (2002) and Darwiche (2020) for more on logic circuits.

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

Example 2.6: 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.



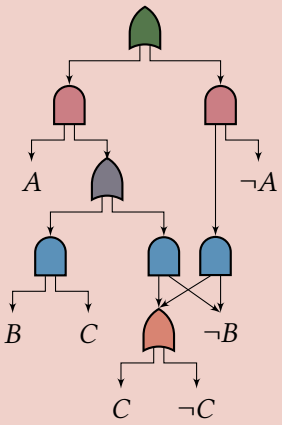
In their usual notation, inner nodes are variables and leaves are constants $\bot$ and $\top$. Evaluating an assignment $\mathbf{x} \in \{0,1\}^n$ is equivalent to a path from the root to a leaf where each variable X determines a decision to go through the dashed line when $x = 0$ or solid line when $x = 1$. 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

$$\phi(A, B, C) = (A \vee \neg B) \wedge (\neg B \vee C), \qquad (2.6)$$

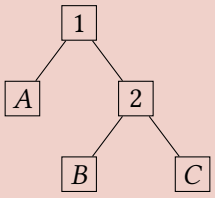
also shown as a truth table on the right, together with a logic circuit that encodes the same truth table and its vtree. Conjunctions take the role of products, with each conjunction node determining a vtree node's scope partition.

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 $\mathcal{C}$ 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 $S_{\mathcal{L}}$ of $\mathcal{L}$ defines

$$S_{\mathcal{L}}(\mathbf{x}) = \bigvee_{C \in \mathrm{Ch}_{\mathcal{C}}(S_{\mathcal{L}})} [\![C_p(\mathbf{x}) > 0]\!] \wedge [\![C_s(\mathbf{x}) > 0]\!], \tag{2.7}$$

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

$$S_{\mathcal{C}}(\mathbf{x}, \mathbf{y}) = \sum_{C \in \mathrm{Ch}(S_{\mathcal{C}})} w_{S_{\mathcal{C}}, C} \cdot C_p(\mathbf{x}) \cdot C_s(\mathbf{y}). \tag{2.8}$$

When a deterministic sum (resp. disjunction) node has the above form, then this composition of a weighted sum (resp. disjunction) is known in PC and LC literature as a *partition*.

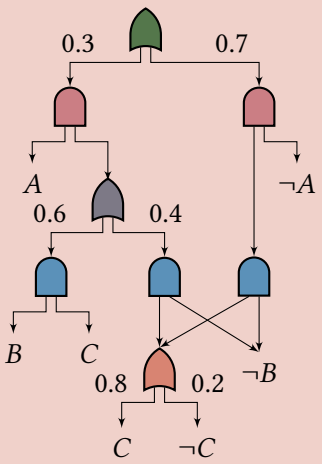
Example 2.7: Embedding certain knowledge in probabilistic circuits

Recall the logic circuit $\mathcal{L}$ from Example 2.6. 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

Algorithm 4 EXP

Input A smooth, structure decomposable PC $\mathcal{C}$ and LC $\mathcal{L}$, both with vtree $\mathcal{V}$
Output The expectation $\mathbb{E}_{\mathcal{C}}[\mathcal{L}] = \int_{\mathbf{x}} \mathcal{C}(\mathbf{x})\mathcal{L}(\mathbf{x})\,\mathrm{d}\mathbf{x}$

Let $\mathcal{H}$ be a hash table mapping a pair of nodes to an expectation
function TRAVERSE($\mathsf{N}_{\mathcal{C}}$, $\mathsf{N}_{\mathcal{L}}$)
    if $(\mathsf{N}_{\mathcal{C}}, \mathsf{N}_{\mathcal{L}}) \in \mathcal{H}$ **then return** $\mathcal{H}(\mathsf{N}_{\mathcal{C}}, \mathsf{N}_{\mathcal{L}})$
    if $\mathsf{N}_{\mathcal{C}}$ is an input **then** $\mathcal{H}(\mathsf{N}_{\mathcal{C}}, \mathsf{N}_{\mathcal{L}}) \leftarrow \mathbb{E}_{\mathsf{N}_{\mathcal{C}}}[\mathsf{N}_{\mathcal{L}}]$
    else if $\mathsf{N}_{\mathcal{C}}$ is a product
        $v \leftarrow 0$
        for each i-th children $\mathsf{C}_{\mathcal{C}}^{(i)}$ and $\mathsf{C}_{\mathcal{L}}^{(i)}$ of $\mathsf{N}_{\mathcal{C}}$ and $\mathsf{N}_{\mathcal{L}}$ respectively **do**
            $v \leftarrow v \cdot$ TRAVERSE($\mathsf{C}_{\mathcal{C}}^{(i)}, \mathsf{C}_{\mathcal{L}}^{(i)}$)
        $\mathcal{H}(\mathsf{N}_{\mathcal{C}}, \mathsf{N}_{\mathcal{L}}) \leftarrow v$
    else if $\mathsf{N}_{\mathcal{C}}$ is a sum
        $v \leftarrow 0$
        for each child $\mathsf{C}_{\mathcal{C}} \in \mathrm{Ch}(\mathsf{N}_{\mathcal{C}})$ **do**
            for each child $\mathsf{C}_{\mathcal{L}} \in \mathrm{Ch}(\mathsf{N}_{\mathcal{L}})$ **do**
                $v \leftarrow v + w_{\mathsf{N}_{\mathcal{C}},\mathsf{C}_{\mathcal{C}}} \cdot$ TRAVERSE($\mathsf{C}_{\mathcal{C}}, \mathsf{C}_{\mathcal{L}}$)
        $\mathcal{H}(\mathsf{N}_{\mathcal{C}}, \mathsf{N}_{\mathcal{L}}) \leftarrow v$
    return $\mathcal{H}(\mathsf{N}_{\mathcal{C}}, \mathsf{N}_{\mathcal{L}})$
return TRAVERSE($\mathcal{C}, \mathcal{L}$)

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, a special case of the previously defined Equation (2.5), where the circuit to be queried $\mathcal{L}$ is a logic circuit representing any logic query and $\mathcal{C}$ is the probabilistic interpretation

$$\mathbb{E}_{\mathcal{C}}[\mathcal{L}] = \int_{\mathbf{x}} \mathcal{C}(\mathbf{x})\mathcal{L}(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{2.9}$$

This computation is known to be tractable when $\mathcal{C}$ and $\mathcal{L}$ both have the same vtree and $\mathcal{C}$ is smooth, as Theorem 2.3.1 shows.

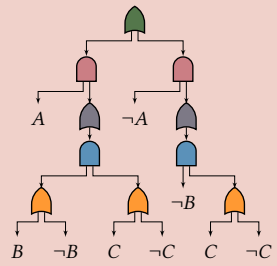
Theorem 2.3.1 (Y. Choi, Vergari, and Broeck, 2020)**.** *If $\mathcal{C}$ is a smooth and structure decomposable probabilistic circuit with vtree $\mathcal{V}$, and $\mathcal{L}$ a structure decomposable logic circuit also respecting $\mathcal{V}$, then $\mathbb{E}_{\mathcal{C}}[\mathcal{L}]$ is polynomial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

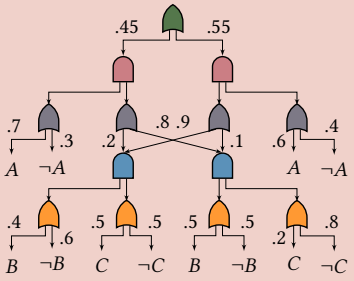
Let $\mathcal{C}$ be a smooth and structure decomposable circuit with vtree $\mathcal{V}$, and $\mathcal{L}$ a logic circuit representing a logical query whose vtree is also $\mathcal{V}$. Computing the probability of $\mathcal{L}$ with respect to the distribution encoded by $\mathcal{C}$ is done by a bottom-up evaluation over both circuits at the same time. Algorithm 4 shows the procedure algorithmically. Importantly, the procedure relies on evaluating the expectation of a node with respect to another, with one coming from the *probabilistic*, and the other from the *logical* side. The algorithm runs in polynomial time by caching expectation values to avoid recomputing already visited nodes. Starting with inputs, where the expectation is delegated to the input's distribution, we go up to inner nodes, where both computation and node pairing is distinct depending on the node's computational unit. 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 2.8: 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 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 2.2: 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 (Stuhlmüller and Goodman, 2012; Saad *et al.*, 2021), and fault localization (Nath and P. M. Domingos, 2016).

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 than, 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[1]. 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 explain LearnSPN however, we must first address how we denote data. Data is commonly represented as a matrix where rows are assignments (of all variables),

[1] The algorithms we shall see in this class are sometimes classified as *constraint-based* (Spirtes and Meek, 1995) learners, as they learn the model by identifying independences within data. Although this is true for the examples here mentioned, the two taxonomies are not equivalent. In fact, we describe a random divide-and-conquer structure learning approach in Section 5.1 that does not (directly) rely on statistical tests.

Algorithm 5 LearnSPN

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

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

and columns are the values that each variable takes at each assignment. Let $\mathbf{D} \in \mathbb{R}^{n \times m}$ be a matrix with n rows and m 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.

3.1.1 LearnSPN

Recall the semantics of sum and product nodes in a smooth and decomposable probabilistic circuit. A sum node encodes 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 node encodes a factorization $p(\mathbf{X}_1, \ldots, \mathbf{X}_m) = \prod_{i=1}^{m} p(\mathbf{X}_i)$, implying that $\mathbf{X}_i \perp\!\!\!\perp \mathbf{X}_j$ for $i, j \in [m]$ 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 $\mathbf{D} \in \mathbb{R}^{n \times m}$, 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 5.

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.

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

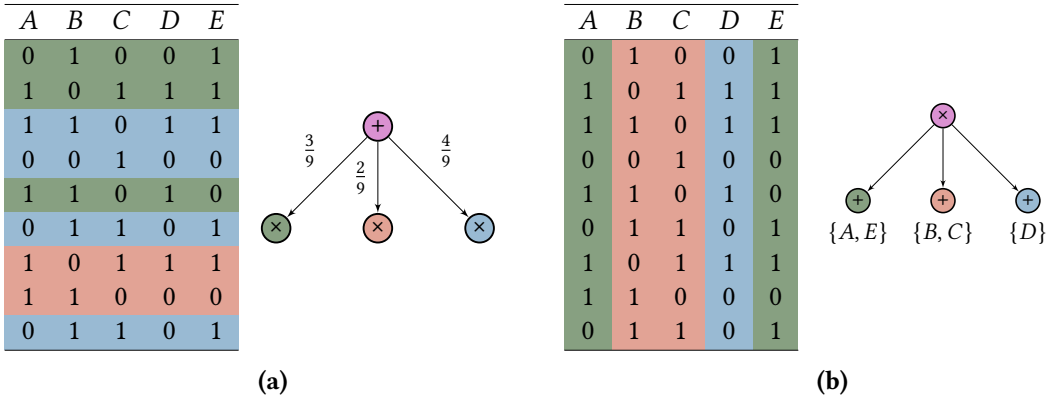


Figure 3.1: *LearnSPN assigns either rows* **(a)** *or columns* **(b)** *for sum and product nodes respectively. For sums, their edge weights are set proportionally to the assignments. For product children, scopes are defined by which columns are assigned to them.*

other hand, are much more costly. The naïve approach would be to find every possible combination of variable partitions of any size and compute statistical independence tests over these subsets of variables, which would take superexponential time on the number of variables. Instead, LearnSPN proposes the faster approach of testing for pairwise independence $X_i \perp\!\!\!\perp X_j$ 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. This is made clear by the following example: suppose we have an incomplete independence graph $\mathcal{G}$ where, at a certain point in the process of finding the (independent) variable partitions, we know there to be two components $\mathbf{X}$ and $\mathbf{Y}$; by hypothesis there is no edge connecting any variable in $\mathbf{X}$ to any other variable in $\mathbf{Y}$. The task of determining whether $\mathbf{X}$ and $\mathbf{Y}$ are, truly, a single component boils down to finding a pair of variables $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$ such that the independence tests on the two returns that $X \perp\!\!\!\perp Y$. If, at the next iteration, we luckily choose such a pair, no other pair of $\mathbf{X}$ and $\mathbf{Y}$ needs to be tested any longer, as we have already shown $\mathbf{X}$ and $\mathbf{Y}$ to belong to the same component. Note that, had we not used this heuristic, every pair would still need to be tested, even if they are known to be in the same component. Even so, this heuristic is still cubic on the number of variables in the 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.

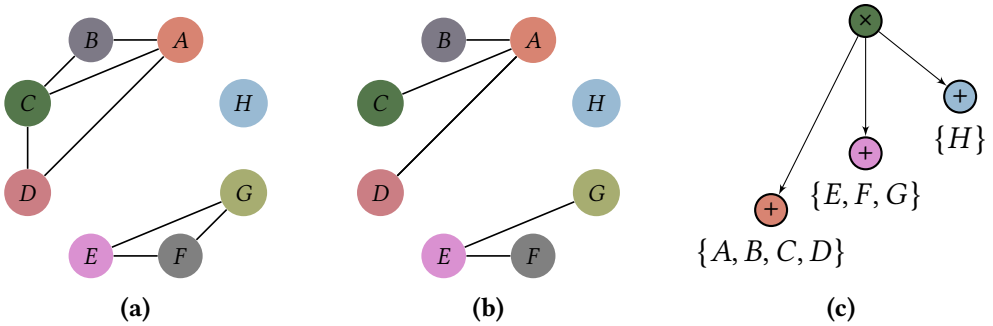


Figure 3.2: *The pairwise (in)dependence graph where each node is a variable. In* **(a)** *we show the full graph, computing independence tests for each pair of variables in* $\mathcal{O}(m^2)$*. However, it suffices to compute for only the connected components* **(b)***, saving up pairwise computation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

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.

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 5 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 tractable 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 7 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. Figure 3.3 shows two hypothetical iterations of ID-SPN, with each call expanding the probabilistic circuit into either a sum or product over Markov networks.

Algorithm 6 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 $\mathbf{D}$

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

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 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 $\mathcal{O}(i \cdot c(r \cdot n + m))$, where i is the number of iterations to run, c is the size of the generated PC, and constant r 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 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

Algorithm 7 ID-SPN

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

Create a single-node PC: $\mathcal{C} \leftarrow$ LearnMarkov($\mathbf{D}, \mathbf{X}$)
Let $\mathcal{M}$ be a memoization function associating a node with a dataset and scope
Call $\mathcal{C}'$ a copy of $\mathcal{C}$
while improving $\mathcal{C}$ yields better likelihood **do**
    Pick worse node N from $\mathcal{C}'$
    Extract sub-data $\mathbf{D}'$ and sub-scope $\mathbf{X}'$ from $\mathcal{M}$(N)
    Replace N with ExtendID($\mathbf{D}', \mathbf{X}', \mathcal{M}$)
    if $\mathcal{C}'$ has better likelihood than $\mathcal{C}$ **then** $\mathcal{C} \leftarrow \mathcal{C}$
return $\mathcal{C}$

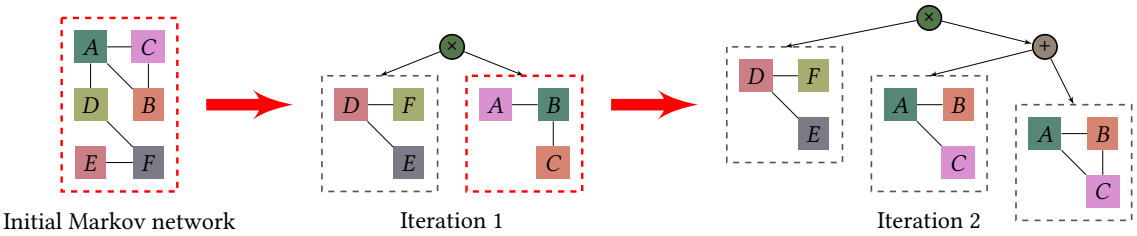


Figure 3.3: *Two iterations of ID-SPN, where the contents inside the dashed line are Markov networks. The red color indicates that a node has been chosen as the best candidate for an extension with ExtendID. Although here we only extend input nodes, inner nodes can in fact be extended as well.*

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

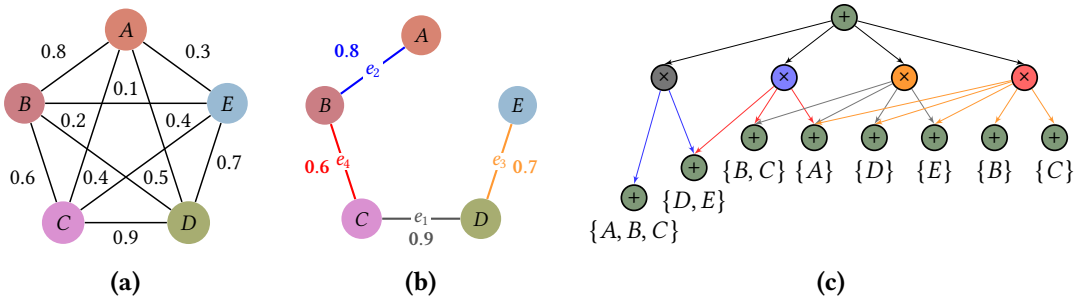


Figure 3.4: *The fully connected correlation graph* **(a)** *with weights as the pairwise correlation measurements for each pair of variables; the maximum spanning tree for determining decompositions* **(b)***; and the mixture of decompositions* **(c)**. *Colors in* **(b)** *match their partitionings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

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 $\mathcal{G}$ be the independence graph for scope $\mathbf{X} = \{X_1, X_2, \ldots, X_m\}$. 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 $X_i \not\perp\!\!\!\perp X_j$. 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 be 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 8 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 8 comes into play). This avoids an exponential growth from the $k \cdot (|\mathbf{X}| - 1)$ potential recursive calls.

Algorithm 8 PROMETHEUS

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

if $|\mathbf{X}|$ is sufficiently small **then return** an input node learned from $\mathbf{D}$
else
  Find subsets of data $\mathbf{x}_1, \ldots, \mathbf{x}_k \subseteq \mathbf{D}$ st all assignments within $\mathbf{x}_i$ are all similar
  Create a sum node S with initially no children and uniform weights
  for each $\mathbf{x}_i$ **do**
    $\mathcal{T} \leftarrow$ CORRELATIONMST($\mathbf{x}_i, \mathbf{X}$)
    for each weighted edge e_j in $\mathcal{T}$ in decreasing order **do**
      Remove edge e_i from $\mathcal{T}$
      Call $\mathsf{S}_1, \ldots, \mathsf{S}_t$ the scopes of each component in $\mathcal{T}$
      Create product node P_j and associate it with $\mathsf{S}_1, \ldots, \mathsf{S}_t$
      Associate P_j with dataset $\mathbf{x}_i$
      Add P_j as a child of S
  Let $\mathcal{H}$ be a hash table (initially empty) associating scopes to sum nodes
  for each $\mathsf{P} \in \mathrm{Ch}(\mathsf{S})$ **do**
    for each scope S associated with P **do**
      if $\mathsf{S} \notin \mathcal{H}$ **then**
        Let $\mathbf{x}$ be the dataset associated with P
        $\mathsf{N} \leftarrow$ PROMETHEUS($\mathbf{x}_{:,\mathsf{S}}, \mathsf{S}$)
        Add N as a child of P
        $\mathcal{H}_{\mathsf{S}} \leftarrow \mathsf{N}$
      else
        Add $\mathcal{H}_{\mathsf{S}}$ as a child of P
  return S

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: 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 possibly large number of hyperparameters involved, usually dependent on the methods chosen for clustering and independence testing. 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 focus our attention to incremental[2] 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. For this reason, these are also sometimes classified as *search-and-score* methods (Teyssier and Koller, 2005). 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 $\mathbf{X} = \mathrm{Sc}(v^{\leftarrow})$ and $\mathbf{Y} = \mathrm{Sc}(v^{\rightarrow})$ 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. With this in mind, Liang, Bekker, *et al.* (2017) propose two approaches to inducing vtrees from data, both of which use mutual information

$$\mathrm{MI}(\mathbf{X}, \mathbf{Y}) = \sum_{\mathbf{X}=\mathbf{x}} \sum_{\mathbf{Y}=\mathbf{y}} p(\mathbf{x}, \mathbf{y}) \log \frac{p(\mathbf{x}, \mathbf{y})}{p(\mathbf{x})p(\mathbf{y})} \tag{3.1}$$

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

$$\mathrm{pMI}(\mathbf{X}, \mathbf{Y}) = \frac{1}{|\mathbf{X}||\mathbf{Y}|} \cdot \sum_{X \in \mathbf{X}} \sum_{Y \in \mathbf{Y}} \mathrm{MI}(X, Y). \tag{3.2}$$

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 $\mathcal{G}$ be a fully connected weighted graph where variables are nodes. For each edge $\overrightarrow{XY}$, attribute its weight as $\mathrm{MI}(X, Y)$. Learning the vtree top-down amounts to partitioning $\mathcal{G}$ such that the cut-set that divides the two partitions $\mathbf{X}$ and $\mathbf{Y}$

[2] Despite the ambiguous name, we draw no connection to *online learning*.

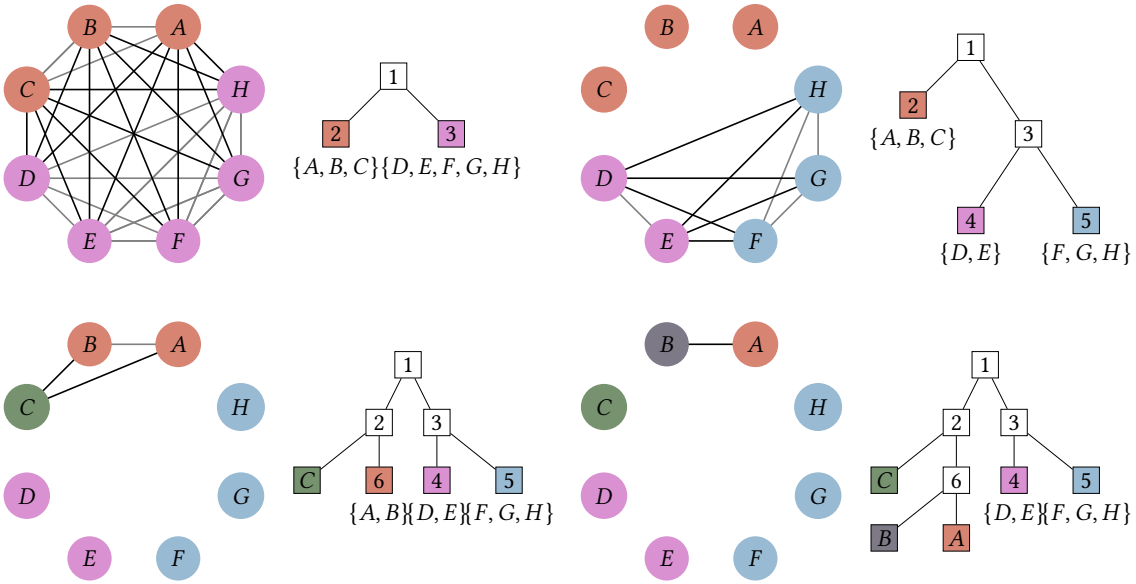


Figure 3.5: *Snapshots of four iterations from running the vtree top-down learning strategy with pairwise mutual information. Each iteration shows a variable partitioning, the cut-set that minimizes the average pairwise mutual information as black edges, and the subsequent (partial) vtree. The algorithm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

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 $\mathbf{Y}$; 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). Figure 3.6 exemplifies the algorithm.

