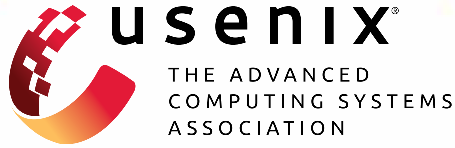


Fast Inference for Probabilistic Graphical Models

Jiantong Jiang, *The University of Western Australia;* Zeyi Wen, *HKUST (Guangzhou) and HKUST;* Atif Mansoor and Ajmal Mian, *The University of Western Australia*

https://www.usenix.org/conference/atc24/presentation/jiang

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the
2024 USENIX Annual Technical Conference
is sponsored by

Fast Inference for Probabilistic Graphical Models

Jiantong Jiang[1], Zeyi Wen[2,3*], Atif Mansoor[1], Ajmal Mian[1]
[1]*The University of Western Australia,* [2]*HKUST (Guangzhou),* [3]*HKUST*

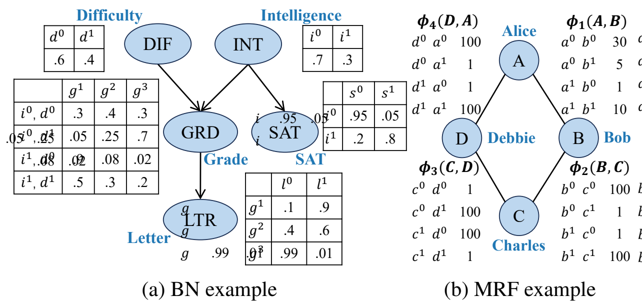
Abstract

Probabilistic graphical models (PGMs) have attracted much attention due to their firm theoretical foundation and inherent interpretability. However, existing PGM inference systems are inefficient and lack sufficient generality, due to issues with irregular memory accesses, high computational complexity, and modular design limitation. In this paper, we present Fast-PGM, a fast and parallel PGM inference system for importance sampling-based approximate inference algorithms. Fast-PGM incorporates careful memory management techniques to reduce memory consumption and enhance data locality. It also employs computation and parallelization optimizations to reduce computational complexity and improve the overall efficiency. Furthermore, Fast-PGM offers high generality and flexibility, allowing easy integration with all the mainstream importance sampling-based algorithms. The system abstraction of Fast-PGM facilitates easy optimizations, extensions, and customization for users. Extensive experiments show that Fast-PGM achieves 3 to 20 times speedup over the state-of-the-art implementation. Fast-PGM source code is freely available at https://github.com/jjiantong/FastPGM.

1 Introduction

Graph data processing is fundamental in parallel and distributed computing, and probabilistic graphical models (PGMs) [32] stand out as a crucial category of graphs that excel in modeling uncertainty in machine learning systems. PGMs employ a transparent and intuitive representation to compactly encode random variables and their interactions, offering a mathematically sound framework grounded in probability theory. Typical examples of PGMs include Bayesian networks (BNs), Markov random fields (MRFs) and factor graphs, covering directed and undirected graphs as shown in Figure 1. They have found applications in broad domains like biomedical informatics [13, 52], computer vision [7, 53], environmental monitoring [23, 34], risk management [4, 40] and

*Zeyi Wen is the corresponding author.



(a) BN example (b) MRF example

Figure 1: Two examples of PGMs. BNs are represented by directed acyclic graphs, while MNs are represented by undirected graphs.

transportation [17, 19]. PGMs also have attracted attention with the recent demand for interpretable machine learning models [9, 36, 42, 50, 54, 56].

Probabilistic inference on PGMs is a crucial task that computes the posterior distribution of any subset of random variables, given some observed values of other variables as *evidence*. Stochastic sampling methods are a popular subclass of approximate inference methods. They estimate the probability of an event from the frequency of event occurrences in sampling trials and can theoretically converge to the exact solutions with sufficient samples [45]. Furthermore, they hold an important *anytime* property [22]: computation can be interrupted at any time to get the best approximate answer available, which is important in time-critical applications. Among the stochastic sampling algorithms, importance sampling-based algorithms [15, 44, 47, 59, 60] are the most widely used ones, since they perform well even with unlikely evidence. They use the *importance function* to generate samples and aim to find a good importance function.

