Paths Not Taken: Structure-Based Pruning in PSDD Learning and Inference

Cory Butz, Alejandro Santoscoy-Rivero, Camilla Lewis

Department of Computer Science, University of Regina, Regina, SK, Canada
cory.butz@uregina.ca, asj799@uregina.ca, cel806@uregina.ca

Abstract

We make three novel contributions to parameter learning and inference in *probabilistic sentential decision diagrams* (PSDDs). First, rather than traversing the entire PSDD during parameter learning for each dataset example, we pioneer the use of determinism to focus only on the activated partition. Second, we demonstrate how to *prune* deterministic computation in inference, thereby eliminating the need to propagate probability over every node in the network for each query. Third, we introduce a technique that *parallelizes* a single circuit evaluation, rather than parallelizing individual multiplications or layer-wise inference. For both learning and inference, experimental results on benchmark PSDDs from various application domains demonstrate state-of-the-art performance.

Code — https://github.com/Santocoyo/Paths
Datasets — https://github.com/Santocoyo/Paths

Introduction

Probabilistic sentential decision diagrams (PSDDs) (Kisa et al. 2014) are an elegant framework for learning from and reasoning about data. They are tractable representations of discrete probability distributions over structured spaces defined by massive logical constraints. The effectiveness of PSDDs has been demonstrated in numerous real-world applications. For example, a traditional Naive Bayes classifier of a board game trace requires 362,879 parameters, whereas a PSDD only needs 1,793 parameters (Choi, Tavabi, and Darwiche 2016). Other successful applications of PSDDs include learning user preferences (Choi, Van den Broeck, and Darwiche 2015), anomaly detection (Choi, Tavabi, and Darwiche 2016), and route distribution modelling (Choi, Shen, and Darwiche 2017; Shen, Choi, and Darwiche 2018).

PSDDs can be learned from complete (Kisa et al. 2014) and incomplete (Choi, Van den Broeck, and Darwiche 2015) datasets. For complete datasets, the maximum likelihood PSDD parameters in closed form are unique. PSDDs provide a tractable representation of probability distributions because they are based on *sentential decision diagrams* (SDDs) (Darwiche 2011), which are a sophisticated representation of logical constraints in the form of a Boolean circuit. A PSDD is an SDD annotated with probabilistic parameters. PSDDs can also be compiled from graphical models such as *Bayesian networks* (BNs) (Pearl 1988).

As a PSDD can be written as an *arithmetic circuit* (AC) (Darwiche 2003), with and-gates and or-gates respectively replaced by multiplication and weighted summation operations, the terms PSDD, AC, and circuit may be used interchangeably. Even though AC inference is linear in the number of edges (Darwiche 2003), it has been argued that ACs cannot be efficiently implemented in GPUs. This is due to their sparseness (Shah et al. 2020) and high demands on memory bandwidth (Vasimuddin, Chockalingam, and Aluru 2018). Specialized hardware can be developed to cope with circuit irregularity (Shah et al. 2020, 2021), making these solutions cost-prohibitive. While utilizing field-programmable gate arrays alleviates the need for customized hardware, communication overhead persists as an obstacle to be overcome, as evidenced by its remaining as future work in a recent attempt to accelerate PSDD inference (Choi et al. 2023). A highly successful solution, called *knowledge layers* (KLay), for implementing sparse AC inference on GPUs that is independent of hardware, was presented in (Maene, Derkinderen, and Zuidberg Dos Martires 2025). However, it was stated that their solution is less effective on dense circuits, such as some of the PSDDs considered here that are compiled from BNs.

In this paper, we make three novel contributions to PSDD learning and exact inference. We pioneer the exploitation of *determinism* when learning the PSDD maximum likelihood parameters from complete datasets. By mapping prime literals a priori to their respective activated partitions, we focus learning on activated partitions rather than traversing the entire PSDD for each example, as is currently done. Second, this is the first work to put forth *structure-based* pruning of deterministic computation in AC inference. Here, pruning refers to pre-compiling a static schedule for a portion of the PSDD that can answer a substantial number of queries later at runtime, in stark contrast to answering every query by propagating probability over every node in the circuit. Lastly, we propose *parallelization* of evaluating a single circuit, whether full or partial, exploiting both cluster-wise and layer-wise parallelization. By introducing *cluster-wise* par-

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

L	K	P	A	Students
0	0	1	0	6
0	0	1	1	54
0	1	1	1	10
1	0	0	0	5
1	0	1	0	1
1	0	1	1	0
1	1	0	0	13
1	1	1	0	8
1	1	1	1	3

Table 1: A student enrolment dataset (Kisa et al. 2014).

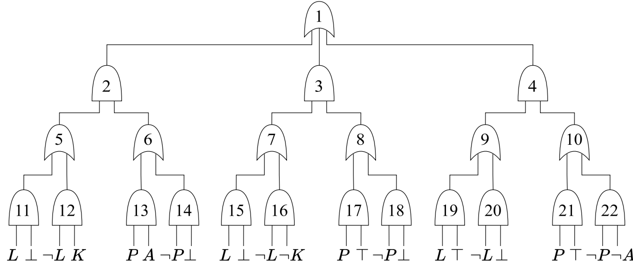

Figure 1: One SDD for the complete dataset in Table 1.

allelization in ACs, we extend this technique beyond models amenable to traditional join tree propagation to those that are not. For both learning and inference, experimental results on benchmark PSDDs, which span a range of densities and application domains, consistently demonstrate runtime improvements over the state of the art. All reported results are statistically significant under the Wilcoxon signed-rank test with p-value < 0.05 (Wilcoxon 1945).

Background Knowledge

Here, we review PSDD parameter learning and inference.

A *sentential decision diagram* (SDD) (Darwiche 2011) can be defined as follows (Kisa et al. 2014). An SDD is either a *decision node* or a *terminal node*. A terminal node is a literal, the constant $\top$ (true), or the constant $\bot$ (false). A decision node is a disjunction of the form $(p_1 \wedge s_1) \vee \cdots \vee (p_n \wedge s_n)$, where each pair (p_i, s_i) is called an *element*. An or-gate depicts a decision node, and and-gates depict its elements. Here, $p_1, \ldots, p_n$ are called *primes* and $s_1, \ldots, s_n$ are called *subs*. The primes of a decision node are always mutually exclusive, exhaustive, and consistent. And-gates will always have exactly two inputs.