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);

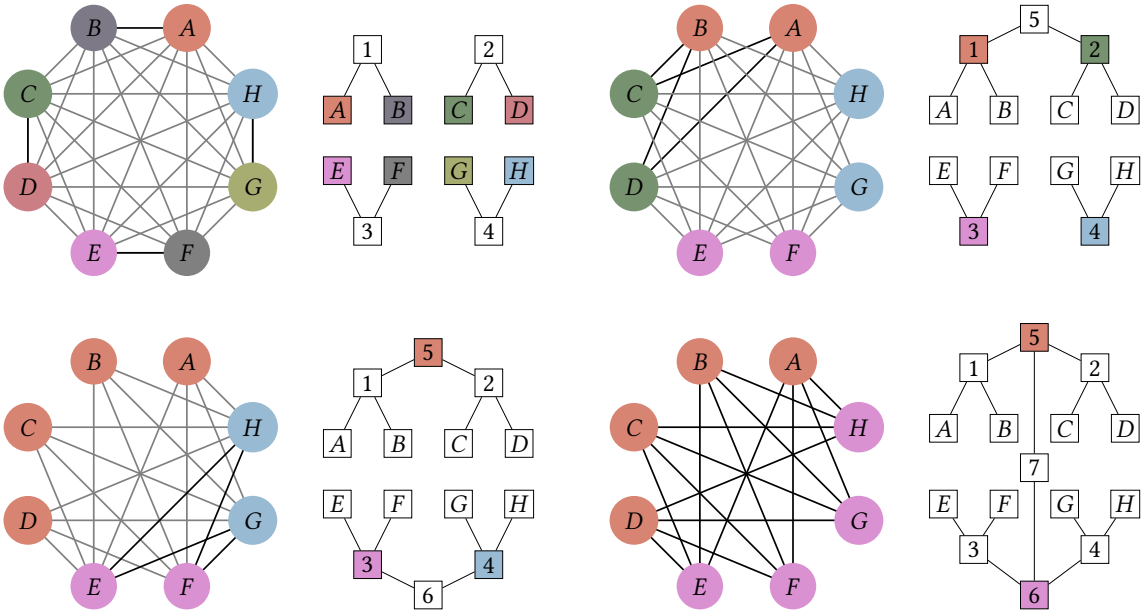


Figure 3.6: *Snapshots from running the vtree bottom-up learning strategy with pairwise mutual information. Snapshots show pairings of two vtrees, with edges between partitions joined into a single edge whose weight is the average pairwise mutual information of all collapsed edges. In black are edges 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.*

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). 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 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.* (2017) 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 $\mathrm{P}_1, \ldots, \mathrm{P}_k$ products such that $\pi_1, \ldots, \pi_k$ (primes of $\mathrm{P}_1, \ldots, \mathrm{P}_k$ respectively) are mutually exclusive. This is done by attributing all possible values of a variable in $\mathrm{Sc}(\mathrm{P})$, 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 $\mathcal{C}_\mathrm{P}$ into k circuits $\mathcal{C}_\mathrm{P}^{(1)}, \ldots, \mathcal{C}_\mathrm{P}^{(k)}$ up to some depth m and then conditioning $\mathcal{C}_\mathrm{P}^{(i)}$ 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.

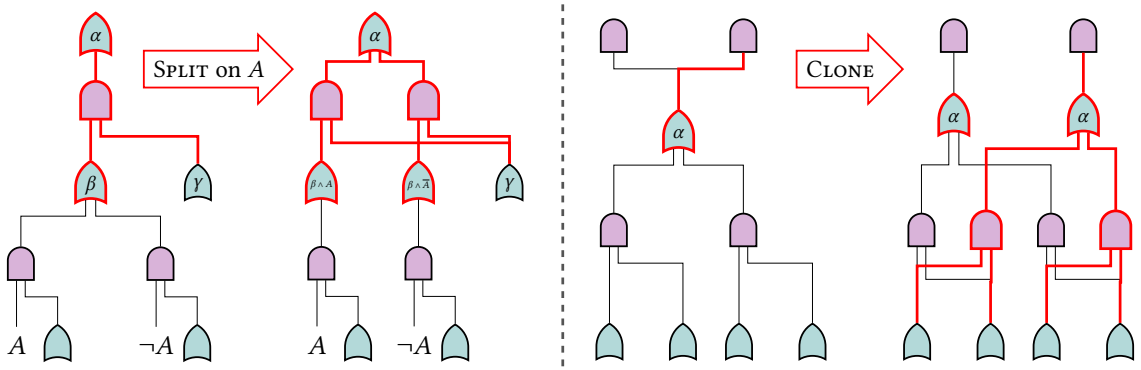


Figure 3.7: *Split (left) and Clone (right) operations for growing a circuit when $m = 1$. Nodes and edges highlighted in red show the modified structure. In both cases smoothness, (structure) decomposability and determinism are evidently preserved.*

The other proposed transformation, Clone, does something similar for sum nodes. Pick a sum node S whose children are $C_1, \ldots, C_k$ and parents P_1 and P_2; double S and $C_1, \ldots, C_k$, producing clones S' and $C'_1, \ldots, C'_k$. Disconnect the edge coming from P_2 to S and instead connect it to S'. Connect all $C'_1, \ldots, C'_k$ 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 $C'_1, \ldots, C'_k$.

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 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

$$\text{Score}(\mathbf{D}, \mathcal{C}, \mathcal{C}') = \frac{\log \mathcal{C}'(\mathbf{D}) - \log \mathcal{C}(\mathbf{D})}{|\mathcal{C}'| - |\mathcal{C}|}, \tag{3.3}$$

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 9 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

Algorithm 9 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 $\mathbf{D}$

while there is score improvement or has not reached the iteration/time limit **do**
  $s_{\mathrm{S}} \leftarrow -\infty$
  Let $(\mathrm{S}^*, \mathrm{P}^*)$ be the best Split candidate seen so far, initially empty
  for each candidate (S, P) of all possible Split candidates **do**
    $\mathcal{C}' \leftarrow \textsc{Split}(\mathcal{C}, \mathrm{S}, \mathrm{P}, \mathcal{V}, m)$
    $s' \leftarrow \text{Score}(\mathbf{D}, \mathcal{C}, \mathcal{C}')$
    if $s' > s_{\mathrm{S}}$ **then** $s_{\mathrm{S}} \leftarrow s'$ and $\mathrm{S}^*, \mathrm{P}^* \leftarrow \mathrm{S}, \mathrm{P}$
  $s_{\mathrm{C}} \leftarrow -\infty$
  Let C^* be the best Clone candidate seen so far, initially empty
  for each candidate C of all possible Clone candidates **do**
    $\mathcal{C}' \leftarrow \textsc{Clone}(\mathcal{C}, \mathrm{C}, \mathcal{V}, m)$
    $s' \leftarrow \text{Score}(\mathbf{D}, \mathcal{C}, \mathcal{C}')$
    if $s' > s_{\mathrm{C}}$ **then** $s_{\mathrm{C}} \leftarrow s'$ and $\mathrm{C}^* \leftarrow \mathrm{C}$
  if $s_{\mathrm{S}} > s_{\mathrm{C}}$ **then** $\mathcal{C} \leftarrow \textsc{Split}(\mathcal{C}, \mathrm{S}^*, \mathrm{P}^*, \mathcal{V}, m)$
  else $\mathcal{C} \leftarrow \textsc{Clone}(\mathcal{C}, \mathrm{C}^*, \mathcal{V}, m)$
return $\mathcal{C}$

in $\mathcal{O}(|\mathbf{X}|^2)$ (Karpis 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 $|\operatorname{Val}(X)|$, the number of possible assignments to X if X is discrete; or the number of intervals to fragment $\operatorname{Val}(X)$ 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 (as well as structure decomposability if different vtrees are used at each component), denying the access to tractably computing queries like divergences, MI and entropies, not to mention the time cost to learn all components. 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). 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.

3.2.2 Strudel

Dang, Vergari, *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, Vergari, *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 $\mathcal{T}$ be a CLT over variables $\mathbf{X} = \{X_1, \ldots, X_m\}$. 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

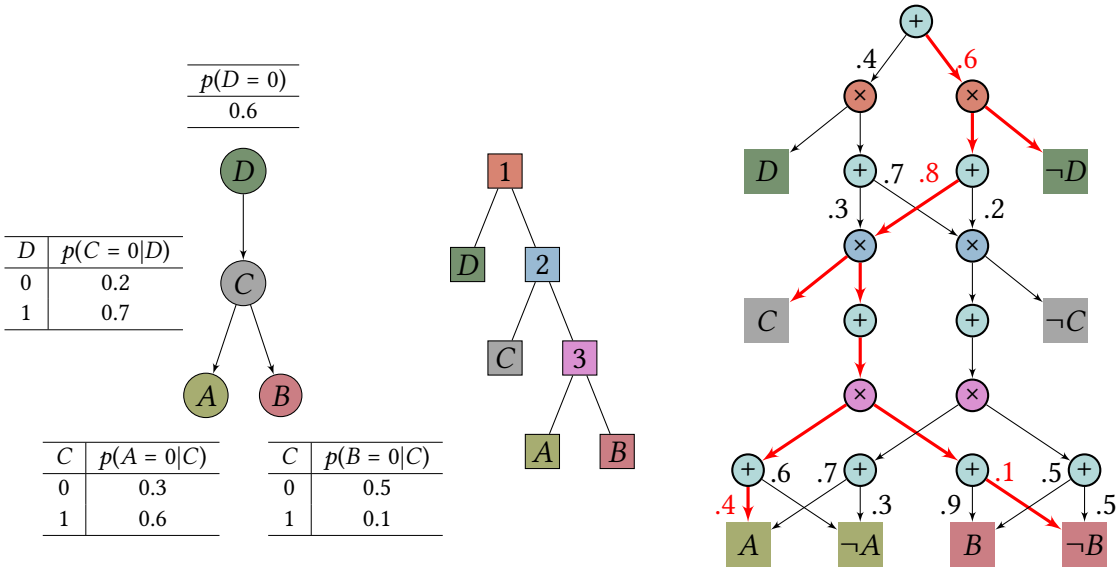


Figure 3.8: *A vtree (middle) and probabilistic circuit (right) compiled from a Chow-Liu Tree (left). Each conditional probability $p(Y|X)$ is encoded as a (deterministic) sum node where each of the two children sets Y to 0 or 1. Colors in the CLT indicate the variables in the PC, while vtree inner node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

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 be 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 $w_{\text{S}_0,\neg Y} = p(Y = 0|X = 0)$, $w_{\text{S}_0,Y} = p(Y = 1|X = 0)$ and $w_{\text{S}_1,\neg Y} = p(Y = 0|X = 1)$, $w_{\text{S}_1,Y} = p(Y = 0|X = 1)$. 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 $\text{P}_1, \ldots, \text{P}_k$, 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 ⊗ 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 ⊗, where C is the

Algorithm 10 InitialStrudel

Input Data $\mathbf{D}$, whose columns are indexed by variables $\mathbf{X}$
Output A smooth, structure decomposable and deterministic initial PC and vtree

$\mathcal{T} \leftarrow$ LearnCLT($\mathbf{D}, \mathbf{X}$)
$\mathcal{V} \leftarrow$ CompileVtree($\mathcal{T}$)
Let $\mathcal{M}$ be a hash table for caching circuits, initially empty
for each vtree node $v \in \mathcal{V}$ in reverse topological order **do**
    if v is a leaf node **then**
        Let $X \in \mathcal{T}$ be the variable represented by v, and Y its parent
        if X is a leaf node in $\mathcal{T}$ **then**
            $\mathrm{S}_j \leftarrow \sum_{i \in \mathrm{Val}(X)} p(X = i | Y = j) \cdot [\![X = i]\!]$ for each $j \in \mathrm{Val}(Y)$
            $\mathcal{M}(v) \leftarrow \mathcal{M}(v) \cup \{\mathrm{S}_j \mid \forall j \in \mathrm{Val}(Y)\}$
        else
            $\mathcal{M}(v) \leftarrow \mathcal{M}(v) \cup \{[\![X = i]\!] \mid i \in \mathrm{Val}(X)\}$
    else
        Attribute $\mathbf{X} \leftarrow \mathrm{Sc}(v^{\leftarrow})$ and $\mathbf{Y} \leftarrow \mathrm{Sc}(v^{\rightarrow})$
        Let $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$ subsets of each scope
        Attribute $\mathbf{N}^{\leftarrow} \leftarrow \mathcal{M}(v^{\leftarrow})$ and $\mathbf{N}^{\rightarrow} \leftarrow \mathcal{M}(v^{\rightarrow})$
        $k \leftarrow |\mathrm{Val}(X)|$
        Construct product nodes $\mathbf{P} = \{\mathbf{N}_i^{\leftarrow} \cdot \mathbf{N}_i^{\rightarrow} \mid \forall i \in [k]\}$
        if $\mathrm{Pa}(X) = \mathrm{Pa}(Y)$ **then**
            Create sum nodes S_i each with only a single child $\mathrm{P}_i \in \mathbf{P}$, for each $i \in [k]$
            $\mathcal{M}(v) \leftarrow \mathcal{M}(v) \cup \{\mathrm{S}_1, \ldots, \mathrm{S}_k\}$
        else
            $\mathrm{S}_j \leftarrow \sum_{i \in \mathrm{Val}(X)} p(\mathbf{Y} | X = j) \cdot [\![X = i]\!]$, for each $j \in \mathrm{Val}(Y)$
            $\mathcal{M}(v) \leftarrow \mathcal{M}(v) \cup \{\mathrm{S}_j \mid \forall j \in \mathrm{Val}(Y)\}$
return $\mathcal{M}(v_r)$, where v_r is $\mathcal{V}$'s root node

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 ⊗ and $p(A, B|C = 1)$ for the left-most sibling. This procedure is shown more formally in Algorithm 10.

Now that 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, Vergari, *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 $\mathcal{F}$

Algorithm 11 STRUDEL

Input Data $\mathbf{D}$, max depth m, scope $\mathbf{X}$
Output A smooth, structure decomposable and deterministic PC learned from $\mathbf{D}$

$\mathcal{C}, \mathcal{V} \leftarrow \text{InitialStrudel}(\mathbf{D}, \mathbf{X})$
while there is score improvement of has not reached the iteration/time limit **do**
    Compute the aggregate flow over all edges
    $w^*_{\mathrm{S,C}} \leftarrow \arg\max_{w \in \mathcal{W}_\mathcal{C}} \text{Score}_{\text{eFlow}}(w|\mathcal{C}, \mathbf{D})$
    $X^* \leftarrow \arg\max_{X \in \text{Sc(S)}} \text{Score}_{\text{vMI}}(X, w^*_{\mathrm{S,C}}|\mathcal{C}, \mathbf{D})$
    $\mathcal{C} \leftarrow \text{Split}(\mathcal{C}, \mathrm{S}, \mathrm{C}, \mathcal{V}, m)$
return $\mathcal{C}$

that encodes the log-likelihood computation

$$\mathcal{C}(\mathbf{x}) = \mathcal{F}_\mathcal{C}(\mathbf{x}) = \prod_{(\mathrm{S,C}) \in \text{Edges}(\mathcal{F}_\mathcal{C})} w_{\mathrm{S,C}} \prod_{\mathrm{L} \in \text{Inputs}(\mathcal{F}_\mathcal{C})} p_\mathrm{L}(\mathbf{x}), \tag{3.4}$$

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 $f_\mathcal{C} : \mathcal{X} \rightarrow \{0, 1\}^{|\mathcal{W}_\mathcal{C}|}$, 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

$$\log \mathcal{C}(\mathbf{x}) = f_\mathcal{C}(\mathbf{x})^\intercal \cdot \log(\mathcal{W}_\mathcal{C}). \tag{3.5}$$

Importantly, by aggregating circuit flows through counting the number of activations of each parameter $w_{\mathrm{S,C}}$ in the entire training dataset $\mathbf{D}$, we get a sense of the number of samples $w_{\mathrm{S,C}}$ 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

$$\text{Score}_{\text{eFlow}}(w_{\mathrm{S,C}}|\mathcal{C}, \mathbf{D}) = \sum_{\mathbf{x} \in \mathbf{D}} f_\mathcal{C}(\mathbf{x})\,[w_{\mathrm{S,C}}], \tag{3.6}$$

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

$$\text{Score}_{\text{vMI}}(X, w_{\mathrm{S,C}}|\mathcal{C}, \mathbf{D}) = \sum_{\substack{Y \in \text{Sc(S)} \\ Y \neq X}} \text{MI}(X, Y). \tag{3.7}$$

The entire algorithm for STRUDEL is showcased in Algorithm 11.

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 $\mathcal{O}(|\mathbf{X}| \cdot |\operatorname{Val}(X)|)$, or linear on $|\mathbf{X}|$ if we assume binary variables as originally intended. The bulk of the computation falls under Strudel, which runs in $\mathcal{O}\left(|\mathbf{X}|^2 \cdot |\mathbf{D}| + i(|C| \cdot |\mathbf{D}| + |\mathbf{X}|^2)\right)$ assuming a bounded max depth m and binary variables. Term $|\mathbf{X}|^2 \cdot |\mathbf{D}|$ corresponds to learning the CLT, $|C| \cdot |\mathbf{D}|$ to the computation of the aggregate circuit flows, $|\mathbf{X}|^2$ to the computation of $\text{Score}_{\text{vMI}}$ 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 of Strudel is its improvement on LearnPSDD's scalability. Compared to LearnPSDD, Strudel can take orders of magnitude less time per iteration, which in practice means a higher number of transformations accomplished in the same range of time. In addition, the nature of circuit flows allows for easy vectorization and thus CPU or GPU parallelization. In terms of data fitness, Dang, Vergari, *et al.* (2020) empirically shows that initial circuits constructed from StrudelInitial greatly improve performance compared to fully factorized initial PCs from LearnPSDD. Similar to LearnPSDD, one can learn an ensemble of Strudels to further boost performance at the cost of losing determinism. Opposite to LearnPSDD however, Dang, Vergari, *et al.* employ structure-sharing components so that the act of learning the circuit's structure is done once, greatly reducing learning time. Parameters are then learned through closed form EM (see Remark 3.3) and bagging.

Cons. Although Strudel's greedy heuristic search strategy translates into possibly more accurate PCs, it also produces more sizable circuits when compared to the exhaustive search of LearnPSDD. Indeed, Dang, Vergari, *et al.* (2020)'s empirical evaluation shows Strudel PCs up to 12 times bigger than LearnPSDD's with the two somewhat tied in terms of fitness. This is especially worrying given that Strudel's complexity grows with its circuit size. In fact, experiments show a sharp increase in seconds per iterations for the two `INCR` algorithms, with both reaching multiple digits for each iteration even in smaller sized datasets; though LearnPSDD much sharper and sooner comparatively (Dang, Vergari, *et al.*, 2020).

Remark 3.2: On the choice of initial circuits

We only briefly mentioned in Section 3.2.1 how we might want to start out with an initial PC conveying a specific support and then run an `INCR` class algorithm to further boost its probabilistic expressiveness without changing the underlying knowledge base. We devote this remark segment to discussing several works in literature that construct a so-called *canonical* (i.e. minimal with respect to their size without sacrificing its logical semantics) logic circuit, becoming perfect candidates to be used as an initial circuit in `INCR` learners.

Just like in probabilistic reasoning, the field of knowledge compilation and symbolic reasoning is often interested in finding succinct representations capable of tractably computing queries, a subject which we briefly touched in Section 2.3.1. For this reason, smooth, structure decomposable and deterministic logic circuits, who usually go by the name of Sentential Decision Diagrams (SDDs, Darwiche, 2011) have proven to be a useful tool in several applications (Vlasselaer, Renkens, *et al.*, 2014; Vlasselaer, Broeck, *et al.*, 2015; Lomuscio and Paquet, 2015; Herrmann and Barros, 2013). Fortunately, both LearnPSDD and Strudel preserve all the necessary structural constraints for both logical *and* probabilistic queries in (P)SDDs. With this in mind, we highlight compilation of SDDs in this short remark.

For most cases, SDDs can be compiled directly from CNFs and DNFs. (A. Choi and Darwiche, 2013) constructs SDDs bottom-up by first compiling C/DNF clauses and then combining smaller SDDs by either conjoining or disjoining them. In contrast, (Oztok and Darwiche, 2015) presents a faster compilation process which recursively breaks down C/DNFs by decomposing the formula into components according to a vtree, and then combines them into an SDD.

Although CNFs and DNFs are the most common form of encoding propositional knowledge bases, they struggle under specific logical constraints such as cardinality constraints (Nishino *et al.*, 2016; Sinz, 2005). Interesting alternatives include BDDs (see Example 2.6) which are also widely used in formal methods and program verification, and for which efficient compilation from cardinality constraints are available (Eén and Sörensson, 2006). Because BDDs are special case SDDs whose vtrees are always right-linear (Darwiche, 2011; Bova, 2016), their reduced representations (Bryant, 1986) are natural initial circuit candidates (see Section 4.2).

We now cover some of the existing literature on producing task specific (P)SDDs. A. Choi, Tavabi, *et al.* (2016) analyzes the feasibility of compiling LCs (and subsequently producing a PC by parameterizing disjunction edges) from tic-tac-toe game traces, and route planning within a city. Both involve exhaustively disjoining all permutations of valid conjoined configurations and compiling the resulting DNF through previously cited SDD compilers. A. Choi, Shen, *et al.* (2017) further studies route planning by compiling them into SDDs, but analyze the feasibility of (P)SDDs in route planning in larger scale maps. A. Choi, Broeck, *et al.* (2015) explore (P)SDDs in preference learning and rankings, providing an algorithm for compiling an SDD from total or partial rankings. Similarly, Shen *et al.* (2017) investigates (P)SDDs in probabilistic cardinality constraint tasks, also known as subset selection or n-choose-k models.

3.3 Random Learning

We now explore random approaches to constructing probabilistic circuits, which we classify as `RAND` class learning algorithms. Essentially, `RAND` class circuits are constructed either by a completely random procedure (Section 3.3.1), or guided by data (Section 3.3.2). We first look at RAT-SPN (Peharz, Vergari, *et al.*, 2020), a connectionist PC structure

learning algorithm for randomly generating tensorized smooth and decomposable probabilistic circuits. We then address XPC (Mauro *et al.*, 2021), a flexible algorithm capable of learning smooth and decomposable PCs as well as structure decomposable and/or deterministic circuits through simple modifications to their method.

3.3.1 RAT-SPN

A key ingredient to RAT-SPN (Peharz, Vergari, *et al.*, 2020) is the concept of *region graphs*. First introduced in PC literature in Dennis and Ventura (2012), region graphs are tensorized templates for PC construction. Informally, a region graph is composed out of *region nodes* and *partition nodes*; the former is a set of sum or input nodes, and the latter of products. Regions can be thought of sets of computational units explaining the same interactions among variables (Dennis and Ventura, 2012), for instance semantically similar pixel regions in an image; while partitions define independencies between these regions. Edges coming out of region (resp. partition) nodes must necessarily connect to a partition (resp. region) node.

Definition 3.3.1 (Region graph)**.** *A* region graph *is a rooted connected DAG whose nodes are either* regions *or* partitions*. Children of regions are partitions, and children of partitions are regions. The root is always a region node.*

Region graphs simplify the process of constructing PCs by ensuring that they are *at the least* smooth and decomposable. Call $\mathcal{G}$ a region graph; $\mathcal{G}$ is easily translated to a PC by applying the procedure described in Algorithm 12. Every region is compiled into a set of sums or inputs, fully connecting children; every partition into a set of products, producing a distinct permutation of children. Evidently, the resulting PC is exponential on s and l, as products must ensure that they encode different permutations. To deal with this blow-up, this number is often restricted to only two children per partition.

RAT-SPN works by randomly generating a region graph in a top-down divide-and-conquer approach similar to LearnSPN, except that the learned structure eventually produces non-tree shaped circuits and the procedure is done *completely* random (see Algorithm 13). In fact, Peharz, Vergari, *et al.* (2020) argue that the parameterization of the circuit by means of sum weights is as important as its structure, looking at probabilistic circuits as a specific subclass of neural networks. Indeed, they show that this connectionist approach heavily inspired by traditional deep learning produces very competitive PCs. However, to do so requires extensive optimization of the circuit's weights, which unsurprisingly is where RAT-SPN shines: because of the tensorized nature of region graphs, the resulting PC is able to exploit the advantages of known deep learning frameworks, making the most of efficient stochastic gradient descent optimizers and GPU parallelization.