However, developing efficient approximate inference algorithms is challenging. Firstly, the problem of approximate inference on PGMs is known to be NP-hard that has a high computational complexity [18, 32]. Secondly, the irregular na-

ture of the graphical structures and the stochastic nature of the sampling process introduce additional inefficiencies in memory and computation. Finally, abstracting and integrating various algorithms into a unified and functional system is a nontrivial endeavor. Several libraries [12, 14, 27, 35, 37, 38, 41, 49] offer implementations of approximate inference methods, but they suffer from limitations such as low efficiency, a narrow range of supported importance sampling-based algorithms, or a lack of complete open-source availability.

To facilitate the widespread adoption of approximate inference to complex problems, we propose a fast PGM inference system named Fast-PGM for importance sampling-based approximate inference algorithms. Fast-PGM is designed for two primary objectives: (i) good generality and flexibility and (ii) high efficiency. Firstly, we abstract all the importance sampling-based algorithms into a general framework with four crucial modules. Fast-PGM provides the implementations of mainstream algorithms under this framework and offers rich interfaces that enable easy optimizations to every module and fast extensions to other inference algorithms. Secondly, we improve the efficiency of Fast-PGM by (i) memory management (i.e., novel data structure design, data fusion and reordering optimizations), (ii) careful computation simplification optimizations, and (iii) parallelization. In summary, the main contributions of this paper are as follows:

- We propose a system named Fast-PGM for importance sampling-based approximate inference. Fast-PGM provides the implementations of various algorithms and allows quick and easy optimizations, extensions, and customization for users with the help of novel systematic abstraction.
- Fast-PGM incorporates memory management to reduce memory consumption and enhance data locality, computation simplification optimizations to reduce computational complexity, and parallelization techniques to further enhance the overall efficiency of Fast-PGM.
- We conduct extensive experiments to study the effectiveness of Fast-PGM and the impact of our optimizations. Results show that Fast-PGM is 3 to 20 times faster than the state-of-the-art solutions, and the practical parallelization speedups of our method approach the theoretical speedups.

2 Background and Related Work

Here we introduce the basis and review the related studies. A summary of the notation used throughout the paper is provided in Appendix A.1 Table 3.

2.1 Probabilistic Graphical Models

Probabilistic graphical models (PGMs) can be described by $\mathbb{G}$ and $\mathbb{P}$, where $\mathbb{G} = (\mathcal{V}, \mathcal{A})$ is the structure and $\mathbb{P}$ is the parameters of the model. The nodes $\mathcal{V} = \{V_0, V_1, ..., V_{n-1}\}$ in $\mathbb{G}$ denote the random variables and the edges $\mathcal{A}$ denote the probabilistic interactions among the variables. We use *conditioning set* $\mathbf{C}(V_j)$ to denote the variables influencing V_j.

Bayesian networks (BNs) are a pivotal type of PGMs. In a BN, $\mathbb{G}$ is a directed graph, and $\mathbb{P}$ is a set of conditional probability distributions (CPDs). Each CPD describes the distribution of a variable given its possible parent configurations. Take Figure 1a as an example, *GRD* is the parent of *LTR*; the direct edge between them indicates that a student's grade impacts the likelihood of obtaining a strong recommendation letter. According to the CPD of *SAT*, the probability of getting a high SAT score is 0.8 for students with high intelligence and 0.05 for those with relatively lower intelligence. The overall joint distribution can be factorized as the product of the CPDs:

$$P(\mathcal{V}) = P(V_0, V_1, ..., V_{n-1}) = \prod_{j=0}^{n-1} P(V_j | \mathbf{C}(V_j)), \qquad (1)$$

where n is the number of random variables, $\mathbf{C}(V_j)$ represents the set of parents of V_j in the context of BNs, and $P(V_j|\mathbf{C}(V_j))$ is the CPD of V_j.

Markov random fields (MRFs) are another subclass of PGMs. The major difference is that $\mathbb{G}$ of an MRF is an undirected graph. Thus, MRFs imply symmetrical interactions between variables. $\mathbb{P}$ of an MRF is a set of potential functions. Each potential function $\phi_f(\mathcal{V}_f)$ defines a joint distribution over a set of variables $\mathcal{V}_f \subseteq \mathcal{V}$. We can also get the CPD $P(V_j|\mathbf{C}(V_j))$ of V_j via the potential functions:

$$P(V_j|\mathbf{C}(V_j)) = \frac{P(V_j, \mathbf{C}(V_j))}{P(\mathbf{C}(V_j))} = \frac{\prod_{f \in \mathbb{P}_{\{V_j\} \cup \mathbf{C}(V_j)}} \phi_f(\mathcal{V}_f)}{\sum_{V_j} \prod_{f \in \mathbb{P}_{\{V_j\} \cup \mathbf{C}(V_j)}} \phi_f(\mathcal{V}_f)}.$$