For example (Kisa et al. 2014), consider a computer science department with four courses: Logic (L), Knowledge Representation (K), Probability (P), and Artificial Intelligence (A). Students enroll under the following three restrictions: a student must take at least one of Probability or Logic; Probability is a prerequisite for AI; and the prerequisite for Knowledge Representation is either AI or Logic. Given the dataset in Table 1, one SDD is shown in Figure 1.

A *probabilistic sentential decision diagram* (PSDD) (Kisa et al. 2014) is a parameterized SDD, i.e., the inputs into each or-gate are probabilities that sum to one. One PSDD for

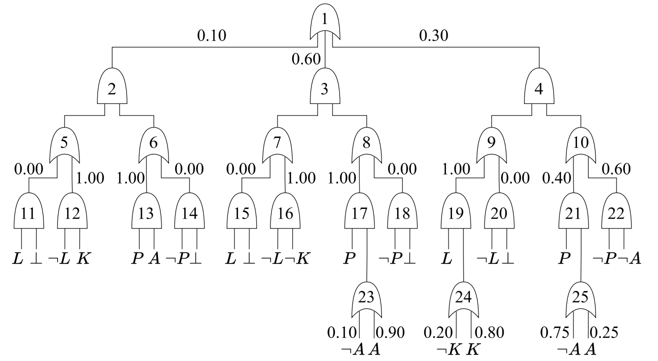

Figure 2: The PSDD showing maximum likelihood parameters for the the SDD in Figure 1 and the dataset in Table 1.

Algorithm 1: Parameter Learn (PL)

Input: An SDD, a dataset D
Output: A parameterized PSDD

1: Initialize the PSDD from the SDD
2: Initialize all PSDD parameters to zero
3: **for** each row r in D **do**
4:     **for** each partition **do**
5:         **if** partition is activated **then**
6:             Add row count to appropriate primes
7:             Add row count to appropriate subs
8:         **end if**
9:     **end for**
10: **end for**
11: Determine counts of the PSDD root or-gate
12: Normalize all or-gate counts as PSDD parameters
13: **return** parameterized PSDD

---

the SDD in Figure 1 is shown in Figure 2. PSDDs possess several favourable properties. The *decomposability* property prohibits inputs to every and-gate from sharing variables (Darwiche 2001a). *Smoothness* ensures that or-gate inputs involve the same variables (Darwiche 2001b). *Determinism* guarantees that at most one input for an or-gate can be high under any circuit input (Darwiche 2001b). This property corresponds to mutual exclusiveness when an or-gate is viewed as a disjunction of its inputs (Darwiche 2021). Here, a *partition* is defined as a sub-circuit rooted at any child node of the PSDD root.

The PSDD maximum likelihood parameters can be learned from complete datasets using the method in (Kisa et al. 2014). Algorithm 1 follows its implementation in the PyPSDD package (Choi, Arthur 2018).

Example 1 *Let us run Algorithm 1 to learn the PSDD parameters in Figure 2 from the student enrolment dataset in Table 1. Consider the last row* $(1, 1, 1, 1, 3)$ *in the dataset. Concerning the node numbers in Figure 2, Algorithm 1 visits nodes* N_1*,* N_2*,* N_5*, and* N_{12} *to determine that* $L = 0$ *and* $K = 1$*, indicating that this partition is not activated. It then visits nodes* N_3*,* N_7*, and* N_{16}*, where* $L = 0$ *and* $K = 0$*, again showing the partition is inactive. Finally, it visits nodes* N_4*,* N_9*, and* N_{19}*, confirming that the right par-*

tition is activated. It adds the row count 3 to the high wires of the prime or-gates N_9 and N_{24} in this partition. The subs are then traversed with the row count 3 added to the high wires for or-gates N_{10} and N_{25}. After repeating this process for the other dataset rows, the counts for the root or-gate are determined, and then the counts are normalized for each or-gate, yielding the maximum likelihood parameters illustrated in Figure 2.

Exact probabilistic inference $P(X = x)$ in a PSDD over variables U, where x is an instantiation of $X \subseteq U$, can be conducted linearly with one upward pass over the circuit by instantiating the leaves accordingly and propagating upwards to the root (Kisa et al. 2014).

Learning Exploiting PSDD Determinism

We suggest the first PSDD parameter learning algorithm exploiting determinism.

We begin by motivating our approach. The learning technique discussed in the previous section does not utilize PSDD properties and can be considered a brute-force approach. As determinism is satisfied, every dataset element will activate precisely one partition. For example, the first row $(0, 0, 1, 0, 6)$ in Table 1 activates the middle partition in Figure 2. Thus, PSDD parameter learning can be focused locally rather than needlessly traversing paths over the entire PSDD. This suggestion relies on dataset elements being quickly assigned to the proper partition, which can be accomplished in two phases by using a pre-compiled mapping of elements to partitions and bitwise checking at runtime.

In the first phase, we formally describe the offline pre-compilation of PSDD parameter learning as Algorithm 2. One traversal of the PSDD constructs a mapping of possible dataset elements to prime nodes in the activated partition. The restriction of dataset element e onto the prime variables X is denoted by $e^{\downarrow X}$. In addition, for a node N with k inputs, we use N^i to denote input i, for $i = 1, \dots, k$.

Example 2 *Let us run Algorithm 2 given the SDD in Figure 1. The set of all variables $\{L, K, P, A\}$ and the set of prime variables $\{L, K\}$ can be obtained from the SDD. The initial PSDD in Figure 2 is constructed, and all parameters are initialized to zero. Consider literals $\neg K$ and K in the right partition of Figure 2. Specify the mapping with the key-value pair*

$$\{(k_0) \mapsto [N_{24}^1]\},$$

and another mapping with one key-value pair

$$\{(k_1) \mapsto [N_{24}^2]\}.$$

Next, consider the or-gate, node N_{24}. The union of its two input mappings is computed as:

$$\{(k_0) \mapsto [N_{24}^1],\ (k_1) \mapsto [N_{24}^2]\}. \tag{1}$$

Now consider the and-gate node N_{19}. The left input is:

$$\{(l_1) \mapsto [N_9^1]\},$$

while the right input is the mapping in Equation (1). Thus, Algorithm 2 computes the Cartesian-product:

$$\{(l_1, k_0) \mapsto [N_9^1, N_{24}^1],\ (l_1, k_1) \mapsto [N_9^1, N_{24}^2]\},$$

which is the mapping for the right partition with L being 1 and K being either 0 or 1. Similarly, the respective mappings for the left and middle partitions are:

$$\{(l_0, k_1) \mapsto [N_5^2]\},$$

and

$$\{(l_0, k_0) \mapsto [N_7^2]\}.$$

Lastly, Algorithm 2 returns the PSDD mapping m:

$$m = \{\ (l_0, k_0) \mapsto [N_7^2],\ (l_0, k_1) \mapsto [N_5^2],\ (l_1, k_0) \mapsto [N_9^1, N_{24}^1],\ (l_1, k_1) \mapsto [N_9^1, N_{24}^2]\ \}.$$