To ensure that the compiled PC is smooth and decomposable, the region graph must also be so. We extend the definition of scope function to region graphs. As long as leaf region nodes are assigned the correct scope, the PC is by construction smooth (every region is fully connected to their children) and decomposable (every partition splits variables into two nonoverlapping regions). Function CreateLayer in Algorithm 13 does exactly that, making sure each partition decomposes into two distinct variable splits down to leaf region nodes. How deep the region graph (and consequentially the resulting PC) goes

Algorithm 12 CompileRegionGraph

Input A region graph $\mathcal{G}$, parameters s for sums and l for inputs
Output A smooth and decomposable probabilistic circuit

Let $\mathcal{M}$ be a mapping of region nodes to PC nodes
for each node N in $\mathcal{G}$ except the root in reverse topological order **do**
    if N is a region **then**
        if N is a leaf node in $\mathcal{G}$ **then**
            Construct $\mathbf{L} = \{\mathsf{L}_1, \dots, \mathsf{L}_l\}$ input nodes over variables Sc(N)
            Associate N with $\mathbf{L}$
        else
            Construct $\mathbf{S} = \{\mathsf{S}_1, \dots, \mathsf{S}_s\}$ sum nodes
            for each partition node $\mathsf{P} \in \mathrm{Ch}(\mathsf{N})$ **do**
                Every sum in $\mathbf{S}$ connects with every product in $\mathcal{M}(\mathsf{P})$
    else if N is a partition **then**
        Let $\mathrm{Ch}(\mathsf{N}) = \{\mathsf{R}_1, \dots, \mathsf{R}_k\}$ be regions and $q = \prod_{i=1}^{k} |\mathcal{M}(\mathsf{R}_i)|$
        Construct $\mathbf{P} = \{\mathsf{P}_1, \dots, \mathsf{P}_q\}$ product nodes
        for every product $\mathsf{P} \in \mathbf{P}$ **do**
            Connect P with a distinct combination of sums in $\mathcal{M}(\mathsf{R}_1), \dots, \mathcal{M}(\mathsf{R}_j)$
Construct a root node R
Connect R to all products in every child of $\mathcal{G}$'s root
return R

depends on a parameter d, which corresponds to half the true depth, as each CreateLayer call produces a partition and their children. After the region graph is randomly buily, a PC is then constructed through CompileRegionGraph, passing the random region graph and number of nodes per region as parameters. The function ultimately produces a dense probabilistic circuit from the region graph blueprint, as Figure 3.9 exemplifies.

Once the PC structure is successfully generated, Peharz, Vergari, *et al.* (2020) suggest Expectation-Maximization (EM, Dempster *et al.*, 1977; Peharz, Gens, Pernkopf, *et al.*, 2016; H. Zhao, Poupart, *et al.*, 2016) for optimizing the circuit parameters (i.e. sum weights and input node distributions). Although parameter learning of PCs is not the focus of this dissertation, we briefly touch the subject in Remark 3.3.

Worthy of note is a discriminative version of RAT-SPN where instead of a single root sum node, k roots are learned, each connecting to every CreateLayer subcircuit. Each i-th root describes the conditional probability $p(\mathbf{X}|Y = i)$, where Y is the query variable and $\mathbf{X}$ evidence. Classification follows directly from Bayes Rule $p(Y|\mathbf{X}) = \frac{p(\mathbf{X}|Y)p(Y)}{\sum_{i=1}^{k} p(\mathbf{X}|Y=i)p(Y=i)}$, where $p(Y)$ is either estimated from the training data or assumed to be fixed. Accordingly, a discriminative objective function is proposed involving cross-entropy and log-likelihood for hybrid generative-discriminative optimization (Bouchard and Triggs, 2004) instead of running EM.

Complexity

Although the procedure described in Algorithm 13 is $\mathcal{O}(r \cdot d(s + l))$ if $d < |\mathbf{X}|$ and $\mathcal{O}\left(r \cdot \log_2 |\mathbf{X}|(s + l)\right)$ otherwise, making the algorithm extremely fast, it does not paint

Algorithm 13 RAT-SPN

Input Data $\mathbf{D}$, variables $\mathbf{X}$, max depth d, r # subcircuits, s # sums, and l # inputs
Output A smooth and decomposable probabilistic circuit

function CreateLayer(R, d, $\mathbf{X}$)
    Assign $\mathbf{X}$ as Sc(R)
    Sample a variable split $(\mathbf{Y}, \mathbf{Z})$ from $\mathbf{X}$
    Create a partition P and add it as a child of R
    if d>1 **then**
        if $|\mathbf{Y}| > 1$ **then**
            Create a region R_1
            CreateLayer(R_1, $d-1$, $\mathbf{Y}$)
        if $|\mathbf{Z}| > 1$ **then**
            Create a region R_2
            CreateLayer(R_2, $d-1$, $\mathbf{Z}$)
Start with a root region node R
for each $i \in [r]$ **do**
    CreateLayer(R, d, $\mathbf{X}$)
$C \leftarrow$ CompileRegionGraph(R, s, l)
return C

the whole picture. The main bulk of the complexity when learning RAT-SPN comes from parameter learning. Peharz, Vergari, *et al.* (2020) calculate the number of sum weights to be

$$|\mathcal{W}_C| = \begin{cases} r \cdot k \cdot l^2 & \text{if } d = 1, \\ r \cdot \left(k \cdot s^2 + (2^{d-1} - 2)s^3 + 2^{d-1} \cdot s \cdot l^2\right) & \text{if } d > 1; \end{cases} \tag{3.8}$$

if we assume that the number of children of partitions is at most two. This means that learning only the non-input parameters of RAT-SPN takes time $\mathcal{O}\left(\left(r \cdot 2^d \cdot \left(s^3 + s \cdot l^2\right) + r \cdot k \cdot s^2\right) \cdot |\mathbf{D}|\right)$. However, given that most structure learning algorithms covered in this dissertation also require parameter learning, one might argue that the true cost of structure learning in RAT-SPN is indeed subquadratic.

Pros and Cons

Pros. As expected from RAND algorithms, the random, data-blind nature of RAT-SPN makes for a very fast structure learning algorithm. More importantly, because the structure is expected to have somewhat uniform layers with a fixed number of computational units in each, the computations from parameter optimization can easily be brought to the GPU. This not only helps with scalability in terms of speed, but also brings all the advantages of deep learning frameworks to the table via well-studied stochastic gradient descent optimizers and diagnostic tools.

Cons. Clearly, RAT-SPN is *completely* random with its structure generation. Particularly, variable splits are done randomly, disregarding the independencies encoded by data, meaning that certain factorizations may be assumed to be true when they would otherwise not be. Although RAT-SPNs are certainly competitive against other learning algorithms

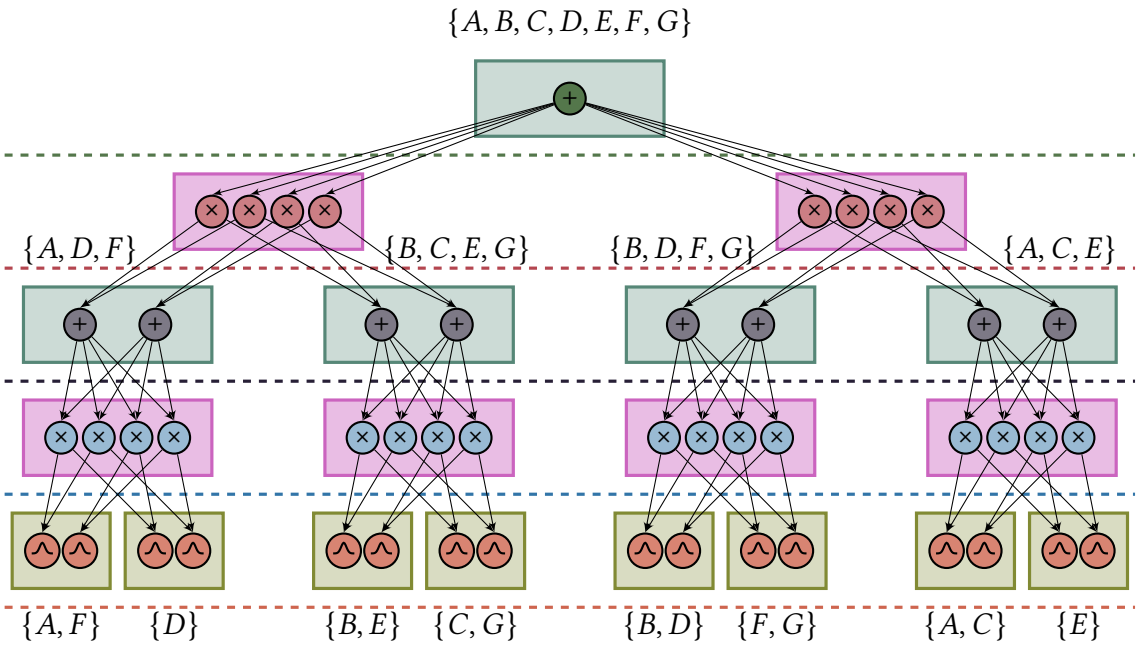


Figure 3.9: *A RAT-SPN generated from parameters $d = 3$, $r = 2$, $s = 2$ and $l = 2$. Nodes within a belong to inner region nodes, to partitions and to leaf regions; dashed lines (and node colors) indicate different PC layers. Scope of region nodes are shown in curly braces.*

for PCs, they only produce smooth and decomposable circuits, denying the access to more complex queries.

Remark 3.3: On parameter learning of probabilistic circuits

Literature in probabilistic circuits often divides learning into two (often distinct) tasks: structure learning and parameter learning. Although in this dissertation we almost exclusively cover *structure* learning algorithms, it is worth also going through (even if superficially) some of the works on parameter optimization in PCs, as most structure learning algorithms assume this as a post-processing procedure.

Expectation-Maximization (EM, Dempster *et al.*, 1977) is perhaps the most common maximum likelihood estimation (MLE) optimization procedure for probabilistic circuits. Poon and P. Domingos (2011), Peharz, Tschiatschek, *et al.* (2015) and H. Zhao, Poupart, *et al.* (2016) derived EM for generative learning in PCs, while Rashwan, Poupart, *et al.* (2018) formulated a discriminative EM version for PCs through Extended Baum-Welch (Gopalakrishnan *et al.*, 1991).

Notably, when a circuit is smooth, decomposable and deterministic, MLE can be easily computed through closed-form by counting (Kisa *et al.*, 2014; Peharz, Gens, and P. Domingos, 2014). Indeed, this is an attractive feature that extends to discriminative PCs (Liang and Van den Broeck, 2019).

Following other more traditional deep learning models, PCs learned with stochastic gradient descent (SGD) have also appeared in literature, especially under convolutional and tensorial extensions (Sharir, Tamari, *et al.*, 2018; Peharz, Vergari,

et al., 2020; Peharz, Lang, *et al.*, 2020; Gens and P. Domingos, 2012).

Bayesian approaches have also received some attention by the PC community. Jaini, Rashwan, *et al.* (2016) and Rashwan, H. Zhao, *et al.* (2016) developed online Bayesian moment matching algorithms to learn from streaming data; H. Zhao, Adel, *et al.* (2016) showed a variational optimization procedure that leverages inference tractability in PCs to efficiently compute the ELBO; Trapp, Peharz, Ge, *et al.* (2019) propose learning both structure and parameters by Gibbs sampling from a generative model on a (restricted) space of PCs; finally Vergari, Molina, *et al.* (2019) propose PCs for automatic Bayesian density analysis.

3.3.2 XPC

While RAT-SPN produces a data-blind PC architecture and then relies on parameter optimization to learn from data, the algorithm that we shall see next does the exact opposite: XPC (Mauro *et al.*, 2021) randomly samples a structure from data and requires no parameter learning. To do this, it restricts the circuit sampling space to a particular class of PCs whose primes are logical restrictions and inputs are CLTs. By assigning a fixed number $k + 1$ of (product) children per sum node and assuming that the k first primes are (random) conjunctions of literals of a fixed length t, with the last $k+1$ prime their negation, the resulting PC is naturally deterministic, as CLTs are themselves deterministic. In more practical terms, both the conjunctions of literals as well as the $(k + 1)$-th prime derived from the negation of the first k conjunctions are translated into products of (degenerate) Bernoullis. Determinism can be relaxed by applying any form of regularization, for instance Laplace smoothing, both on CLTs and on the products of Bernoullis.

Following the footsteps of RAT-SPN, they generate a tree-shaped random region graph and produce a PC from it. Despite both employing region graphs, the graph in Mauro *et al.* (2021) is only used as an artifice for formalizing the structure construction: their process for reconstructing a PC from a region graph boils down to replacing an inner region with a single sum, a leaf region with a single input and a partition with a single product. Although one *could* generate a non-tree shaped PC by setting $s > 1$ and $l > 1$ (i.e. number of sums and inputs per inner and leaf region respectively), the resulting circuit could be reduced to a tree, since both sums (and inputs) coming from the same region would be syntatically the same[3].

A critical step to efficiently ensuring consistency with the logical restrictions is to assign only consistent subsets of data to subcircuits. Just like in `DIV` class algorithms, partition (i.e. product) nodes in the region graph define column splits over data and regions (i.e. sums and inputs) correspond to row splits. The algorithm then associates a node with a portion of data according to its scope and logical constraint. More specifically, when the i-th prime defines a conjunction of literals α_i, only assignments $\mathbf{x}$ whose application $\alpha_i(\mathbf{x})$ are true are transferred down, effectively splitting data row-wise. To ensure that the

[3] Whether this "expanded" circuit could have its performance improved if one were to run, say EM, to exploit this increase in capacity is an interesting question that unfortunately was left unexplored in Mauro *et al.* (2021).

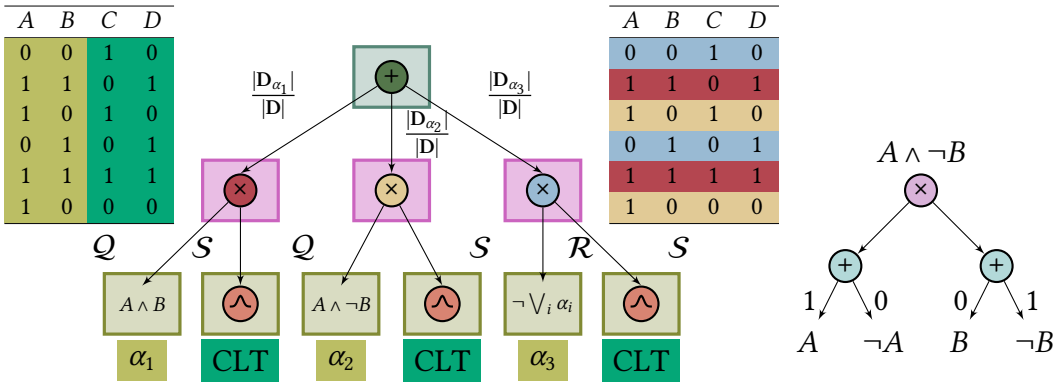


Figure 3.10: *The first iteration of XPC, where $t = 2$ variables are selected, A and B; $k = 2$ conjunctions of literals are sampled, $\alpha_1 = A \wedge B$, $\alpha_2 = A \wedge \neg B$ and $\alpha_3 = \neg(\alpha_1 \vee \alpha_2)$; with primes set to a product of Bernoullis corresponding to each α_i and subs to CLTs. Leaf region nodes $\mathcal{S}$ are candidates for expansion. Sums, products and CLT input nodes are the resulting probabilistic circuit from the sampled region graph. Conjunctions of literals are expanded into product of Bernoullis whose weights are inferred from data, as the circuit on right shows; if no smoothing is applied, the circuit is deterministic.*

resulting circuit is also decomposable, only $\text{Sc}(\alpha_i)$ variables are passed to primes, with the remaining variables going to the subs. This way, sum weights can be estimated as the subdata size ratios just like in LearnSPN, as Figure 3.10 shows. Each subdata $\mathbf{D}_{\alpha_i}$ corresponds to joining all assignments consistent with α_i, as shown through matching colors on the table on the right. Data is divided by rows according to the satisfiability of each prime α_i and by column according to the scope of each prime.

At each iteration of Algorithm 15, sub leaf region nodes, denoted here as $\mathcal{S}$ regions, are randomly selected for further expansion. Algorithm 14 takes a candidate region R and samples k conjunctions $\alpha_1, \ldots, \alpha_k$ of length t that appear at least δ times in the subdata associated with R. If no such constraint has been found, R is discarded as a candidate for expansion. Otherwise, another layer of partitions and regions similar to Figure 3.10 is constructed, making sure that data splits obey both scope and constraints α_i. Once no more candidates are available or a limit on the number of expansions has been reached, the region graph is translated into a probabilistic circuit by replacing leaf region nodes of type $\mathcal{Q}$ into products of Bernoullis, type $\mathcal{S}$'s into CLTs and inner regions as sums whose weights are the proportions of (row-wise) split data; partitions are replaced with product nodes.

Determinism is not the only constraint that can be enforced. In fact, since every product is by construction 2-standardized, the circuit is structure decomposable if not only the scopes of $\mathcal{Q}$ regions follow a vtree, but the CLTs in $\mathcal{S}$ regions do as well. This forces the sampling done in line 3 to instead deterministically assign $\mathbf{Y}$ the scope of $\text{Sc}(v^{\leftarrow})$, where v is the vtree node associated with the product splitting $\mathbf{X}$.

Complexity

To simplify, we assume that all k distinct logical constraints pass the condition of containing at least δ samples in the dataset. With this, the complexity of ExpandXPC is $\mathcal{O}(t + k \cdot |\mathbf{D}|)$, where the first term comes from sampling a subset of size t from $\mathbf{X}$ and the

Algorithm 14 ExpandXPC

Input Region R, data $\mathbf{D}$, variables $\mathbf{X}$, min. # of assignments per partition δ, # of conjunctions of literals k, length of conjunctions t

Let $\mathbf{A}$ be a set of logical constraints initially empty
Copy all data from $\mathbf{D}$ to S
Sample a subset Y of size t from $\mathbf{X}$
while $|\mathbf{A}| < k$ **do**
    Sample a conjunction of literals α over Y distinct from any in $\mathbf{A}$
    $\mathbf{Q} \longleftarrow \{\mathbf{x} \mid \forall \mathbf{x} \in \mathbf{S} \wedge \alpha(\mathbf{x}) = 1\}$
    if $|\mathbf{Q}| \geq \delta$ **and** $|\mathbf{S} \setminus \mathbf{Q}| \geq \delta$ **then**
        Append α to $\mathbf{A}$
        $\mathbf{S} \longleftarrow \mathbf{D} \setminus \mathbf{Q}$
        Create a partition node P as a child of R
        Create a region R'_p of type $\mathcal{Q}$ and assign it as a prime of P
        Assign scope Y and data $\mathbf{Q}$ to R'_p
        Create a region R'_s of type $\mathcal{S}$ and assign it as a sub of P
        Assign scope $\mathbf{X} \setminus \mathbf{Y}$ and data $\mathbf{S} \setminus \mathbf{Q}$ to R'_s
if no constraint is suitable **then** unset R as a candidate and **return**
else
    Create a partition node P as a child of R
    Create a region R'_p of type $\mathcal{R}$ and assign it as a prime of P
    Assign scope Y and data S to R'_p
    Create a region R'_s of type $\mathcal{S}$ and assign it as a sub of P
    Assign scope $\mathbf{X} \setminus \mathbf{Y}$ to R'_s

second from selecting all assignments in $\mathbf{D}$ that agree with the constraint α. Sampling conjunctions of literals can be done in constant time by representing α as a bit vector where a 1 indicates a positive literal and 0 a negative literal; sampling a number in $\left[0, 2^{|\mathrm{Y}|-1}\right]$ (here assumed to be done in $\mathcal{O}(1)$) is equivalent to producing α.

The analysis for XPC relies on either the number of maximum iterations or available candidates. At every call to ExpandXPC, we create at least $3(k+1)$ new PC nodes, of which $k+1$ of them are $\mathcal{S}$ regions. Assuming that we let the algorithm run a fixed number of iterations i, we get a total runtime of $\mathcal{O}\left(i \cdot (t + k \cdot |\mathbf{D}|)\right)$. for the main loop in XPC. We then need to compile the PC and learn CLTs for every $\mathcal{S}$ leaf. Because we ran for i iterations, we should have $i \cdot k$ CLTs to learn, which is done in $\mathcal{O}(|\mathbf{X}|^2 \cdot \mathbf{D})$, bringing the total runtime to $\mathcal{O}\left(i \cdot (t + k \cdot |\mathbf{D}|) + i \cdot k \cdot |\mathbf{X}|^2 \cdot |\mathbf{D}|\right)$. Note, however, that this is a rough upper bound on the true complexity, as both scope and data shrink considerably at each depth.

Pros and Cons

Pros. XPC is flexible in the sense that it can produce both deterministic and structure decomposable circuits with little change to the algorithm. More importantly, because it essentially divides data in a `DIV` approach, the most costly operation, i.e. learning CLTs at the leaves, is done extremely fast, since the optimization is done only on a fraction of the data and scope. As an example, learning XPCs from binary datasets of hundreds of variables

Algorithm 15 XPC

Input Dataset $\mathbf{D}$, variables $\mathbf{X}$, min. # of assignments per partition δ, # of conjunctions of literals k, length of conjunctions t

Output A smooth, decomposable and deterministic probabilistic circuit

Start with a region graph $\mathcal{G}$ with a single region node as root
while there are candidate region nodes of type $\mathcal{S}$ **do**
    Select a random region R of type $\mathcal{S}$
    Let $\mathbf{Q}$ be the subdata associated with R
    ExpandXPC(R, $\mathbf{Q}$, Sc(R), δ, k, t)
for each node $\mathsf{N} \in \mathcal{G}$ in reverse topological order **do**
    Let $\mathbf{D}_{\mathsf{N}}$ be the data associated with N
    if N is a leaf region node of type $\mathcal{Q}$ **then**
        Replace N with a product of Bernoullis according to $\mathbf{D}$
    else if N is a leaf region node of type $\mathcal{S}$ **then**
        Replace N with a CLT learned from $\mathbf{D}$
    else if N is a region node **then**
        Replace N with a sum node whose weights are $w_{\mathsf{N},\mathsf{C}} = \frac{\mathbf{D}_{\mathsf{C}}}{\mathbf{D}_{\mathsf{N}}}$ for each $\mathsf{C} \in \mathrm{Ch}(\mathsf{N})$
    else
        Replace N with a product node
return $\mathcal{G}$'s root sum node

and tens of thousands of instances takes a matter of seconds, while most competitors usually take hours for learning from the same data. In terms of performance, although single circuit XPCs rarely beat state-of-the-art competitors, Mauro *et al.* (2021) showed that by merely aggregating sampled circuits into a simple mixture boosts performance considerably, reaching competitive results.

Cons. When it comes to circuit size, although a single XPC generated circuit has comparable size to other state-of-the-art structure learning competitors, for these to be competitive requires ensembles of a few dozens of components, meaning that in its final form, these can be tens of times the size of other structure learners. Moreover, because of the number of parameters involved in sampling these PCs (δ, k, t, number of components per ensemble, and whether to produce deterministic and/or structure decomposable PCs), a grid-search over parameters is necessary to produce optimal results. Although this is generally faster than other structure learning algorithms, the sheer size of all generated circuits from every hyperparameter combination can easily overwhelm memory space.

