STRUDEL: A Fast and Accurate Learner of Structured-Decomposable Probabilistic Circuits

Meihua Dang, Antonio Vergari, Guy Van den Broeck

{*mhdang, aver, guyvdb*}*@cs.ucla.edu*
Computer Science Department, University of California, Los Angeles

Abstract

Probabilistic circuits (PCs) represent a probability distribution as a computational graph. Enforcing structural properties on these graphs guarantees that several inference scenarios become tractable. Among these properties, structured decomposability is a particularly appealing one: it enables the efficient and exact computations of the probability of complex logical formulas, and can be used to reason about the expected output of certain predictive models under missing data. This paper proposes STRUDEL, a simple, fast and accurate learning algorithm for structured-decomposable PCs. Compared to prior work for learning structured-decomposable PCs, STRUDEL delivers more accurate single PC models in fewer iterations, and dramatically scales learning when building ensembles of PCs. It achieves this scalability by exploiting another structural property of PCs, called determinism, and by sharing the same computational graph across mixture components. We show these advantages on standard density estimation benchmarks and challenging inference scenarios.

Keywords: Probabilistic circuits, structure learning, structured decomposability

1. Introduction

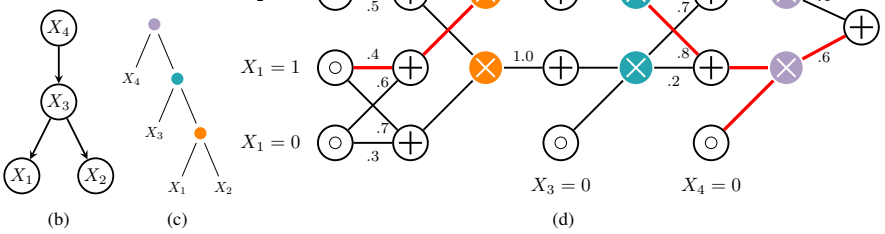
In several real-world scenarios, decision making requires *advanced* probabilistic reasoning, i.e., the ability to answer *complex probabilistic queries* [1]. Consider, for instance, querying a generative model for the probability of events described as logical constraints [2], e.g., rankings of user preferences [3, 4]; or the probability of Bayesian classifiers agreeing on their prediction [5]; or to conform to human expectations [6, 7]. Answering these queries goes beyond the capabilities of intractable probabilistic models like classical Bayesian networks (BNs) and more recent neural estimators such as variational autoencoders (VAEs) and normalizing flows [8]. Moreover, in many sensitive domains like healthcare and finance, the result of these queries is required (i) to be *exact*, as approximations without guarantees would make the decision making process brittle, and (ii) to be provided in a *limited amount of time*. This explains the recently growing interest around tractable probabilistic models (TPMs) which guarantee both (i) and (ii) by design.

Probabilistic circuits (PCs) [9, 1] propose a unifying framework to abstract from the myriad of different TPM representations. Among these, arithmetic circuits [10], probabilistic sentential decision diagrams [11], sum-product networks [12], and cutset networks [13] naturally fit under the umbrella of PCs. Classical bounded-treewidth graphical models [14] and their mixtures [15] are easily cast into a PC. Within the framework of PCs, one can reason about the tractable inference capabilities of a model via the structural properties of its computational graph. In turn, this enables learning routines that, by enforcing such specific structural properties, deliver PCs guaranteeing tractable inference for the desired classes of queries.

The *structured decomposability* [11] property of PCs enables the largest class of tractable inference scenarios. Indeed, all the advanced probabilistic queries we mentioned in the introductory paragraph can be exactly and efficiently answered using structured-decomposable PCs. In a nutshell, a structured-decomposable PC encodes a probability distribution in a computational graph by recursively decomposing it into smaller distributions according to a hierarchical partitioning of the random variables, also called *vtree*. However, while inference on structured-decomposable PCs has been extensively studied [2, 3, 16, 17], relatively little attention has gone to *learning* these circuits from data. The

$\mathsf{p}(X_i = 1 \mid X_{\mathrm{Pa}_i})$	$X_{\mathrm{Pa}_i} = 0$	$X_{\mathrm{Pa}_i} = 1$
$\mathsf{p}(X_4 = 1)$	0.4	0.4
$\mathsf{p}(X_3 = 1 \mid X_4)$	0.8	0.3
$\mathsf{p}(X_2 = 1 \mid X_3)$	0.5	0.9
$\mathsf{p}(X_1 = 1 \mid X_3)$	0.7	0.4

(a)



(b) (c) (d)

Figure 1: ***A structural decomposable PC and its corresponding vtree and equivalent BN.*** The CLT over RVs $\boldsymbol{X} = \{X_1, X_2, X_3, X_4\}$ in (1b) with CPTs in (1a) is compiled into the structured-decomposable PC in (1d) whose extracted vtree is shown in (1c). In (1d), $\odot$ are input distributions, $\bigotimes$ are products, and $\bigoplus$ are sums. Each product node is colored as its corresponding vtree node in (1c). All the edges in red are "active" in the circuit flow for the input configuration $\{X_1 = 1, X_2 = 0, X_3 = 1, X_4 = 0\}$. The feed forward computational order is from left to right, and the root node is on the right.

only prior work fully tailored towards structured-decomposable PCs is LearnPsdd [18]. LearnPsdd first introduces the task of learning a vtree, and starts from a fully-factorized circuit normalized for the vtree. Furthermore, LearnPsdd performs a local search: it evaluates many candidate PC structures and computes for each of them their penalized likelihood scores. This learning procedure is costly and prevents PCs from scaling to larger real-world datasets. Ideas on how to possibly design alternative score-based learners for structured-decomposable PCs are discussed in [19].

In this paper we introduce Strudel, a simpler and faster way to learn structured-decomposable PCs. Specifically, we do not perform vtree learning and instead initialize the PC structure in Strudel using the best TPM that can be learned with guarantees. Moreover, Strudel drastically simplifies the search by not computing a likelihood score for each candidate structure but greedily growing the circuit. It considerably speeds up learning while still delivering accurate PCs. This is even more relevant when learning large mixtures of PCs; here we propose to scale even further by mixing components that share the same structure, and exploiting CPU/GPU parallelism. We demonstrate these performance gains on 20 standard benchmarks and on the more challenging task of computing the expected predictions of regression models in the context of missing data [7].

The rest of paper is organized as follows. Sections 2 and 3 introduce the framework of PCs for tractable probabilistic inference. Sections 4 and 5 describe Strudel for learning single PCs and mixtures thereof. Then, Section 6 discusses implementation details on how to parallelize the computation on both CPU and GPU. Lastly, Section 7 discusses our experimental results.

2. Probabilistic Circuits

2.1. Representation

Recently, the great interest in tractable probabilistic modeling propelled the introduction of a multitude of representations. Many of these representations can be understood under a unifying computational framework, which we refer to as *probabilistic circuits* (PCs) [9, 1]. PCs reconcile and abstract from the different graphical and syntactic representations of recently introduced TPM formalisms such as arithmetic circuits [10], probabilistic sentential decision diagrams (PSDDs) [11], sum-product networks (SPNs) [12], and cutset networks [13]. In contrast to the intractable probabilistic models such as VAEs, PCs enable reasoning about the tractable inference scenarios they support. Their answers are guaranteed to be exact, and for many queries, inference runs in time linear in the size of the circuit, long as it satisfies certain *structural properties*.

Notation. We use upper-case letters for random variables (RVs), e.g., X, Y, and lowercase ones for their assignments e.g., x, y. Analogously, sets of RVs are denoted by upper-case bold letters, e.g., $\boldsymbol{X}$, $\boldsymbol{Y}$, and their joint values by the corresponding lower-case ones, e.g., $\boldsymbol{x}$, $\boldsymbol{y}$. Here we consider discrete RVs, specifically represented as Boolean variables, i.e., having values in $\{0, 1\}$.

Representation. A probabilistic circuit (PC) C over RVs $\boldsymbol{X}$ is a pair $(\mathcal{G}, \boldsymbol{\theta})$ where $\mathcal{G}$ is a directed acyclic graph (DAG) representing a computational graph, also called the *circuit structure*; and $\boldsymbol{\theta}$ are the *circuit parameters*. The PC C encodes a probability distribution $\mathsf{p}_C(\boldsymbol{X})$ in a recursive manner.

From the perspective of a DAG, $\mathcal{G}$ has three kinds of nodes: *input distributions* (leaves), *product* nodes and *sum* nodes. Figure 1d shows an example of a PC. Each node $n \in \mathcal{G}$ encodes a distribution p_n, defined as follows. An input distribution n encodes a tractable probability distribution p_n over some RVs $\phi(n) \subseteq \boldsymbol{X}$, where ϕ is called the *scope*. In this work targeting Boolean RVs, we consider univariate input distributions, specifically leaves for RV $X_i \in \boldsymbol{X}$ will be indicator functions of the form $\mathsf{p}(X_i = 1) = [\![X_i = 1]\!]$. A product node n defines the factorized distribution $\mathsf{p}_n(\boldsymbol{X}) = \prod_{c \in \mathsf{ch}(n)} \mathsf{p}_c(\boldsymbol{X})$ over its children $\mathsf{ch}(n)$. Without loss of generality, we will consider product nodes to have only two children since product nodes with multiple children are semantically equivalent with and can be tractably transformed into two children representations. A sum node n defines a mixture model $\mathsf{p}_n(\boldsymbol{X}) = \sum_{c \in \mathsf{ch}(n)} \theta_{n,c} \mathsf{p}_c(\boldsymbol{X})$ parameterized by edge weights $\theta_{n,c}$. The scope of a product or sum node is the union of the scopes of its children: $\phi(n) = \cup_{c \in \mathsf{ch}(n)} \phi(c)$. Thus $\boldsymbol{\theta}$ is the set of all sum weights $\theta_{n,c}$ denoting all the parameters in a circuit C (parameters are only attached to sum node edges, since input distributions are indicators). From the perspective of a computational graph $\mathcal{G}$, nodes are computational units, specifically input distribution units, product units and sum units and edges define an order of execution. Let $\mathsf{in}(n) = \mathsf{ch}(n)$ be the set of inputs of an inner node n and $\mathsf{out}(n)$ the output. The feedforward evaluation of a PC C following $\mathsf{p}_n(\boldsymbol{X})$ defined above computes $\mathsf{p}_r(\boldsymbol{X})$, with r as the circuit's root, to be the output of this PC, namely $\mathsf{p}_C(\boldsymbol{X})$.

PCs are not PGMs. Even if PCs are *probabilistic* models expressed via a *graphical* formalism, probabilistic circuits are *not* classical PGMs. The clear *operational* semantics we described above makes PCs peculiar neural networks [20, 21] whose inner units are either products (acting as non-linearities) or convex combinations of their inputs. Overall, a PC C defines a multilinear polynomial [10] whose indeterminates are the distributions equipping the leaves of C.

2.2. Inference

EVI. Any PC C over RVs $\boldsymbol{X}$ that represents a normalized distribution supports computing the likelihood $\mathsf{p}_C(\boldsymbol{x})$ given a *complete configuration* $\boldsymbol{x}$ (a complete evidence query, EVI) by evaluating the circuit feed forward: starting from the input distributions and computing the output of children before parents.

Additional structural properties of the PC, such as *decomposability*, *smoothness* and *determinism*, can extend the set of probabilistic queries that are guaranteed to be answered exactly and in time linear in the *size* of the PC, that is, its number of edges. Then, query answering reduces to traversing the PC feed forward and backward given values for the leaf nodes.

MAR and CON. A PC is *decomposable* if for every product node, the children have disjoint scopes. That is, product nodes encode well-defined factorized probability distributions. A PC is *smooth* if for every sum node, the children have the same scope. That is, sum nodes encode mixtures of distributions that are well-defined over identical sets of RVs. Smooth and decomposable PCs enable linear time computation of any marginal query (MAR) [10]. This also implies linear time computation of conditional probabilities (CON), which are ratios of marginals. SPNs [12] are examples of smooth and decomposable PCs.

MAP. A circuit is *deterministic* if for every sum node n and complete assignment $\boldsymbol{x}$, at most one of the children of n have a non-zero output. That is, a deterministic sum defines a mixture model whose components have disjoint support. Smoothness, decomposability and determinism enable tractable maximum a posterior queries (MAP) [22]. [1] Examples of smooth, decomposable and deterministic PCs are cutset networks [13, 24] and selective SPNs [25].

2.3. Structured-Decomposable PCs

More recently, the stronger property of *structured decomposability* has been introduced to enable a larger class of tractable inference scenarios [26, 11]. Briefly, the product nodes in a structured-decomposable PC cannot decompose in arbitrary ways, but must agree on a "contract." A PC is structured-decomposable if it is *normalized* for a *vtree*, a binary tree encoding a hierarchical decomposition of RVs. Each leaf in a vtree denotes a RV, while an internal node

[1] We adopt the terminology of [14] and [9]: our MAP queries are also called *most probable explanation (MPE)* queries in [23], and what is called MAP there, which involves marginalizing over a set of random variables before maximising, corresponds to marginal MAP in our setting.

indicates how to decompose a set of RVs in two subsets mapping to its left and right branch. A PC is normalized for a vtree if the scope of every product node decomposes over its children as its corresponding node in the vtree. An example of a vtree and a structured-decomposable PC normalized for it are shown in Figure 1c and Figure 1d, where each product node in Figure 1c is colored as its corresponding vtree node as in Figure 1d.

AND-OR graphs [27] and PSDDs [11] are examples of smooth, deterministic and structured-decomposable PCs.[2] Despite all these advanced inference scenarios that structured-decomposable PCs enable, relatively little attention has been dedicated to learning these circuits from data, and the only attempt so far is difficult to scale [18].

By enforcing structured decomposability, several classes of advanced probabilistic queries become computable exactly and efficiently. For instance, structured-decomposable PCs allow to compute symmetric and group queries [2] and, given certain constrained vtrees, same-decision probabilities [28], their expected version [29] and classifier agreement [5]. Moreover, if two PCs conform to the same vtree, it is possible to efficiently compute the KL divergence between them [17] or to multiply them [16]. Besides, it becomes possible to also compute the *expected predictions* of a discriminative model—a classifier or a regressor—in a tractable manner with respect to the input distribution modeled by a generative model if both are circuits conforming to the same vtree and if the discriminative model defines its predictions in a certain way [7, 30]. We showcase this advanced inference scenario in Section 7, where we employ *regression circuits* [7] as discriminative models and structured-decomposable PCs as generative ones. Regression circuits are deep regressors comprising a structured-decomposable circuit that acts as a feature extractor and a linear regressor that is learned on top of the circuit extracted features. Crucial to our purposed, we can build a regression circuit that conforms to the same vtree of a PC we have learned with our structure learning algorithm.

3. Circuit Flows: Fast Inference and Parameter Learning

Before explaining STRUDEL, we briefly introduce *circuit flows* – a computational tool that allows us to scale up our learner. Determinism not only makes MAP inference tractable, but also enables closed-form parameter estimation in PCs [11] and dramatically speeds up inference by leveraging circuit flows [31].

3.1. Definition and Computation

We first introduce *context* and then formally define *circuit flows*.