Algorithm 2: Pre-compile PSDD Parameter Learn

Input: An SDD
Output: A map m of PSDD prime configurations to or-gate inputs in the activated partition

1: Determine the set of all variables from the SDD
2: Determine the set of prime variables from the SDD
3: Initialize a PSDD from the SDD
4: Initialize all PSDD parameters to zero
5: **for** each PSDD partition **do**
6: Bottom-up traversal of prime nodes in this partition
7: Let N be the next node to be processed
8: **if** node N is literal l **then**
9: Map l to the wire feeding into the nearest or-gate
10: **else if** node N is an or-gate **then**
11: Let $m_1, \dots, m_k$ be disjoint input mappings of N
12: $m = \bigcup_{i=1}^k m_i$, where $m(x) = m_i(x)$ for $i = 1, \dots, k$.
13: **else if** node N is an and-gate **then**
14: Let m_1 and m_2 be the two input mappings of N
15: $m = m_1 \times m_2 : (x_1, x_2) \mapsto (m_1(x_1), m_2(x_2))$
16: **end if**
17: **end for**
18: Process the root or-gate, yielding the final mapping m
19: **return** m

Algorithm 3: Deterministic Parameter Learn (Det-PL)

Input: A mapping m, a dataset D, prime variables X
Output: A parameterized PSDD

1: **for** each element e in D **do**
2: Add element count to primes cached in $m(e^{\downarrow X})$
3: Add element count to subs following Algorithm 1
4: **end for**
5: Determine counts of the PSDD root or-gate
6: Normalize all or-gate counts as PSDD parameters
7: **return** parameterized PSDD

In the second phase, runtime PSDD parameter learning is formally given as Algorithm 3.

Example 3 *Let us run Algorithm 3 to learn the PSDD parameters given the pre-compiled mapping m in Example 2, the complete dataset in Table 1, and prime variables $\{L, K\}$. Consider the last row e in the dataset. Algorithm 3*

extracts the values $e^{\downarrow(L,K)} = (1, 1)$ of the prime variables. Using this as the key for the mapping m in Example 2, the row count 3 is immediately added to the high wires N_9^1 and N_{24}^2 of the prime or-gate nodes N_9 and N_{24}. Next, the row count 3 is added to high wires for sub or-gates following Algorithm 1 as in Example 1. The rest of the example follows a similar pattern.

The important point in Example 3 is that Algorithm 3 did not needlessly traverse PSDD paths. By exploiting determinism, only paths within the activated partition for each dataset element are explored during PSDD parameter learning.

Theorem 1 *Given an SDD for a complete dataset, Algorithms 2 and 3 correctly learn the PSDD maximum likelihood parameters.*

Proof. Since Algorithm 1 is known to learn the PSDD maximum likelihood parameters, it only needs to be shown that Algorithms 2 and 3 add a row count to a PSDD node if and only if Algorithm 1 does. By line 3 in Algorithm 3, row counts are added to the subs of the activated partition in the exact same manner. Algorithm 1 will traverse partitions that are not activated by the dataset row currently under consideration. Since the row count will not be added to any of the nodes in non-active partitions, it is only necessary to demonstrate that the algorithms perform the same work within the prime nodes of the activated partition. In lines 8 and 9 of Algorithm 2, the literal value of every leaf in each PSDD sub-circuit is mapped to its truth value. As the inputs to an or-gate are mutually exclusive and exhaustive, there can be no conflict between the keys of the mappings of the two inputs. Thus, lines 10 to 12 of Algorithm 2 correctly record activated or-gates given a dataset element. Similarly, since PSDD and-gates are decomposable, the Boolean variables of the left input are disjoint from those of the right input. Hence, the Cartesian-product of the key-value pairs in lines 13 to 15 of Algorithm 2 correctly record activated and-gates. Consequently, during runtime, line 2 in Algorithm 3 adds the row count precisely to the necessary prime nodes of the activated partition. Thereby, Algorithms 2 and 3 are equivalent to Algorithm 1. ■

Inference Exploiting Determinism

Here, we cluster AC nodes to avoid deterministic computation and show how clustering facilitates parallelization.

AC *indicators* are binary leaf nodes corresponding to PSDD literals and are set to zero or one for each query. A zero means an indicator is incompatible with the query, and a one means it is compatible. Although every AC node is relevant to at least one query, its numeric value is deterministic whenever all descendant indicators are set to one.

Definition 1 *In an AC, let N be a non-leaf node and I be the set of all descendant indicators of N. N is called* deterministic *whenever all indicators in I are set to 1.*

A deterministic node separates all possible AC queries into two halves: one in which the node is deterministic and one in which it is not. Discussion of determinism at the level of one AC node is too fine-grained to be useful in practice. Thereby, we group together AC nodes into *clusters*.

Our clustering procedure follows a fixed node evaluation order used by the *Arithmetic Circuit Evaluator* (ACE) software package (UCLA Automated Reasoning Group 2015). While AC inference proceeds bottom-up from leaves to the root, the traversal order is not uniquely determined by the circuit structure. We therefore adopt one node evaluation sequence in which each step corresponds to a literal, multiplication, or summation operation in one circuit evaluation.

Definition 2 *Given the AC inference sequence generated by ACE for a PSDD, a* cluster *is a consecutive sequence of one or more multiplication operations. This sequence must be immediately preceded by either a literal statement or a summation operation. If the sequence is followed by a summation operation, the cluster includes all subsequent consecutive summation operations as well.*

This notion of a cluster seems to be a natural pattern in AC inference conducted by ACE. Note, however, that a cluster does not necessarily correspond to classical variable elimination in a BN, since ACE employs a logical optimization approach to compile a BN when its treewidth is high (UCLA Automated Reasoning Group 2015).