3.4 A Summary

We finish this review of structure learning algorithms in probabilistic circuits by summarizing some of the more important points raised throughout this chapter in Section 3.4. We list each algorithm seen in Chapter 3, describing their class, time complexity of learning the probabilistic circuit and any other auxiliary data structure, number of hyperparameters needed during learning, which structural constraints are guaranteed to hold in the resulting PCs, and which data (binary $\{0, 1\}$, discrete $\mathbb{N}$ or continuous $\mathbb{R}$) are supported. We use

the same notation used throughout this section for data dimensions: n is the number of examples in a dataset and m is the number of variables of dataset. Other variable names differ in their meaning depending on each learning technique. We next describe some of the assumptions made in order to more concisely summarize the information set in Section 3.4.

For LearnSPN, we assume sums to be learned by c iterations of k-means and products through the G-test. We call k the number of clusters to be learned, and only assume the bare minimum as hyperparameters: k and the G-test p-value, giving a lower bound of 2 on the number of hyperparameters. LearnSPN is easily extensible to continuous data by replacing the G-test with any other continuous alternative, such as mutual information, and learning continuous univariate distributions as inputs.

Recall that ID-SPN learns Markov networks as inputs. If we assume this process to follow the same procedure proposed in Rooshenas and Lowd (2014), then the number of hyperparameters needed for just learning the structure of the Markov network inputs is at least three (per-edge penalty, per-split penalty and score tolerance heuristic[4]). Just like in LearnSPN, we also assume sums and products to be learned from k-means and the G-test for ID-SPN, raising the number of hyperparameters to 5, 2 for sums and products and 3 for inputs. We use the same notation as Section 3.1.2: i is the number of iterations for learning the Markov networks, c is the size of the Markov network being learned, r is a constant bounding the number of improvements, and k is the number of clusters used for sums.

We assume Prometheus to use its more scalable version of learning products by sampling edges from the correlation graph and sums learned from k-means. Because the procedure for learning products is parameterless, we are left with only k as a hyperparameter for sums. We use the same variable notation as the other `DIV` algorithms.

For both LearnPSDD and Strudel, we consider only the maximum depth m when partially copying the circuit during a local transformation as a hyperparameter. We do not consider the maximum number of iterations i as a hyperparameter, as it acts more like a time constraint rather than a parameter to be optimized. We denote C as the probabilistic circuit being learned.

When describing RAT-SPN, we denote r, d, s and l as the number of subcircuits learned, maximum depth of the generated region graph, number of sums per inner region node, and number of inputs per leaf region node, all of which are accounted as hyperparameters in Section 3.4.

In the case of XPC, we call i the number of expansions to carry out in total, t the length of sampled conjunctions of literals and k the number of conjunctions to be sampled per region. Of these, t and k, together with the number of assignments per partition δ, are considered XPC hyperparameters, bringing the total number to three.

[4] See the Libra Toolkit manual for more information (Lowd and Rooshenas, 2015).

Name	Class	Time Complexity	# hyperparams	Smooth?	Dec?	Det?	Str. Dec?	$\{0,1\}$?	$\mathbb{N}$?	$\mathbb{R}$?	Reference
LearnSPN	DIV	$\begin{cases}\mathcal{O}(nkmc) & \text{, if sum}\\ \mathcal{O}(nm^3) & \text{, if product}\end{cases}$	≥ 2	✓	✓	✗	✗	✓	✓	✓	Section 3.1.1
ID-SPN	DIV	$\begin{cases}\mathcal{O}(nkmc) & \text{, if sum}\\ \mathcal{O}(nm^3) & \text{, if product}\\ \mathcal{O}(ic(rn+m)) & \text{, if input}\end{cases}$	$\geq 2+3$	✓	✓	✗	✗	✓	✓	✗	Section 3.1.2
Prometheus	DIV	$\begin{cases}\mathcal{O}(nkmc) & \text{, if sum}\\ \mathcal{O}(m(\log m)^2) & \text{, if product}\end{cases}$	≥ 1	✓	✓	✗	✗	✓	✓	✓	Section 3.1.3
LearnPSDD	INCR	$\begin{cases}\mathcal{O}(m^2) & \text{, top-down vtree}\\ \mathcal{O}(m^4) & \text{, bottom-up vtree}\\ \mathcal{O}(i\lvert C\rvert^2) & \text{, circuit structure}\end{cases}$	1	✓	✓	✓	✓	✓	✗	✗	Section 3.2.1
Strudel	INCR	$\begin{cases}\mathcal{O}(m^2 n) & \text{, CLT + vtree}\\ \mathcal{O}(i(\lvert C\rvert n + m^2)) & \text{, circuit structure}\end{cases}$	1	✓	✓	✓	✓	✓	✗	✗	Section 3.2.2
RAT-SPN	RAND	$\mathcal{O}(rd(s+l))$	4	✓	✓	✗	✗	✓	✓	✓	Section 3.3.1
XPC	RAND	$\mathcal{O}(i(t+kn)+ikm^2n)$	3	✓	✓	✓	✓	✓	✗	✗	Section 3.3.2

4

A Logical Perspective to Scalable Learning

Considering the many benefits and drawbacks of the current state-of-the-art learning algorithms addressed in Chapter 3, and emphasizing the need for scalability and accessibility, we now present one of the main contributions of this dissertation, proposing the first of two novel structure learning algorithms for probabilistic circuits, the second of which is touched on in Chapter 5. We approach the problem of learning scalable PCs from two distinct points of view. In this chapter, we are interested in PCs whose support encodes a given logical constraint as certain knowledge; we show how both the probabilistic issue of data fitness, as well as the logical question of whether the circuit successfully compiles a knowledge base can be accomplished by aggregating PC samples into ensembles of models. The contents of this chapter come from our contributions in R. L. Geh and Denis Deratani Mauá (2021b).

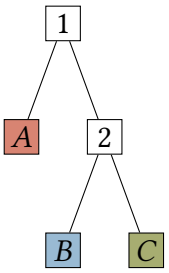
4.1 Sampling PSDDs

Remark 3.2 briefly mentioned the question of compiling logical constraints into smooth, structure decomposable and deterministic logic circuits (i.e. (P)SDDs Darwiche, 2011; Kisa *et al.*, 2014). Indeed, although there are many existing approaches to learning circuits from logical formulae, most are only useful for specific tasks (A. Choi, Tavabi, *et al.*, 2016; A. Choi, Broeck, *et al.*, 2015; Shen *et al.*, 2017; A. Choi, Shen, *et al.*, 2017). Although there are more generalistic ways of producing circuits, namely from CNFs and DNFs (Oztok and Darwiche, 2015; A. Choi and Darwiche, 2013); logic formulae which incorporate more complex relationships such as cardinality constraints either have no tractable representation (Nishino *et al.*, 2016) or require the addition of latent variables (Sinz, 2005). More importantly, because variables which do not play a role in the logical formulae are completely discarded in the compilation process, translating these *logic* circuits into *probabilistic* circuits involves naïve assumptions on the discarded variables, such as fully factorizing them.

Surprisingly, to our knowledge there have been next to no work on learning the structure of PCs from scratch by looking at both logical formulae *and* data. Even worse, the couple that do are restricted to very preliminary work: Mattei, Soares, *et al.* (2019) came up with a `DIV` prototype for a top-down approach to sampling a special class of PSDDs

A	B	C	$p(\mathbf{x})$
0	0	0	0.1
0	1	0	0.1
1	0	0	0.2
1	0	1	0.6

$\phi(A, B, C) = (A \rightarrow \neg B) \wedge (C \rightarrow A)$



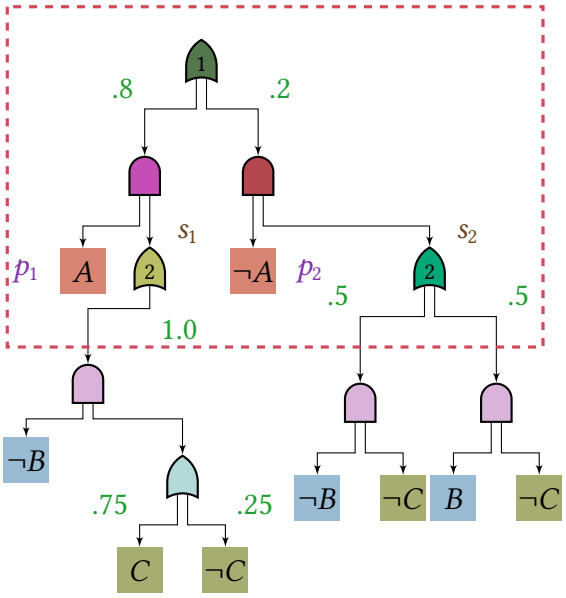


Figure 4.1: *A PSDD encoding the logical constraint $\phi(A, B, C) = (A \rightarrow \neg B) \wedge (C \rightarrow A)$, following the distribution set by the probability table on the top left corner and whose structure is defined by the vtree pictured on the bottom left corner.*

whose primes are conjunctions of literals in a similar manner to XPCs, proposing a Bayesian information criterion to searching the sample space, yet no practical algorithm was fully formulated; R. Geh, D. Mauá, and Antonucci (2020) expanded on Mattei, Soares, *et al.*'s work by formalizing an algorithm and introducing a BDD to guide sampling, however the generated circuits suffered from an exponential blow-up in size.

In this section, we propose a solution inspired by R. Geh, D. Mauá, and Antonucci (2020) and Mattei, Soares, *et al.* (2019), yet without the previously mentioned problems that come with them. In summary, we propose a sampling procedure to efficiently generating PSDDs whose primes are always conjunctions of literals; to overcome the exponential blow-up, these PSDDs only partially encode their prior logical restrictions. To diversify sampling, local transformations similar in spirit to `INCR` algorithms are used. Of worth, we found that not only is this process incredibly fast even under intricate logical formulae, but by combining samples into an ensemble we achieve competitive results against the state-of-the-art.

Before we address our contributions, we should first fix some notation on the issue of propositional logic. We treat propositional variables as 0/1-valued random variables and use them interchangeably. Given a Boolean formula, 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, i.e. 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 overload the scope function once again for logical formulae: $\mathrm{Sc}(f)$ denotes the set of variables that appear in f. We say that the restriction of f to an assignment $\mathbf{X} = \mathbf{x}$, where $\mathbf{X} \subseteq \mathrm{Sc}(f)$, is a Boolean function resulting from setting all literal nodes to the corresponding values of $\mathbf{x}$; we denote this operation as $f|_{\mathbf{x}}$. As an example, consider $f(a, b, c, d) = (a \vee \neg b) \wedge (\neg c \vee \neg d)$ and say we wish to restrict f to $\mathbf{x} = \{a = 0, c = 1\}$, then $f|_{\mathbf{x}} = b \wedge \neg d$. At times, we might wish to restrict a function to

a conjunction of literals consistent with an assignment. When this happens, we overload the function to return the restriction to the equivalent assignment. For instance, given the same function as the previous example, and calling $p(a, c) = \neg a \wedge c$, we use $f|_p$ to mean the restriction of f to p's equivalent assignment $(a = 0, c = 1)$, that is, the only assignment where p would return true.

Because we are only interested in smooth, structure decomposable and deterministic PCs whose support is defined by a logical formula, we shall adopt the usual notation of PSDDs, which we present next.

Definition 4.1.1 (Partition). *Let $\phi(\mathbf{x}, \mathbf{y})$ be a Boolean function over disjoint sets of variables $\mathbf{X}$ and $\mathbf{Y}$, and $\mathcal{D} = \{(p_i, s_i)\}_{i=1}^k$ be a set of tuples where p_i (the prime) and s_i (the sub) are formulae over $\mathbf{X}$ and $\mathbf{Y}$ respectively, satisfying $p_i \wedge p_j \equiv \bot$ for each $i \neq j$ and $\bigvee_{i=1}^k p_i \equiv \top$. We say that $\mathcal{D}$ is an $(\mathbf{X}, \mathbf{Y})$-partition of ϕ if and only if $\phi \equiv \bigvee_{i=1}^k (p_i \wedge s_i)$.*

An (exact) partition[1] is no more than a smooth, structure decomposable and deterministic circuit rooted at a sum (or disjunction) node whose children are products (or conjunctions); the primes of these products must necessarily be mutual exclusive (formally, $p_i \wedge p_j \equiv \bot$) and exhaustive (formally, $\bigvee_{i=1}^k p_i \equiv \top$). Semantically, a partition states that a logical formula decomposes into k exact conjunctions of pairs of prime and sub. The box in Figure 4.1 shows a partition whose primes are $p_1 = A$ and $p_2 = \neg A$, and subs are $s_1 = \neg B$ (represented as a PC rooted at) and $s_2 = \neg C$ (represented as the PC rooted at). Recall that the conjunction between a prime and sub is called an *element*, here shown as and .

4.2 SamplePSDD

We now describe how to efficiently learn PSDDs by sampling and averaging. The procedure takes inspiration from `DIV` algorithms in the sense that we construct a PSDD structure top-down by recursively decomposing a logical formula (instead of data). At the same time, we employ local transformations similar to `INCR` approaches on a partially constructed circuit to diversify samples. All this procedure is done randomly in a similar fashion to XPCs, where we restrict primes to be random conjunctions of literals and sample variables according to a previously learned or randomly sampled vtree. To better understand how this is done, we must first consider a naïve approach.

Let ϕ be a logical formula acting as our knowledge base, and assume that a vtree $\mathcal{V}$ is given beforehand. To obtain a PSDD whose support is ϕ, we decompose it down to a disjunction of prime and sub conjunctions. This is a non-trivial problem, as primes must not only be mutual exclusive (to ensure determinism) but exhaustive (to make sure the circuit is coherent with ϕ in all possible assignments). If we assume primes to be conjunctions of literals, then to adhere to Definition 4.1.1 there will be, in the worst case, an exponential number of elements $2^{|\mathrm{Sc}(v^{\leftarrow})|}$, where v is the vtree node that corresponds to

[1] The naming *partition* is unfortunate. The nomenclature in probabilistic circuits is full of many other partitions, either using the term to conjure meaning from set theory when dealing with data splits (see Section 3.1), partition nodes in region graphs (see Section 3.3) or in PSDD literature in this section. Here (and only here), partitions will mean stricly the latter.

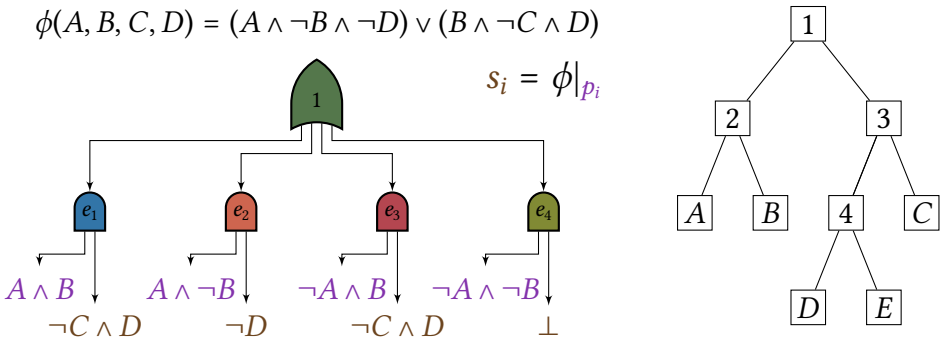


Figure 4.2: *A(n exact) partition of ϕ where we assume that primes are conjunctions of literals. Primes must be exhaustive, mutually exclusive, and have to follow the vtree's scope, here* $\mathrm{Sc}(1^{\leftarrow}) = \{A, B\}$. *The subs are then the restriction of ϕ under the assignment induced by the primes.*

the partition. Subs, however, are easy to retrieve as they correspond to the restriction of ϕ under the assignment induced by the prime. Figure 4.2 shows a partition whose primes are conjunctions over A and B. This problem is the same as the one faced by R. Geh, D. Mauá, and Antonucci (2020): under the assumption of conjunctions of literals as primes, ϕ can only be faithfully represented as a PSDD if the circuit is exponential in the number of variables.

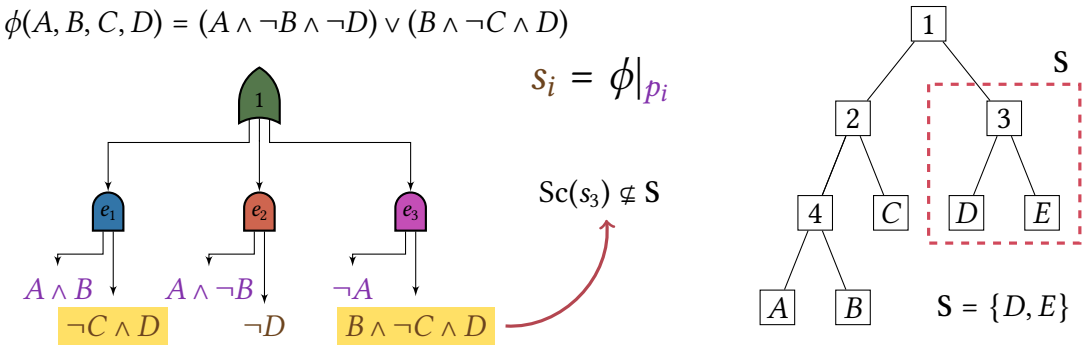
To overcome this exponential blow-up, we might restrict the number of primes at each partition. Unfortunately, if we upper bound this number by a constant, say k, and randomly sample primes from the space of all possible conjunctions of literals, then we face yet another problem: the scope of subs might contain variables not in their corresponding vtree node. Take the top circuit in Figure 4.3 as an example. Note that the scope for primes is defined by $\mathrm{Sc}(1^{\leftarrow}) = \{A, B, C\}$, with $\mathrm{Sc}(1^{\rightarrow}) = \{D, E\}$ for subs; so sampled primes $p_1 = A \wedge B$, $p_2 = A \wedge \neg B$ and $p_3 = \neg A$ must come from $\mathrm{Sc}(1^{\leftarrow})$. However, because $s_1 = \phi|_{p_1} = \neg C \wedge D$ and $s_3 = \phi|_{p_3} = B \wedge \neg C \wedge D$, meaning that $\mathrm{Sc}(s_1) \not\subseteq \mathrm{Sc}(1^{\rightarrow})$ and $\mathrm{Sc}(s_3) \not\subseteq \mathrm{Sc}(1^{\rightarrow})$, subs violate the factorization imposed by the vtree $\mathcal{V}$, making the circuit non-structure decomposable (albeit decomposable). Here, we point out that the scope of the formula needs to be a subset of the scope of its corresponding vtree for the PC to be structure decomposable, and not necessarily the set itself, as variables that do not appear in the formula yet are part of the vtree's scope play a *probabilistic* role in the PSDD.

For these circuits to both preserve structure decomposability *and* have tractable representation, we resort to a weaker definition of a partition that relaxes the logical constraints.

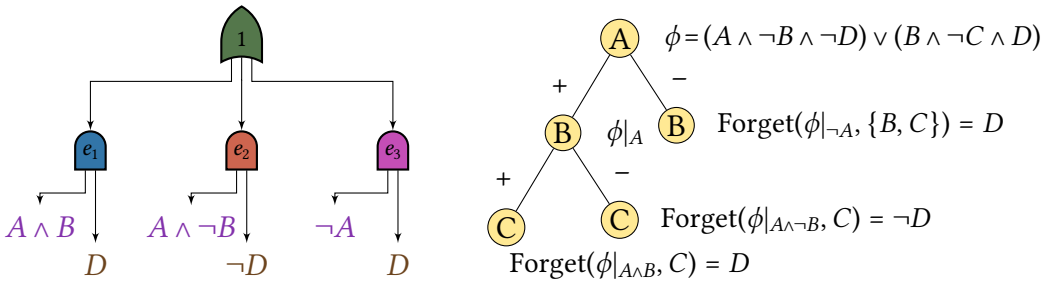
Definition 4.2.1 (Partial partition). *Let $\phi(\mathbf{x}, \mathbf{y})$ be a Boolean function over disjoint sets of variables $\mathbf{X}$ and $\mathbf{Y}$, and $\mathcal{D} = \{(p_i, s_i)\}_{i=1}^{k}$ be a set of tuples where p_i (the prime) and and s_i (the sub) are formulae over $\mathbf{X}$ and $\mathbf{Y}$ respectively, satisfying $p_i \wedge p_j \equiv \bot$ for each $i \neq j$ and $\bigvee_{i=1}^{k} p_i \equiv \top$. We say that $\mathcal{D}$ is a partial partition of ϕ if*

$$\left\langle \bigvee_{i=1}^{k} (p_i \wedge s_i) \right\rangle \geq \phi, \tag{4.1}$$

where the inequality is taken coordinate-wise.



(a) *Partially constructed PSDD with vtree infracting subs*



(b) *Partial partition where subs are relaxed*

Figure 4.3: *An example of an invalid partition* **(a)** *due to subs disrespecting the vtree's right branch, here shown as the box with scope* S. *To fix this infraction, variables who do not belong to* S *are* forgotten, *as* **(b)** *shows.*

Definition 4.2.1 essentially states that the disjunction over elements has to only encode a relaxation of the original formula ϕ. This is somewhat similar to what Gatterbauer and Suciu (2014) propose in probabilistic databases, where they relax a formula in such a way that the approximate probabilities provide an upper bound independent of the actual probabilities.

Looking back to the issue of bounding the number of elements per partition, a solution to the problem of tractability and structure decomposability comes by employing *partial* partitions instead of *exact* partitions. Let S be a sum node, and call s_i one of its sub and v the vtree node for S; denote by $\mathbf{F} = \mathrm{Sc}(s_i) \setminus \mathrm{Sc}(v^{\rightarrow})$, that is, the variables in s_i which should *not* have been in the sub. We already know that S cannot be turned into an exact partition unless it has an exponential number of elements, and so we look to *partial* partitions. The *forget* operation takes a formula ψ and marginalizes variable X: $\mathrm{Forget}(\psi, X) = \psi|_X \vee \psi|_{\neg X}$; by construction $\mathrm{Forget}(\psi, X) \geq \psi$. By forgetting all variables in $\mathbf{F}$, we secure structure decomposability and produce a relaxation of the original formula. To do this efficiently, we make use of a BDD for representing formulae, as reduced BDDs are canonical and permit polynomial time restricting and forgetting.

In more practical terms, the overall process of sampling a PSDD starts with a logical formula ϕ, a vtree $\mathcal{V}$ and scope $\mathbf{X}$. We recursively construct partial partitions by first sampling a fixed number of k primes and evaluating subs with restrictions and Forgets. Elements whose subs are $\bot$ are removed, as their probability is always zero. For each

Algorithm 16 SamplePartialPartition

Input BDD ϕ, vtree node v, number of primes k
Output A set of pairs of primes and subs

Define **E** as an empty collection of pairs
Sample an ordering $X_1, \ldots, X_m$ of $\text{Sc}(v^{\leftarrow}) \cap \text{Sc}(\phi)$
Let **Q** be a queue initially containing $(\phi, 1, \{\})$
$j \leftarrow 1$ ▷ Counter of sampled elements
while $|\mathbf{E} < k|$ **do**
    Pop top item (ψ, i, p) from **Q**
    if $j \geq k$ **or** $i > m$ **or** $\psi \equiv \top$ **then**
        Add $(p, \text{Forget}(\phi|_p, \text{Sc}(v^{\leftarrow})))$ to **E**
        continue
    $\alpha \leftarrow \psi|_{X_i}, \beta \leftarrow \psi|_{\neg X_i}$
    if $\alpha \equiv \beta$ **then** enqueue $(\psi, i+1, p)$ into **Q**
    else
        if $\alpha \not\equiv \bot$ **then** enqueue $(\alpha, i+1, p \wedge X_i)$ into **Q**
        if $\beta \not\equiv \bot$ **then** enqueue $(\beta, i+1, p \wedge \neg X_i)$ into **Q**
        $j \leftarrow j + 1$
return E

prime and sub, we recursively call the same procedure on their formulae. Just like in most DIV class algorithms, if $|\mathbf{X}| = 1$, then we either return a literal node consistent with ϕ, or a Bernoulli distribution input node over **X**'s only variable. Another special case arises when $\phi \equiv \top$, in which case any smooth, structure decomposable and deterministic PC will do. This PC can either be learned purely from data or generated from a template. Alternatively, we might even choose to continue sampling partitions as before, except in this case all partial partitions are also exact partitions. Algorithm 17 shows the entire recursive procedure of SamplePSDD.