Definition 1 (Context). *Let C be a PC over RVs $\boldsymbol{X}$ and n be one of its nodes. The* context γ_n *of node n denotes all joint assignments that return a nonzero value for all nodes in a path between the root of C and n.*

$$\gamma_n := \bigcup_{p \in \mathsf{pa}(n)} \gamma_p \cap \mathsf{supp}(n)$$

where $\mathsf{pa}(n)$ *refers to the parent nodes of n and* $\mathsf{supp}(n) := \{\boldsymbol{x} : \mathsf{p}_n(\boldsymbol{x}) > 0\}$ *is the support of node n.*

Note that the context of a node is different from its support. Even if the node returns a non-zero value for some input, its output may be multiplied by 0 at its ancestor nodes; i.e., such node does not contribute to the circuit output of that assignment.

We can now express circuit flows in terms of contexts. Intuitively, the context of a circuit node is the set of all complete inputs that "activate" the node. Hence, an edge is "activated" by an input if it is in the contexts of both nodes for that edge.

Definition 2 (Circuit flow). *Let C be a PC over variables $\boldsymbol{X}$, (n, c) its edge, and $\boldsymbol{x}$ a joint assignment to $\boldsymbol{X}$. The circuit flow of (n, c) given $\boldsymbol{x}$ is*

$$f_C(n, c; \boldsymbol{x}) = [\boldsymbol{x} \in \gamma_n \cap \gamma_c]. \tag{1}$$

[2] In their original formulation [11], PSDDs required a stronger notion of determinism, which is related to its logical constraints well has no practical implication for tractable probabilistic inference.

Algorithm 1: circuitFlows($C, \boldsymbol{x}$)

Input : PC C, one data sample $\boldsymbol{x}$
Output : circuit flows of sample $\boldsymbol{x}$ for each node n and edge c, cached in f

```
// visit children before parents, compute support s
foreach n ∈ C do
    if n is a leaf then
        X_i ← literal that n represent
        s(n) ← [[X_i = x_i]]
    else if n is a sum node then
        s(n) ← ⋁_{c∈ch(n)} s(c)
    else
        s(n) ← ⋀_{c∈ch(n)} s(c)
// visit parents before children, compute circuit flows
foreach n ∈ C do
    if n is root then
        f_C(n, c; x) ← s(n) for every c ∈ ch(n)
    else if n is a sum node then
        f_C(n, c; x) ← s(c) ∧ ⋁_{p∈pa(n)} f_C(p, n; x) for every c ∈ ch(n)
    else if n is a product node then
        f_C(n, c; x) ← ⋁_{p∈pa(n)} f_C(p, n; x) for every c ∈ ch(n)
```

Note that the determinism property guarantees that for every sum node, at most one input has a flow of 1, and the rest has a flow of 0. Figure 1d shows an example of a circuit flow, all the edges in red are "active" for input configuration $\{X_1 = 1, X_2 = 0, X_3 = 1, X_4 = 0\}$. We can compute the circuit flows via feed forward evaluation (to compute the support) followed by a backward pass as shown in Algorithm 1. We cache intermediate results to avoid redundant computations and to ensure a linear-time evaluation. To compute the circuit flows on a dataset, we can compute the flow of each data sample in parallel via vectorization.

Intuitively, a circuit flow encodes which parameters are activated by different input configurations. As such, circuit flows characterize how a particular complete assignment $\boldsymbol{x}$ propagates through the circuit and they represent a binary encoding of a tree circuit. An analogous concept has been introduced in the literature of non-deterministic circuits such as sum-product networks under the name of induced sub-circuit [32, 33]. Note that the semantic of circuit flows and induced sub-circuits differ in that to materialize the latter we need to set evidence over the latent variables that are associated to the sum units of the considered circuit, thus making it "temporarily deterministic". Induced sub-circuits have been used as a metaphor to understand non-deterministic circuits as mixture models with exponentially many components, linking them to the notion of network polynomials [34, 35]. While this intermediate representation has been helpful in inspiring novel parameter learning schemes for non-deterministic circuits such as sum-product networks [36, 37], their concrete connection to the likelihood computation for a deterministic circuit has not been investigated. We are the first to do that as shown next.

3.2. Fast Inference

For a given sample $\boldsymbol{x}$, a circuit flow acts as a mapping $f_C : \boldsymbol{X} \mapsto \{0, 1\}^{|\boldsymbol{\theta}|}$ from sample $\boldsymbol{x}$ to a binary vector, called *flow embedding*, with as many entries as there are parameters in C. The k-th entry in $f_C(\boldsymbol{x})$, which also defines the *flow* at edge e associated with the k-th parameter in $\boldsymbol{\theta}$, is 1 if sample $\boldsymbol{x}$ *flows* through the edge e reaching the output, and 0 otherwise.

As such, the log-likelihood $\mathcal{LL}_C(\boldsymbol{\theta}; \boldsymbol{x})$ of a deterministic circuit C parameterized by $\boldsymbol{\theta}$, given a single input configuration $\boldsymbol{x}$, is efficiently computed as

$$\mathcal{LL}_C(\boldsymbol{\theta}; \boldsymbol{x}) = \log(\mathsf{p}_C(\boldsymbol{x})) = f_C(\boldsymbol{x})^T \cdot \log(\boldsymbol{\theta}).$$

Similarly, the circuit flow of a batch of samples $\mathcal{D}$ can be represented as a binary matrix $\mathbf{F}_C(\mathcal{D}) \in \{0,1\}^{|\mathcal{D}|\times|\boldsymbol{\theta}|}$. Then, the log-likelihood given the entire batch at once is efficiently vectorized as

$$\mathcal{LL}_C(\boldsymbol{\theta}; \mathcal{D}) = \mathbf{F}_C(\mathcal{D}) \cdot \log(\boldsymbol{\theta}).$$

This formulation of the likelihood has the clear benefit that a vectorized computation can yield significant speedups. Furthermore, the matrix $\mathbf{F}_C(\mathcal{D})$ can be computed by propagating bit-vectors forward and backward through the circuit C. Recall that flow embeddings are binary vectors by definition, and they can be used to compute the log-likelihoods because of determinism property. Bit-vector arithmetic is extremely efficient, much more so than using floating-point vectors to compute the same likelihoods. Lastly, circuit flows will greatly benefit inference in our large ensembles sharing the same structure (cf. Section 5). Since a flow only depends on the circuit structure, for a mixture $\mathcal{M} = \{C_i\}_{i=1}^k$ of k PCs sharing the same structure, we need to evaluate a single flow $f_{\mathcal{M}}$ once. As such, the log-likelihood of $\mathcal{M}$ given $\boldsymbol{x}$ can be efficiently computed as:

$$\mathcal{LL}_{\mathcal{M}}(\boldsymbol{\Theta}; \boldsymbol{x}) = \mathsf{logsumexp}(f_{\mathcal{M}}(\boldsymbol{x})^T \cdot \log(\boldsymbol{\Theta}) + \log(\boldsymbol{w})), \tag{2}$$

where $\boldsymbol{w} = \{w_i\}_{i=1}^k$ are the mixture weights, $\boldsymbol{\Theta}$ is the matrix whose columns are the parameters $\boldsymbol{\theta}_i$ of the ith PC in the mixture, and logsumexp sums probabilities over all mixture components in their logarithmic representation. We empirically show these speedups in Section 7 and Appendix D.

3.3. Parameter Learning

More instrumental to our purpose, flow embeddings can be used for parameter learning of a PC C. We define the *aggregate flow* $a_C(i, j; \mathcal{D})$ of one edge $e_{i,j}$ associated with the kth weight $\theta_{i,j}$, as the total number of configurations in the dataset $\mathcal{D}$ that flow through edge $e_{i,j}$, and whose likelihood therefore contains the kth weight $\theta_{i,j}$ as a factor. That is, $a_C(i, j; \mathcal{D}) = \sum_{h=1}^{|\mathcal{D}|} \mathbf{F}_C(\mathcal{D})[h, k]$. Now, the maximum likelihood estimator (MLE) of weight $\theta_{i,j}$ can be computed in closed form as the ratio

$$\theta_{i,j}^{\mathsf{MLE}} = \frac{a_C(i, j; \mathcal{D})}{\sum_* a_C(i, *; \mathcal{D})}. \tag{3}$$

In other words, $\theta_{i,j}^{\mathsf{MLE}}$ can be computed as the ratio of the number of samples in $\mathcal{D}$ flowing through edge $e_{i,j}$ over the total number of samples flowing through node i.

In summary, the circuit flow formulation has the benefit of (1) vectorizing the computation, (2) allowing for a single computation of the flow embeddings to be reused across PCs with the same structure but different parameters, for example in large ensembles, and (3) yielding simple closed-form parameter estimates. Moreover, we will show in Section 4 that flows can act as one of the building blocks in structure learning. Thus, both our learner for expressive single models in Section 4 and for scalable ensembles in Section 5 will make use of this efficient formulation.

4. Strudel: Learning Structured-Decomposable Probabilistic Circuits

The objective of structure learning for PCs is to find a circuit structure and parameters that approximate well the data distribution. If the learned PC has to guarantee tractable inference for certain classes of queries, its structure has to enforce the corresponding properties discussed in Section 2. For the advanced inference scenarios we are interested in, and to retain efficient parameter learning (cf. Section 3), we require structured decomposability and determinism. So far, the only alternative learner to deliver such PCs is LearnPsdd [18], while ideas to develop alternative score-based learners are discussed in [19]. Since the development of Strudel, several new learners were proposed that target structured-decomposable PCs [38, 39, 40], as well as novel regularization techniques for large PCs [41, 42].

In this section, we first propose Strudel, a STRUctured-DEcomposable Learner, and then explain how it compares against LearnPsdd [18].

Briefly, Strudel starts from the best tree shaped Bayesian Network learned from data – *Chow-Liu trees*. And then it performs a greedy search or beam search over the space of possible structured-decomposable PCs iteratively to optimize the circuit structures and parameters. Specifically, at each iteration we perform a local modification on circuit structures, which is guided by some heuristic scores calculated from data.

Algorithm 2: getVtree($\mathcal{T}, opt$)

Input : a CLT $\mathcal{T}$, option opt decides how to organize a set of sub-vtrees from a set of children into a vtree
Output : vtree $\mathcal{V}$

1 $X \leftarrow$ root node in $\mathcal{T}$
2 **if** *X is a leaf node* **then**
3 **return** makeVtreeLeafNode(X)
4 **else**
5 $v_l \leftarrow$ makeVtreeLeafNode(X)
6 $vs \leftarrow$ getVtree(X_c, opt) for every $X_c \in$ ch(X)
7 $v_r \leftarrow$ a binary tree with vs as its leaves built according to option opt
8 **return** makeVtreeInnerNode(v_l, v_r)

4.1. From Chow-Liu Trees to Structured-Decomposable PCs

Chow-Liu trees. Providing a good initial candidate structure to a structure learning algorithm is crucial, as it might save considerable time during search. We use *Chow-Liu trees* (CLTs) as the "best" initial PC structure possible as they are tree-shaped BNs that: (1) guarantee to encode the best tree model in terms of KL divergence with the data distribution; (2) support linear time marginals and MAP inference; (3) and can be learned in time $O(|\boldsymbol{X}|^2|\mathcal{D}|)$. Furthermore, as we will show next, we can quickly compile CLTs into smooth, deterministic and structured-decomposable PCs and extract a vtree from them.

More formally, a CLT $\mathcal{T}$ over RVs $\boldsymbol{X}$ is a tree-shaped BN equipped with parameters $\theta_{i|\mathrm{Pa}_i}$ defining the conditional probability table (CPT) of node i associated to RV X_i with parent node Pa_i. An example of a CLT is shown in Figure 1b. The classic Chow-Liu algorithm [43] learns a CLT $\mathcal{T}$ from data $\mathcal{D}$ by running a maximum spanning tree algorithm over a complete graph induced by the pairwise mutual information (MI) matrix over variables $\boldsymbol{X}$ as estimated from data $\mathcal{D}$. These MI estimates are used to compute the $\theta_{i|\mathrm{Pa}_i}$ parameters, and can be smoothed by adding a Laplace correction factor α. See Appendix A for a detailed algorithm.

Compiling CLTs. We now turn our attention to compiling a CLT into a structured-decomposable PC and distilling a corresponding vtree from it. Compiling generic BNs into smooth, deterministic and decomposable PCs[3] has been extensively researched in the literature [10], and compilation of a BN into a structured-decomposable circuit has also been explored [44, 16, 45]. However, compiling a BN (even a CLT) for an arbitrary vtree can lead to an exponentially larger PC. Therefore, we adopt a simple strategy tailored for CLTs that extracts a vtree guaranteeing a linear-size PC in the number of RVs.

We start from the observation that a *rooted* CLT provides a natural variable decomposition. While rooting the CLT can be done arbitrarily, we root it at its *Jordan center* as a heuristics to minimize the resulting vtree depth and thus yielding smaller PCs. Then we traverse the CLT top-down to build the vtree as shown in Algorithm 2, for each node $X_i \in \mathcal{T}$, if X_i is a leaf node, makeVtreeLeafNode compiles it to a vtree leaf node v_i containing variable X_i; otherwise makeVtreeInnerNode builds an inner node with v_i as its one branch, and the vtree for its children ch(X_i) as its other branch. To turn a set of children ch(X_i), which is conditionally independent given their parent X_i, into a vtree, we compile each child as a sub-vtree separately (line 6) and then make a binary vtree where each leaf is one sub-vtree, with option $opts$ decides how to organize a set of sub-vtrees (line 7). Figure 2 shows an example of all possible vtrees built from the given CLT/BN: $X_1 \rightarrow X_i$ for every $i \in \{2, 3, 4, 5, 6\}$ ignoring variable imputations and left-right branch symmetries. It also illustrates how to turn a set of sub-vtrees normalized for conditionally independent variables $X_2, ..., X_6$ given X_1 into a sub-vtree. opt decides how to organize a set of sub-vtrees, if we want the resulting PC to be *shallow* then the vtree is more *balanced* as in (2b) and (2c); otherwise, if we want the resulting PC to be *deep*, then the vtree has a more *linear* shape, such as in (2d).

After a vtree is fully grown, we proceed compiling the CLT bottom-up as shown in Algorithm 3. During compilation, caching the previously compiled sub-circuits (line 1) guarantees that we obtain a PC of linear size [10]. For every node $X_i \in \mathcal{T}$ that we visit, and for every parent configuration, we introduce sum nodes selecting a value of X_i

[3] Specifically to Arithmetic Circuits [10] represented as DAGs having parameters attached to leaf nodes.