Algorithm 4 first groups the nodes of the AC compiled by ACE into clusters, producing the full clustered AC $\mathcal{C}_{full}$. Here, $\mathcal{C}_{full}$ (line 3) is a directed acyclic graph whose nodes are the clusters $c_1, \ldots, c_k$, and a directed edge (c_i, c_j) is added whenever cluster c_i depends on the numeric output produced by c_j. The algorithm then prunes deterministic computation to obtain the partial clustered AC $\mathcal{C}_{partial}$, where I_C denotes the set of indicator variables appearing in a given set of clusters C. Line 12 performs a disjointness check to ensure that a cluster and its descendants share no indicator variables with the rest of $\mathcal{C}_{partial}$. Provided that the rest of $\mathcal{C}_{partial}$ has more than half of the original variables, line 14 prunes away these clusters from $\mathcal{C}_{partial}$.

Example 4 *Given the PSDD for the real-world BN* Mildew *(Jensen and Jensen 1996) over U with 35 variables, we apply Algorithm 4 to compute the clustered ACs $\mathcal{C}_{full}$ and $\mathcal{C}_{partial}$. The full clustered AC $\mathcal{C}_{full}$ is formed as a directed acyclic graph with 34 clusters, as illustrated in Figure 3. A copy $\mathcal{C}_{partial}$ is then initialized from $\mathcal{C}_{full}$ in line 4, and the pruning loop begins in line 5.*

Consider cluster c_{30} during the top-down traversal. In line 9, C_{30} is defined as cluster c_{30} together with its fifteen descendant clusters shown in Figure 3. The set $\mathcal{C}_{partial} \setminus C_{30}$ therefore contains all clusters in the smaller rectangle of Figure 3, as well as cluster c_{12}. The corresponding indicator sets $I_{C_{30}}$ and $I_{\mathcal{C}_{partial} \setminus C_{30}}$ are disjoint, satisfying the condition in line 12 (indicators are not depicted in Figure 3). Since $|I_{\mathcal{C}_{partial} \setminus C_{30}}| > |U|/2$ in line 13, C_{30} is pruned from $\mathcal{C}_{partial}$.

The next interesting cluster is c_{13}. Here, $C_{13} = c_{13}$, but the indicator sets are not disjoint because one variable has indicators appearing in both c_{13} and c_{14}. Thus, the disjointness condition in line 12 fails. Next, for cluster c_{12}, $C_{12} = c_{12}$, and the indicator sets are disjoint. Since $|I_{\mathcal{C}_{partial} \setminus C_{12}}| > |U|/2$, c_{12} is pruned from $\mathcal{C}_{partial}$.

No further pruning applies, and Algorithm 4 returns both the clustered ACs $\mathcal{C}_{full}$ and $\mathcal{C}_{partial}$, as depicted in Figure 3.

Algorithm 4: Compile Clustered ACs from a PSDD

Input: A PSDD $\mathcal{P}$ over variables U
Output: $\mathcal{C}_{full}$ and $\mathcal{C}_{partial}$

1: Let $\mathcal{A}$ be the AC compiled by ACE for $\mathcal{P}$
2: Let $c_1, \ldots, c_k$ be $\mathcal{A}$'s clusters according to Definition 2
3: Let $\mathcal{C}_{full}$ be the directed acyclic graph whose nodes are $\{c_1, \ldots, c_k\}$, with a directed edge (c_i, c_j) iff cluster c_i depends on numeric output from c_j
4: $\mathcal{C}_{partial} \leftarrow \mathcal{C}_{full}$
5: **for** $j = k, \ldots, 1$ **do**
6: **if** $c_j \notin \mathcal{C}_{partial}$ **then**
7: **continue**
8: **end if**
9: Let C_j be the subgraph induced by cluster c_j and all its descendants in $\mathcal{C}_{partial}$
10: Let I_{C_j} be the set of indicator variables in C_j
11: Let $I_{\mathcal{C}_{partial} \setminus C_j}$ be the set of indicator variables over clusters in $\mathcal{C}_{partial} \setminus C_j$
12: **if** $I_{C_j} \cap I_{\mathcal{C}_{partial} \setminus C_j} = \emptyset$ **then**
13: **if** $|I_{\mathcal{C}_{partial} \setminus C_j}| > |U|/2$ **then**
14: $\mathcal{C}_{partial} \leftarrow \mathcal{C}_{partial} \setminus C_j$
15: **end if**
16: **end if**
17: **end for**
18: **return** clustered ACs $\mathcal{C}_{full}$ and $\mathcal{C}_{partial}$

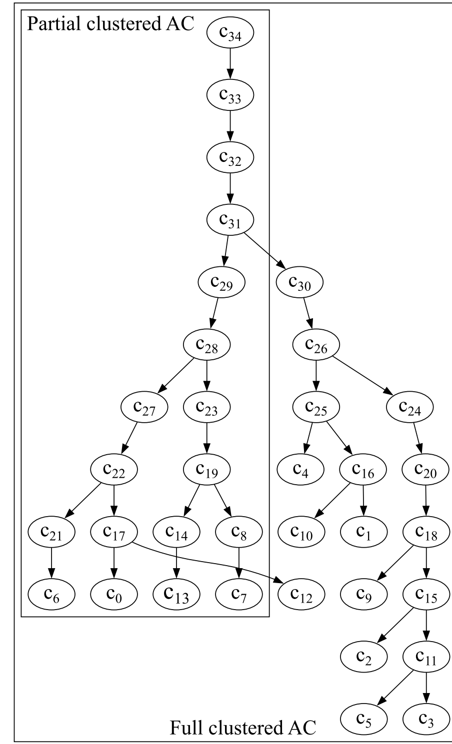

Figure 3: Given the Mildew PSDD, Algorithm 4 computes the full and partial *clustered* ACs $\mathcal{C}_{full}$ and $\mathcal{C}_{partial}$.

We now turn our attention from pruning to parallelization. *Layer-based* parallelization of probabilistic inference was suggested as long ago as (Kozlov and Singh 1994) and is still used today (Maene, Derkinderen, and Zuidberg Dos Martires 2025). We use layer-based parallelization when scheduling for clustered ACs $\mathcal{C}_{partial}$ and $\mathcal{C}_{full}$.