The sampling process of generating primes and their subs is shown in Algorithm 16 and goes as follows. To produce primes (and subs), we must look at the space of all possible variable assignments coming from $\text{Sc}(v^{\leftarrow})$, where v is the relevant vtree node. If we fix a variable ordering to $\text{Sc}(v^{\leftarrow})$, say an m-uple $\mathbf{O} = \{X_1, \ldots, X_m\}$, then we might structure this space as a binary decision tree whose nodes are labeled as a variable and every edge denotes positive or negative literals over that variable. The path coming from a node to a leaf in this decision tree represents all chosen literals in a prime. An example of such a tree is shown as the right tree in Figure 4.3b. We efficiently generate k primes by starting from the root node labeled as variable X_1 and repeatedly expanding a leaf labeled X_i with two children X_{i+1} until the number of leaves is between $k-1$ and k (expanding further would mean violating the bound on the number of primes). Every time we expand a leaf, we must generate the restrictions $\psi|_{X_i}$ and $\psi|_{\neg X_i}$, where ψ is the formula up to that path, and associate them with the left and right children respectively. If $\psi|_{X_i} \equiv \psi|_{\neg X_i}$, or in other words the assignment of X_i does not change ψ's semantics, then we relabel the node as X_{i+1} and re-expand it with children X_{i+2}, effectively ignoring X_{i+1}. When this process terminates, we have at most k conjunctive primes represented by all the paths coming from the root down to the leaves, each of these with an associated formula equivalent to

Algorithm 17 SamplePSDD

Input BDD ϕ, vtree node v, number of primes k
Output A sampled PSDD structure

if $|\text{Sc}(v)| = 1$ **then**
  if ϕ is a literal **then return** ϕ as a literal node
  else return a Bernoulli distribution input node over variable $\text{Sc}(v)$
else if $\phi \equiv \top$ **then**
  return any smooth, structure decomposable and deterministic PC over $\text{Sc}(v)$
E $\leftarrow$ SamplePartialPartition$(\phi, \text{Sc}(v^{\leftarrow}), k)$
Create a sum node S
Randomly compress elements in **E** with equal subs
Randomly merge elements in **E** with equal subs
for each element $(p, s) \in$ **E do**
  $l \leftarrow$ SampleExactPSDD$(p, v^{\leftarrow}, k)$
  $r \leftarrow$ SamplePSDD$(s, v^{\rightarrow}, k)$
  Add a product node with children l and r as a child of S
return S

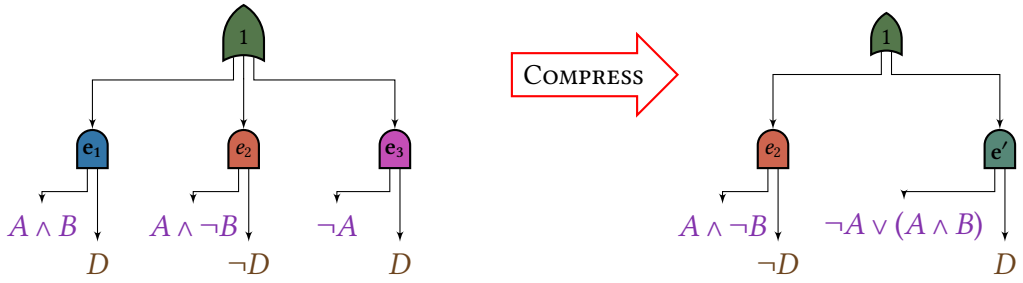
restricting ϕ to all assignments $\phi|_{\mathbf{x}}$. Now, to obtain valid subs as previously mentioned, we apply the Forget operation to each sub over the scope of $\text{Sc}(v^{\leftarrow})$, removing any variables from the wrong side of the vtree.

Once primes and subs are generated, SamplePSDD randomly applies local transformations to add variety to sampled circuits and reduce their size. Here we introduce two *shrinking* local transformations, directly opposed to INCR's *growing* local transformations. We borrow the concept of *compression*, used to minimize a logic circuit down to a canonical representation (Darwiche, 2011), and use it to join multiple elements into a single one during learning. Let $e_1, \ldots, e_q$ be elements whose subs $s_1, \ldots, s_q$ are all equivalent to s, or more formally $s \equiv s_i \equiv s_j$, $i \neq j$; in this case, the disjunction over these elements factorizes over s

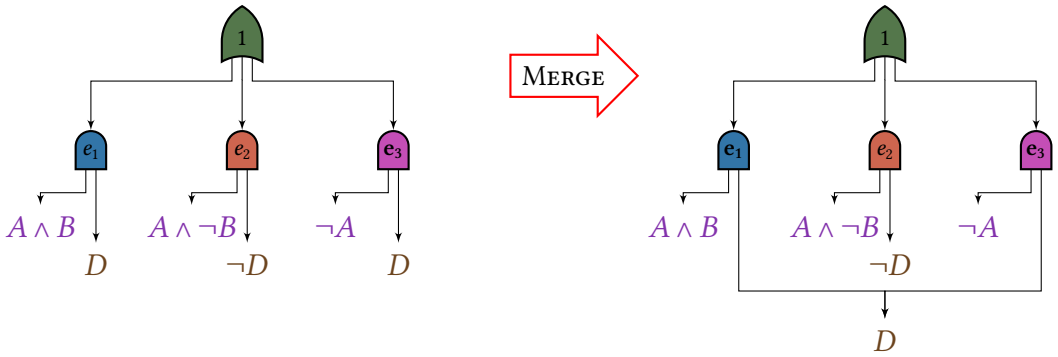
$$\bigvee_{i=1}^{q} (p_i \wedge s_i) = \bigvee_{i=1}^{q} (p_i \wedge s) = s \wedge \left(\bigvee_{i=1}^{q} p_i \right). \tag{4.2}$$

Figure 4.4a shows a compression of elements e_1 and e_3 whose subs are both D. The resulting compressed element e' is equivalent to the disjunction of the primes with no change to the sub. Compression is the exact inverse of Split, seen in Section 3.2 (cf. Figure 3.7). Apart from compression, we propose *merging* two equivalent subs into the same circuit as shown in Figure 4.4b. Merging is a common (previously nameless) operation in PC literature and is the inverse of Clone (cf. Figure 3.7). In both cases, shrinking local transformations preserve smoothness, structure decomposability, determinism and the circuit's formula, although they change the PSDD's underlying distribution by reducing the number of parameters.

To ensure that elements are mutual exclusive (and by consequence the partition is deterministic), we need to disallow relaxations in recursive calls to primes. For instance, if we had not imposed this restriction, a possible relaxation of e_1's prime $A \wedge B$ into, say B,



(a) *Compressing elements e_1 and e_3 into e'*



(b) *Merging subs s_1 and s_3 into a single sub*

Figure 4.4: *Examples of compression* **(a)** *and merging* **(b)** *as local transformations for reducing the size of PSDDs. Both act on elements whose subs are logically equivalent.*

in Figure 4.3b might contradict prime mutual exclusivity, as B conflicts with e_3's prime $\neg B$ (because $B \wedge \neg A \not\equiv \bot$). This is trivially solved by making sure that every partition in subcircuits rooted at primes is exact (here denoted by the function SampleExactPartition). If we bound the number of primes to a constant k, these exact subcircuits will never suffer from an exponential blow-up, as all subsequent (exact) partitions contain primes with at most $\lceil \log_2(k) \rceil$ variables and thus at most $2^{\lceil \log_2(k) \rceil}$ elements are constructed at each call.

4.3 Experiments

Just like in Strudel and LearnPSDD, to boost the performance of our approach we resort to mixtures of PSDDs. We separately sample t models in parallel with SamplePSDD, learn their parameters through closed-form smoothed MLE (Kisa *et al.*, 2014) and then learn only mixture weights in five different approaches: (1) likelihood weighting (LLW), where each component's weight is proportional to its train likelihood; (2) uniform weights, (3) Expectation-Maximization, (4) stacking (Smyth and Wolpert, 1998), and (5) Bayesian Model Combination (BMC, Monteith *et al.*, 2011).

We compare our results against Strudel, mixtures of 10 Strudels, LearnPSDD and LearnSPN. We used existing implementations coming from the Juice probabilistic circuits library (Dang, Khosravi, *et al.*, 2021) for the first two, while LearnPSDD and SamplePSDD

were implemented into the library. For LearnSPN, we used the PySPN library[2] whose implementation uses k-means for learning sums and G-test for products. We look at four different datasets that contain some kind of structure to them, modeling them as logical formulae. We observe the impact of this prior knowledge by learning PCs both under low data regimes and lots of data. For Strudel, as proposed in Dang, Vergari, *et al.* (2020), we used an initial PC compiled from a CLT learned purely from data (see Section 3.2.2). Initial circuits for LearnPSDD were compiled into canonical logic circuits from either a CNF or DNF when the logical restrictions permitted a tractable representation in such forms; when this was not the case, a BDD was compiled into a circuit instead. For the resulting initial circuit to contain variables not present in the formula, a product node whose children were the compiled PC and a fully factorized circuit over absent variables was created and set as root.

All experiments were run on an Intel i7-8700K 3.70 GHz machine with 12 cores and 64GB. We limited LearnPSDD and Strudel both to 100 iterations, although runs with 1000 iterations can be found in the supplementary material of Annex B.1. For all experiments with SamplePSDD, when $\phi \equiv \top$ (line 5 of Algorithm 17) we produced a fully factorized circuit. In the remaining part of this section, we first address the performance of our proposed approach compared to the state-of-the-art and then provide an empirical analysis on the impact of vtrees and parameter choice for ensembles of SamplePSDD

4.3.1 LED Display

A seven-segment LED display consists of LED light segments which are separately turned on or off in order to represent a digit. Figure 4.5 shows some digits represented by a seven-segment display. Each digit is associated with a local constraint on the values of each segment. We adapt the approach by Mattei, Antonucci, *et al.* (2020b) and generate a `led` dataset of faulty observations of the segments as follows. Each segment is represented by a pair of variables (X_i, Y_i), where Y_i is the observable state of segment i (i.e. whether the segment is on or off) and X_i is the latent state of i. We randomly sampled a PSDD over variables X_i and Y_i whose support are the valid configurations of segments X_i representing the digits, and use that model to generate a dataset of 5,000.00 training instances and 10,000.00 test instances.

A more complex alternative configuration for the LED setting, `led-pixels`, is the interpretation of digits as images and segments as pixel regions. The segment constraints remain unchanged, but now pixel regions act as the latent variables. Figure 4.5 (bottom) shows ten samples for each the ten digits; each instance from the dataset is a 10×15 black-and-white image. In this pixelized version, we do not explicitly describe, in the form of logical constraints, a one-to-one mapping of pixel regions as segments; instead, we visually identify key points where pixels most often activated given a segment's value. Let $\mathbf{R}_s$ be pixel variables which are most oftn set to 1 when a segment s is on. We build a constraint for each segment: $\psi(s) = s \longrightarrow \bigvee_{r \in \mathbf{R}_s} r$. We futher recognize which pixels are always off given a valid digit segment configuraton: $\phi(\mathbf{s}) = \left(\bigwedge_{p\,:\,p=0|\mathbf{s}} \neg p\right) \wedge \left(\bigwedge_{s \in \mathbf{s}} s\right)$. The full logic formula encoding all constraints is the conjunction of every possible ϕ and ψ.

[2] https://gitlab.com/pgm-usp/pyspn

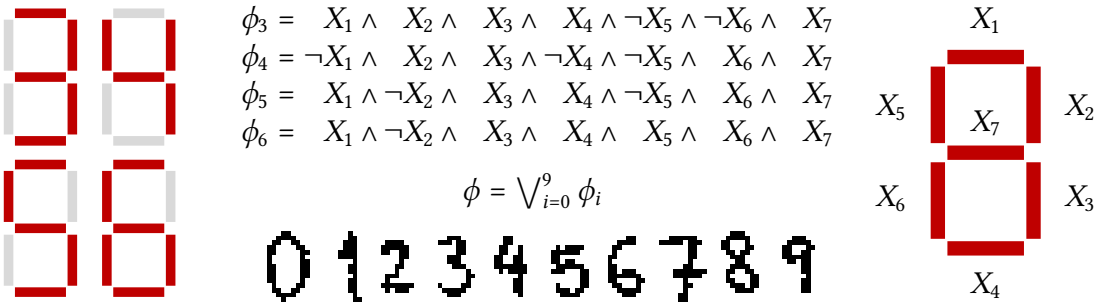


Figure 4.5: *Seven-segment LED digits for 3, 4, 5 and 6 (left), the logical constraints for each of these digit ϕ_i (top middle) and the resulting formula derived from listing all valid configurations ϕ (middle), each latent variable X_i corresponding to a segment's supposed state, and samples of pixel variants* `led-pixels` *for each digit (bottom middle).*

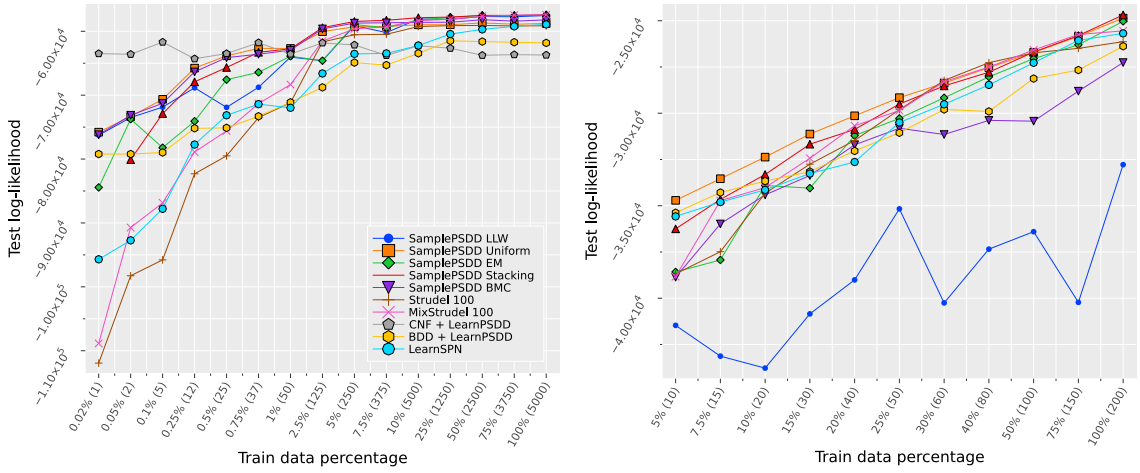
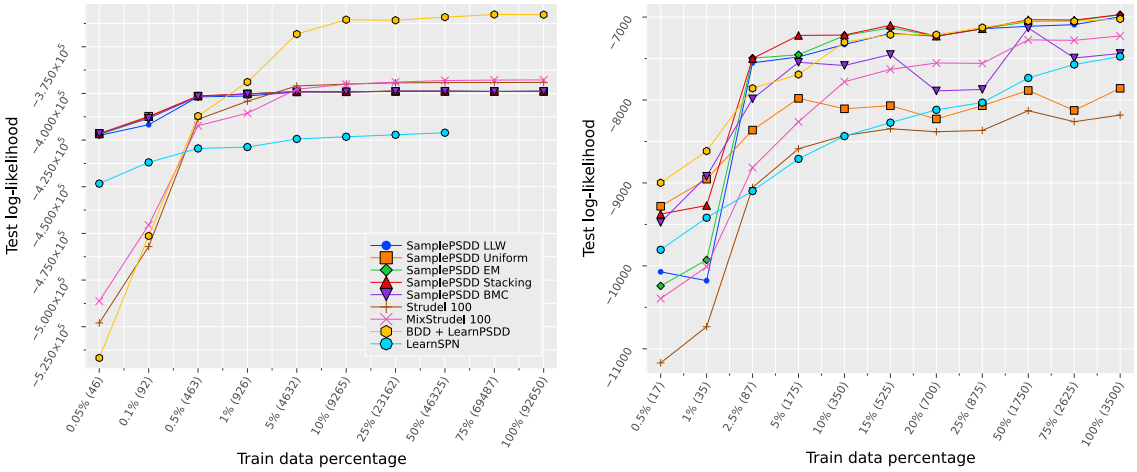


Figure 4.6: *Log-likelihoods for the unpixelized* `led` **(a)** *and pixelized* `led-pixels` **(b)** *datasets.*

Figure 4.6 shows how our approach fairs against competitors using different percentages of available training data on the unpixelized and pixelized versions of the dataset. The labels on the x-axis indicate percentage and number of training instances. We sampled $t = 100$ circuits for both settings, with $k = 32$ and $k = 8$ for `led` and `led-pixels` respectively. Note how the use of logical constraints greatly improves performance even under extremely scarce data (with 1 or 2 datapoints). Most of the SamplePSDD approaches obtain the best performance with the full dataset, and ranks among the best when data size is small.

4.3.2 Cardinality Constraints

The DOTA dataset contains the result of 102,944.00 online matches of the Dota 2 videogame, made available at the UCI Repository. In this game, each team is composed of 5 players players, with each one controlling a single character out of a pool of 113. Each character can be controlled by at most one single player in a match. We represent the domain by 2 groups of 113 Boolean variables $C_1^{(i)}$ and $C_2^{(i)}$, denoting whether the i-th



(a) **(b)**

Figure 4.7: *Log-likelihoods for the* `dota` **(a)** *and* 10*-choose-*5 `sushi` **(b)** *datasets.*

character was selected by the first or second team, respectively. We then encode 113-choose-5 cardinality constraints on the selection of each team (i.e. $\sum_{i=1}^{1} 13C_j^{(i)} = 5$ for $j \in \{1, 2\}$). Unfortunately, adding the constraint that no character can be selected by both teams $\neg(C_1^{(i)} \wedge C_2^{(i)})$ made the BDD representation of the formula intractable, and so was ignored. Since the CNF representation of cardinality constraints is intractable, we used a PSDD compiled from a BDD to generate an initial circuit for LearnPSDD (as BDDs can efficiently encode such restrictions (Eén and Sörensson, 2006)). We set the number of components for SamplePSDD to $t = 30$ and bound the number of sampled elements to $k = 3$.

The plot in Figure 4.7a shows the test log-likelihood of the tested approaches. Despite accurately encoding logical constraints, LearnPSDD initially obtains worse performance when compared to SamplePSDD, but quickly picks up, outperforming other models by a large margin. SamplePSDD ranks first for small data regimes, and is comparable to Strudel (and mixtures of) for large training datasets. LearnSPN encountered problems scaling to more than 50,000.00 instances due to intensive memory usage in both clustering and pairwise independence testing.

We also compared methods on the `sushi` dataset (Kamishima, 2003), using the setting proposed in Shen *et al.* (2017). The data contains a collection of 5,000.00 rankings of 10.00 different types of sushi. For each ranking, we create 10.00 Boolean variables denoting whether an item was ranked among the top 5, and ignore their relative position. The logical constraints represent the selection of 5.00 out of 10.00 items. We split the dataset into 3,500.00 instances for training and 1,500.00 for the test set and evaluated the log-likelihood on both tasks. The plot in Figure 4.7b shows the log-likelihood for this data. For this dataset, we set $t = 30$ and $k = 3$ for ensembles of SamplePSDD. LearPSDD obtains superior performance accross some of the low sample sizes, but our approaches were able to quickly pick up and tie with LearnPSDD when using the LLW, stacking and EM strategies.

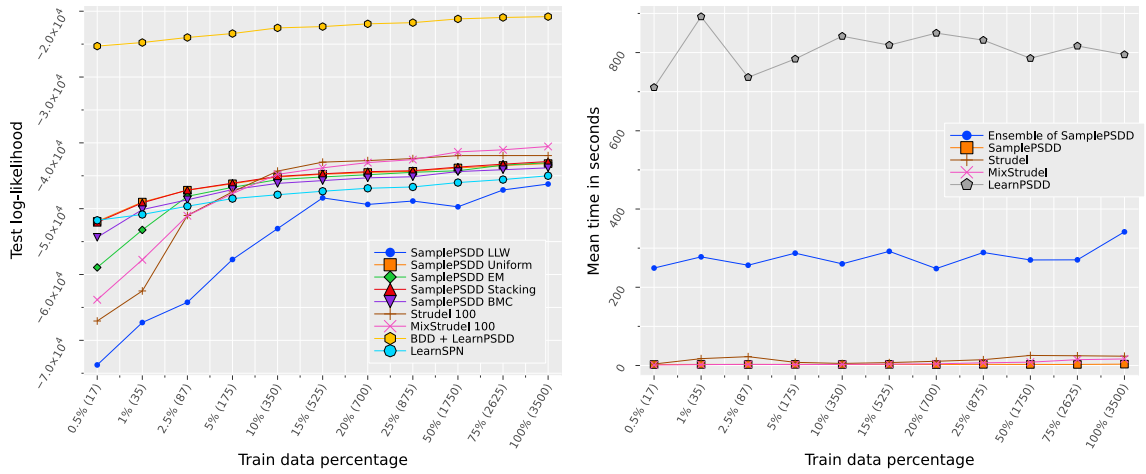


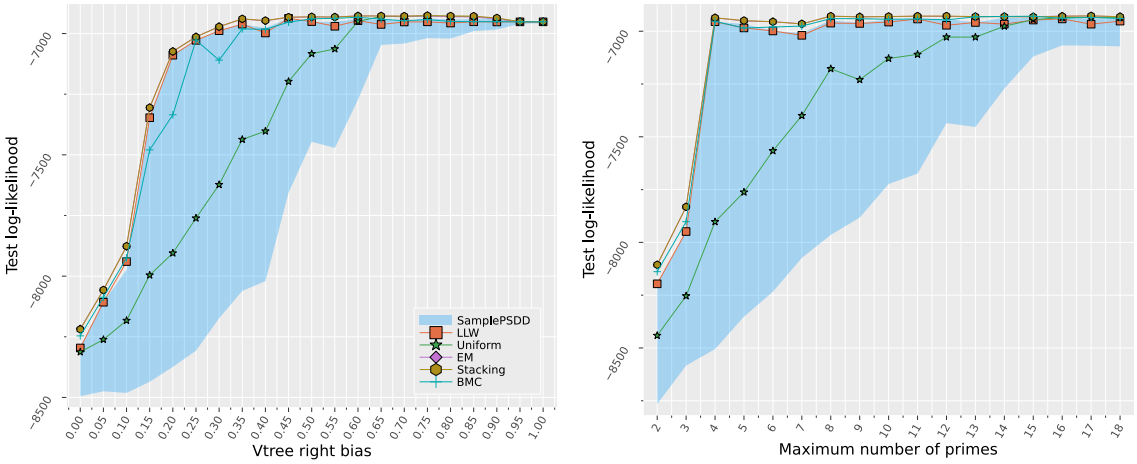
Figure 4.8: *Log-likelihood for the* `sushi` *ranking* **(a)** *dataset and curves for mean average time (in seconds) of learning a single* LearnPSDD *circuit, one* Strudel *circuit (CLT initialized), a mixture of 10 shared-structure* Strudel *components, a single* SamplePSDD *PC and an ensemble of 100* SamplePSDD *circuits* **(b)**.

4.3.3 Preference Learning

We also evaluated performance on the original task of ranking items on the `sushi` dataset. We adopt the same encoding and methodology as (A. Choi, Broeck, *et al.*, 2015), where each ranking is encoded by a set of Boolean variables X_{ij} indicating whether the i-th item was ranked in the j-th position. We use same parameters for ensembles of SamplePSDD as the previous dataset and set $t = 30$ and $k = 3$. The test log-likelihood performance of each different method is shown in Figure 4.8a. The results are qualitatively similar to the previous experiments: SamplePSDD performed better than pure data approaches under low data yet achieved competitive results when given the full data. In this case, however, we found that LearnPSDD ranked first by a large margin compared to others.