The core is to multiply all the potential functions $\mathbb{P}_{\{V_j\} \cup \mathbf{C}(V_j)}$ that are related to $\{V_j\} \cup \mathbf{C}(V_j)$ and sum over all possible values of V_j. In the context of MRFs, $\mathbf{C}(V_j)$ is the Markov blanket of V_j. This computation allows for a unified perspective on BNs and MRFs. The conditioning set $\mathbf{C}(V_j)$ encompasses both the parents of V_j in the context of BNs and the Markov blanket of V_j in the context of MRFs.

2.2 Probabilistic inference on PGMs

Given a PGM, we often have some observed variables $\mathcal{E} \subseteq \mathcal{V}$ and the observed values $\mathbf{e}$ of $\mathcal{E}$ are called *evidence*. Probabilistic inference on PGMs is to compute the posterior distributions of the variables of interest $\mathcal{Y} \subseteq \mathcal{V}$ given $\mathbf{e}$. In Figure 1a, suppose we know that the course is hard ($DIF = d^1$) and the student is intelligent ($INT = i^1$), and we want to query the quality of the professor's recommendation letter (LTR). Thus, we compute distribution over LTR given the evidence of $\{DIF = d^1, INT = i^1\}$, i.e., $P(LTR|DIF = d^1, INT = i^1)$.

To infer the posterior distribution of $Y_j \in \mathcal{Y}$ given $\mathcal{E} = \mathbf{e}$, a common practice is to first compute $P(Y_j, \mathcal{E} = \mathbf{e})$ and $P(\mathcal{E} =$

e), then combine them according to the Bayes' Theorem.

$$P(Y_j|\mathcal{E}=\mathbf{e}) = \frac{P(Y_j, \mathcal{E}=\mathbf{e})}{P(\mathcal{E}=\mathbf{e})}. \tag{2}$$

2.3 Importance Sampling

The importance sampling approach can be applied to probabilistic inference. It is an alternative to exact numerical integration. Consider the problem of estimating the integral

$$I = \int_{\Omega} g(\mathcal{X}) d\mathcal{X}, \tag{3}$$

where $g(\mathcal{X})$ is a function of m variables $\mathcal{X} = (X_0, X_1, ..., X_{m-1})$ over the domain $\Omega \subset R^m$. Importance sampling approaches this problem by rewriting Equation 3 into

$$I = \int_{\Omega} \frac{g(\mathcal{X})}{f(\mathcal{X})} f(\mathcal{X}) d\mathcal{X},$$

where $f(\mathcal{X})$ is the *importance function*, which is a probability density function over Ω and is easy to sample from. After generating q samples from $f(\mathcal{X})$, I can be estimated by

$$\hat{I} = \frac{1}{q} \sum_{i=0}^{q-1} \frac{g(s_i)}{f(s_i)},$$

where $\frac{g(s_i)}{f(s_i)}$ is called the *importance score*. The accuracy of $\hat{I}$ largely depends on the quality of the importance function, and if $g(\mathcal{X}) > 0$, the optimal importance function [43] is

$$f(\mathcal{X}) = \frac{g(\mathcal{X})}{I}. \tag{4}$$

In the context of PGMs, $P(\mathcal{E}=\mathbf{e})$ is calculated by marginalizing all variables except for $\mathcal{E}$ from the joint distribution:

$$P(\mathcal{E}=\mathbf{e}) = \sum_{\mathcal{V} \setminus \mathcal{E}} P(\mathcal{V} \setminus \mathcal{E}, \mathcal{E}=\mathbf{e}),$$

which is almost identical to Equation 3. According to Equation 4, its optimal importance function is actually the posterior distribution $P(\mathcal{V}|\mathcal{E}=\mathbf{e})$, which is impossible to obtain [61]. However, the goal is to find a function that is sufficiently close to the posterior distribution. For the purpose of efficiency, it is common practice to use the same importance function to estimate both $P(Y_j, \mathcal{E}=\mathbf{e})$ and $P(\mathcal{E}=\mathbf{e})$, and compute the posterior distribution according to Equation 2.