Example 5 *Consider exploiting layer-based parallelization when performing a single circuit evaluation on the partial clustered AC $\mathcal{C}_{partial}$ in Figure 3. The leaf clusters c_0, c_6, c_7, and c_{13} form the first layer of the parallel schedule. These four clusters can be processed concurrently since each is independent of the others. Once this layer has been completed, the next layer, consisting of c_8, c_{14}, c_{17}, and c_{21}, can be started. In the next round, each of these four clusters can be processed in parallel, and so on.*

Experience has shown that layer-wise parallelization is not as effective as parallelism within clusters (Kozlov and Singh 1994; Madsen and Jensen 1999) when inference is implemented using join tree propagation. ACs, however, allow inference to scale sometimes to models with very high treewidth, which are beyond the scope of classical inference algorithms such as variable elimination and join tree propagation (Shen, Choi, and Darwiche 2016). For instance, ACE was the only system to exactly solve all 251 relational networks in the UAI'08 inference competition (Darwiche et al. 2008). Thus, we propose integrating layer-wise parallelism with *cluster-wise* parallelism for AC inference.

The computation within each cluster is partitioned into at most one serial block and zero or more parallel blocks.

Definition 3 *Given ACE's node evaluation sequence, we define the following within each cluster. The* serial block *is the set of all nodes that are referenced more than once during evaluation, along with all their descendant nodes. For each root node in the cluster, a* parallel block *is defined as the root node together with all of its descendant nodes that are not part of the serial block.*

For example, the cluster c_1 in Figure 4 has 15 numeric operations, partitioned into one serial and two parallel blocks, respectively.

Better load balancing can be achieved only by dividing the calculation inside a given cluster between different processors (Kozlov and Singh 1994). In our case, scheduling within clusters will be done by blocks, with all parallel blocks waiting until the serial block has completed.

Algorithm 5, called *Parallelized Circuit Evaluation* (PCE), constructs a static time-ordered schedule for evaluating a clustered AC in parallel. Each cluster is decomposed into one serial block and zero or more parallel blocks, which are assigned to jobs according to a fixed-capacity constraint. The parameter $J_{\max}$ denotes the job limit, i.e., the maximum number of numeric operations (additions and multiplications) that may be included in a single job. A job thus represents a bounded-capacity container of operations scheduled for concurrent execution within one time unit. Within each time unit, jobs are indexed sequentially, e.g., $\mathcal{S}[t, j]$ denotes job j in time unit t. Thus, we may write $\mathcal{S}[t, j]$ as $\mathcal{S}[t]$ with the job number j understood, since job numbers

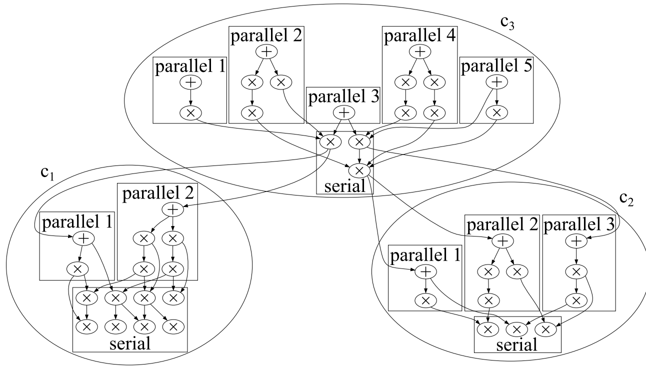

Figure 4: Illustrating PCE's integration of *cluster*- and layer-based parallelization to help cope with circuit irregularity.

are unique in each time unit. For every cluster, the serial block must complete before any parallel blocks can start. For each serial block, up to the first J_{max} numeric operations are scheduled in the current time unit, with consecutive time units used to schedule jobs for any remaining operations. Each parallel block of this cluster is scheduled either in the same job as the last operation of the serial block in time unit t or in the current or a new job of time unit $t+1$. In our experience, each parallel block fit entirely within a single job. Note that PCE checks whether a new block fits only within the most recently created job in each time unit, not in earlier jobs of that time unit.

Example 6 *Consider how Algorithm 5 schedules evaluation over a clustered AC $\mathcal{C}$, where the bottom layer has two clusters, c_1 and c_2, with a parent cluster c_3 in the next layer, as depicted in Figure 4. Let $c_1 = [8, [2, 5]]$, $c_2 = [3, [2, 4, 3]]$, and $c_3 = [3, [2, 4, 1, 5, 2]]$, where the first value denotes the number of numeric operations in the serial block, and the list specifies the sizes of its parallel blocks. Assume a job limit of $J_{\max} = 6$, meaning that each job can contain at most six numeric operations. By line 2, scheduling proceeds in a bottom-up, layer-wise manner.*

Consider cluster c_1 in line 4, and then set $t_1 = 1$. The serial block of c_1 contains 8 operations. By line 6, PCE first creates job $\mathcal{S}[1, 1]$ (job 1 in time unit 1) and assigns six operations, filling its capacity. The remaining two operations are placed in $\mathcal{S}[2, 1]$ (job 1 in time unit 2), by line 9. Next, the first parallel block of size 2 is considered in line 12. By line 13, since it can fit within the remaining capacity of $\mathcal{S}[2, 1]$ ($2 < 4$), it is scheduled there, leaving room for two additional operations. However, the second parallel block of size 5 does not fit in $\mathcal{S}[2, 1]$ ($5 > 2$). By line 17, PCE creates $\mathcal{S}[3, 1]$ and schedules it there. Cluster c_1 is now complete.

Now consider cluster c_2 in line 4. By line 5, $t_2 = 1$. The serial block of c_2 (3 operations) is scheduled in a new job $\mathcal{S}[1, 2]$, leaving space for three more operations ($6-3 = 3$). The first parallel block (size 2) is placed in $\mathcal{S}[1, 2]$, reducing the remaining space to one. The second parallel block (size 4) does not fit in $\mathcal{S}[1, 2]$ or in $\mathcal{S}[2, 1]$, so PCE creates $\mathcal{S}[2, 2]$ for it. The third parallel block (size 3) similarly requires a new job, $\mathcal{S}[2, 3]$. At this point, both c_1 and c_2 are scheduled, completing the bottom layer.

A new layer begins in line 2. For cluster c_3, the time unit is set as $t_3 = 4$. The serial block (size 3) is scheduled in $\mathcal{S}[4, 1]$, leaving three remaining operations ($6 - 3 = 3$). The first parallel block (size 2) is scheduled in the same job, leaving one slot. The second parallel block (size 4) exceeds this remaining capacity, so PCE creates $\mathcal{S}[5, 1]$ for it, leaving two available operations. The third parallel block (size 1) fits in $\mathcal{S}[4, 1]$. The fourth (size 5) and fifth (size 2) parallel blocks are scheduled as $\mathcal{S}[5, 2]$ and $\mathcal{S}[5, 3]$, respectively. Cluster c_3 is now fully scheduled, completing the schedule $\mathcal{S}$ to this point.