4.3.4 Scalability, Complexity and Learning Time

The major advantage of SamplePSDD compared to other PSDD learning algorithms comes from its ability to learn from both data and a logical formula ϕ even when ϕ defines an intricate Boolean formula over many variables. Interestingly, how capable SamplePSDD is of learning from ϕ comes not only from the algorithm itself, but how ϕ is represented. In fact, any data structure with tractable restriction and reduction (to a canonical form) can be used in place of the BDD shown in Algorithm 17. We do not require forgetting to be tractable due to an implementational "trick" in SamplePSDD that allows a fast implementation to ignore variables not in the scope of the vtree. More details can be found in the supplemental material of Annex B.1. Consequently, how scalable our proposed algorithm is depends on the representational power of the tool used for manipulating logical formulae. This is shown more concretely when we learn from the constraints set by the `dota` dataset: had we tried to manipulate formulae with a CNF, the intractability of



(a) **(b)**

Figure 4.9: *Impact of the structure of vtree (left) and number of bounded primes (right) on the test log-likelihood for the* 10*-choose-*5 `sushi` *dataset.*

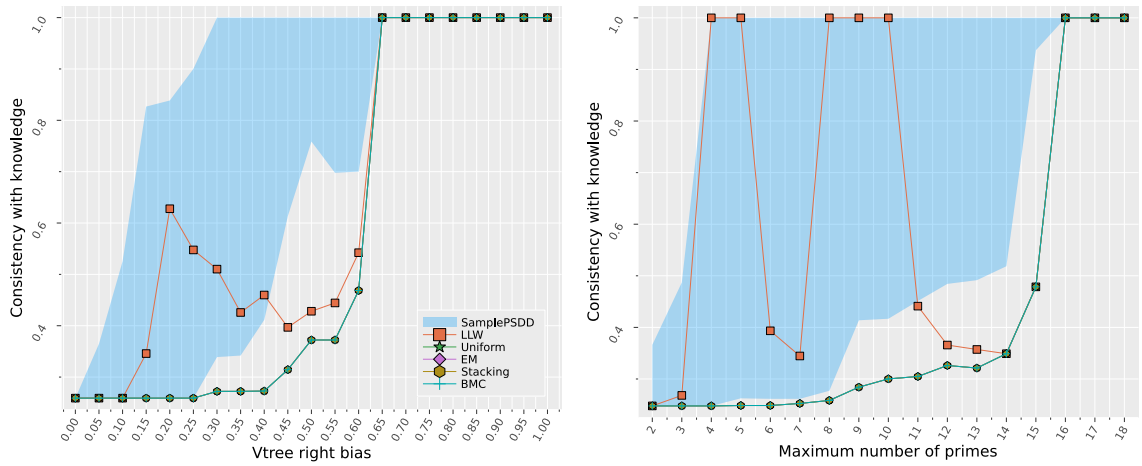
cardinality constraints would unfortunately impede any progress.

We now provide an analysis on the complexity of SamplePSDD. We start with the SamplePartialPartition subroutine. If we choose a BDD for manipulating formulae, then restrictions are done in $\mathcal{O}(c \log c)$, where c is the size of the BDD's graph. At every iteration of the main loop in line 5 of Algorithm 16, a leaf of the binary decision tree is split into two, increasing the number of leaves (and therefore primes) by one. This is repeated until k leaves of the decision tree have been expanded, bringing SamplePartialPartition's complexity to $\mathcal{O}(k \cdot c \log c)$. Every call of SamplePSDD (apart from base cases) produces a new partition and randomly compress and merge elements. Compression requires applying a disjunction over two primes, both of which are conjunctions of literals, represented in BDDs as a graph of size linear to the number of terms. Because primes can have at most $\lceil \log_2(k) \rceil$, the disjunction of two such conjunctions is done in $\mathcal{O}(\log_2^2 k)$. Merging (or compressing) n elements essentially subtracts the number of recursive calls for each merge (or compression) by $n - 1$. Therefore, every SamplePSDD recursive call is $\mathcal{O}\left(k \cdot c \log c + \log_2^2 k\right)$.

We empirically evaluate the time it takes to learn a single circuit from each of LearnPSDD, Strudel and SamplePSDD. We also measure running times for learning 10 structure-sharing circuits as components of a mixture of Strudels and 100 (structurally) distinct PCs sampled from SamplePSDD, each with parameters learned from closed-form MLE. Figure 4.8b shows the time it takes to run each of these settings on the `sushi` ranking dataset. On average, Strudel took approximately 15.00 seconds, LearnPSDD 13.00 minutes and 25.00 seconds, and SamplePSDD about 2.76 seconds for learning a single PSDD.

4.3.5 Performance and Sampling Bias

The approximation quality of SamplePSDD is highly dependent on both the vtree and maximum number of primes. In this section, we compare the impact of both in terms of



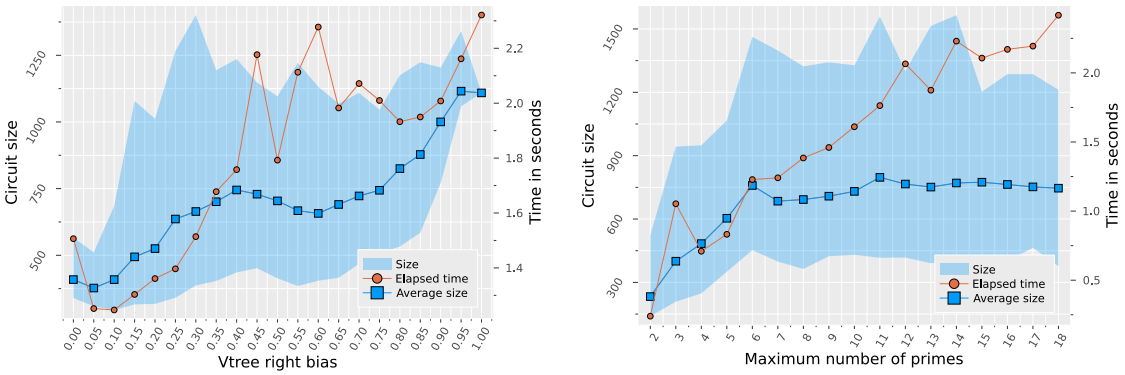
(a) (b)

Figure 4.10: *Impact of the structure of vtree (left) and number of bounded primes (right) on the consistency of sampled PSDDs with the original logical constraints for the* 10*-choose-*5 `sushi` *dataset.*

performance and circuit complexity. We assess performance by the log-likelihoods in the test set, as well as consistency with the original logical constraints. The latter is measured by randomly sampling 5,000.00 (possibly invalid) instances and evaluating whether the circuit correctly decides their value. A set of the top 100.00 sampled PSDDs (in terms of log-likelihood in the train set) are selected out of 500.00 circuits learned on the 10-choose-5 `sushi` dataset to compose the ensemble. Circuit complexity is estimated in terms of both time taken to sample all 500 circuits and graph size (i.e. number of nodes) of each individually generated PSDD.

It is quite clear that the structure of the vtree is tightly linked to the structure of a PSDD, especially given the graphical constraints imposed by SamplePSDD and the fact that subs need to obey a vtree's scope (and thus its structure). For instance, (near) right vtrees keep the number of primes fixed and require no approximation, while (near) left vtrees discard a large number of primes. In order to evaluate the effect of the type of vtree on the quality of sampled structures, we compared the performance of SamplePSDD as we vary the bias towards generation of right-leaning vtrees. Given a parameter p, we grow a vtree in a top-down manner where at each node we independently assign each variable to the right child with probability p. Small values of p produce left-leaning vtrees, while vtrees are more likely to lean to the right when $p > 0.5$. Left leaning vtrees produce especially small networks compared to other vtrees, as more variables are left unmentioned because of relaxations coming from the need for a bounded number of primes. To produce decently sized circuits, we increase the number of sampled primes k when p is low and decrease k when p is high.

Figure 4.9 shows the log-likelihood, Figure 4.10 shows consistency and Figure 4.11 shows circuit complexity when varying the bound on the number of primes (left) and the type of vtrees used for guiding the PSDD construction (right). The blue shaded area represents the interval of values (worse to best ranked) for individual circuits. To verify consistency, we evaluate the PSDDs in terms of satisfiability of a given example. An ensemble returned a configuration as satisfiable if any of its models attributed some nonzero



(a) **(b)**

Figure 4.11: *Impact of the structure of vtree (left) and number of bounded primes (right) on circuit size (in number of nodes) and learning time (in seconds) for the* 10*-choose-*5 `sushi` *dataset.*

probability to it; and unsatisfiable if all models gave zero probability. This evaluation gives a lower bound to consistency, which means all models eventually unanimously agreed on satisfiability when vtree right bias ≥ 0.65. Alternatively, since SamplePSDD is a relaxation of the original formula, an upper bound on consistency could be achieved by evaluating whether any model within the ensemble gave a zero probability to the example; this upper bound curve on consistency would be equivalent to the top side of the shaded area. Interestingly, we note that the likelihood weighting strategy (LLW) dominates over others on consistency. This is because LLW often degenerates to a few models, giving zero probability to lower scoring PSDDs, which means only a small subset of circuits decide on satisfiability, and thus a more relaxed model is less likely to disagree with the consensus. On the other hand, this does not translate to better data fitness on the general case, as we clearly see in Figures 4.6 to 4.8.

5

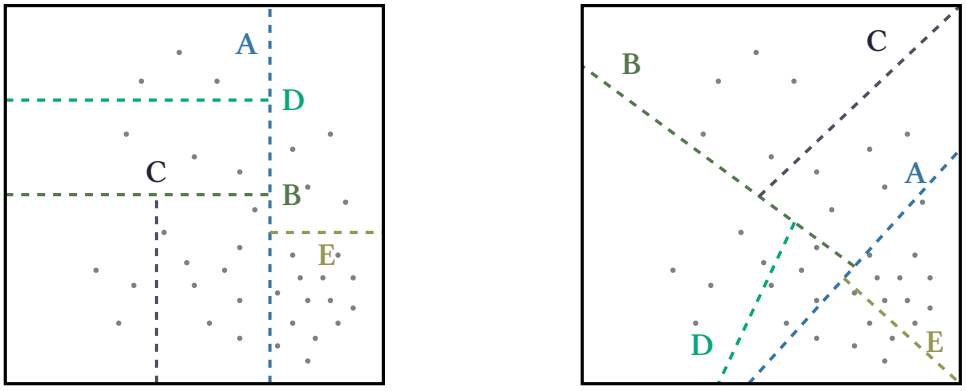
A Data Perspective to Scalable Learning

We now turn our attention to scalably learning a PC purely from data. In this chapter, we look at PCs solely from the perspective of data fitness; we exploit the connection between PCs and generative random forests (Correia *et al.*, 2020; Ho, 1995) and revisit a well-known technique based on random projections for constructing random trees (Freund *et al.*, 2008; Dasgupta and Freund, 2008), presenting a simple and fast yet effective way of learning PCs. This approach learns smooth and structure decomposable circuits by randomly partitioning the data space with random projections in a `DIV` class fashion. We show that our method produces competitive PCs at a fraction of the time. The contributions in this chapter come, in part, from R. L. Geh and Denis Deratani Mauá (2021a).

5.1 Probabilistic Circuits and Decision Trees

Before we go through with our proposal in detail, we must first lay the groundwork and motivate the decisions behind our structure learning algorithm. We begin by re-emphasizing the connection between probabilistic circuits and density estimation trees briefly discussed in Example 2.3. We follow by revisiting *random projections* (Freund *et al.*, 2008; Dasgupta and Freund, 2008), a well-known technique for hierarchically partitioning data through oblique hyperplanes. Next, we present in detail a very fast structure learning algorithm for quickly generating smooth and structure decomposable circuits. Despite their simplicity, we empirically show their competitive performance compared to state-of-the-art.

Recently, Correia *et al.* (2020) showed that (ensembles of) decision trees (DTs) learned for prediction tasks can be easily extended into full probabilistic models represented as probabilistic circuits. Besides equipping decision forests with more principles approaches to handling missing data and diagnosing outliers, this bridge between decision trees and PCs suggests an interesting alternative to learning the latter using the efficient inductive algorithms available for the former (Correia *et al.*, 2020; Ram and Gray, 2011; Khosravi *et al.*, 2020). Despite this, most works addressing such a connection have focused on the discriminative side of DTs, with much of the effort put onto classification rather than generative tasks such as density estimation. Here, we explore the generative side of DTs, often referred as density estimation trees (DETs, Ram and Gray, 2011; Hang and Wen,



(a) *Axis-aligned projections* **(b)** *Random projections*

Figure 5.1: *Two partitionings induced by k-d trees:* ***(a)*** *shows axis-aligned splits and* ***(b)*** *random projection splits. Gray dots are datapoints, dashed lines are (hyper)planes.*

2019; Smyth, Gray, *et al.*, 1995), within the framework of PCs, taking inspiration from known algorithms for building DTs and DETs, and transplanting them to PCs.

As shortly discussed in Example 2.3, a DET can be represented as a smooth and deterministic PC with only sums and input nodes, the latter's supports restricted to the data cells induced by the partitioning data hyperplanes. The resulting density of this DET PC is given by

$$p_C(\mathbf{x}) = \sum_{\mathsf{L} \in \mathrm{Inputs}(C)} w_{\mathsf{L}} \cdot \mathsf{L}(\mathbf{x}) \cdot [\![\mathbf{x} \in \mathsf{L}]\!], \tag{5.1}$$

where $[\![\mathbf{x} \in \mathsf{L}]\!]$ is an indicator function that returns 1 if $\mathbf{x}$ is within L's cell and 0 otherwise. The above formula comes from collapsing a circuit with only sum layers, where each sum corresponds to a latent variable representing a partitioning of the data, into a single-layer shallow PC. These latent variables usually consist of partitioning data through hyperplanes, dividing data into two parts, each represented as the subcircuit rooted at each child of the sum node.

A k-d tree is a subclass of decision trees which hierarchically partitions data into more or less equally sized parts, usually by splitting the data according to the value of a single variable at a time (Bentley, 1975; Hang and Wen, 2019; Ho, 1995), essentially producing axis-aligned hyperplanes. Dasgupta and Freund (2008) noted that such an approach cannot ensure that the resulting partitioning of the input space approximates the intrinsic dimensionality of data (roughly understood as a manifold of low dimension). In contrast, they provide a simple strategy for space partitioning that consists in recursively partitioning the space according to a random separating hyperplane. This approximates a random projection of the data and has the following theoretical guarantee (Dasgupta and Freund, 2008):

> If the data has intrinsic dimension d, then with constant probability the part of the data at level d or higher of the tree has average diameter less than half of the data.

Accordingly, the depth of the tree needs only to grow proportionally to the intrinsic dimension and not to the number of variables. In addition to that and to other theoretical

Algorithm 18 SplitSID

Input Dataset $\mathbf{D} \subset \mathbb{R}^m$
Output A partition $\mathbf{S}_1, \mathbf{S}_2$ of $\mathbf{D}$

Let n be the number of examples in $\mathbf{D}$
Sample a random unit direction $\mathbf{w}$
Sort $\mathbf{b} = \mathbf{a} \cdot \mathbf{x}$ for $\mathbf{x} \in \mathbf{D}$ s.t. $b_1 \leq b_2 \leq \cdots \leq b_n$
for each $i \in [n-1]$ **do**
    $\mu_1 = \frac{1}{i} \sum_{j=1}^{i} b_i, \quad \mu_2 = \frac{1}{n-i} \sum_{j=i+1}^{n} b_i$
    $c_i = \sum_{j=1}^{i} (b_j - \mu_1)^2 + \sum_{j+1}^{n} (b_j - \mu_2)^2$
Find i that minimizes c_i and set $\theta = (b_i + b_{i+1})/2$
$\mathbf{S}_1 \longleftarrow \{\mathbf{x} \,|\, \forall \mathbf{x} \in \mathbf{D} \wedge \mathbf{a} \cdot \mathbf{x} \leq \theta\}$
return $(\mathbf{S}_1, \mathbf{D} \setminus \mathbf{S}_1)$

Algorithm 19 SplitMax

Input Dataset $\mathbf{D} \subset \mathbb{R}^m$ and constant r
Output A partition $(\mathbf{S}_1, \mathbf{S}_2)$ of $\mathbf{D}$

Sample a random unit direction $\mathbf{a}$
Pick any $\mathbf{x} \in \mathbf{D}$ and let $\mathbf{y}$ be $\mathbf{x}$'s farthest point in $\mathbf{D}$
Sample δ uniformly in $[-c, c]$, where $c = r \cdot \text{dist}(\mathbf{x}, \mathbf{y})/\sqrt{m}$
$\mathbf{S}_1 \longleftarrow \{\mathbf{x} \,|\, \forall \mathbf{x} \in \mathbf{D} \wedge \mathbf{a} \cdot \mathbf{x} \leq \text{median}(\{\mathbf{z} \cdot \mathbf{a} \,|\, \mathbf{z} \in \mathbf{D}\}) + \delta\}$
return $(\mathbf{S}_1, \mathbf{D} \setminus \mathbf{S}_1)$

insurances (Dhesi and Kar, 2010), the recursive partitioning scheme proposed is extremely fast, taking linearithmic time in the dataset size (number of instances and variables). Freund *et al.* (2008) further empirically show that employing random projections boosts performance significantly compared to regular axis-aligned projections.

5.2 Random Projections

Let $\mathbf{D}$ be a dataset with $\mathbf{X}$ variables. A function $f : \mathcal{X} \longrightarrow \{0, 1\}$ describes a hyperplane over variables in $\mathbf{X}$ and is here called a *rule*. A rule partitions data by assigning observations to either one, $\mathbf{S}_1 = \{\mathbf{x} \,|\, \forall \mathbf{x} \in \mathbf{D} \wedge f(\mathbf{x}) = 0\}$, or the other, $\mathbf{S}_2 = \{\mathbf{x} \,|\, \forall \mathbf{x} \in \mathbf{D} \wedge f(\mathbf{x} = 1)\}$, of two data partitions. In k-d trees, this partitioning proceeds recursively until $|\mathbf{D}|$ is sufficiently small. When employing axis-aligned partitions, f typically selects the variable with the largest variance (or some other measure of spread) in $\mathbf{D}$ and separates instance according to the median value of that variable. The process is similar to the induction of decision trees, except that in this case the rules discriminate against a target variable (Breiman, 2001).

The statistical properties of estimates obtained from the instances at the leaves of a k-d tree depend on the rate at which the diameter of the partitions are reduced once we move down the tree. For a space of dimension m, a k-d tree induced by the process described might require m levels to halve the diameter of the original data (Dasgupta and Freund, 2008). This is true even for datasets of low intrinsic dimension. The latter

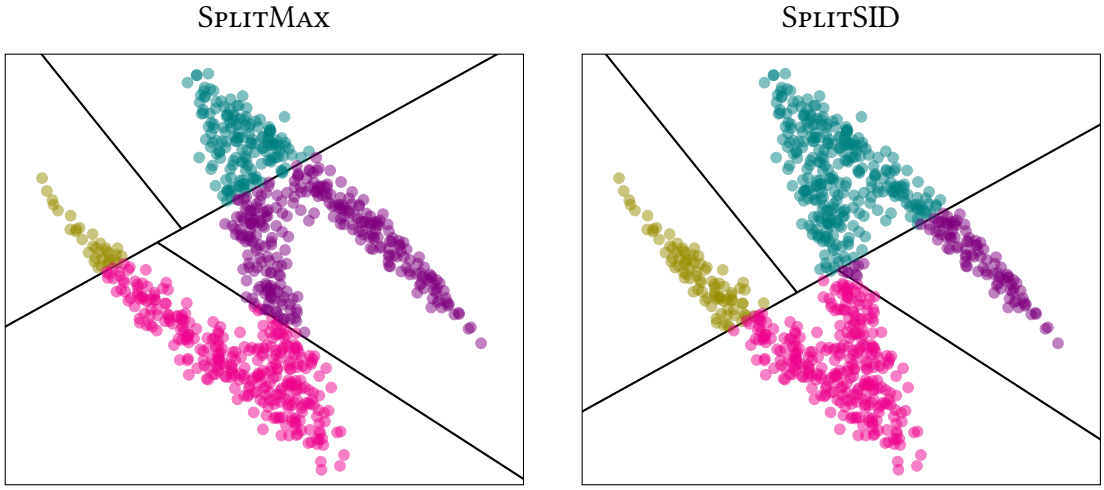


Figure 5.2: *Example of space partitioning by RPTrees grown using different split rules but the same random directions.*

is variously defined, and different definitions lead to different theoretical properties. A common surrogate metric is the *doubling dimension* of the dataset $\mathbf{D} \subset \mathbb{R}^m$, given by the smallest integer d such that the intersection of $\mathbf{D}$ and any ball of radius r centered at $\mathbf{x} \in \mathbf{D}$ can be covered by at most 2^d balls of radius $\frac{r}{2}$ (Dhesi and Kar, 2010).

Random Projection Trees (RPTrees) are a special type of k-d trees that split along a random direction of the space. Two such splitting rules are given by Algorithms 18 and 19, where in the latter dist$(\mathbf{x}, \mathbf{y})$ refers to the Euclidean distance. The intuition behind either rule is to generate a random hyperplane (unit direction) and then find a threshold projection value that roughly divides dataset $\mathbf{D}$ into two approximately equal sized subsets. More concretely, we wish to find a mapping $f : \mathbf{x} \mapsto \left[\!\left[\sum_{i=1}^{d} x_i \cdot a_i > \theta \right]\!\right]$ that discriminates an assignment $\mathbf{x}$ to one or the other side of a hyperplane described as the random unit vector $\mathbf{a} = (a_1, \ldots, a_d)$. Algorithm 18 attempts at finding the projection threshold θ by minimizing the average squared interpoint distance (SID), while Algorithm 19 uses a random noise proportional to the average diameter of $\mathbf{D}$. As discussed by Dasgupta and Freund (2008) and by Freund *et al.* (2008), either optimizing or randomizing the threshold leads to better separation of data than simply selecting the median point. Figure 5.1 shows the difference between axis-aligned and random projections, while Figure 5.2 shows an example of space partitioning induced by 2-level RPTrees using each of the rules with the same direction vectors w. Note that the rules produce quite different splits despite using the same random directions.

Unlike standard k-d trees, RPTrees ensure that, for a data with doubling dimension d, at most d levels are necessary to half the diameter of the data, irrespective of its dimension. This leads to improved statistical properties that are connected to that notion of low intrinsic dimensionality (Dasgupta and Freund, 2008; Dhesi and Kar, 2010). Inspired by these findings, we propose a fast randomized structure learning algorithm for learning probabilistic circuits by recursively stacking random projections in a divide-and-conquer manner similar to what is done in LearnSPN. Albeit our contributions are minor, we found that our approach is extremely fast and reaches competitive performance in binary

Algorithm 20 LearnRP

Input Dataset $\mathbf{D}$, variables $\mathbf{X}$, vtree $\mathcal{V}$ and k projection tryouts
Output A smooth and structure decomposable probabilistic circuit

1: **if** $|\mathbf{X}| = 1$ **then return** an input node learned from $\mathbf{D}$
2: **else**
3: Sample k projections and call f the one which minimizes the avg. diameter of $\mathbf{D}$
4: $\mathbf{S}_1 \longleftarrow \{\mathbf{x} \,|\, \forall \mathbf{x} \in \mathbf{D} \wedge f(\mathbf{x}) = 1\}$, $\mathbf{S}_2 = \{\mathbf{x} \,|\, \forall \mathbf{x} \in \mathbf{D} \wedge f(\mathbf{x}) = 0\}$
5: Let v the root of $\mathcal{V}$
6: $\mathrm{C}_1^{(1)} \longleftarrow \textsc{LearnRP}(\mathbf{S}_1, \mathrm{Sc}(v^{\leftarrow}), v^{\leftarrow}, k)$
7: $\mathrm{C}_2^{(1)} \longleftarrow \textsc{LearnRP}(\mathbf{S}_1, \mathrm{Sc}(v^{\rightarrow}), v^{\rightarrow}, k)$
8: $\mathrm{C}_1^{(2)} \longleftarrow \textsc{LearnRP}(\mathbf{S}_2, \mathrm{Sc}(v^{\leftarrow}), v^{\leftarrow}, k)$
9: $\mathrm{C}_2^{(2)} \longleftarrow \textsc{LearnRP}(\mathbf{S}_2, \mathrm{Sc}(v^{\rightarrow}), v^{\rightarrow}, k)$
10: Construct products $\mathrm{P}_1 \longleftarrow \mathrm{C}_1^{(1)} \cdot \mathrm{C}_2^{(1)}$ and $\mathrm{P}_2 \longleftarrow \mathrm{C}_1^{(2)} \cdot \mathrm{C}_2^{(2)}$
11: **return** sum $\frac{\mathrm{S}_1}{\mathrm{D}} \cdot \mathrm{P}_1 + \frac{\mathrm{S}_2}{\mathrm{D}} \cdot \mathrm{P}_2$

benchmark datasets.