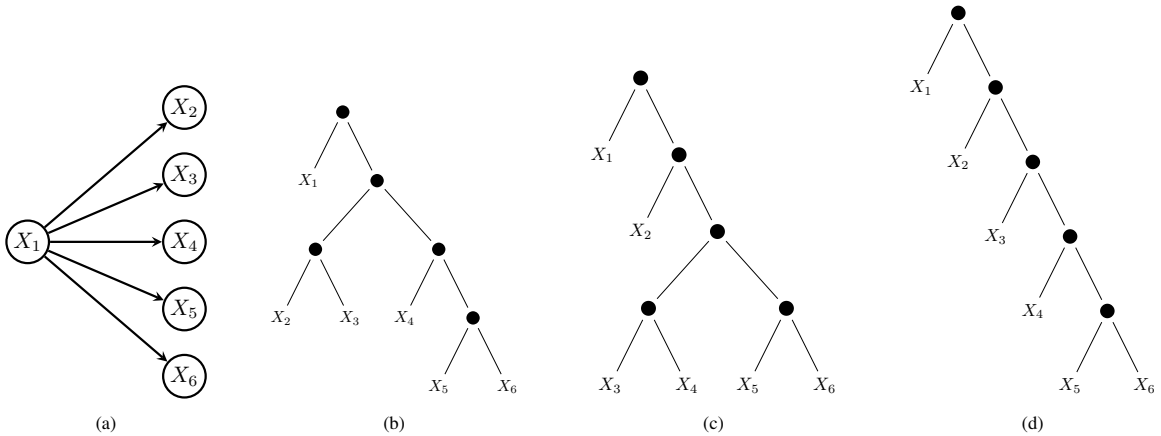


Figure 2: ***All possible vtrees built from given CLT ignoring all variable imputations and left-right branch symmetries.*** The CLT over RVs $\boldsymbol{X} = \{X_1, X_2, X_3, X_4, X_5, X_6\}$ in (2a) and its vtrees in (2b, 2c, 2d). *opt* is *balanced* for (2b) and (2c), and *linear* for (2d).

Algorithm 3: compileCLT($\mathcal{T}, \mathcal{V}$)

Input : a CLT $\mathcal{T}$ and vtree $\mathcal{V}$
Output : compiled PC $\mathcal{C}$

1 $v2n(v) \leftarrow \emptyset$ for every $v \in \mathcal{V}$
2 `// iterate children before parents`
3 **foreach** $v \in \mathcal{V}$ **do**
4 **if** *v is a leaf node* **then**
5 $X \leftarrow \mathsf{RV}(v)$
6 $n_+, n_- \leftarrow \mathsf{makeLiteralNodes}(X)$
7 **if** *X is leaf node in CLT* $\mathcal{T}$ **then**
8 $v2n(v) \leftarrow \mathsf{sumNodes}([n_+, n_-], X)$
9 **else**
10 $v2n(v) \leftarrow [n_+, n_-]$
11 **else**
12 $X_l \leftarrow \mathsf{RV}(\mathsf{leftMostDescendent}(\mathsf{leftBranch}(v)))$
13 $X_r \leftarrow \mathsf{RV}(\mathsf{leftMostDescendent}(\mathsf{rightBranch}(v)))$
14 $ns_l \leftarrow v2n(\mathsf{leftBranch}(v))$
15 $ns_r \leftarrow v2n(\mathsf{rightBranch}(v))$
16 $products \leftarrow n_l \bigotimes n_r$ for $n_l, n_r \in \mathsf{zip}(ns_l, ns_r)$
17 **if** X_{Pa_l} *is* X_{Pa_r} **then**
18 $v2n(v) \leftarrow \bigoplus(n)$ for $n \in products$
19 **else if** X_l *is* X_{Pa_r} **then**
20 $v2n(v) \leftarrow \mathsf{sumNodes}(products, X_l)$
21 **else**
22 $\mathsf{error}(\text{Vtree } \mathcal{V} \text{ is not valid})$
23 **return** $v2n(r)[0]$ where r is root of vtree $\mathcal{V}$

with edge weights $\mathsf{p}(X_i|X_{\mathrm{Pa}_i} = x_{\mathrm{Pa}_i})$ (line 17 $\sim$ 22), i.e., its distribution conditioned on the parent configuration. A corresponding indicator leaf, following the vtree structure, is introduced in the product (line 16). This yields a smooth deterministic sum node branching over the possible values for the considered RV. Figure 1d illustrates the compiled PC for the example CLT in Figure 1b and a step-by-step compilation progress is in Appendix A. The detailed pseudo-

code is listed in Algorithm 3, where makeLiteralNodes, $\bigotimes$, and $\bigoplus$ refer to building leaves, product nodes and sum nodes given a random variable or inputs. Subroutine sumNodes returns two sum nodes branching over products and indicator leaves, which is listed in Appendix A.

4.2. How And What to Split

From this initial circuit, STRUDEL applies the *split* operation at each step, performing a *greedy search*.

Our split operation is based on the one proposed in LEARNPSDD [18], which builds on [46]. Given a PC structure C^t at iteration t, we create C^{t+1} in a *two-step procedure*. First, we select one *edge* $e_{n,c}$ to split, from one sum node n to one of its child product nodes c, and one *variable* X_i to split in the scope of c. Second, we make two *partial copies* of sub-circuits rooted at c conditioned on $[\![X_i = 0]\!]$ and $[\![X_i = 1]\!]$ respectively. These partial copies are carried out by copying nodes up to a certain depth bound. We thus create C^{t+1} by removing child c from node n in C^t, while adding both of the new copies. This splitting operation preserves smoothness and determinism since sums are conditioned on the RV X_i, while it also preserves structured decomposability since the new circuit confroms to the same vtree as the old one. The pseudo-code is listed in Algorithm 4, in which to perform the construction of the split subcircuits as described above, we use the subroutine conjoin to conjoin the PC's logical formula with given literal constraints, and partialCopy to create a copy of the circuit up to a certain depth (see Appendix B).

Algorithm 4: splitOperation($C, e_{n,c}, X$)

Input : a PC C, edge $e_{n,c}$ between sum node n and product node c, a random variable $X \in \phi(c)$
Output : PC splited on edge $e_{n,c}$ given variable X

$C_c \leftarrow$ sub-circuit rooted at node c
$C_{c+} \leftarrow \mathsf{partialCopy}(\bigodot(X = 1) \bigotimes \mathsf{conjoin}(C_c, X = 1))$
$C_{c-} \leftarrow \mathsf{partialCopy}(\bigodot(X = 0) \bigotimes \mathsf{conjoin}(C_c, X = 0))$
remove edge $e_{n,c}$ from C
add edge $e_{n,c+}$ and edge $e_{n,c-}$ to C
return C

Selecting edges to split. To select an edge and variable to split, the simplest but uninformed strategy would be to pick one edge and one variable randomly. We name these two strategies ERAND and vRAND. Clearly, more informed heuristics, but less expensive than the computation of the likelihood, would benefit search. We introduce two novel heuristics, EFLOW and vMI for selecting an edge and a variable respectively.

To select an edge to split, we are implicitly selecting an sub-circuit, therefore we prefer the sub-circuit responsible for most of the samples such that we can substitute it with a larger circuits containing more parameters, which can better fit that multitude of data points. Specifically, EFLOW selects edge $e_{i,j}$ which maximizes the aggregate circuit flow (cf. Section 3):

$$\mathsf{score}_{\mathsf{eFLOW}}(e_{i,j}; C^t, \mathcal{D}) := a_{C^t}(i, j; \mathcal{D}) \tag{4}$$

That is, the EFLOW picks the edges where more samples in $\mathcal{D}$ flow through, indicating that introducing a new sub-circuit there could potentially better model the distribution over those samples.

Selecting RVs to split. Once we pick edge $e_{i,j}$, we then select the RV among those in the scope of node j. Specifically our vMI heuristics selects the RV X_k sharing more dependencies with the others in the scope. That is, we maximize the score:

$$\mathsf{score}_{\mathsf{vMI}}(X_k; C^t, \mathcal{D}) := \sum\nolimits_{X_h \neq X_k} \overline{\mathsf{MI}}(X_h; X_k) \tag{5}$$

where $\overline{\mathsf{MI}}$ is the pairwise mutual information estimated on the samples of $\mathcal{D}$ "flowing" through edge $e_{i,j}$. By introducing new parameters for highly dependent RVs we can learn more accurate PCs. The entire greedy local search loop performed by STRUDEL is summarized in Algorithm 5.

4.3. From Greedy Search to Beam Search

Issues in greedy search. The greedy search approach we just introduced is efficient. This is because, at each training step, we only calculate the simple heuristic scores and perform the split operation once to find the best next

Algorithm 5: STRUDEL($\mathcal{D}, \boldsymbol{X}$)

Input : a dataset $\mathcal{D}$ over RVs $\boldsymbol{X}$
Output : a structured-decomposable PC C

$\mathcal{T} \leftarrow \mathsf{LearnCLT}(\mathcal{D}, \mathbf{X})$
$\mathcal{V} \leftarrow \mathsf{getVTree}(\mathcal{T})$
$C \leftarrow \mathsf{compile}(\mathcal{T}, \mathcal{V})$
while C *is not overfitting* **do**
  $e^*_{i,j} \leftarrow \text{argmax}_{e_{i,j} \in \mathsf{edges}(C)} \mathsf{score}_{\mathsf{eFLOW}}(e_{i,j}; C, \mathcal{D})$
  $X^* \leftarrow \text{argmax}_{X_k \in \phi(C_i)} \mathsf{score}_{\mathsf{vMI}}(X_k; C, \mathcal{D})$
  $C \leftarrow \mathsf{SplitOperation}(C, e^*_{i,j}, X^*; \mathcal{D})$
return C

circuit. However, by being more greedy, it is possible that our simple heuristics perform a split operation at a certain iteration that can lead to a less optimal circuit structure in the long run, without any possibility to "go back" and fix it. To circumvent this possible issue, we introduce STRUDELBEAM, where we modify the structure learning loop of STRUDEL to perform beam search, i.e., allowing the evaluation of the likelihood of a small set of the most promising candidate structures.

Beam search. At each iteration during training, instead of getting one best circuit based on heuristics, we maintain β best circuits to apply the split operation to, where β is called *beam width*. If $\beta = 1$, beam search is reduced to greedy search; with an infinite β, it explores every possible circuit structures and is identical to breadth-first search (BFS).

Given a set of β PCs $\mathcal{B}^t = \{C^t_1, C^t_2, ...C^t_\beta\}$ at iteration t, we create $\mathcal{B}^{t+1}$ in the following way. First, for each PC C^t_i where $i \in \{1, 2, ...\beta\}$, we select from all possible next step PCs and pick the best β candidates with the top heuristic scores to form $\mathcal{B}^{t+1}_i$. Then we have $\bigcup_{i=1}^{\beta} \mathcal{B}^{t+1}_i$, which is a set with maximum size β^2. Note that $\mathcal{B}^{t+1}_i$ and $\mathcal{B}^{t+1}_j$ $(i \neq j)$ may have exactly the same PCs, for example, they may come from the same ancestor and split on the same edges and variables but with different sequences. Finally we need to pick the best β PCs from $\bigcup_{i=1}^{k} \mathcal{B}^{t+1}_i$. It is not reasonable to compare the heuristic scores directly as they come from different ancestors. Therefore, as a fair comparison, we evaluate the PCs' log-likelihoods (for upto β^2 PCs) from which we pick the top β to form $\mathcal{B}^{t+1}$.

The entire beam search algorithm pseudocode performed by STRUDEL is listed in Algorithm 6.

4.4. LEARNPSDD *And Its Limitations*

LEARNPSDD performs a local search over the space of possible structured-decomposable PCs, given a vtree as input. To learn a vtree from data, a hierarchical clustering step is performed over the RVs discovering some independence relationships: they are recursively grouped bottom up (or split top down) so as to maximize their pairwise mutual information. Next, local search starts from a fully-factorized PC, that is, one where all RVs are considered to be independent, reshaped to conform to the learned vtree. Each search iteration locally changes the circuit while preserving its semantics and structural properties of smoothness, determinism and structured decomposability. To propose candidates, LEARNPSDD consistently applies two structural transformations – *split* and *clone* – to all possible nodes in the circuits. These candidates are then ranked by their log-likelihood score, penalized by their circuit size.

We highlight the following shortcomings of LEARNPSDD: (i) vtree learning as a separate pre-processing step has a limited effect on structure learning, which starts from a fully-factorized distribution, discarding the dependencies discovered in vtree learning. Moreover, while circuit flows speed up likelihood computation in deterministic circuits, (ii) using likelihood to score candidate structures drastically slows down learning, especially in large data regimes. As a result, when employed in mixture models LEARNPSDD has not been able to scale beyond tens of components.

To overcome these shortcomings, we propose to: (i) extract a vtree structure from the best graphical model that can be learned in tractable time, and then compile it into a structured-decomposable PC, which provides a more informative starting point, (ii) dramatically reduce learning time by employing a greedier local/beam search using a single transformation, split, and (iii) effectively use circuit flows, CPU, and GPU parallelism to speed up parameter learning and likelihood computation. The resulting algorithm is a *simpler*, *faster* structure learning scheme, yet

Algorithm 6: StrudelBeamSearch($\mathcal{D}, \boldsymbol{X}, \beta$)

Input : a dataset $\mathcal{D}$ over RVs $\boldsymbol{X}$, beam width β
Output : a structured-decomposable PC $\mathcal{C}$

$\mathcal{T} \leftarrow \mathsf{LearnCLT}(\mathcal{D}, \mathbf{X})$
$\mathcal{V} \leftarrow \mathsf{getVTree}(\mathcal{T})$
$\mathcal{C} \leftarrow \mathsf{compileCLT}(\mathcal{T}, \mathcal{V})$
$\mathcal{B} \leftarrow \{\mathcal{C}\}$
while $\mathcal{C}$ *is not overfitting* **do**
  foreach $\mathcal{C} \in \mathcal{B}$ **do**
    $E \leftarrow \mathsf{edges}(\mathcal{C})$
    foreach $k \in 1 : \beta$ **do**
      $e^*_{i,j} \leftarrow \mathrm{argmax}_{e_{i,j} \in E} \mathsf{score}_{\mathsf{eFLOW}}(e_{i,j}; \mathcal{C}, \mathcal{D})$
      $E \leftarrow E - e_{i,k}$
      $X^* \leftarrow \mathrm{argmax}_{X_k \in \phi(\mathcal{C}_i)} \mathsf{score}_{\mathsf{vMI}}(X_k; \mathcal{C}, \mathcal{D})$
      $\mathcal{C} \leftarrow \mathsf{SplitOperation}(\mathcal{C}, e^*_{i,j}, X^*; \mathcal{D})$
      $\mathcal{B} \leftarrow \mathcal{B} + \mathcal{C}$
  $\mathcal{B} \leftarrow \mathsf{topk}(\mathcal{B}, k = \beta, by = \textbf{loglikelihood})$
  $\mathcal{C} \leftarrow \mathrm{argmax}_{\mathcal{C} \in \mathcal{B}}$ **loglikelihood**$(\mathcal{C})$
return $\mathcal{C}$

yielding competitively *accurate* PCs and enabling fast learning of large mixtures. We name it Strudel: a STRUctured-DEcomposable Learner.

5. Fast Mixtures with Strudel

Learning mixtures of PCs greatly improves their performance as density estimators [47, 48, 49, 50].

Issues in learning structured-decomposable mixtures. Building a mixture of several PCs results in a joint non-deterministic PC, as it introduces a sum node marginalizing a latent variable. While such a PC would not allow exact MAP inference, it could still be used for queries requiring structured decomposability. However, to answer complex queries like the expectation of predictive models [7], one would require a *structured-decomposable mixture of PCs*, i.e., an ensemble whose components are structured-decomposable *and* share the same vtree. *To force such a constraint, while preserving the mixture expressiveness or containing its circuit size, is a non-trivial research question.* Consider learning several PCs with Strudel while requiring them to share the same vtree. If we learn each component from a different CLT, compiling them to PCs while enforcing a unique vtree might lead to an exponential blow-up in the size of some PCs. Alternatively, enforcing Algorithm 7 to output a CLT that can be compactly compiled according to a vtree, would result in losing the algorithm's optimality guarantee.