Algorithm 5: Parallelized Circuit Evaluation (PCE)

Input: Layered clusters $\{\mathcal{L}_1, \ldots, \mathcal{L}_m\}$, each cluster c_i has a serial block s_i and zero or more parallel blocks $[p_{i0}, \ldots, p_{ik}]$
Parameter: Job limit $J_{\max}$ // maximum ops per job
Output: Parallel schedule $\mathcal{S}$

```
 1: Initialize empty schedule S and time unit t
 2: for each layer L_ℓ from bottom to top do
 3:     t ← the next time unit having no scheduled jobs
 4:     for each cluster c_i ∈ L_ℓ do
 5:         t_i ← t
 6:         Schedule min(J_max, s_i) numeric operations of s_i in S[t_i]
 7:         while s_i > 0 do
 8:             t_i ← t_i + 1   // continue serial in next time unit
 9:             Schedule min(J_max, s_i) remaining operations of s_i in S[t_i]
10:         end while
11:         remaining ← available space in S[t_i]
12:         for each parallel block p_ij of c_i do
13:             if p_ij ≤ remaining then
14:                 Schedule p_ij in S[t_i]
15:                 remaining ← remaining − p_ij
16:             else
17:                 Schedule p_ij in S[t_i + 1]
18:             end if
19:         end for
20:     end for
21: end for
22: return cluster- and layer-wise parallel schedule S
```

PCE is a static approach to scheduling and is an offline compilation prior to runtime inference, as done in (Choi et al. 2023). Offline compilation exploits the fact that the PSDD structure of a dataset is fixed, even for different queries. Then, at runtime, simply execute the corresponding pre-compiled PCE schedule for either $\mathcal{C}_{full}$ or $\mathcal{C}_{partial}$.

Example 7 *Consider an input query $P(X = x)$ posed to the Mildew PSDD. The partial clustered AC $\mathcal{C}_{partial}$ in Figure 3 for Mildew contains the following 18 variables: dm$_2$, dm$_3$, dm$_4$, foto$_2$, foto$_3$, foto$_4$, lai$_2$, lai$_3$, lai$_4$, meldug$_3$, meldug$_4$, middel$_3$, mikro$_3$, straaling$_3$, straaling$_4$, temp$_3$, temp$_4$, udbytte. If X is a subset of these 18 variables, execute the pre-compiled PCE schedule for $\mathcal{C}_{partial}$ in Figure 3; otherwise, execute the pre-compiled PCE schedule for $\mathcal{C}_{full}$.*

Theorem 2 *Given a PSDD defining a joint distribution $P(U)$ and $X \subseteq U$, PCE correctly answers exact inference queries of the form $P(X = x)$.*

Proof. The claim holds whenever an input query is processed in $\mathcal{C}_{full}$, since PCE is faithful to a bottom-up traversal of the underlying AC with children evaluated before their parents. Thus, consider the case where $P(X = x)$ is processed in the partial clustered AC $\mathcal{C}_{partial}$. All indicators in $U \setminus X$ are set to 1 by definition. Thus, all subsequent numeric computations based on these indicators are deterministic. Hence, the same numeric values are obtained regardless of whether the computation is performed at runtime or in a clustered, pre-compiled environment. ■

Experimental Results

We present experimental results aimed at evaluating the effectiveness of exploiting determinism in PSDD parameter learning and inference.

The empirical evaluation was conducted on a laptop running Windows 11 Pro (version 23H2), equipped with a 3.3 GHz AMD Ryzen 5 5600H processor (six physical cores, twelve logical cores), integrated with an NVIDIA GeForce GTX 1650 (G6) GPU, and 64 GB of RAM. Software used includes KLay (v0.0.2), ACE (v3.0), PyPSDD (v0.1) and PyPy (v2.7) as the recommended interpreter. KLay inference is conducted in Python (v3.12). The schedule output by PCE is written in C++23 and compiled using G++ (v14.2).

Table 2 describes five PSDDs for learning user preferences used in our experiments. For context, the benchmark *Sushi* dataset (Kamishima 2003) contains the preference rankings for ten types of sushi by 5,000 people. As total rankings correspond to complete datasets, PyPSDD (Choi, Arthur 2018) was used to learn PSDDs as in (Choi, Van den Broeck, and Darwiche 2015). The average running times and standard deviations to learn the PSDD maximum likelihood parameters by PL and Det-PL are reported in seconds, where 10 runs were performed for each dataset size. To evaluate scalability, we use three synthetic datasets that span three orders of magnitude.

Table 2 shows that Det-PL was always faster than PL regardless of the number of elements in the dataset and the number of PSDD partitions. All boldface winners are statistically significant under the Wilcoxon signed-rank test with p-value < 0.05. This empirically demonstrates the effectiveness of exploiting determinism when learning the maximum likelihood PSDD parameters.

The PSDDs in Table 3 used to evaluate inference performance were compiled from benchmark BNs (Scutari, Marco 2007), with AC sizes ranging from 627,098 to over 18 million nodes. PCE was compared with ACE, a well-established AC evaluator that remains state-of-the-art according to a recent empirical study (Agrawal, Pote, and Meel 2021), and with KLay (Maene, Derkinderen, and Zuidberg Dos Martires 2025), a recent GPU-accelerated evaluator showing competitive performance. Each experiment involved ten randomly generated queries, where the query scope was set to 5% of the network variables. Each query was executed ten times using the three approaches. To isolate the effects of pruning and parallelism, we conducted evaluations of (i) parallel inference over the full clustered AC (no pruning) and (ii) serial inference over the partial clustered AC (no parallelism). In our experiments, a job limit of $J_{\max} = 20{,}000$ operations provided a good balance between load granularity and scheduling overhead. Reported results represent averages over 100 runs, with standard deviations provided as customary. All boldface winners are statistically significant under the Wilcoxon signed-rank test with p-value < 0.05, where only the top two methods per benchmark were compared to reduce the likelihood of spurious conclusions.

The final line of Table 3 indicates that PCE attains the state-of-the-art, as it is faster than both ACE and KLay on four of the five PSDDs. KLay was the fastest on one PSDD. These results emphasize the worthiness of exploiting a fixed structure to pre-compile a static parallelized schedule.

(Maene, Derkinderen, and Zuidberg Dos Martires 2025) demonstrated that sparse ACs can be executed efficiently on GPUs, yielding brilliant results. However, it was acknowledged that their approach is less effective on dense circuits. The results in Table 3 show that KLay is not as effective on ACs compiled from benchmark BNs. PCE is also faster than ACE because ACE evaluates every node one by one, whereas PCE clusters nodes to exploit layer- and cluster-wise parallelization.