5.3 LearnRP

A random projection (RP) naturally induces a clustering of data: given a rule f, two clusters are formed from the partitions induced by f's hyperplane. We use this simple yet extremely fast clustering method to replace clustering techniques in LearnSPN. We justify this move from a theoretical and practical aspect. From a theoretical perspective, every sum node created this way defines a latent variable corresponding to a hyperplane, giving some interpretability (akin to decision trees) to the model. From a more practical point of view, we point to the work of Vergari, Mauro, *et al.* (2015), showing that binary partitions (both row-wise and column-wise) favor a deeper architecture and produce smaller models. We further strengthen this last point by restricting the learned structure to a vtree, effectively constructing smooth and structure decomposable PCs.

Of note is the fact that DETs are deterministic by nature, as the leaves of the binary tree are constrained to the corresponding cells, meaning that only one path from leaf to root is active at a time. Here, determinism could be enforced if, for every input node (line 1 in Algorithm 20) its support is truncated to the cell induced by all random projections above it, similar to what is done when representing DETs as PCs. For the discrete case, we might attribute zero mass to assignments outside its cell, normalizing the distribution with the remaining mass. However, apart from the fact that doing so is not so trivial in the general case (i.e. in the continuous) as the projections are oblique and the resulting truncated distributions must have tractable marginalization, this process also violates decomposability: by truncating inputs, we are essentially turning the previously univariate inputs from line 1 into multivariate distributions covering the entire scope, making products learned in line 10 nondecomposable. This comes from the fact that each hyperplane is a function $f(\mathbf{x}) = \left[\!\left[\sum_{i=1}^{d} x_i \cdot a_i > \theta \right]\!\right]$ with $\mathrm{Sc}(f) = \mathrm{Sc}(v)$, where v is the vtree node of line 5. Further, because each variable $X \in \mathrm{Sc}(v)$ contributes (linearly) to f, marginalization in this multivariate distribution is not as straightforward. We therefore choose not to constraint

Dataset	Vars	Train	Test	Domain	Dataset	Vars	Train	Test	Domain
ACCIDENTS	111	12758	2551	$\{0,1\}$	NLTCS	16	16181	3236	$\{0,1\}$
AD	1556	2461	491	$\{0,1\}$	PLANTS	69	17412	3482	$\{0,1\}$
AUDIO	100	15000	3000	$\{0,1\}$	PUMSB-STAR	163	12262	2452	$\{0,1\}$
BBC	1058	1670	330	$\{0,1\}$	EACHMOVIE	500	4524	591	$\{0,1\}$
NETFLIX	100	15000	3000	$\{0,1\}$	RETAIL	135	22041	4408	$\{0,1\}$
BOOK	500	8700	1739	$\{0,1\}$	ABALONE	8	3760	417	$\mathbb{R}$
20-NEWSGRP	910	11293	3764	$\{0,1\}$	CA	22	7373	819	$\mathbb{R}$
REUTERS-52	889	6532	1540	$\{0,1\}$	QUAKE	4	1961	217	$\mathbb{R}$
WEBKB	839	2803	838	$\{0,1\}$	SENSORLESS	48	52659	5850	$\mathbb{R}$
DNA	180	1600	1186	$\{0,1\}$	BANKNOTE	4	1235	137	$\mathbb{R}$
JESTER	100	9000	4116	$\{0,1\}$	FLOWSIZE	3	1358674	150963	$\mathbb{R}$
KDD	65	180092	34955	$\{0,1\}$	KINEMATICS	8	7373	819	$\mathbb{R}$
KOSAREK	190	33375	6675	$\{0,1\}$	IRIS	4	90	10	$\mathbb{R}$
MSNBC	17	291326	58265	$\{0,1\}$	OLDFAITH	2	245	27	$\mathbb{R}$
MSWEB	294	29441	5000	$\{0,1\}$	CHEMDIABET	3	131	14	$\mathbb{R}$

Table 5.1: *Details for all binary and continuous benchmark datasets.*

inputs.

5.4 Experiments

For binary data, we evaluate LearnRP on the 20 well-known binary datasets for density estimation (Lowd and Davis, 2010; Van Haaren and Davis, 2012)[1] and compare against reported results from LearnSPN (Gens and P. Domingos, 2013), Strudel, LearnPSDD (both from the benchmarks reported in Dang, Vergari, *et al.*, 2020), Prometheus (Jaini, Ghose, *et al.*, 2018) and XPC (Mauro *et al.*, 2021). To measure performance in the continuous, we compare LearnRP against the performance of Prometheus, deep Boltzmann machines (SRBMs, Salakhutdinov and Hinton, 2009), an offline version of Hsu *et al.*'s online structure learning with Gaussian leaves (oSLRAU, Hsu *et al.*, 2017), Gaussian mixture models with Bayesian moment matching (GBMMs, Jaini, Rashwan, *et al.*, 2016) and infinite sum-product trees (iSPTs, Trapp, Peharz, Skowron, *et al.*, 2016) all of which are reported in Jaini, Ghose, *et al.* (2018). We used the same 10 continuous datasets as Jaini, Ghose, *et al.* (2018)[2], which although are reported as coming from the UCI Machine Learning Repository (Dua and Graff, 2017) and Bilkent University's Function Approximation Repository (Güvenir and Uysal, 2000), are numerically distinct from the ones found in these repositories. Table 5.1 shows detailed information of every dataset evaluated.

When evaluating the performance of LearnRP, we did not see a significant difference between SplitSID and SplitMax, and so we only report figures for the latter. When the domain is discrete, the vtree was learned by the top-down pairwise mutual information algorithm discussed in Section 3.2.1; when in the continuous, we replace mutual information for Pearson's correlation. We use Gaussian mixture models as input nodes and fine-tune the parameters of the resulting structure by standard batch EM. Experiments were carried out on a single computer with a 12-core Intel i7 3.7GHz processor and 64GB RAM.

[1] Taken from https://github.com/UCLA-StarAI/Density-Estimation-Datasets.

[2] Which we compiled to https://github.com/RenatoGeh/CDEBD.

Dataset	LearnSPN	Strudel	LearnPSDD	XPC	LearnRP-F	LearnRP-10	LearnRP-20	LearnRP-30
nltcs	7m	3m	6m	17s	3m19s	4s	8s	12s
plants	50m	41m	26m	1m3s	26m	52s	1m47s	2m32s
audio	2h	33m	51m	1m58s	37m	1m25s	2m45s	4m11s
jester	52m	24m	37m	1m20s	29m	1m20s	2m37s	3m54s
netflix	1h	14m	33m	2m8s	46m	1m40s	3m22s	4m48s
accidents	47m	20m	41m	1m47s	33m	1m30s	2m45s	4m10s
book	>3h	8m	1h22m	2m26s	2h	7m24s	15m17s	21m55s
dna	>3h	>3h	>3h	17s	11m	1m27s	2m47s	3m13s

Table 5.2: *Learning time benchmark for a single circuit of LearnSPN, Strudel, LearnPSDD, XPC and LearnRP.*

LearnRP was implemented in Julia[3], separately from Strudel's and LearnPSDD's implementations found in the Juice package (Dang, Khosravi, *et al.*, 2021). For fairness, when comparing runtimes for speed benchmarking we reran the same or similar setups as originally reported for XPC, Strudel and LearnPSDD in the same machine used for LearnRP, setting the number of iterations to 1000 for the two last `INCR` algorithms. Similarly, we compare runtimes of a Rust implementation[4] of LearnSPN on the same machine; we use k-means for learning sums and G-test for products, setting $k = 2$ for a shorter learning time. We do not report running times for the original Prometheus code since, as far as we know, the source was not made public and no working implementation was found.

5.4.1 Binary data

Table 5.3 shows benchmark results for binary data. For fairness, we report results for best ensembles of XPC, Strudel and LearnPSDD so that all models compared are smooth and structure decomposable (with the exception of LearnSPN and Prometheus which are only decomposable) but nondeterministic. Columns LearnRP-10, LearnRP-20, LearnRP-30 and LearnRP-F correspond to runs of a single LearnRP circuit where we optimize the learned structure with 10, 20, 30 and 100 iterations of batch EM with a batch size of 500 instances. After this, for LearnRP-F, we proceed to run 30 iterations of full EM, a regularization technique suggested in A. Liu and Van den Broeck (2021). The last two rows of Table 5.3 correspond to the average rank of each algorithm, with the last one ignoring LearnRP-10, LearnRP-20 and LearnRP-30.

Table 5.2 shows average running time for each learning algorithm to learn a *single* circuit for each dataset. Note that the performance reported in Table 5.3 for Strudel, LearnPSDD and XPC correspond to ensemble results, meaning that in all but Strudel (where a shared-structure mixture was learned) one has to multiply the averages in Table 5.2 with the number of components in the ensemble to get a better picture of the speed difference.

Table 5.4 shows circuit sizes for each learning algorithm. All except for LearnSPN are reported as in their original works to better reflect the log-likelihood results shown in Table 5.3. Because Gens and P. Domingos (2013) do not report circuit sizes, we report

[3] The source code can be found at https://github.com/RenatoGeh/RPCircuits.jl.

[4] Source code found at https://gitlab.com/marcheing/spn-rs.

values from our own runs. We do not show circuit sizes for PROMETHEUS since we could not find the source code and JAINI, GHOSE, *et al.* (2018) do not report circuit sizes.

Overall, we found that LEARNRP was competitive against the state-of-the-art, often reaching second or first place in the case of LEARNRP-F. As expected for such a simple learning algorithm, it was hardly the best, although even under few EM iterations it was capable of producing somewhat competitive models. Arguably, LEARNRP's strengths come from its speed, learning circuits in a fraction of the time when compared to LEARNSPN, STRUDEL and LEARNPSDD. Perhaps the more direct competitor of LEARNRP in terms of scalability is XPC, showing the strengths of RAND when it comes to speed. Recall from Section 3.3.2, however that XPC requires an extensive grid search on the hyperparameters, while the performance of LEARNRP mainly depends on how much time one is willing to spend to fine-tune weights with parameter learning. Notably, we found that LEARNRP performs worse on data with fewer variables (e.g. NLTCS, MSNBC, KDD) and better on data with more variables (e.g. 20-NEWSGRP, BOOK, WEBKB), which is perhaps correlated with the smaller, and respectively larger, circuits from Table 5.4.

5.4.2 Continuous data

Table 5.5 shows results for continuous data. Note that some entries are positive since the log-likelihood of continuous variables can be positive. Because the evaluated continuous datasets were small in size, we only show results for LEARNRP under 100 iterations of EM. We also do not run full EM after the batch variant as we found performance to degrade after doing so. When constructing the circuit with LEARNRP, we use mixtures of three Gaussians as input nodes. We run learn both sum weights and input GMMs through EM. The last column of Table 5.5 shows the circuit sizes of LEARNRP.

Strangely, we found that LEARNRP behaves extremely poorly on continuous datasets. This could be both due to the smaller number of variables in these datasets (as evidenced by the reduced circuit sizes in Table 5.5's last column), causing LEARNRP to produce very small models; and because of the unpredictability and inconsistency of log-likelihood as a score in the continuous.

Dataset	LearnSPN	Strudel	LearnPSDD	XPC	Prometheus	LearnRP-F	LearnRP-10	LearnRP-20	LearnRP-30
accidents	-30.03	|-28.73|	-30.16	-31.02	**-27.91**	-28.71	-30.18	-29.66	-29.17
ad	-19.73	-16.38	-31.78	**-15.50**	-23.96	|-19.09|	-22.51	-20.87	-20.37
audio	-40.50	-41.50	-39.94	-40.91	**-39.80**	|-40.16|	-41.05	-40.66	-40.68
bbc	|-250.68|	-254.41	-253.19	**-248.34**	-248.50	-255.16	-259.17	-260.32	-254.09
netflix	-57.02	-58.69	**-55.71**	-57.58	-56.47	|-56.97|	-59.91	-60.06	-57.60
book	-35.88	-34.99	-34.97	-34.75	|-34.40|	**-33.51**	-36.15	-34.45	-33.95
20-newsgrp	-155.92	-154.47	-155.97	-153.75	|-154.17|	**-152.94**	-158.10	-156.74	-154.35
reuters-52	|-85.06|	-86.22	-89.61	-84.70	**-84.59**	-85.56	-91.61	-87.73	-86.62
webkb	-158.20	-155.33	-161.09	**-153.67**	|-155.21|	-154.33	-156.40	-155.70	-156.40
dna	**-82.52**	-86.22	-88.01	-86.61	-84.45	|-85.12|	-86.55	-85.30	-85.61
jester	-75.98	-55.03	**-51.29**	-53.43	-52.80	|-52.97|	-53.67	-53.28	-53.13
kdd	-2.18	|-2.13|	**-2.11**	-2.15	-2.12	-2.13	-2.20	-2.19	-2.15
kosarek	-10.98	-10.68	**-10.52**	-10.77	-10.59	|-10.61|	-10.97	-10.86	-10.85
msnbc	-6.11	**-6.04**	**-6.04**	|-6.18|	**-6.04**	-6.34	-6.39	-6.42	-6.39
msweb	-10.25	**-9.71**	|-9.89|	-9.93	-9.86	|-9.89|	-10.23	-10.17	-10.10
nltcs	-6.11	-6.06	**-5.99**	|-6.05|	-6.01	-6.22	-6.30	-6.29	-6.37
plants	-12.97	|-12.98|	-13.02	-14.19	**-12.81**	-13.80	-14.76	-14.33	-14.26
pumsb-star	|-24.78|	-24.12	-26.12	-26.06	**-22.75**	-26.45	-27.14	-26.78	-26.75
eachmovie	-52.48	-53.67	-58.01	-54.82	|-51.49|	-51.46	-52.68	-52.34	**-51.43**
retail	-11.04	-10.81	**-10.72**	-10.94	-10.87	|-10.83|	-11.08	-10.98	-10.95
Avg. Rank	**4.3**	|4.8|	5.2	4.85	4.9	4.6	5.95	5.45	4.95
	4.3	3.65	|3.55|	3.9	**2.15**	3.45			

Table 5.3: *Performance of LearnRP in log-likelihood against state-of-the-art competitors in the twenty binary datasets for density estimation. Entries in* **bold** *correspond to best performance,* underlined *entries are second best, and* |barred| *entries are third place.*

Dataset	LearnSPN	Strudel	LearnPSDD	XPC	LearnRP-F	LearnRP-10	LearnRP-20	LearnRP-30
accidents	32708	75363	8418	11921	17609	17693	17309	17977
ad	40901	13152	12238	22093	89525	88877	94111	87117
audio	50130	55675	18208	29317	17319	17457	17369	17621
bbc	39389	29532	12335	14578	124439	124717	123765	126019
netflix	36286	27173	10997	39868	20663	20539	20631	20447
book	51493	54839	10978	13678	109881	108057	112303	109367
20-newsgrp	119060	58749	15793	65881	486105	485427	485733	481613
reuters-52	155191	36343	10410	36440	285060	286162	289292	285870
webkb	223847	25406	11033	17122	207462	209069	205128	208299
dna	12180	17507	3068	2616	26567	26527	26007	26495
jester	25076	27713	11322	20273	20679	20887	21025	21065
kdd	8755	6572	2915	13040	5389	5219	5229	5251
kosarek	19512	37583	7173	20938	32357	33455	33087	32993
msnbc	11606	20795	5465	4887	789	801	765	778
msweb	10743	2347	6581	12135	48545	48805	49715	50235
nltcs	1855	4373	1304	4401	515	511	491	519
plants	36596	119194	11583	13960	9335	9317	9941	9655
pumsb-star	26206	108876	8298	8866	28725	28937	27731	29930
eachmovie	54184	123996	20648	21369	69357	67047	68623	69193
retail	2158	3979	2989	6651	24345	24459	25031	24103

Table 5.4: *Circuit size (in the number of nodes) comparison between LearnRP and the state-of-the-art in the twenty binary datasets for density estimation.*

Dataset	SRBMs	oSLRAU	GBMMs	GMMs	Prometheus	iSPTs	LearnRP	Size
abalone	-2.28	-0.94	-1.17	—	-0.85	—	-6.13	317
ca	-4.95	21.19	3.42	—	27.82	—	-5.84	2765
quake	-2.38	-1.21	-3.76	—	-1.50	—	-3.76	79
sensorless	-26.91	60.72	8.56	—	62.03	—	-38.46	12589
banknote	-2.76	-1.39	-4.64	—	-1.96	—	-6.06	79
flowsize	-0.79	15.32	5.72	—	18.03	—	2.20	49
kinematics	-5.55	-11.13	-11.20	—	-11.12	—	-11.02	319
iris	—	—	—	-3.94	-1.06	-3.74	-3.47	79
oldfaith	—	—	—	-1.73	-1.48	-1.70	-4.33	19
chemdiabet	—	—	—	-3.02	-2.59	-2.88	-18.68	48

Table 5.5: *Performance of LearnRP in log-likelihood against state-of-the-art competitors in ten continuous datasets for density estimation and function approximation. Entries in* **bold** *correspond to best performance,* underlined *entries are second best, and* |barred| *entries are third place. Last column shows size (in the number of nodes) of circuits learned with LearnRP.*

6

Contributions, Discussion and Future Work

In this chapter we conclude this dissertation and provide a brief discussion on the topics touched throughout this work, highlighting our contributions and pointing to possible future work on scalably learning probabilistic circuits.

6.1 Contributions

The objectives of this dissertation were two-fold: to provide a concise review of state-of-the-art literature on the subject of structure learning of probabilistic circuits; and propose two ideas for scalably learning the structure of PCs. For the former, we classified learning algorithms into three classes (Chapter 3): *divide-and-conquer* (`DIV`) learning, where we recursively divide available knowledge (data or logical formula) into smaller partitions, eventually joining them together (Section 3.1); *incremental* learning (`INCR`), where we iteratively grow a circuit through local transformations usually guided by a score (Section 3.2); and finally *random* learning, based around the concept of sampling circuits either from knowledge or completely random (Section 3.3).

We follow this literature review and taxonomy on structure learning by addressing two cases of `RAND` algorithms, both of which draw inspiration from `DIV` and `INCR` (Chapters 4 and 5). Each tackles the problem of learning PCs from distinct point of views: we first propose SamplePSDD to learn a circuit from certain knowledge, constructing a smooth, structure decomposable and deterministic PC (i.e. a PSDD) from both logical formula and data. To scale up to the hundreds of variables we restrict the PSDD's support to only a relaxation of the formula, showing that by aggregating several sampled circuits into an ensemble we achieve competitive performance against the state-of-the-art (Chapter 4). Next, we describe a very simple `RAND` structure learning algorithm based on random projections, which we call LearnRP, for producing smooth and structure decomposable PCs solely from data. We show that our approach is orders of magnitude faster compared to competitors, achieving relatively competitive performance on binary data (Chapter 5).

6.2 Discussion and Future Work

Despite the interesting results reported in Chapters 4 and 5, there is much room for improvement. We end this dissertation by addressing the flaws of both SamplePSDD and LearnRP, pointing to their weaknesses and suggesting possible ideas for further work.

6.2.1 SamplePSDD

We now explore some interesting paths to take for further work on SamplePSDD. We summarize them through the topics below, providing a short discussion on future work.

Primes are conjunctions of literals. Although this is a common assumption, it restricts the space of sampled PSDDs to a very specific class of PSDDs. One may draw connections to BDDs and argue that this class of PSDDs is a generalization BDDs in the sense that a (PSDD) partition of this nature is akin to a more general case of Shannon's decomposition where instead of conditioning on a single variable, we (randomly) choose a subset of variables. We suggest possible further work on sampling PSDDs from other kinds of decompositions other than Shannon's.

Search-based sampling. SamplePSDD blindly samples circuits from a logical formula regardless of how well it models data. Guiding which variables are sampled as primes and which primes are compressed or merged could provide a better data fit model.

Simultaneously learning the vtree. Our proposed algorithm for SamplePSDD is completely decoupled from the process of learning a vtree. Learning the vtree during sampling could potentially provide a better fit to the overall model.

6.2.2 LearnRP

In this section, we provide a discussion on LearnRP and possible paths from both a theoretical as well as a more practical point of view.

Extending the theoretical works from random projections. As mentioned in Section 5.2, there are several interesting theoretical results coming primarily from the decision tree literature. Now that Correia *et al.* (2020) have made clear the connection between decision trees and PCs, it would be interesting to understand whether these same results extend to more general PCs.

Enforcing determinism. Random projection trees are naturally deterministic, however circuits learned through LearnRP are not. This means LearnRP PCs are not as interpretable as, say density estimation trees, where each assignment produces a clear path "explaining" the decisions taken by the model. As far as we know, to retain the same semantics of random projections *and* enforce determinism would require explicitly laying out the latent

variables representing each (hyperplane) rule, effectively creating an exponential number of latent variable indicators for each sum node.

Simultaneously learning the vtree. Similar to SamplePSDD, LearnRP fixes a learned vtree and produces a PC from it. The choice of which variables to partition deeply impacts how the random projections are sampled. Choosing a partitioning according to some score could greatly enhance data fitness.

A

Appendices

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

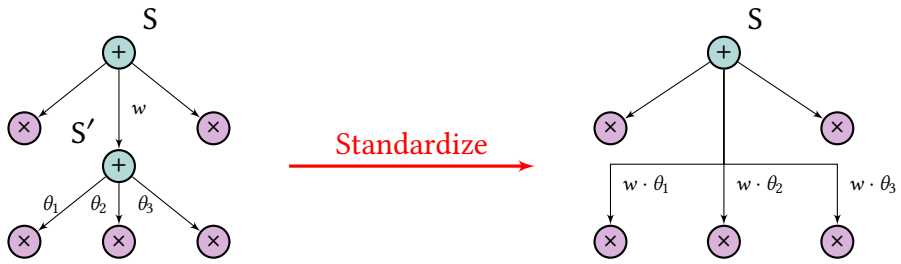
procedure until there are no more consecutive sum nodes. This results in a PC of the form

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

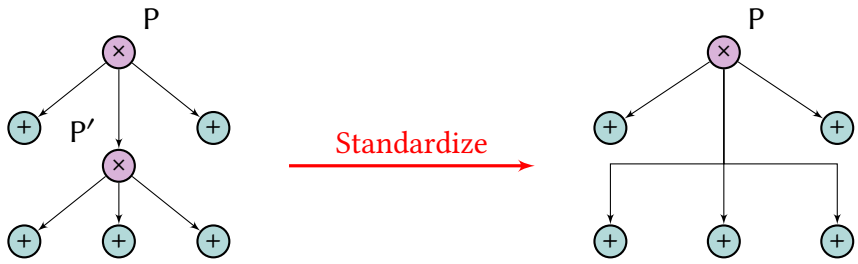
where $\mathrm{N} \in \mathbf{N}$. The number of children of the resulting root sum node will be exponential on the number of edges of its children. □

Theorem A.1.2 (Standardization). *Any probabilistic circuit $\mathcal{C}$ 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 $\mathcal{C}$ 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 $\overrightarrow{\mathrm{S\,S'}}$ 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.