Shared-structure mixtures. We propose a simpler and faster ensembling strategy which proves to be very effective in practice. *We build ensembles of PCs sharing the same structure*, concretely the structure learned by Strudel for single models on that data, while letting each mixture component have different parameters. This has a number of advantages: we (i) need to perform structure learning only once, (ii) materialize a single flow $f_{\mathcal{M}}$ once (as identical structures will generate the same flow), and (ii) can evaluate the likelihood of the whole mixture efficiently, as shown in Eq. 2.

This strategy is compatible with classical ensembling schemes such as expectation-maximization (EM), bagging and boosting. All these scenarios, e.g., each M step in EM, reduce to learning each mixture component parameters on a weighted version of the original data.

We briefly review how EM operates in our ensembling scenario. In each E step, the log-likelihoods of mixtures are evaluated as in Equation 2. And in each M step, we have a weighted data set for each component, the weight for

sample $\boldsymbol{x}$ and component c is the probability of this sample $\boldsymbol{x}$ selecting component c from all components:

$$\mathsf{Pr}(c|\boldsymbol{x}) = \frac{\mathsf{Pr}(\boldsymbol{x}, c)}{\mathsf{Pr}(\boldsymbol{x})}$$

The numerator is cached from Equation 2 since we can compute the log-probability $\mathsf{Pr}(\boldsymbol{x}, c)$ for each component via $f_{\mathcal{M}}(\boldsymbol{x})^T \cdot \log(\boldsymbol{\Theta})$ and the denominator is just normalizing all components in it. Therefore, we can learn the parameters for each component in closed form as in Equation 3, except that we use *expected aggregate circuit flows*:

$$ea_c(i, j; \mathcal{D}) = \sum\nolimits_{h=1}^{|\mathcal{D}|} \mathsf{Pr}(c|\mathcal{D}[h]) \cdot \mathbf{F}_{\mathcal{M}}(\mathcal{D})[h, k]$$

Instead of counting the samples, it sums all sample weights flow through. Finally, we update the combination weights $\boldsymbol{w}$ with $\mathsf{Pr}(c)$ for every $c \in \mathcal{M}$.

6. Parallel Computing on CPU and GPU

From Section 3 and Section 5, we can see that most of the arithmetic computation in learning, such as evaluating the log-likelihood and estimating the parameters, is closely related to circuit flows and is computed by matrix manipulation. Moreover, if two nodes appear in the same 'layer' of the PC, their value can be computed independently given the previous layer. Therefore, it is natural to additionally speed up our algorithms by parallel computation, using CPU multi-threading, CPU SIMD instructions, and GPU parallelism.

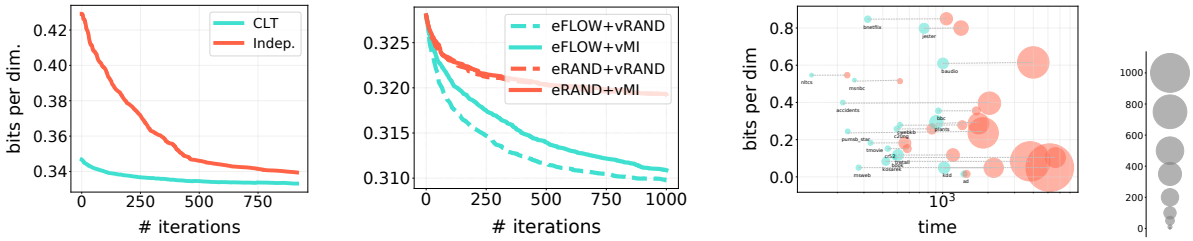
Efficient data structures for PCs. The most intuitive way to encode PCs is by a linked node data structure. Then the inference and learning algorithms would require iterating over the nodes of the PC either in a feedforward (children to parents) or backward (parents to children) fashion. However, this representation makes computations sparse, thus it is harder to leverage parallelism. To optimize performance during inference and learning, we translate the PCs' DAG into a layered computational graph [20, 51]. The nodes of a PC are cached in a layered vector by some unique identifiers. We also explicitly cache the mapping from parents to children for forward traversal and children to parents for backward traversal.

Parallel computation. Since the computations on the nodes in the same layer are cached in one large vector, we can simultaneously parallelize our computation over the nodes in the layer on the one hand, and training examples or inference task data on the other hand. Such parallelism is most useful in fast mixtures. As discussed in Section 5, the core bottleneck operation there is parameter learning, i.e., efficiently computing flows, evaluating the PCs and estimating their parameters. Concretely, when training the mixtures using the EM algorithm, we only need to compute the flows once at all, since they are not changed during training; and then for each iteration, we perform the expectation step by Equation 2 and the maximization step by Equation 3.

We use customized kernels to accelerate computation on both CPUs and GPUs (using SIMD and CUDA kernels respectively). Experiments show that CPU parallelism gives significant speed-ups, which even become an order of magnitude faster with GPU parallelism, all using the same underlying data structures.

7. Experiments

In this section, we rigorously evaluate Strudel empirically. We implement our learning algorithms in the open-source Juice library for probabilistic circuits [52]. The natural competitor for Strudel is LearnPsdd, as they both aim to learn PCs with the same structural properties (cf. Section 4). We evaluate both learners as density estimators on a series of 20 standard benchmark datasets. Specifically, we aim to answer the following research questions: **(Q1)** What is the effect of initializing structure learning with a CLT? **(Q2)** How is the splitting heuristic in Strudel affecting structure learning? **(Q3)** How does the beam search strategy improve the greedy search structure learning? **(Q4)** How do single PCs learned by Strudel compare to those learned by LearnPsdd? **(Q5)** Are ensembles of PCs learned by Strudel competitive with LearnPsdd? **(Q6)** Is our inference approach based on circuit flows speeding up likelihood computations on ensembles of PCs? **(Q7)** How do CPU parallelism and GPU parallelism help speed up training of mixtures? **(Q8)** Are ensembles of PCs learned by Strudel helpful for advanced probabilistic queries?



(a) (b) (c) (d)

Figure 3: ***Effect of different initializations and heuristics in* STRUDEL.** We report the mean test bits per dimensions (bpd) averaged across all datasets (y-axis) for each iteration (x-axis) as scored by the different initialization schemes (3a) and with four possible splitting heuristics (cf. Section 4) in STRUDEL when using the CLT initialization (3b). In Figure 3c the per-dataset bpd scores (y-axis) versus total learning time (in secs, x-axis) and learned PC sizes (proportional to circle radiuses times 1000, cf. legend) of the heuristics EFLOW+VMI(blue) and EFLOW+VRAND(red). (3d) indicates mapping from the circuit sizes (number of nodes) to the circuit shape.

7.1. Evaluation of Single Models

(Q1) Effect of using CLTs. To evaluate the impact of different initializations in STRUDEL, we compare the CLT initialization scheme we proposed in Section 4.1 against the scheme of LEARNPSDD, where the initial PC is a fully-factorized distribution, normalized for a vtree learned in advance. For both, we learn PCs with up to 1000 splits on every dataset, then report the mean test bits-per-dimension (bpd)[4] averaged across all datasets as a function of iterations in Figure 3 (top left). On average, employing CLTs in STRUDEL not only delivers more accurate initial PCs as expected, but better bpds in the long run. Detailed per-dataset curves can be found in Appendix C.4.

We have additionally experimented with different heuristics to select the root of a CLT, such as picking a random node or one with highest degree in addition to selecting the Jordan center of the tree. In the end, we observed that all these variants do not affect the circuit likelihoods significantly. However, selecting the Jordan center can be preferred as it delivers the CLT with minimal depth and hence it produces smaller circuits in the end, which in turn means faster inference.

(Q2) Effect of splitting heuristics. We adopt the same setting of **Q1** and we mix and match all possible combinations of splitting heuristics: 1) EFLOW-VMI, 2) EFLOW-VRAND, 3) ERAND-VMI, 4) ERAND-VRAND (cf. Section 4.2). Figure 3 (top right) reports the mean test bpd per iteration averaged across all datasets. Detailed per-datasets plots can be found in Appendix C.5. It is apparent how selecting edges at random with ERAND delivers suboptimal PCs when compared to EFLOW, regardless of the heuristic to select the RV. On the other hand, EFLOW-VRAND delivers slightly more accurate PCs, on average, than EFLOW-VMI. However, this comes at a high price which is highlighted in Figure 3 (bottom): circuits learned by EFLOW-VRAND are one or two orders of magnitude larger and each splitting iteration on them is much slower than for EFLOW-VMI. This can be explained as follows: VRAND can greatly increase the PC size by arbitrarily picking a RV far from the root of the sub-circuit selected by EFLOW hence duplicating a larger PC, while VMI picks more informative RVs which generally are closer to the root of such sub-circuit. Given all the above, we employ EFLOW-VMI and the CLT initialization scheme in our remaining experiments.

(Q3) Beam search. We refer with STRUDELBEAM to the beam search algorithm and with STRUDELGREEDY to STRUDEL without quantifier. To evaluate the impact of STRUDELBEAM, we choose the beam width β by a grid search on {1, 5, 10, 30}. A beam width being 1 directly reduces to STRUDELGREEDY. At each iteration, we select the top β PCs to perform split operation on from all candidates according to log-likelihoods on validation set. Taking dataset JESTER as an example, Figure 4 reports the average log-likelihood, circuit sizes, and training times as a function of iteration. As we can see, the beam width helps improve the log-likelihoods, especially in early training iterations, though later the margin between different beam widths are getting smaller. Also, the circuits' sizes are close as beam width increases. Besides, as shown in Figure 4 (right), the training time increases as beam width increases: there is a trade-off between accuracy and time. For efficient learning, STRUDELGREEDY is a nice choice point as it only computes simple heuristics and gets a relatively effective circuit in a short amount of time; however, if you have enough time, STRUDELBEAM performs better as it delivers more accurate PCs. We perform early stopping after 100 iterations with no

[4] $\mathsf{bpd}(\boldsymbol{\theta}; \mathcal{D}) = -\sum_{i=1}^{|\mathcal{D}|} \mathcal{LL}(\boldsymbol{\theta}; \boldsymbol{x}_i)/(\log(2) \cdot |\mathcal{D}| \cdot m)$ where m is the number of features in dataset $\mathcal{D}$.

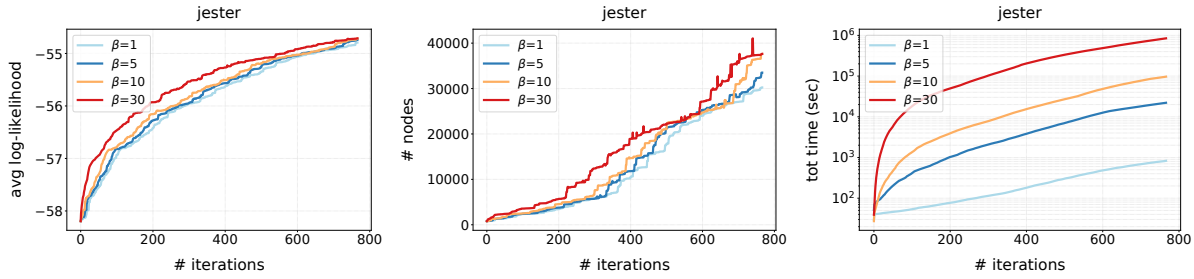


Figure 4: ***Comparing different beam width***. We report the averaged training set log-likelihood (left), number of nodes in PC (center), and total time (right) (y-axis) as a function of iteration (x-axis) as scored by different beam width in {1, 5, 10, 30}.

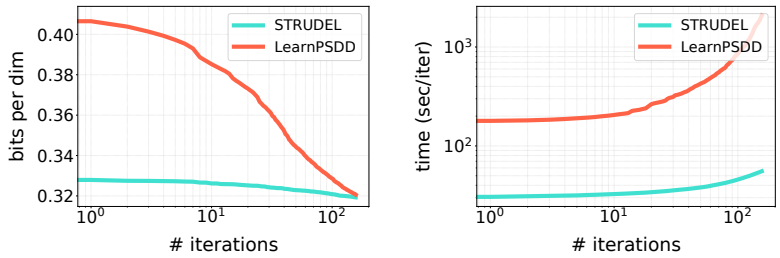


Figure 5: ***Comparing* STRUDEL *and* LEARNPSDD *accuracy and learning times***. We report the mean bpd averaged across all datasets (y-axis) (left), and seconds per iteration during learning (y-axis) (right) for each iteration (x-axis) as scored by single models learned by STRUDEL (blue) and LEARNPSDD (red).

improvements for STRUDELGREEDY and STRUDELBEAM and report the mean test log-likelihood for each dataset and the beam width each dataset in Table 1. STRUDELBEAM learns more effective PCs than STRUDELGREEDY and usually with larger beam width.

(Q4) Single models. We perform early stopping after 100 iterations with no improvements for STRUDELGREEDY and STRUDELBEAM. For LEARNPSDD, we re-run on all datasets by using the hyperparameters reported in [18].

Table 1 reports the mean test log-likelihood for each dataset. For single models, STRUDEL consistently learns more (or equally) accurate PCs than LEARNPSDD on 11 datasets out of 20.[5] More strikingly, STRUDELGREEDY *delivers equivalently accurate PCs sooner than* LEARNPSDD. This is clearly shown in Figure 5 (left & center) for all datasets: it takes fewer iterations for STRUDELGREEDY to achieve comparable bpds and each of its iterations takes at least one order of magnitude less time than LEARNPSDD. Lastly, PCs learned with STRUDEL are still comparable on many datasets to other PC learners performing local search like selective SPNs (SELSPN) [25] and normal SPNs (SEASPN) [53]. Note however, that these competitors learn PCs with less structural requirements (SELSPNs are not structured-decomposable and SEASPNs are not deterministic), and hence support fewer tractable inference scenarios (cf. Section 2). We report detailed learning times, circuit sizes and statistical tests are reported in Appendix C.

7.2. Evaluation of Mixtures

(Q5) Mixtures with STRUDEL. We evaluate the shared-structure mixtures of PCs learned by STRUDEL by reusing for each dataset the best single PC learned for setting **Q1**, and performing parameter learning with EM (STRUDEL-EM) or a combination of EM and bagging (STRUDEL-BEM) to alleviate overfitting as in [18]. For STRUDEL-EM, we choose the number of mixture components k_{EM} by a grid search on {2, 5, 10, 15, 20, 25, 30}. STRUDEL-BEM instead trains mixtures of PCs using EM on 10 different bagged datasets. Table 2 reports the mean test log-likelihoods of our mixtures and the corresponding ones learned by LEARNPSDD. On 15 datasets STRUDEL-EM outperforms LEARNPSDD-EM, while STRUDEL-BEM is more accurate than its counterpart 11 times. This is remarkable if one notes that PCs

[5]The likelihoods we are reporting for LEARNPSDD are significantly better than those originally reported in [18]. Compared to those original results, STRUDEL is more accurate than LEARNPSDD 16 times out of 20.