Restricting attention to cases where a query can be evaluated in a pruned AC, a serial execution was faster than PCE in two of the networks, while PCE was faster in the other three networks. A serial execution can be faster on smaller networks, as the overhead of parallelization may outweigh its benefits. These results show that structure-based pruning of deterministic computation is significant in its own right, and that it further complements PCE.

Our static scheduling approach requires the identification of the computation per task, the sets of parallelizable tasks, and the job limit hyperparameter. We reuse shared arrays since traversal is fully known in advance. While this approach avoids explicit handling of memory management, data locality, and thread synchronization, future work could investigate cache-aware array partitioning and adapting cluster- and layer-based parallelization to GPUs.

Lastly, it is worth noting that *query-based* pruning techniques applied during runtime have been developed for BNs (Zhang and Poole 1994) and for sum–product networks (Butz et al. 2018), whereas the work here introduces a *structure-based* pruning technique utilized in a precompilation step. Rather than pruning dynamically for a single query, the partial clustered AC $\mathcal{C}_{partial}$ in Figure 3 can be reused across an enormous family of queries. Specifically, $\mathcal{C}_{partial}$ contains 18 observable variables whose discrete domains range in cardinality from 4 up to 100. Considering all possible non-empty subsets of these 18 variables and all combinations of their domain values yields approximately 6.4×10^{20} distinct instantiations, i.e., more than 640 quintillion unique queries $P(X = x)$. Thus, the precompiled $\mathcal{C}_{partial}$ can correctly answer every one of these queries without recompilation, demonstrating the scalability and generality of the proposed pruning method.

	Rank-16	Rank-12	Rank-11	Rank-10	Rank-9
# vars	256	144	121	100	81
# nodes	331,430	22,258	11,660	6,230	3,405
PL on 10,000 rows	4.49 ± 0.08	1.64 ± 0.02	1.37 ± 0.03	1.19 ± 0.01	0.97 ± 0.03
Det-PL on 10,000 rows	**2.74 ± 0.16**	**0.94 ± 0.01**	**0.78 ± 0.02**	**0.58 ± 0.01**	**0.39 ± 0.01**
PL on 100,000 rows	26.31 ± 0.83	10.09 ± 0.11	7.27 ± 0.05	4.56 ± 0.12	2.36 ± 0.04
Det-PL on 100,000 rows	**23.65 ± 1.30**	**9.19 ± 0.10**	**6.12 ± 0.03**	**3.39 ± 0.07**	**1.56 ± 0.02**
PL on 1,000,000 rows	248.07 ± 6.94	76.53 ± 1.11	45.64 ± 3.36	18.24 ± 0.18	5.34 ± 0.12
Det-PL on 1,000,000 rows	**216.91 ± 4.35**	**74.76 ± 2.75**	**42.02 ± 3.66**	**16.62 ± 0.17**	**4.24 ± 0.04**

Table 2: PSDD learning times (in seconds) by PL and Det-PL for user preference total rankings. Boldface indicates the best-performing method with statistical significance based on the Wilcoxon signed-rank test with p-value < 0.05.

	Pigs	Andes	Mildew	Munin	Barley
Full AC nodes	627,098	725,333	1,109,622	2,036,422	18,174,464
Partial AC nodes	239,948	711,335	446,562	1,915,257	17,174,174
Full AC clusters	441	223	35	1,041	48
Partial AC clusters	219	108	17	619	20
Serial execution on partial AC	2.85 ± 0.19	5.75 ± 0.55	6.17 ± 0.17	20.85 ± 2.22	199.93 ± 24.19
PCE on partial AC	3.97 ± 0.19	6.05 ± 0.36	5.29 ± 0.17	10.93 ± 1.05	59.14 ± 7.31
KLay (CPU) on full AC	55.60 ± 3.46	20.61 ± 5.51	167.60 ± 13.09	251.16 ± 13.61	316.23 ± 27.43
KLay (GPU) on full AC	24.46 ± 7.53	29.57 ± 11.22	14.05 ± 4.41	32.96 ± 10.47	**23.43 ± 7.14**
ACE on full AC	7.27 ± 0.84	8.56 ± 0.94	13.78 ± 1.12	26.67 ± 2.93	222.21 ± 28.09
PCE on full AC	**6.69 ± 0.18**	**7.63 ± 0.34**	**8.81 ± 0.22**	**13.16 ± 0.49**	69.93 ± 6.30

Table 3: AC inference times (in milliseconds) for KLay, ACE, and PCE on benchmark networks. Boldface indicates the best-performing method with statistical significance based on the Wilcoxon signed-rank test with p-value < 0.05.

Conclusion

To the best of our knowledge, this is the first work to accelerate PSDD maximum likelihood parameter learning, opening the door to further improvement. The results in Table 2 suggest that exploiting PSDD *determinism* to focus parameter learning on the activated partition is better than needlessly exploring unnecessary paths across the entire PSDD. Since pre-compilation is only done once, future work will look at online learning applications where concept drift occurs, necessitating the relearning of parameters over the same fixed structure due to shifts in the distribution (Li et al. 2021). Investigating how the SDD topology affects Det-PL is left for future work.

Our PSDD inference algorithm, called *Parallelized Circuit Evaluation* (PCE), integrates cluster- and layer-based parallelism to reach state-of-the-art results, as shown in the final line of Table 3. PCE can run on a full clustered AC or on a partial clustered AC obtained via structure-based pruning of deterministic computation. For both learning and inference, all boldface winners are statistically significant under the Wilcoxon signed-rank test with p-value < 0.05. Additional studies are needed to compare PCE with a broader suite of related works (Gala et al. 2024; Liu, Ahmed, and Van den Broeck 2024). Future work can also investigate the role of cluster formation in Definition 2.

Acknowledgments

The authors thank Arthur Choi for several insightful discussions that significantly benefited this work. This research was partially supported by the NSERC Discovery Grant Program (Grant No. 238880).

References

Agrawal, D.; Pote, Y.; and Meel, K. S. 2021. Partition Function Estimation: A Quantitative Study. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, 4276–4285. Virtual: IJCAI.

Butz, C. J.; Oliveira, J. S.; dos Santos, A. E.; Teixeira, A. L.; Poupart, P.; and Kalra, A. 2018. An Empirical Study of Methods for SPN Learning and Inference. In *Proceedings of the Ninth International Conference on Probabilistic Graphical Models*, 49–60. Prague, Czech Republic: PMLR.