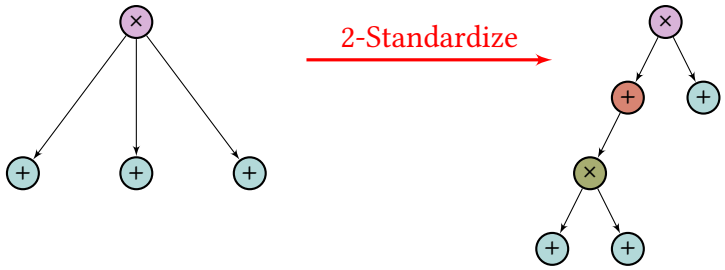
□

Theorem A.1.3 (2-Standardization). *Any probabilistic circuit $\mathcal{C}$ 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.*

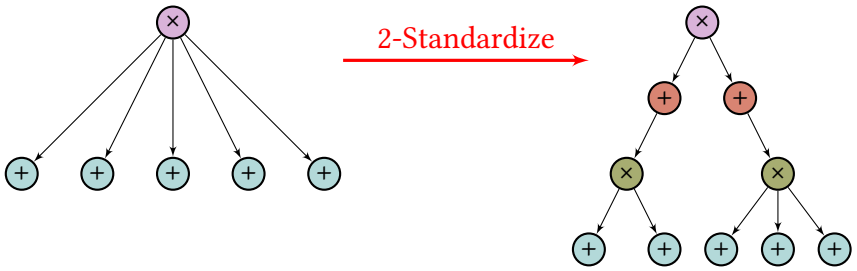
Proof. For sums, apply the same standardization procedure as theorem A.1.2. Let P a product and call $n = |\mathrm{Ch(P)}|$. If $n = 1$ and Ch(P) is a product, then remove P and connect

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 $\mathcal{C}$ be a* smooth *and* decomposable *PC. Any one of EVI, MAR or CON can be computed in linear time (in the size of $\mathcal{C}$).*

Proof. For a sum node S and query variables $\mathbf{X}$, we have the following marginalization of variables $\mathbf{Y}$

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

Analogously, for a product node

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

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 $p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{x},\mathbf{y})}{p(\mathbf{y})}$ which are both done in linear time as we have seen here. □

Theorem 2.2.2 (Peharz, Gens, Pernkopf, *et al.*, 2016)**.** *Let $\mathcal{C}$ be a smooth, decomposable and* deterministic *PC.* MaxProduct *computes the MAP in $\mathcal{C}$ in linear time (on the size of $\mathcal{C}$).*

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

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

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

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

For a product node P, we compute

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

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 $\mathbb{E}_{\mathcal{C}}[\mathcal{L}]$ has the following form when the root of $\mathcal{C}$ is a product

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

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

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

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

B

Annexes

B.1 Learning PSDDs Under Logic Constraints by Sampling and Averaging

This page intentionally left blank.

B.2 Fast And Accurate Learning of PCs by Random Projections

This page intentionally left blank.

References

[Akers 1978] S. B. Akers. "Binary decision diagrams". In: *IEEE Transactions on Computers* 27.6 (1978), pp. 509–516 (cit. on p. 10).

[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. 24).

[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. 24).

[Bach and Jordan 2001] Francis R. Bach and Michael I. Jordan. "Thin junction trees". In: *Proceedings of the 14th International Conference on Neural Information Processing Systems*. NeurIPS. 2001, pp. 569–576 (cit. on p. 15).

[Barwise 1982] Jon Barwise. *Handbook of Mathematical Logic*. 1982 (cit. on p. 10).

[Bentley 1975] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. issn: 0001-0782. doi: 10.1145/361002.361007. url: https://doi.org/10.1145/361002.361007 (cit. on p. 74).

[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. 31).

[Bouchard and Triggs 2004] Guillaume Bouchard and Bill Triggs. "The Tradeoff Between Generative and Discriminative Classifiers". In: *16th IASC International Symposium on Computational Statistics (COMPSTAT '04)*. Prague, Czech Republic, Aug. 2004, pp. 721–728. url: https://hal.inria.fr/inria-00548546 (cit. on p. 48).

[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. 42).

[Bova 2016] Simone Bova. "Sdds are exponentially more succinct than obdds". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, 2016, pp. 929–935 (cit. on p. 46).

[Breiman 2001] Leo Breiman. "Random forests". In: *Machine Learning* 45 (2001), pp. 5–32 (cit. on p. 75).

[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. 20, 21, 46).

[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. 35).

[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. 24).

[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. 36).

[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. 25).

[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. 2, 24).

[A. Choi, Broeck, *et al.* 2015] Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. "Tractable learning for structured probability spaces: A case study in learning preference distributions". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015, pp. 2861–2868 (cit. on pp. 46, 57, 68).

[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 pp. 46, 57).

[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 pp. 46, 57).

[A. Choi, Tavabi, *et al.* 2016] Arthur Choi, Nazgol Tavabi, and Adnan Darwiche. "Structured features in naive Bayes classification". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 2016, pp. 3233–3240 (cit. on pp. 46, 57).

[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. 18, 23, 90).

[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. 12, 89).

[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. 41, 45).

[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. 11).

[Correia *et al.* 2020] Alvaro H. C. Correia, Robert Peharz, and Cassio de Campos. "Joints in random forests". In: *Advances in Neural Information Processing Systems 33 (NeurIPS)*. 2020 (cit. on pp. 3, 13, 73, 84).

[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. 10).

[Dang, Khosravi, *et al.* 2021] Meihua Dang, Pasha Khosravi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. "Juice: a julia package for logic and probabilistic circuits". In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence (Demo Track)*. 2021 (cit. on pp. 64, 79).

[Dang, Vergari, *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. 2, 41, 43, 45, 65, 78).

[Darwiche 1999] Adnan Darwiche. "Compiling knowledge into decomposable negation normal form". In: *IJCAI*. Vol. 99. 1999, pp. 284–289 (cit. on p. 20).

[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. 20).

[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. 1, 15, 17).

[Darwiche 2009] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511811357 (cit. on p. 11).

[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. 10, 20, 46, 57, 63).

[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. 20).

[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 p. 20).

[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 pp. 3, 4, 73–76).

[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. 11).

[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 pp. 1, 17).

[Dempster *et al.* 1977] A. P. Dempster, N. M. Laird, and D. B. Rubin. "Maximum likelihood from incomplete data via the em algorithm". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 39.1 (1977), pp. 1–38. ISSN: 00359246. URL: http://www.jstor.org/stable/2984875 (cit. on pp. 48, 50).

[Dennis and Ventura 2012] Aaron Dennis and Dan Ventura. "Learning the architecture of sum-product networks using clustering on variables". In: *Advances in Neural Information Processing Systems 25*. NIPS, 2012, pp. 2033–2041 (cit. on p. 47).

[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. 24).

[Dhesi and Kar 2010] Aman Dhesi and Purushottam Kar. "Random projection trees revisited". In: *Advances in Neural Information Processing Systems*. Vol. 23. NeurIPS. 2010 (cit. on pp. 75, 76).

[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. 2, 35).

[Dua and Graff 2017] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml (cit. on p. 78).

[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. 37, 40).

[Eén and Sörensson 2006] Niklas Eén and Niklas Sörensson. "Translating pseudo-boolean constraints into SAT". In: *Journal on Satisfiability, Boolean Modeling and Computation* (2006) (cit. on pp. 46, 67).

[Enshaei and Naderkhani 2019] Nastaran Enshaei and Farnoosh Naderkhani. "Application of deep learning for fault diagnostic in induction machine's bearings". In: *2019 IEEE International Conference on Prognostics and Health Management (ICPHM)*. 2019, pp. 1–7. DOI: 10.1109/ICPHM.2019.8819421 (cit. on p. 1).

[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. 35).

[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 pp. 3, 4, 73, 75, 76).

[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. 10).

[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. 24).

[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. 24).

[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. 10, 25).

[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. 37).

[Gatterbauer and Suciu 2014] Wolfgang Gatterbauer and Dan Suciu. "Oblivious bounds on the probability of boolean functions". In: *ACM Transactions on Database Systems* 39.1 (2014) (cit. on p. 61).

[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 pp. 24, 25).

[R. Geh, D. Mauá, and Antonucci 2020] Renato Geh, Denis Mauá, and Alessandro Antonucci. "Learning probabilistic sentential decision diagrams by sampling". In: *Proceedings of the VIII Symposium on Knowledge Discovery, Mining and Learning*. SBC, 2020 (cit. on pp. 58, 60).

[R. L. Geh and Denis Deratani Mauá 2021a] Renato Lui Geh and Denis Deratani Mauá. "Fast and accurate learning of probabilistic circuits by random projections". In: *The 4th Tractable Probabilistic Modeling Workshop*. 2021 (cit. on p. 73).

[R. L. Geh and Denis Deratani Mauá 2021b] 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. 2–4, 57).

[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. 24, 51).

[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. 2, 28, 29, 78, 79).

[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. 20).

[Gopalakrishnan *et al.* 1991] P.S. Gopalakrishnan, D. Kanevsky, A. Nadas, and D. Nahamoo. "An inequality for rational functions with applications to some statistical estimation problems". In: *IEEE Transactions on Information Theory* 37.1 (1991), pp. 107–113. doi: 10.1109/18.61108 (cit. on p. 50).

[Grigorescu *et al.* 2020] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. "A survey of deep learning techniques for autonomous driving". In: *Journal of Field Robotics* 37.3 (2020), pp. 362–386 (cit. on p. 1).

[Güvenir and Uysal 2000] Halil Altay Güvenir and İlhan Uysal. *Bilkent University Function Approximation Repository*. Jan. 2000 (cit. on p. 78).

[Hang and Wen 2019] Hanyuan Hang and Hongwei Wen. *Best-scored Random Forest Density Estimation*. 2019. arXiv: 1905.03729 [stat.ML] (cit. on pp. 73, 74).

[Herrmann and Barros 2013] Ricardo G. Herrmann and Leliane N. de Barros. "Algebraic sentential decision diagrams in symbolic probabilistic planning". In: *2013 Brazilian Conference on Intelligent Systems*. 2013, pp. 175–181. doi: 10.1109/BRACIS.2013.37 (cit. on p. 46).

[Ho 1995] Tin Kam Ho. "Random decision forests". In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. Vol. 1. 1995, 278–282 vol.1. doi: 10.1109/ICDAR.1995.598994 (cit. on pp. 73, 74).

[Hsu *et al.* 2017] Wilson Hsu, Agastya Kalra, and Pascal Poupart. *Online Structure Learning for Sum-Product Networks with Gaussian Leaves*. 2017. arXiv: 1701.05265 [stat.ML] (cit. on p. 78).

[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 pp. 1, 17).

[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. 2, 31, 33, 34, 78, 80).

[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. 87).

[Jaini, Rashwan, *et al.* 2016] Priyank Jaini, Abdullah Rashwan, *et al.* "Online algorithms for sum-product networks with continuous variables". In: *Proceedings of the Eighth International Conference on Probabilistic Graphical Models*. Ed. by Alessandro Antonucci, Giorgio Corani, and Cassio Polpo Campos. Vol. 52. Proceedings of Machine Learning Research. Lugano, Switzerland: PMLR, June 2016, pp. 228–239. URL: https://proceedings.mlr.press/v52/jaini16.html (cit. on pp. 51, 78).

[Kamishima 2003] Toshihiro Kamishima. "Nantonac collaborative filtering: recommendation based on order responses". In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 2003 (cit. on p. 67).

[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. 37, 40).

[Khan and Yairi 2018] Samir Khan and Takehisa Yairi. "A review on the application of deep learning in system health management". In: *Mechanical Systems and Signal Processing* 107 (2018), pp. 241–265. ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2017.11.024. URL: https://www.sciencedirect.com/science/article/pii/S0888327017306064 (cit. on p. 1).

[Khosravi *et al.* 2020] Pasha Khosravi, Antonio Vergari, YooJung Choi, Yitao Liang, and Guy Van den Broeck. *Handling Missing Data in Decision Trees: A Probabilistic Approach*. 2020 (cit. on p. 73).

[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. 1, 17, 23, 50, 57, 64).

[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. 37, 40).

[Lan *et al.* 2018] Kun Lan *et al.* "A survey of data mining and deep learning in bioinformatics". In: *Journal of Medical Systems* 42.8 (June 2018), p. 139. ISSN: 1573-689X. DOI: 10.1007/s10916-018-1003-9 (cit. on p. 1).

[Li *et al.* 2019] Yu Li *et al.* "Deep learning in bioinformatics: introduction, application, and perspective in the big data era". In: *Methods* 166 (2019). Deep Learning in Bioinformatics, pp. 4–21. ISSN: 1046-2023 (cit. on p. 1).

[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. 2, 36–38, 40, 41).

[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 pp. 10, 50).

[A. Liu and Van den Broeck 2021] Anji Liu and Guy Van den Broeck. "Tractable regularization of probabilistic circuits". In: *Advances in Neural Information Processing Systems 35 (NeurIPS)*. Dec. 2021. URL: http://starai.cs.ucla.edu/papers/LiuNeurIPS21.pdf (cit. on p. 79).

[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. 35).

[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. 24).

[Lomuscio and Paquet 2015] Alessio Lomuscio and Hugo Paquet. "Verification of multi-agent systems via sdd-based model checking". In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. AAMAS '15. Istanbul, Turkey: International Foundation for Autonomous Agents and Multiagent Systems, 2015, pp. 1713–1714. ISBN: 9781450334136 (cit. on p. 46).

[Lou *et al.* 2019] Bin Lou *et al.* "An image-based deep learning framework for individualising radiotherapy dose: a retrospective analysis of outcome prediction". In: *The Lancet Digital Health* 1.3 (2019), e136–e147. ISSN: 2589-7500 (cit. on p. 1).

[Lowd and Davis 2010] Daniel Lowd and Jesse Davis. "Learning markov network structure with decision trees". In: *2010 IEEE International Conference on Data Mining*. 2010 (cit. on p. 78).

[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 pp. 15, 31).

[Lowd and Rooshenas 2015] Daniel Lowd and Amirmohammad Rooshenas. "The libra toolkit for probabilistic models". In: *Journal of Machine Learning Research* 16 (2015), pp. 2459–2463 (cit. on p. 55).

[Lu *et al.* 2013] Wei-Lwun Lu, Jo-Anne Ting, James J. Little, and Kevin P. Murphy. "Learning to track and identify players from broadcast sports videos". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.7 (2013), pp. 1704–1716. DOI: 10.1109/TPAMI.2012.242 (cit. on p. 1).

[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 p. 30).

[Mattei, Antonucci, *et al.* 2020a] Lilith Mattei, Alessandro Antonucci, Denis Deratani Mauá, Alessandro Facchini, and Julissa Villanueva Llerena. "Tractable inference in credal sentential decision diagrams". In: *International Journal of Approximate Reasoning* 125 (2020), pp. 26–48. ISSN: 0888-613X. DOI: https://doi.org/10.1016/j.ijar.2020.06.005. URL: https://www.sciencedirect.com/science/article/pii/S0888613X20301845 (cit. on p. 10).

[Mattei, Antonucci, *et al.* 2020b] Lilith Mattei, Alessandro Antonucci, Denis Deratani Mauá, Alessandro Facchini, and Julissa Villanueva Llerena. "Tractable inference in credal sentential decision diagrams". In: *International Journal of Approximate Reasoning* 125 (2020), pp. 26–48 (cit. on p. 65).

[Mattei, Soares, *et al.* 2019] Lilith Mattei, Décio L. Soares, Alessandro Antonucci, Denis D. Mauá, and Alessandro Facchini. "Exploring the space of probabilistic sentential decision diagrams". In: *3rd Workshop of Tractable Probabilistic Modeling*. 2019 (cit. on pp. 57, 58).

[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. 10).

[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 pp. 2, 47, 51, 54, 78).

[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. 11).

[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. 10).

[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. 24).

[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. 35).

[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. 35).

[Monteith *et al.* 2011] Kristine Monteith, James L. Carroll, Kevin Seppi, and Tony Martinez. "Turning bayesian model averaging into bayesian model combination". In: *The 2011 International Joint Conference on Neural Networks*. 2011 (cit. on p. 64).

[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. 35).

[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. 2, 25).

[Niehues *et al.* 2018] Jan Niehues, Ngoc-Quan Pham, Thanh-Le Ha, Matthias Sperber, and Alex Waibel. "Low-latency neural speech translation". In: *Interspeech 2018, 19th Annual Conference of the International Speech Communication Association, Hyderabad, India, 2-6 September 2018*. Ed. by B. Yegnanarayana. ISCA, 2018, pp. 1293–1297. doi: 10.21437/Interspeech.2018-1055 (cit. on p. 1).

[Nishino *et al.* 2016] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. "Zero-suppressed sentential decision diagrams". In: *AAAI Conference on Artificial Intelligence*. 2016 (cit. on pp. 46, 57).

[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. 24).

[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. 25).

[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 pp. 46, 57).

[Papadimitriou 1994] C.H. Papadimitriou. *Computational Complexity*. Theoretical computer science. Addison-Wesley, 1994. ISBN: 9780201530827 (cit. on p. 20).

[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. 50).

[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. 7, 11, 14, 48, 90).

[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. 24).

[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. 2, 10, 24, 51).

[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, 12, 50).

[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. 2, 24, 46–50).

[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. 10).

[Pogancic *et al.* 2020] Marin Vlastelica Pogancic, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. "Differentiation of blackbox combinatorial solvers". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. 2020 (cit. on p. 1).

[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. 1, 2, 7, 12, 17, 24, 50, 89).

[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. 25).

[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. 36).

[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 pp. 1, 17).

[Ram and Gray 2011] Parikshit Ram and Alexander G. Gray. "Density estimation trees". In: *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '11. San Diego, California, USA: Association for Computing Machinery, 2011, pp. 627–635. ISBN: 9781450308137. DOI: 10.1145/2020408.2020507. URL: https://doi.org/10.1145/2020408.2020507 (cit. on pp. 13, 73).

[Rashwan, Poupart, *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. 50).

[Rashwan, H. Zhao, *et al.* 2016] Abdullah Rashwan, Han Zhao, and Pascal Poupart. "Online and distributed bayesian moment matching for parameter learning in sum-product networks". 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. 1469–1477. URL: http://proceedings.mlr.press/v51/rashwan16.html (cit. on p. 51).

[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. 24).

[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. 24).

[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. 24).

[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. 30, 31, 55).

[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. 25).

[Salakhutdinov and Hinton 2009] Ruslan Salakhutdinov and Geoffrey Hinton. "Deep boltzmann machines". In: *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*. Ed. by David van Dyk and Max Welling. Vol. 5. Proceedings of Machine Learning Research. Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA: PMLR, Apr. 2009, pp. 448–455. URL: https://proceedings.mlr.press/v5/salakhutdinov09a.html (cit. on p. 78).

[Sezer *et al.* 2020] Omer Berat Sezer, Mehmet Ugur Gudelek, and Ahmet Murat Ozbayoglu. "Financial time series forecasting with deep learning : a systematic literature review: 2005–2019". In: *Applied Soft Computing* 90 (2020), p. 106181. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2020.106181. URL: https://www.sciencedirect.com/science/article/pii/S1568494620301216 (cit. on p. 1).

[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 pp. 24, 25).

[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. 25).

[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. 25).

[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. 25).

[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. 10).

[Sharir, Tamari, *et al.* 2018] Or Sharir, Ronen Tamari, Nadav Cohen, and Amnon Shashua. *Tensorial Mixture Models*. 2018. arXiv: 1610.04167 [cs.LG] (cit. on p. 50).

[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 pp. 46, 57, 67).

[Sinz 2005] Carsten Sinz. "Towards an optimal CNF encoding of boolean cardinality constraints". In: *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*. 2005 (cit. on pp. 46, 57).

[Smyth, Gray, *et al.* 1995] Padhraic Smyth, Alexander G. Gray, and Usama M. Fayyad. "Retrofitting decision tree classifiers using kernel density estimation". In: *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning, Tahoe City, California, USA, July 9-12, 1995*. Ed. by Armand Prieditis and Stuart J. Russell. Morgan Kaufmann, 1995, pp. 506–514. doi: 10.1016/b978-1-55860-377-6.50069-4. url: https://doi.org/10.1016/b978-1-55860-377-6.50069-4 (cit. on p. 74).

[Smyth and Wolpert 1998] Padhraic Smyth and David Wolpert. "Stacked density estimation". In: *Advances in Neural Information Processing Systems*. Ed. by M. Jordan, M. Kearns, and S. Solla. Vol. 10. MIT Press, 1998 (cit. on p. 64).

[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. 25).

[Spirtes and Meek 1995] Peter Spirtes and Christopher Meek. "Learning bayesian networks with discrete variables from data". In: *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*. KDD'95. Montréal, Québec, Canada: AAAI Press, 1995, pp. 294–299 (cit. on p. 27).

[Stuhlmüller and Goodman 2012] Andreas Stuhlmüller 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. 25).

[Teyssier and Koller 2005] Marc Teyssier and Daphne Koller. "Ordering-based search: a simple and effective algorithm for learning bayesian networks". In: *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*. UAI'05. Edinburgh, Scotland: AUAI Press, 2005, pp. 584–590. isbn: 0974903914 (cit. on p. 36).

[Trapp, Peharz, Ge, *et al.* 2019] Martin Trapp, Robert Peharz, Hong Ge, Franz Pernkopf, and Zoubin Ghahramani. "Bayesian learning of sum-product networks". In: *Advances in Neural Information Processing Systems 32*. 2019, pp. 6347–6358 (cit. on p. 51).

[Trapp, Peharz, Skowron, *et al.* 2016] Martin Trapp, Robert Peharz, M. Skowron, *et al.* "Structure inference in sum-product networks using infinite sum-product trees". In: *Neural Information Processing Systems workshop*. 2016 (cit. on p. 78).

[Van Haaren and Davis 2012] Jan Van Haaren and Jesse Davis. "Markov network structure learning: a randomized feature generation approach". In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. 2012 (cit. on p. 78).

[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. 10, 12, 17, 18, 89).

[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. 2, 35, 77).

[Vergari, Molina, *et al.* 2019] Antonio Vergari, Alejandro Molina, *et al.* "Automatic bayesian density analysis". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 5207–5215. doi: 10.1609/aaai.v33i01.33015207. url: https://ojs.aaai.org/index.php/AAAI/article/view/4977 (cit. on p. 51).

[Vlasselaer, Broeck, *et al.* 2015] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. "Anytime inference in probabilistic logic programs with tp-compilation". In: (2015). url: https://www.aaai.org/ocs/index.php/IJCAI/IJCAI15/paper/view/11005 (cit. on p. 46).

[Vlasselaer, Renkens, *et al.* 2014] Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. "Compiling probabilistic logic programs into sentential decision diagrams". eng. In: 2014, pp. 1–10. url: $$Uhttps://lirias.kuleuven.be/retrieve/275254$$Dplp_sdd_paper.pdf%20[freely%20available] (cit. on p. 46).

[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. 20).

[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. 24).

[Wong *et al.* 2012] Lawson L.S. Wong, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. "Collision-free state estimation". In: *2012 IEEE International Conference on Robotics and Automation*. 2012, pp. 223–228. doi: 10.1109/ICRA.2012.6225309 (cit. on p. 1).

[Xu *et al.* 2018] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. "A semantic loss function for deep learning with symbolic knowledge". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 5502–5511. url: https://proceedings.mlr.press/v80/xu18h.html (cit. on p. 1).

[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. 24).

[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. 10).

[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 pp. 9, 51).

[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).

[H. Zhao, Poupart, *et al.* 2016] Han Zhao, Pascal Poupart, and Geoffrey J Gordon. "A unified approach for learning the parameters of sum-product networks". In: *Advances in Neural Information Processing Systems*. NeurIPS. 2016 (cit. on pp. 48, 50).

[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. 25).