DATASET	LEARNPSDD	STRUDEL GREEDY	STRUDEL BEAM	SELSPN	SEASPN
NLTCS	-6.03	**-6.07**	**-6.06**	-6.03↓	-6.07↑
MSNBC	-6.04	-6.05	-6.05	-6.04↓	-6.06↑
KDD	-2.17	**-2.17**	**-2.16**	-2.16↑	-2.16↑
PLANTS	-13.49	-13.95	-13.78	-12.97↓	-13.12↓
AUDIO	-41.51	-42.29	-42.17	-41.23↓	-40.13↓
JESTER	-54.63	-55.23	-55.21	-54.38↓	-53.08↓
NETFLIX	-58.53	-58.66	**-58.59**	-57.98↓	-56.91↓
ACCIDENTS	-28.29	-29.63	-29.41	-26.88↓	-30.02↑
RETAIL	-10.92	**-10.90**	**-10.90**	-10.88↓	-10.97↑
PUMSB-STAR	-25.40	-26.11	-25.70	-22.66↓	-28.69↑
DNA	-83.02	-87.20	-86.68	-80.44↓	-81.76↓
KOSAREK	-10.99	**-10.98**	**-10.87**	-10.85↑	-11.00↑
MSWEB	-9.93	-10.19	-10.13	-9.93↓	-10.25↑
BOOK	-36.06	**-35.80**	**-35.76**	-36.01↑	-34.91↓
EACHMOVIE	-55.41	-60.49	-59.69	-55.73↓	-53.28↓
WEBKB	-161.42	**-159.95**	**-159.90**	-158.52↑	-157.88↓
ROUTERS-52	-93.30	**-91.82**	**-91.73**	-88.48↓	-86.38↓
20NEWS-GRP	-160.43	**-160.77**	**-160.67**	-158.68↓	-153.63↓
BBC	-260.24	**-260.15**	**-258.35**	-259.35↑	-253.13↓
AD	-20.13	**-16.52**	**-16.40**	-16.94↑	-16.77↑

Table 1: ***Density estimation benchmarks: single models***. Average test log-likelihood for STRUDEL and LEARNPSDD models. The bold values indicate STRUDEL is better than or statistically equivalent with (cf. Appendix C.3) LEARNPSDD. On the right other state-of-the-art structure learners, which are *not* targeting structured-decomposable circuits (see text). ↑ (resp. ↓) indicates that STRUDEL is more accurate (resp. less accurate).

DATASET	LEARNPSDD EM	STRUDEL EM	LEARNPSDD BEM	STRUDEL BEM
NLTCS	-6.03↓	-6.07	**-5.99**↓	-6.06
MSNBC	**-6.04**↑	**-6.04**	**-6.04**↑	**-6.04**
KDD	-2.12↓	-2.14	**-2.11**↓	-2.13
PLANTS	-13.79↑	-13.22	-13.02↑	**-12.98**
AUDIO	-41.98↑	-41.20	**-39.94**↓	-41.50
JESTER	-53.47↓	-54.24	**-51.29**↓	-55.03
NETFLIX	-58.41↑	-57.93	**-55.71**↓	-58.69
ACCIDENTS	-33.64↑	-29.05	-30.16↑	**-28.73**
RETAIL	-10.81↓	-10.83	**-10.72**↓	-10.81
PUMSB-STAR	-33.67↑	-24.39	-26.12↑	**-24.12**
DNA	-92.67↑	-87.15	-88.01↑	**-86.22**
KOSAREK	-10.81↑	-10.70	**-10.52**↓	-10.68
MSWEB	-9.97↑	-9.74	-9.89↑	**-9.71**
BOOK	-34.97↑	**-34.49**	-34.97↓	-34.99
EACHMOVIE	-58.01↑	-53.72	-58.01↑	**-53.67**
WEBKB	-161.09↑	**-154.83**	-161.09↑	-155.33
ROUTERS-52	-89.61↑	-86.35	-89.61↑	**-86.22**
20NEWS-GRP	-160.09↑	**-153.87**	-155.97↑	-154.47
BBC	-253.19↓	-256.53	**-253.19**↓	-254.41
AD	-31.78↑	-16.52	-31.78↑	**-16.38**

Table 2: ***Density estimation benchmarks: ensembles***. Average test log-likelihood for STRUDEL and LEARNPSDD models. Two versions of ensembles (EM vs. BEM) are compared separately, ↑ (resp. ↓) indicates that STRUDEL is more accurate (resp. less accurate). Bold values indicates the best on certain dataset over 4 methods.

in LEARNPSDD-(B)EM are allowed to take arbitrary structures and update the structures for each component during learning. As expected, STRUDEL drastically reduces the learning times of large mixtures, as single PCs can be learned much faster, and mixtures can be learned in a fraction of the time by virtue of shared flows (cf. Section 3).

(Q6) Effectiveness of flows. The efficiency from classical circuit evaluation to our circuit flow approach on a single circuit comes from that computing conjunction and disjunction of bit-vectors is much more efficient than computing sum or product of floating point numbers. Experiment shows that the speed-up is around 3 times.

The speed-up is more significant in mixtures. Figure 6 (left) shows timings for evaluating the circuitscomputing mixture likelihoods on the 'plants' dataset. The circuit flow approach is orders of magnitude faster (around 10^2–10^3), and up to 4591 times faster on the 'msnbc' dataset. Table D.8 in Appendix D reports the detailed timings for all datasets.

(Q7) Effectiveness of parallel computing. We compare the effectiveness of the naive circuit flow approach, CPU parallel computing and GPU parallel computing for learning mixtures. Figure 6 (right) reports the seconds it takes to perform one expectation-maximization step while learning mixtures on the 'plants' dataset. We can see that CPU parallelism gives significant speed-ups, which even become an order of magnitude faster with GPU parallelism, all using the same underlying data structure. Table E.9 in Appendix D reports the detailed timings for all datasets.

7.3. Application of Structured-decomposable Probabilistic Circuits

(Q8) Advanced probabilistic queries. Finally, we evaluate how PCs learned with STRUDEL can be exploited for advanced inference scenarios requiring structured decomposability. We adopt the experimental setting of [7] aiming to compute the expected predictions of a regressor r w.r.t. a generative model represented as a structured-decomposable PC sharing the same vtree of r in the challenging scenario of predicting a real target variable in the presence of missing values over the input features. Specifically, on 4 different real-world regression benchmarks, Abalone, Delta, Elevators and Insurance, we first learn either a structured-decomposable PC with STRUDELor a mixture learned by STRUDEL-BEM with 5 bags and a number of EM components cross-validated in {5, 10, 15, 20}. Then, we learn a regression circuit sharing the same vtree as the learned generative model by employing the learning algorithm of [7]. Figure 7 shows the root mean squared error (RMSE) of our models for different percentages of missing values, when compared to a

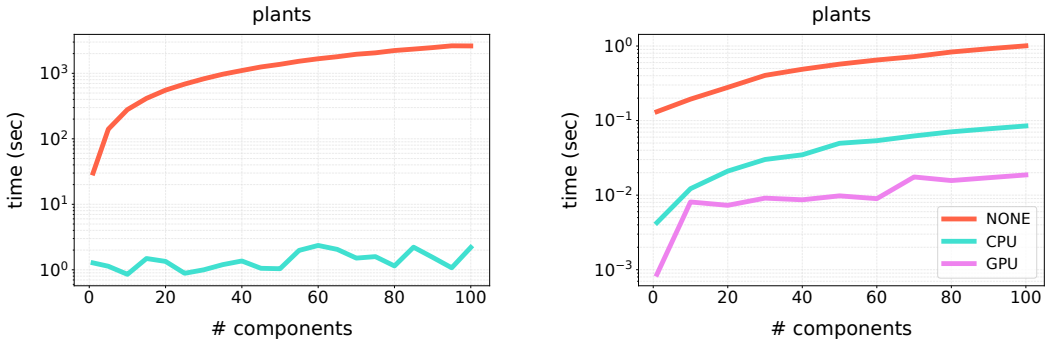
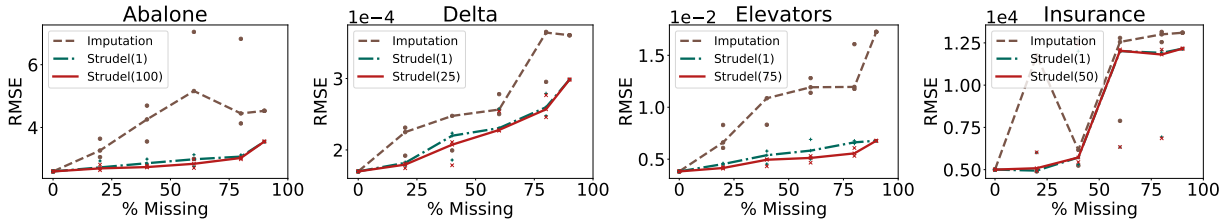


Figure 6: ***Circuit flows for fast inference of ensembles for 'plants' dataset.*** We report the time (seconds) used evaluating the ensemble circuits (y-axis) for different number of component in the ensemble (x-axis), comparing the circuit flows implementation (blue) and the classical algorithm of naively evaluating the circuits bottom-up (red). ***CPU and GPU parallelism for fast parameter learning of ensembles for 'plants' dataset.*** We report the time (seconds) used learning the ensemble circuits per iteration (y-axis) for different number of component in the ensemble (x-axis), comparing the non-parallelism circuit flows implementation (red), CPU parallelism version (blue) and GPU parallelism version (violet).



(a) (b) (c) (d)

Figure 7: ***Expected prediction benchmarks***. Root Mean Square Error (RMSE) for median imputation, STRUDEL and STRUDEL-Ensemble models (number in parenthesis shows the number of components in the ensemble models).

common imputation schemes like median imputation. We can see that not only that single PCs perform better than the baseline, but the cheap mixtures of PCs with shared structure help further reduce the error.

8. Conclusions

In this paper we introduced STRUDEL to learn structured-decomposable PCs in a fast and simple way. STRUDEL delivers accurate single PCs in a fraction of the time of its competitor and effectively scales up to learning mixtures of PCs sharing the same structure. We consider STRUDEL as an initial stepping stone to learn PCs for several application scenarios where advanced probabilistic inference is required and out-of-the-scope of the current landscape of tractable probabilistic models; for example in explainable AI [54] and algorithmic fairness [55]. Another interesting research direction is to extend STRUDEL to deal with mixed continuous-discrete settings. While extending it to accommodate categorical variables is easy to implement, as the latter can be encoded as finite mixtures of indicator functions, and therefore we could reuse our split operator and heuristics for them, we could not do the same for continuous random variables and discrete distributions with infinite supports. This challenge calls for novel heuristics and data-dependent split operator variants which can potentially discretize an unbounded support into a finite set while maximizing the data likelihood.

Acknowledgements

This work is supported in part by NSF grants #CCF-1837129, #IIS-1956441, #IIS-1943641 and a Sloan Fellowship.

References

[1] A. Vergari, Y. Choi, R. Peharz, G. Van den Broeck, Probabilistic circuits: Representations, inference, learning and applications, The 34th AAAI Conference on Artificial Intelligence (Tutorial) (2020).

[2] J. Bekker, J. Davis, A. Choi, A. Darwiche, G. Van den Broeck, Tractable learning for complex probability queries, in: Advances in Neural Information Processing Systems 28 (NeurIPS), 2015.

[3] A. Choi, G. Van den Broeck, A. Darwiche, Tractable learning for structured probability spaces: A case study in learning preference distributions, in: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI), 2015.

[4] Y. Shen, A. Choi, A. Darwiche, A tractable probabilistic model for subset selection., in: Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI), 2017.

[5] Y. Choi, G. Van den Broeck, On robust trimming of bayesian network classifiers, in: Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI), 2018.

[6] P. Khosravi, Y. Liang, Y. Choi, G. Van den Broeck, What to expect of classifiers? Reasoning about logistic regression with missing features, in: Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), 2019.

[7] P. Khosravi, Y. Choi, Y. Liang, A. Vergari, G. Van den Broeck, On tractable computation of expected predictions, in: Advances in Neural Information Processing Systems 32 (NeurIPS), 2019.

[8] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, B. Lakshminarayanan, Normalizing flows for probabilistic modeling and inference, Journal of Machine Learning Research.

[9] Y. Choi, A. Vergari, G. Van den Broeck, Probabilistic circuits: A unifying framework for tractable probabilistic modeling.

[10] A. Darwiche, A differential approach to inference in Bayesian networks, Journal of the ACM.

[11] D. Kisa, G. Van den Broeck, A. Choi, A. Darwiche, Probabilistic sentential decision diagrams, in: Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR), 2014.

[12] H. Poon, P. Domingos, Sum-Product Networks: a New Deep Architecture, in: Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI), 2011.

[13] T. Rahman, P. Kothalkar, V. 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 (ECML-PKDD), 2014.

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

[15] M. Meilă, M. I. Jordan, Learning with mixtures of trees, Journal of Machine Learning Research.

[16] Y. Shen, A. Choi, A. Darwiche, Tractable operations for arithmetic circuits of probabilistic models, in: Advances in Neural Information Processing Systems 29 (NeurIPS), 2016.

[17] Y. Liang, G. Van den Broeck, Towards compact interpretable models: Shrinking of learned probabilistic sentential decision diagrams, in: IJCAI 2017 Workshop on Explainable Artificial Intelligence (XAI), 2017.

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

[19] L. Mattei, D. Soares, A. Antonucci, D. Mauà, A. Facchini, Exploring the space of probabilistic sentential decision diagrams, in: 3rd Workshop of Tractable Probabilistic Modeling, 2019.

[20] A. Vergari, N. Di Mauro, F. Esposito, Visualizing and understanding sum-product networks, Machine Learning Journal.

[21] R. Peharz, S. Lang, A. Vergari, K. Stelzner, A. Molina, M. Trapp, G. V. d. Broeck, K. Kersting, Z. Ghahramani, Einsum networks: Fast and scalable learning of tractable probabilistic circuits, in: Proceedings of the 37th International Conference on Machine Learning (ICML), 2020.

[22] H. Chan, A. Darwiche, On the robustness of most probable explanations, in: Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI), 2006.

[23] A. Darwiche, Modeling and reasoning with Bayesian networks, Cambridge University Press, 2009.

[24] N. Di Mauro, A. Vergari, F. Esposito, Learning accurate cutset networks by exploiting decomposability, in: AI*IA 2015, Advances in Artificial Intelligence, 2015.

[25] R. Peharz, R. Gens, P. Domingos, Learning selective sum-product networks, in: Workshop on Learning Tractable Probabilistic Models, 2014.

[26] K. Pipatsrisawat, A. Darwiche, New compilation languages based on structured decomposability., in: Proceedings of the 23rd AAAI Conference on Artificial Intelligence, 2008.

[27] R. Dechter, R. Mateescu, And/or search spaces for graphical models, Artificial Intelligence.

[28] U. Oztok, A. Darwiche, A top-down compiler for sentential decision diagrams, in: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI), 2015.

[29] Y. Choi, A. Darwiche, G. Van den Broeck, Optimal feature selection for decision robustness in bayesian networks, in: Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI), 2017.

[30] P. Khosravi, A. Vergari, Y. Choi, Y. Liang, G. Van den Broeck, Handling missing data in decision trees: A probabilistic approach, in: The Art of Learning with Missing Values Workshop at ICML (Artemiss), 2020.

[31] Y. Liang, G. Van den Broeck, Learning logistic circuits, in: Proceedings of the 33rd Conference on Artificial Intelligence, 2019.

[32] H. Zhao, M. Melibari, P. Poupart, On the Relationship between Sum-Product Networks and Bayesian Networks, in: Proceedings of the 32th International Conference on Machine Learning (ICML), 2015.