Choi, A.; Shen, Y.; and Darwiche, A. 2017. Tractability in Structured Probability Spaces. In *Advances in Neural Information Processing Systems 30*. Long Beach, Calif.: NeurIPS.

Choi, A.; Tavabi, N.; and Darwiche, A. 2016. Structured Features in Naive Bayes Classification. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 3233–3240. Phoenix, Ariz.: AAAI Press.

Choi, A.; Van den Broeck, G.; and Darwiche, A. 2015. Tractable Learning for Structured Probability Spaces: A Case Study in Learning Preference Distributions. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2861–2868. Buenos Aires, Argentina: AAAI Press.

Choi, Y.-K.; Santillana, C.; Shen, Y.; Darwiche, A.; and Cong, J. 2023. FPGA Acceleration of Probabilistic Senten-

tial Decision Diagrams with High-Level Synthesis. *ACM Transactions on Reconfigurable Technology and Systems*, 16(2): 1–22.

Choi, Arthur. 2018. The PyPSDD Package. https://github.com/art-ai/pypsdd. Accessed: 2025-07-10.

Darwiche, A. 2001a. Decomposable Negation Normal Form. *Journal of the ACM*, 48(4): 608–647.

Darwiche, A. 2001b. On the Tractable Counting of Theory Models and Its Application to Truth Maintenance and Belief Revision. *Journal of Applied Non-Classical Logics*, 11(1–2): 11–34.

Darwiche, A. 2003. A Differential Approach to Inference in Bayesian Networks. *Journal of the ACM*, 50(3): 280–305.

Darwiche, A. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, 819–826. Barcelona, Spain: AAAI Press/IJCAI.

Darwiche, A. 2021. Tractable Boolean and Arithmetic Circuits. In Diligenti, M.; d'Avila Garcez, A. S.; and Lamb, L. C., eds., *Neuro-Symbolic Artificial Intelligence: The State of the Art*, volume 342 of *Frontiers in Artificial Intelligence and Applications*, 146–172. Amsterdam: IOS Press.

Darwiche, A.; Dechter, R.; Choi, A.; Gogate, V.; and Otten, L. 2008. Probabilistic Reasoning Evaluation UAI-2008: Results from the Probabilistic Inference Evaluation of UAI-08. https://ics.uci.edu/~dechter/softwares/benchmarks/UAI08/uai08-evaluation-2008-09-15.pdf. Accessed: 2025-07-10.

Gala, G.; de Campos, C.; Vergari, A.; and Quaeghebeur, E. 2024. Scaling Continuous Latent Variable Models as Probabilistic Integral Circuits. In *Advances in Neural Information Processing Systems 37*.

Jensen, A. L.; and Jensen, F. V. 1996. MIDAS - An Influence Diagram for Management of Mildew in Winter Wheat. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, 349–356. Portland, Oregon: Morgan Kaufmann.

Kamishima, T. 2003. Nantonac Collaborative Filtering: Recommendation Based on Order Responses. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 583–588. Washington, D.C.: ACM.

Kisa, D.; Van den Broeck, G.; Choi, A.; and Darwiche, A. 2014. Probabilistic Sentential Decision Diagrams. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, 558–567. Vienna, Austria: AAAI Press.

Kozlov, A.; and Singh, J. P. 1994. A Parallel Lauritzen-Spiegelhalter Algorithm for Probabilistic Inference. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, 320–329. Seattle, Wash.: Morgan Kaufmann.

Li, A.; Boyd, A.; Smyth, P.; and Mandt, S. 2021. Detecting and Adapting to Irregular Distribution Shifts in Bayesian Online Learning. In *Proceedings of the Thirty-Fourth Conference on Neural Information Processing Systems*. Virtual: NeurIPS.

Liu, A.; Ahmed, K.; and Van den Broeck, G. 2024. Scaling Tractable Probabilistic Circuits: A Systems Perspective. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, 30630–30646. Vienna, Austria: PMLR.

Madsen, A. L.; and Jensen, F. V. 1999. Parallelization of Inference in Bayesian Networks. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, 229–238. Stockholm, Sweden: Morgan Kaufmann.

Maene, J.; Derkinderen, V.; and Zuidberg Dos Martires, P. 2025. KLay: Accelerating Arithmetic Circuits for Neurosymbolic AI. In *Proceedings of the Thirteenth International Conference on Learning Representations*. Singapore: ICLR.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, Calif.: Morgan Kaufmann.

Scutari, Marco. 2007. Bayesian Network Repository. https://www.bnlearn.com/bnrepository/. Accessed: 2025-07-10.

Shah, N.; Galindez Olascoaga, L. I.; Meert, W.; and Verhelst, M. 2020. Acceleration of Probabilistic Reasoning through Custom Processor Architecture. In *Proceedings of the 2020 Design, Automation & Test in Europe Conference & Exhibition*, 322–325. Grenoble, France: IEEE.

Shah, N.; Galindez Olascoaga, L. I.; Zhao, S.; Meert, W.; and Verhelst, M. 2021. DPU: DAG Processing Unit for Irregular Graphs With Precision-Scalable Posit Arithmetic in 28 nm. *IEEE Journal of Solid-State Circuits*, 57(8): 2586–2596.

Shen, Y.; Choi, A.; and Darwiche, A. 2016. Tractable Operations for Arithmetic Circuits of Probabilistic Models. In *Proceedings of the 30th Conference on Neural Information Processing Systems*, 3886–3894. Barcelona, Spain: NeurIPS.

Shen, Y.; Choi, A.; and Darwiche, A. 2018. Conditional PSDDs: Modeling and Learning with Modular Knowledge. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 6433–6442. New Orleans: AAAI Press.

UCLA Automated Reasoning Group. 2015. ACE: Arithmetic Circuit Evaluator. http://reasoning.cs.ucla.edu/ace. Accessed: 2025-07-10.

Vasimuddin, M.; Chockalingam, S. P.; and Aluru, S. 2018. A Parallel Algorithm for Bayesian Network Inference Using Arithmetic Circuits. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium*, 34–43. Vancouver, Canada: IEEE.

Wilcoxon, F. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6): 80–83.

Zhang, N. L.; and Poole, D. 1994. A Simple Approach to Bayesian Network Computations. In *Proceedings of the Tenth Canadian Artificial Intelligence Conference*, 171–178. Banff, Canada: CIPS.