[33] R. Peharz, Foundations of sum-product networks for probabilistic modeling, Ph.D. thesis, Graz University of Technology, SPSC (2015).

[34] A. Darwiche, Modeling and Reasoning with Bayesian Networks, Cambridge University Press, 2009.

[35] R. Peharz, S. Tschiatschek, F. Pernkopf, P. Domingos, On theoretical properties of sum-product networks, The Journal of Machine Learning Research.

[36] H. Zhao, T. Adel, G. Gordon, B. Amos, Collapsed variational inference for sum-product networks, in: In Proceedings of the 33rd International Conference on Machine Learning (ICML), 2016.

[37] H. Zhao, P. Poupart, G. J. Gordon, A unified approach for learning the parameters of sum-product networks, in: Advances in Neural Information Processing Systems 29 (NeurIPS), 2016.
[38] S. Kowshik, Y. Liang, G. Van den Broeck, IL-Strudel: Independence-based learning of structured-decomposable probabilistic circuit ensembles, in: The 4th Workshop on Tractable Probabilistic Modeling (TPM), 2021.
[39] R. L. Geh, D. D. Mauá, Learning probabilistic sentential decision diagrams under logic constraints by sampling and averaging, in: Proceedings of the 37th Conference on Uncertainty in Artificial Intelligence (UAI), 2021.
[40] N. D. Mauro, G. Gala, M. Iannotta, T. M. Basile, Random probabilistic circuits, in: Proceedings of the 37th Conference on Uncertainty in Artificial Intelligence (UAI), 2021.
[41] A. Shih, D. Sadigh, S. Ermon, HyperSPNs: Compact and expressive probabilistic circuits, in: The 4th Workshop on Tractable Probabilistic Modeling (TPM), 2021.
[42] A. Liu, G. Van den Broeck, Tractable regularization of probabilistic circuits, in: The 4th Workshop on Tractable Probabilistic Modeling (TPM), 2021.
[43] C. K. Chow, C. N. Liu, Approximating discrete probability distributions with dependence trees, IEEE Transactions on Information Theory.
[44] A. Choi, D. Kisa, A. Darwiche, Compiling probabilistic graphical models using sentential decision diagrams, in: Proceedings of the 12th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU), 2013.
[45] S. Holtzen, G. Van den Broeck, T. Millstein, Scaling exact inference for discrete probabilistic programs, Proc. ACM Program. Lang. (OOPSLA).
[46] D. Lowd, P. Domingos, Learning arithmetic circuits, in: Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence (UAI), 2008.
[47] A. Vergari, N. Di Mauro, F. Esposito, Simplifying, Regularizing and Strengthening Sum-Product Network Structure Learning, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD), 2015.
[48] N. Di Mauro, A. Vergari, T. M. Basile, Learning bayesian random cutset forests, in: International Symposium on Methodologies for Intelligent Systems (ISMIS), 2015.
[49] N. Di Mauro, A. Vergari, T. M. Basile, F. Esposito, Fast and accurate density estimation with extremely randomized cutset networks, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD), 2017.
[50] T. Rahman, V. Gogate, Learning ensembles of cutset networks, in: Proceedings of the 30th AAAI Conference on Artificial Intelligence, 2016.
[51] R. Peharz, A. Vergari, K. Stelzner, A. Molina, X. Shao, M. Trapp, K. Kersting, Z. Ghahramani, Random sum-product networks: A simple but effective approach to probabilistic deep learning, in: Proceedings of The 35th Uncertainty in Artificial Intelligence Conference (UAI), 2019.
[52] M. Dang, P. Khosravi, Y. Liang, A. Vergari, G. 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.
[53] A. Dennis, D. Ventura, Greedy Structure Search for Sum-product Networks, in: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI), 2015.
[54] G. Van den Broeck, A. Lykov, M. Schleich, D. Suciu, On the tractability of SHAP explanations, in: Proceedings of the 35th AAAI Conference on Artificial Intelligence, 2021.
[55] Y. Choi, M. Dang, G. Van den Broeck, Group fairness by probabilistic modeling with latent fair decisions, in: Proceedings of the 35th AAAI Conference on Artificial Intelligence, 2021.

Appendix A. Learning a Chow-Liu Tree

We list the algorithm for learning Chow-Liu Trees [43] and sub-routines sumNodes to help compiling a CLT here.

Algorithm 7: LearnCLT($\mathcal{D}, \boldsymbol{X}, \alpha$)

Input : a dataset $\mathcal{D}$ over RVs, $\boldsymbol{X} = \{X_i\}_{i=1}^n$, Laplace smoothing factor α
Output : a Chow-Liu tree model $\langle \mathcal{T}, \boldsymbol{\theta} = \{\theta_{i|\mathrm{Pa}_i}\}_{i=1}^n \rangle$ estimating $\mathsf{p}(\boldsymbol{X})$

$\mathsf{MI} \leftarrow \mathbf{0}_{n \times n}$
foreach $X_i, X_j \in \mathbf{X}$ **do**
  $\mathsf{MI}_{ij} \leftarrow \mathsf{estimateMI}(\mathcal{D}, X_i, X_j, \alpha)$
end
$T \leftarrow \mathsf{maximumSpanningTree}(\mathsf{MI})$
$\mathcal{T} \leftarrow \mathsf{traverseTree}(T)$
$\boldsymbol{\theta} \leftarrow \{\theta_{i,\mathrm{Pa}_i} \leftarrow \mathsf{estimateCPT}(\mathcal{D}, X_i, X_{\mathrm{Pa}_i}, \alpha)\}$
return $\langle \mathcal{T}, \boldsymbol{\theta} \rangle$

Algorithm 8: sumNodes(ns, X)

Input : list of sub-circuits ns with length two, RV X
Output : sum nodes with children ns, parameterized by $\mathsf{p}(X)$ or $\mathsf{p}(X|X_{\mathrm{Pa}})$

$sums \leftarrow \emptyset$
$n_l, n_r \leftarrow ns$
if *X is root in CLT* **then**
  $\theta \leftarrow \mathsf{p}(X = 1)$
  $\mathsf{append}(sums, \theta \cdot n_l \bigoplus (1 - \theta) \cdot n_r)$
else
  for $x_{\mathrm{Pa}} \in X_{\mathrm{Pa}}$ **do**
    $\theta \leftarrow \mathsf{p}(X = 1 | X_{\mathrm{Pa}} = x_{\mathrm{Pa}})$
    $\mathsf{append}(sums, \theta \cdot n_l \bigoplus (1 - \theta) \cdot n_r)$
return $sums$

Figure A.8 illustrates the compilation progress of CLT in Figure 1.

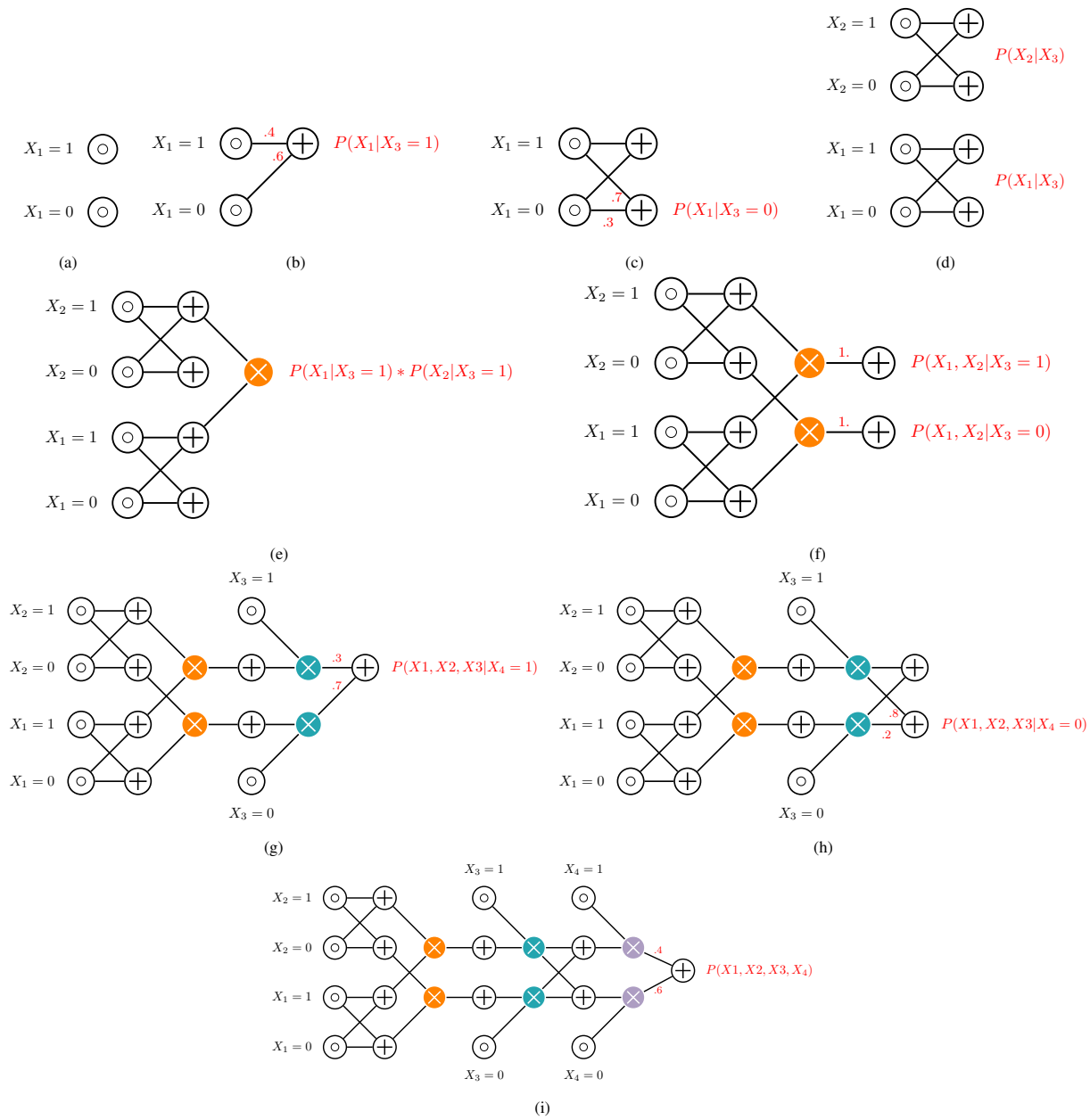


Figure A.8: ***Compilation progress of CLT into its corresponding PC.*** (A.8a) compiles literal nodes for RV X_1 (line 4-6 in Algorithm 3). For parent configuration $X_3 = 1$, (A.8b) introduce sum node selecting values of X_1 with edge parameterized by conditional probability $P(X_1|X3 = 1)$. (A.8b) and (A.8c) represent $P(X_1|X3))$ (line 7-8 in Algorithm 3). (A.8d) compiles $P(X_2|X_3)$ following the same rule, and (A.8e) connects two sum nodes with a product node representing that X_1 and X_2 are independent conditioned on X_3; the product node is colored as its corresponding vtree node in Figure 1. (A.8f) add one layer sum nodes to maintain semantic properties (line 17-18 in Algorithm 3). (A.8g) and (A.8h) create two sub-circuits under conditions $X_4 = 1$ and $X_4 = 0$ separately (line 19-20 in Algorithm 3). (A.8i) finishes the whole algorithm.

Appendix B. Subroutines of Split Operation

We list subroutines conjoin and partialCopy for split operation here. Given a smooth PC, algorithm conjoin returns a circuit with the logical formula conjoined with a given literal constraint. There is a similar implementation for the disjoin algorithm. Algorithm partialCopy recursively creates a copy of a circuit up to a certain depth, and circuit structures beyond that depth are reused in the new copy.

Algorithm 9: conjoin($\mathcal{G}, X = x$)

Input : a smooth PC structure $\mathcal{G}$, a random variable $X \in \phi(\mathcal{G})$ and its assignment x
Output : the PC structure $\mathcal{G}$ with logical formula conjoined with $X = x$

$old2new(n) \leftarrow \emptyset$ for every $n \in \mathcal{G}$ `// map old sub-circuits to new ones`
`// iterate children before parents`
foreach $n \in \mathcal{G}$ **do**
  if *n is a literal node* **then**
    if variable(n) = X *and* literal(n)! = x **then**
      $old2new(n) \leftarrow \emptyset$
    else
      $old2new(n) \leftarrow n$
  else
    $ch \leftarrow old2new(c)$ for every $c \in \mathsf{ch}(n)$
    if *n is a product node* **then**
      if $\emptyset \in ch$ **then**
        $old2new(n) \leftarrow \emptyset$
      else
        $old2new(n) \leftarrow \bigoplus(ch)$
    else
      if *c is* $\emptyset$ *for every* $c \in ch$ **then**
        $old2new(n) \leftarrow \emptyset$
      else
        remove $\emptyset$ from ch
        $old2new(n) \leftarrow \bigotimes(ch)$
return $old2new(r)$ for PC root r

Algorithm 10: partialCopy($\mathcal{G}, depth, old2new = \{\}$)

Input : a smooth PC structure $\mathcal{G}$, a certain depth $depth$ to create the copy, a dictionary $old2new$ mapping from old circuits to new circuits
Output : a copy of the PC structure to the certain depth

$n \leftarrow$ root of $\mathcal{G}$
if *depth is 0 or n is a leaf* **then**
  return $\mathcal{G}$
else if $n \in old2new$ **then**
  return $old2new[n]$
else
  $ch \leftarrow$ partialCopy($c, depth - 1, old2new$) for $c \in \mathsf{ch}(n)$
  if *n is product node* **then**
    $n^* \leftarrow$ conjoin(ch)
  else
    $n^* \leftarrow$ disjoin(ch)
  $old2new(n) \leftarrow n^*$
return graph rooted at n^*

Appendix C. Single Models with STRUDEL

Appendix C.1. Circuit Sizes

We compare the circuit size of models learned by LEARNPSDD STRUDEL in Table C.3. As expected, STRUDELGREEDY and STRUDELBEAM learned circuits have similar circuit sizes, and also STRUDEL delivers larger circuits than LEARNPSDD but almost always on the same order of magnitude, which makes sense since the latter explicitly penalizes larger sizes via its likelihood score. Nevertheless, the increase in size is contained with the vMI heuristics (cf. Figure 3 (bottom)) and is a negligible price to pay for a much faster learner (cf. Table C.4).

DATASET	LEARNPSDD	STRUDEL GREEDY	STRUDEL BEAM
NLTCS	1304	7909	7166
MSNBC	5465	20509	20360
KDD	2915	9744	30174
PLANTS	11583	143805	148310
BAUDIO	18208	50170	57860
JESTER	11322	44438	41079
BNETFLIX	10997	25598	34791
ACCIDENTS	8418	41401	54242
TRETAIL	2989	3984	3984
PUMSB_STAR	8298	40471	68171
DNA	3068	13461	8430
KOSAREK	7173	38201	34395
MSWEB	6581	2345	7400
BOOK	10978	82637	37969
TMOVIE	20648	141193	149979
CWEBKB	11033	48777	64459
CR52	10410	109933	107556
C20NG	15793	117556	82876
BBC	12335	15080	36848
AD	12238	13152	17447

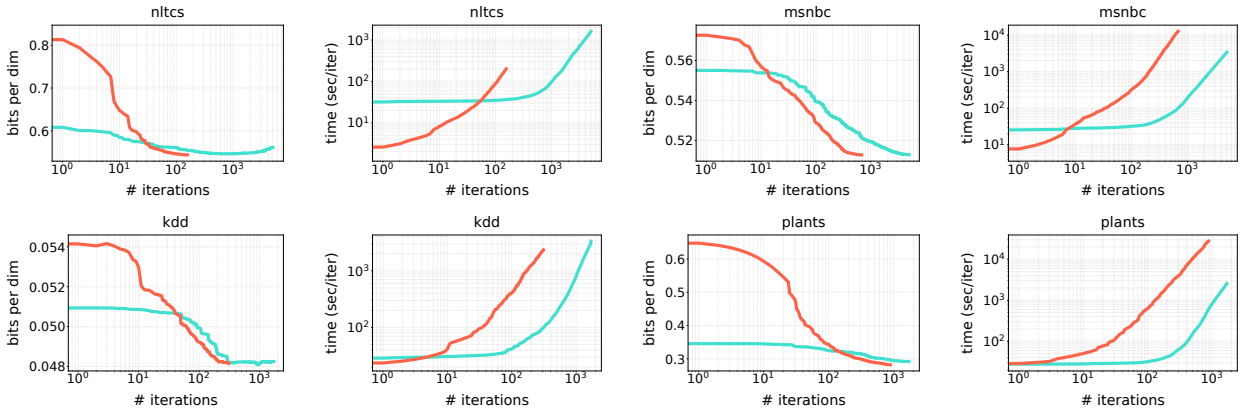
Table C.3: ***Density estimation benchmarks: single models***. The circuit sizes of best learned single models for LEARNPSDD STRUDELGREEDY and STRUDELBEAM.

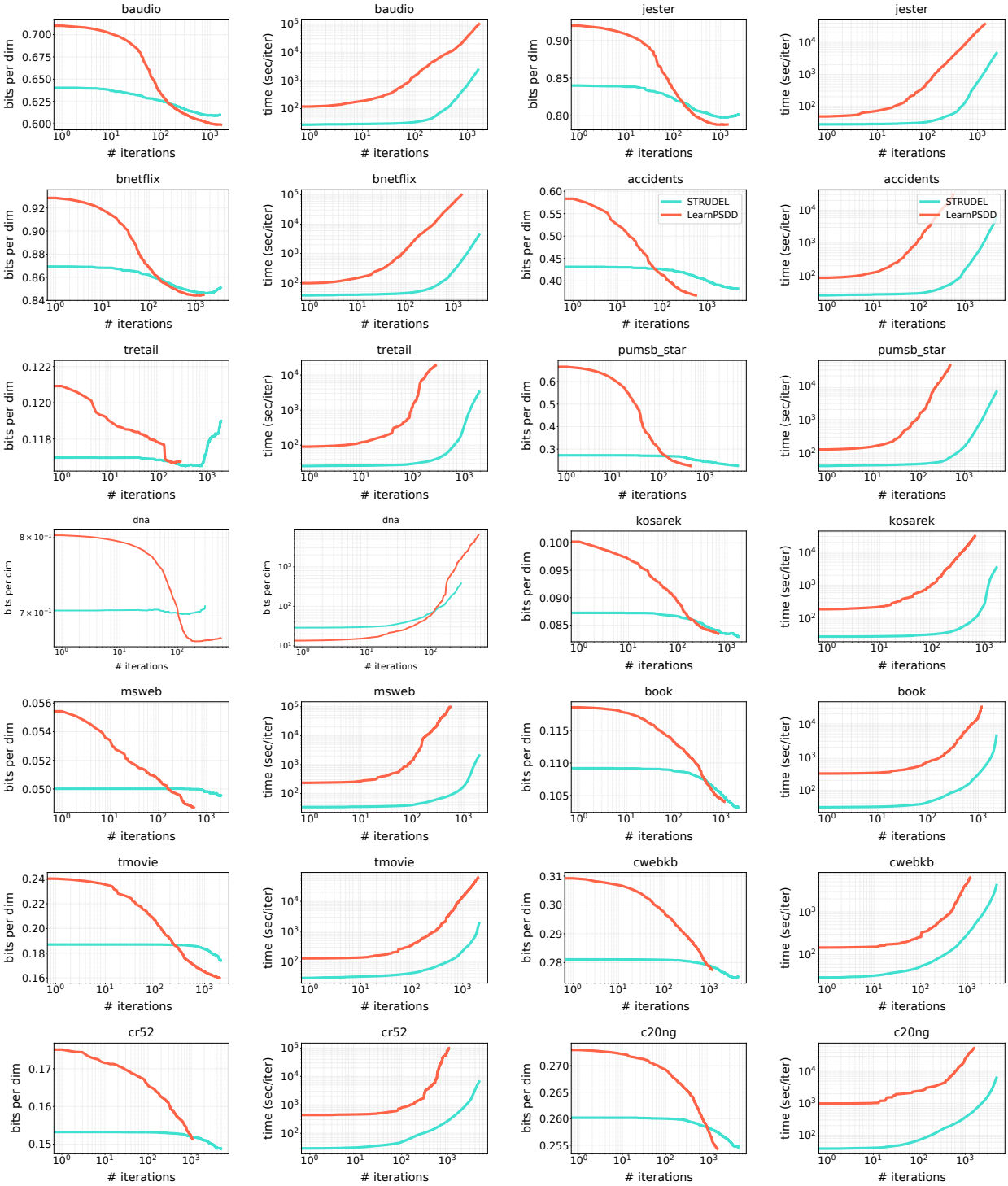
DATASET	LEARNPSDD SEC/ITER	STRUDELGREEDY SEC/ITER	LEARNPSDD SEC TOT	STRUDELGREEDY SEC TOT
NLTCS	1.3	0.1	193.0	42.1
MSNBC	18.6	0.5	12573.9	1930.3
KDD	7.5	0.4	2325.3	131.4
PLANTS	31.2	1.3	27438.4	1911.3
AUDIO	59.0	0.8	98457.1	848.4
JESTER	23.0	0.4	23573.1	279.6
NETFLIX	52.3	0.3	51300.7	436.4
ACCIDENTS	54.0	0.8	29664.2	3393.9
RETAIL	66.5	0.1	16018.3	44.2
PUMSB-STAR	83.7	1.1	40508.6	4737.3
DNA	3.0	0.5	595.4	86.6
KOSAREK	45.8	0.4	30196.8	398.9
MSWEB	176.6	4.3	93589.7	12.9
BOOK	28.2	1.1	32567.2	2316.8
EACHMOVIE	31.1	0.7	61792.3	1427.2
WEBKB	5.6	0.4	6333.0	884.3
ROUTERS-52	93.2	0.5	98796.0	1561.0
20NEWS-GRP	34.6	0.7	53829.2	2593.2
BBC	6.2	0.9	8000.8	2136.3
AD	3.7	1.1	3748.6	300.8

Table C.4: Total times (in seconds) taken (SEC TOT) and averaged times per iteration (SEC/ITER) to learn the best single models on each dataset for LEARNPSDD and STRUDEL. STRUDEL requires a fraction of the time of LEARNPSDD greatly speeding up learning.

Appendix C.2. Learning Times

To compare the efficiency of LEARNPSDD and STRUDELGREEDY, we reproduce the experimental setting of LEARNPSDD and rerun the single model experiments with the latest version of the LEARNPSDD code. Here we report the learning times – both the seconds per iteration (SEC PER ITER) and total seconds during learning (SEC SUM) – in Table C.4, from which it is clear that STRUDELGREEDY is more efficient than LEARNPSDD.




baudio
bits per dim
time (sec/iter)
jester
bnetflix
accidents
STRUDEL
LearnPSDD
tretail
pumsb_star
dna
kosarek
msweb
book
tmovie
cwebkb
cr52
c20ng
iterations

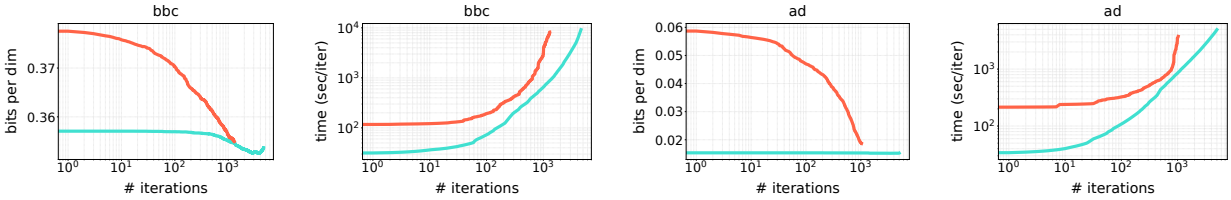


Table C.5: ***Comparing* STRUDEL *and* LEARNPSDD *single models performance and learning times.*** For each of the 20 benchmark datasets, we report 1) the test bits per dimensions (bpd) (y-axis) for each iteration (x-axis) on column 2 and column 4, and 2) seconds per iteration during learning (y-axis) for each iteration (x-axis) on column 2 and column 4 as scored by STRUDEL (blue) and LEARNPSDD (red). The plots for the same dataset is aligned next to each other as comparison.

Appendix C.3. Statistical Tests

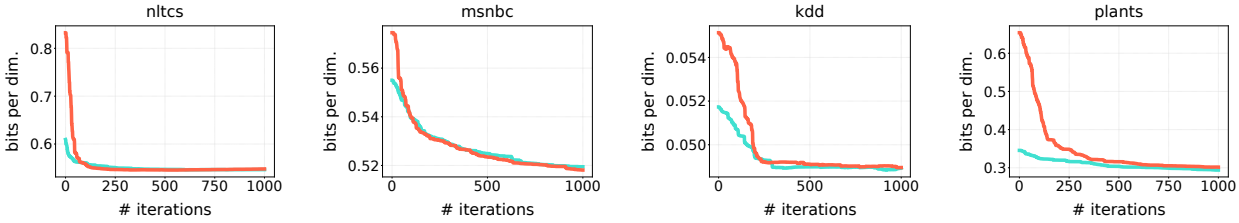
Since we only have the log-likelihood per sample available for SelSPN, we run pairwise Wilcoxon signed rank tests to compare STRUDEL and SelSPN more rigorously. The p-values are reported in Table C.9, in 4 out of 20 datasets, the results are statistically equivalent

DATASET	LEARNPSDD		SELSPN	
	STRUDELGREEDY	STRUDELBEAM	STRUDELGREEDY	STRUDELBEAM
NLTCS	**7.93E-01**	**2.31E-01**	1.24E-03	2.05E-03
MSNBC	0.00E+00	0.00E+00	0.00E+00	0.00E+00
KDD	0.00E+00	0.00E+00	0.00E+00	0.00E+00
PLANTS	7.78E-24	8.06E-08	3.33E-53	1.37E-27
BAUDIO	1.22E-33	5.82E-24	7.19E-42	5.81E-30
JESTER	6.98E-28	2.96E-26	8.07E-09	9.66E-08
BNETFLIX	1.10E-03	**6.55E-02**	5.26E-08	5.94E-05
ACCIDENTS	6.67E-125	5.51E-96	1.13E-297	1.76E-276
TRETAIL	**2.75E-01**	**2.75E-01**	1.08E-04	1.08E-04
PUMSB_STAR	1.70E-49	1.17E-20	1.31E-289	1.26E-274
DNA	1.05E-167	1.03E-155	5.15E-190	8.33E-190
KOSAREK	1.65E-03	5.28E-41	4.83E-21	**1.86E-01**
MSWEB	2.76E-100	9.34E-83	7.06E-114	2.40E-95
BOOK	**3.72E-02**	**2.20E-02**	**6.19E-01**	**9.30E-01**
TMOVIE	2.10E-54	1.92E-46	1.23E-45	1.31E-38
CWEBKB	2.06E-05	9.58E-06	**2.34E-01**	**2.06E-01**
CR52	3.72E-16	4.53E-17	2.18E-71	4.70E-70
C20NG	3.87E-03	**4.04E-02**	1.19E-08	2.04E-06
BBC	**6.74E-01**	2.24E-07	**1.24E-02**	**1.08E-02**
AD	9.90E-65	8.23E-66	1.64E-09	9.20E-08

Figure C.9: Pairwise Wilcoxon signed rank test p-values for the comparisons of test log-likelihoods for STRUDEL against LEARNPSDD SELSPN on all datasets. Bold values indicate the results are not statistically significant when picking confidence value 99%.

Appendix C.4. Initializations

Here we report the effect of different initialization methods (CLT and independent) on each dataset in Table C.6.



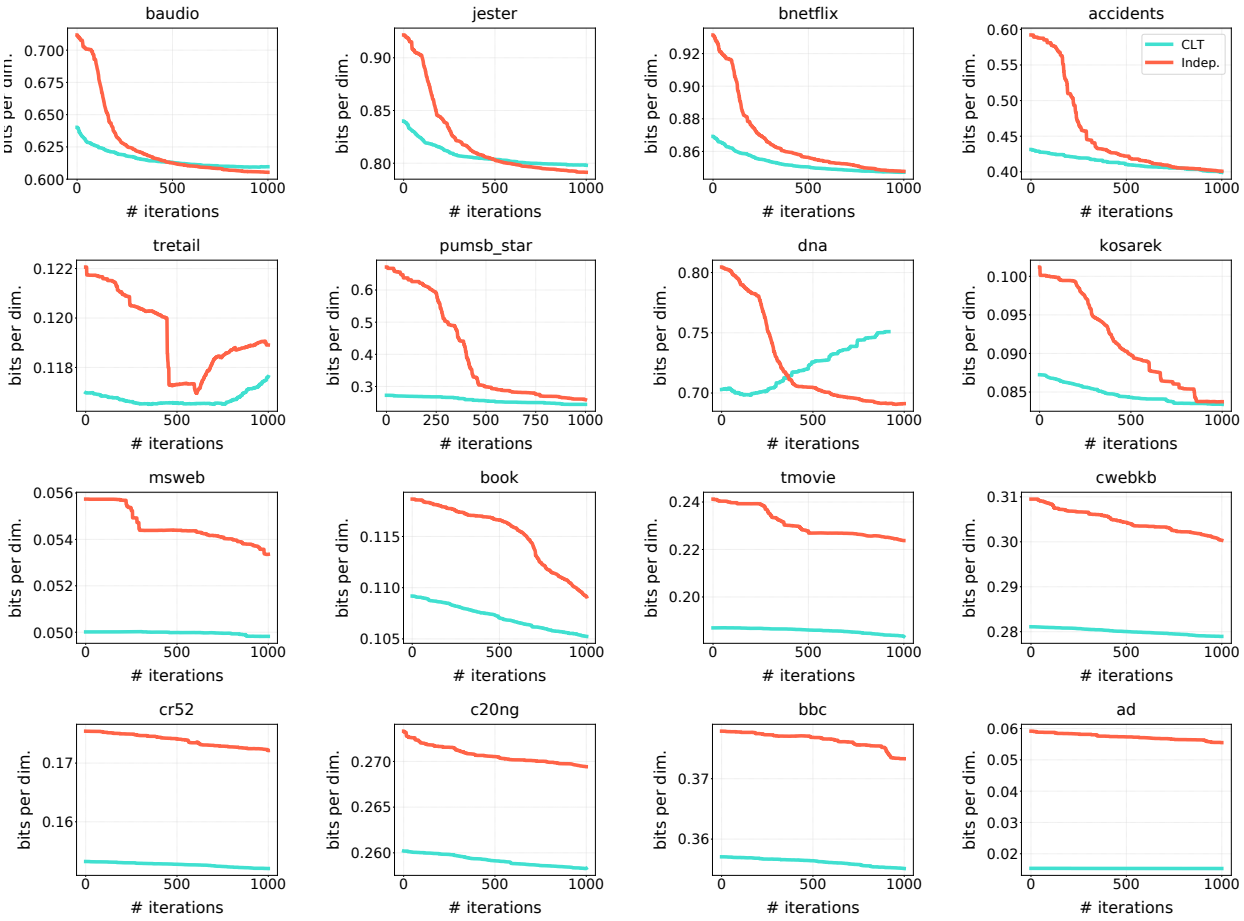
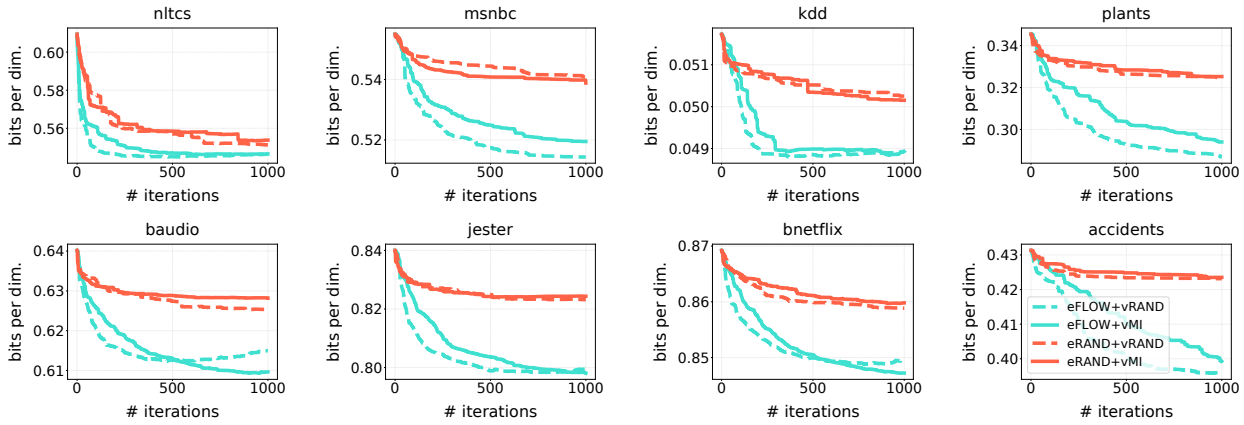


Table C.6: ***Effect of different initializations in* STRUDEL *for each dataset.*** We report the mean test bits per dimensions (bpd) on each dataset (y-axis) for each iteration (x-axis) as scored by the different initializations: CLT (blue) and independent (red).

Appendix C.5. Heuristics

Here we report the effect of different heuristics on each dataset in Table C.7.



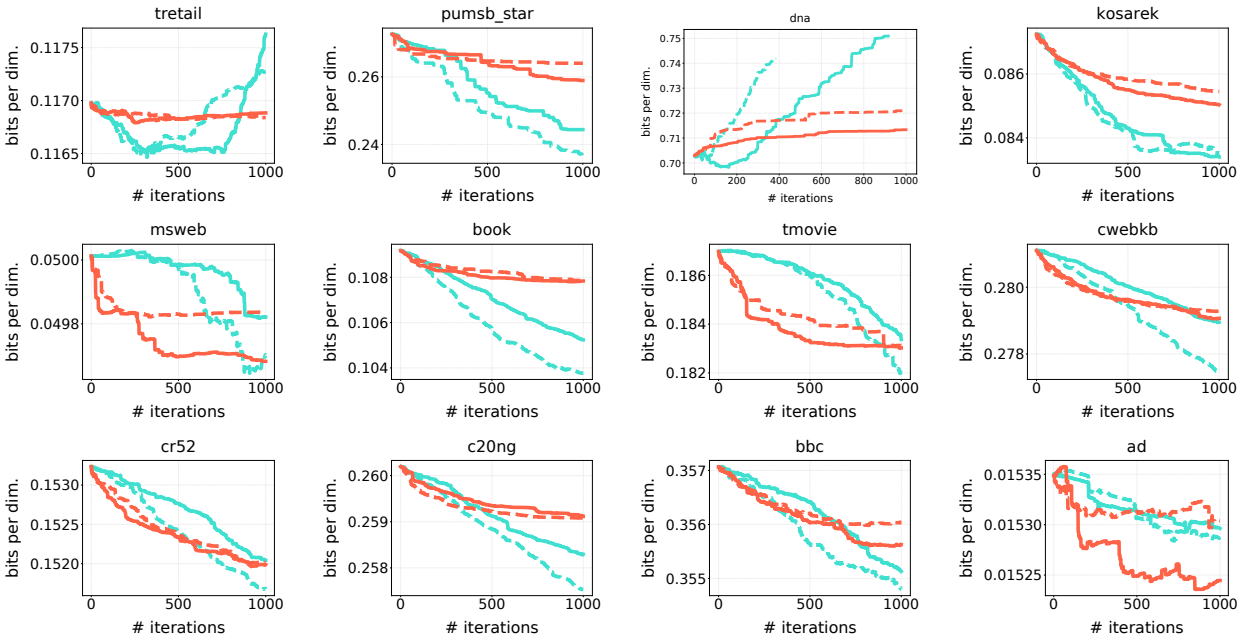
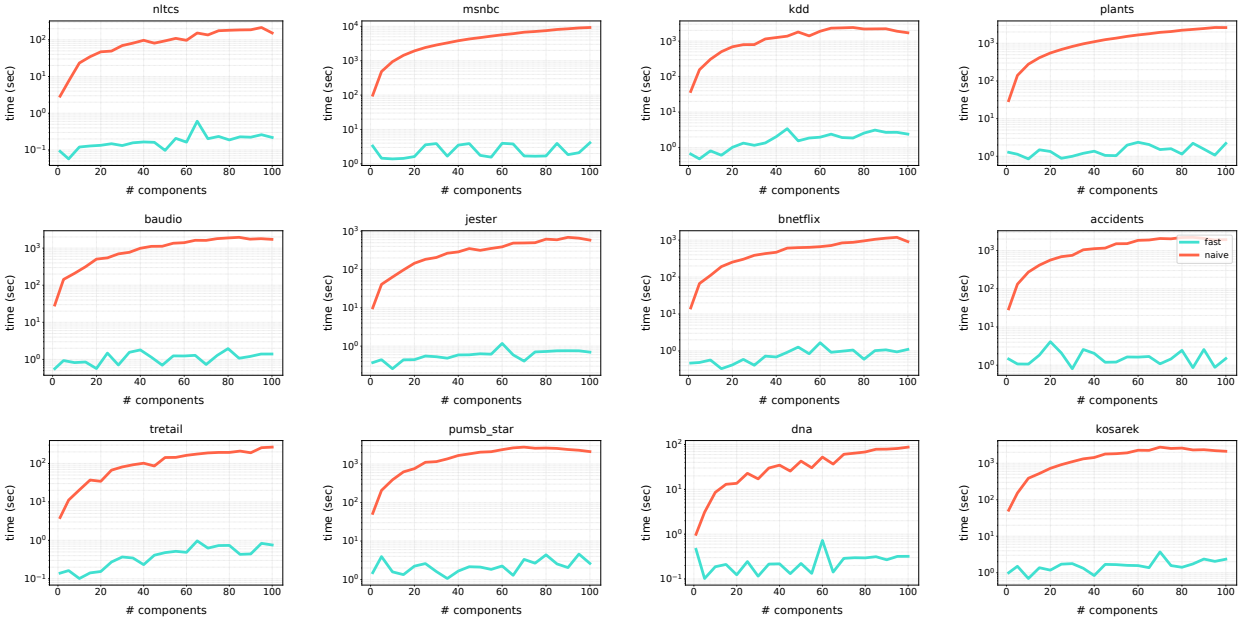


Table C.7: ***Effect of different heuristics in* STRUDEL *for each dataset.*** We report the mean test bits per dimensions (bpd) on each dataset (y-axis) for each iteration (x-axis) as scored by the 4 combinations of different heuristics.

Appendix D. Circuit Flows for Fast Inference

To empirically show that our shared circuit flows implementation (Section 3) benefits the efficient evaluation of ensembles, compared a vectorized version of the classical algorithm that evaluates the circuit bottom-up [10]. We report the time taken to compute likelihoods for an ensemble as a function of the number of components in Table D.8.



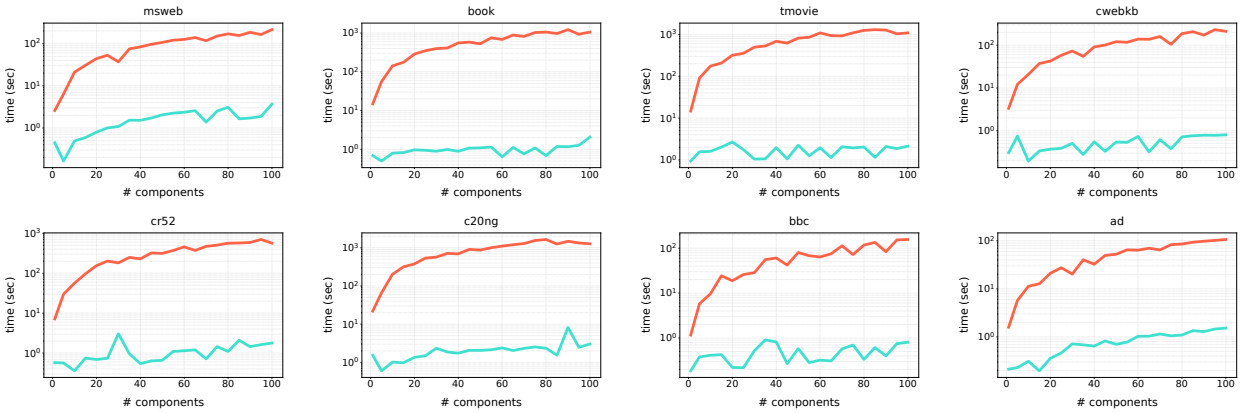
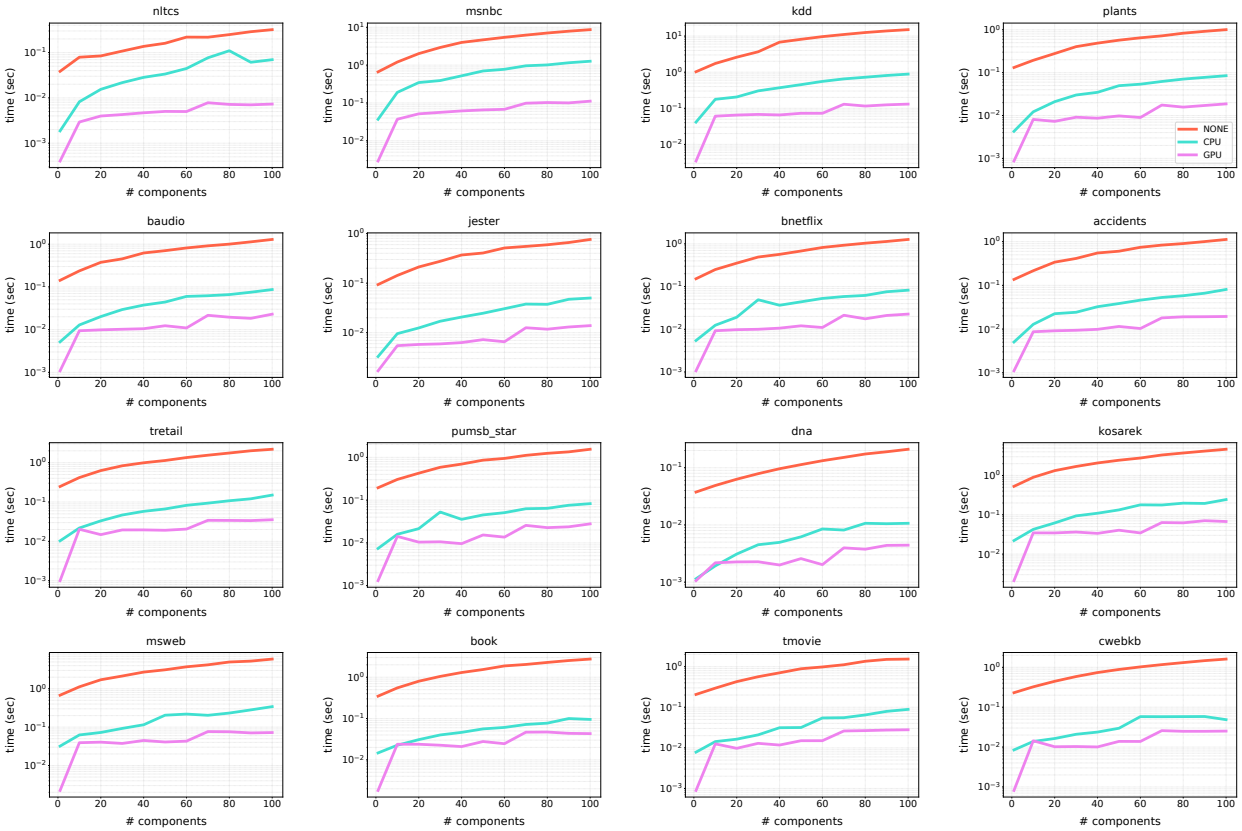


Table D.8: ***Circuit flows for fast inference of ensembles for each dataset.*** We report the time (seconds) used evaluating the ensemble circuits on each dataset (y-axis) for different number of component in the ensemble (x-axis), comparing the circuit flows implementation (blue) and the classical algorithm of naively evaluating the circuits bottom-up (red).

Appendix E. Parallelism Computation

To empirically show that CPU and GPU parallelism (Section 6) benefits the efficient learning of ensembles We report the time taken to perform one EM step for an ensemble as a function of the number of components in Table E.9.



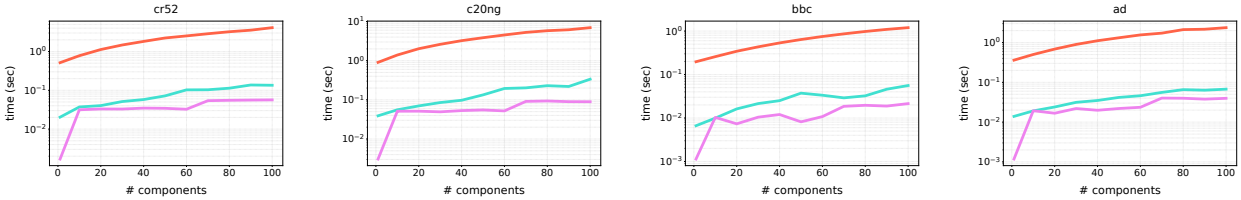


Table E.9: ***CPU and GPU parallelism for fast parameter learning of ensembles for each dataset.*** We report the time (seconds) used learning the ensemble circuits per iteration on each dataset (y-axis) for different number of component in the ensemble (x-axis), comparing the non-parallelism circuit flows implementation (red), CPU parallelism version (blue) and GPU parallelism version (violet).