2.4 Related Studies on PGM Inference

PGMs are critical in machine learning for modeling uncertainty in complex systems, aligning well with the growing interest in explainable artificial intelligence [6, 20, 55, 57]. This work focuses on approximate inference on PGMs. Here, we review the most relevant studies from two categories: approximate inference algorithms and implementations.

Approximate Inference Algorithms. One class of approximate inference methods is optimization-based methods [2, 11, 24, 39, 51, 58]. They construct an approximation to the target distribution, which involves optimizing an objective under a constraint space. Our focus in this paper is on another category of methods known as stochastic sampling methods. One notable method within this subclass is probabilistic logic sampling [26]. It instantiates all variables in the sampling process and discards samples that are inconsistent with the evidence. To improve the sample efficiency, likelihood weighting [21] only instantiates non-evidence variables when generating samples. Follow-up studies investigate the importance sampling methods to handle extremely unlikely evidence. To find a better importance function, some algorithms, such as self-importance sampling [16, 47] and adaptive importance sampling (AIS-BN) [15], introduce a learning process to gradually learn an importance function. On the other hand, some algorithms propose to pre-compute a good importance function. Examples include evidence pre-propagation importance sampling (EPIS-BN) [59, 60] and importance sampling with probability trees [44].

Existing PGM Inference Libraries. There are some libraries supporting inference on PGMs. FastInf [27], libDAI [37], Factorie [35] and PNL [41] focus more on exact inference or optimization-based methods. On the other hand, BNT [38] and UnBBayes [38] implement stochastic sampling algorithms, while they support only a limited set of importance sampling-based methods. BNJ [12] supports various importance sampling-based methods but suffers from inefficiency issues. SMILE [49] is a state-of-the-art software tool for PGMs. However, it is mainly for business purposes and its source code is not fully available to academic users, hindering reproducibility and extension efforts. Furthermore, most of the libraries lack support for acceleration techniques such as parallelization, which is increasingly critical for handling large-scale data and complex problems [30, 62]. To address these limitations and advance the utilization of approximate inference on PGMs, we propose a general and efficient solution tailored for importance sampling-based approximate inference. A summarized comparison of Fast-PGM with the above libraries is provided in Appendix A.2 Table 4.

3 Overview of Fast-PGM

Here, we present Fast-PGM, a system for importance sampling-based inference on PGMs. Fast-PGM is designed for the following two primary objectives (O1 and O2).

O1: Good Generality and Flexibility. On the one hand, we discover four crucial steps that can form a skeleton for any importance sampling-based algorithm. Fast-PGM provides four corresponding modules with well-designed abstraction and interface for each step, thereby achieving a high level of gen-

Algorithm 1: Fast-PGM

```
input  : prior probability P(V), weight v^0, # of
         samples required q, updating interval l
output : estimated posterior marginal probability for
         all non-evidence variables
1 k ← 0, scrArr ← 0, w_curScr ← 0, w_allScr ← 0
  /* importance function initialization module      */
2 f^0(V) ← initImpFunc(P(V))
3 for i ← 1 to q do
4    if i%l == 0 then
5       k ← k+1
        /* importance function update module        */
6       f^k(V), v^k ← updImpFunc(w_curScr, w_allScr)
     /* sample generation module                    */
7    s_i, w_iScr ← genSamp(f^k(V), P(V))
     /* importance score accumulation module        */
8    w_curScr, w_allScr, scrArr ← accScr(s_i, w_iScr, v^k)
9 normalize scrArr for each variable
```

erality. On the other hand, to maximize flexibility, Fast-PGM supports various functionalities within each module, allowing easy combinations of modules with different functionalities. Specifically, Fast-PGM naturally supports the following importance sampling-based algorithms: (i) probabilistic logic sampling (PLS) [26], (ii) likelihood weighting (LW) [21], (iii) self-importance sampling (SIS) [47], (iv) self-importance sampling variant (SISv1) [16], (v) adaptive importance sampling (AIS-BN) [15], (vi) evidence pre-propagation importance sampling (EPIS-BN) [59,60], while also offering a high degree of flexibility to accommodate additional optimizations and extensions to any module or functionality. We describe the details of our system abstraction in Section 4.

O2: High Efficiency. Fast-PGM equips an efficient underlying data structure and applies data fusion and reordering techniques to enhance data locality. Meanwhile, Fast-PGM reduces the computational complexity of the most time-consuming operations. Finally, parallelization techniques are developed to further boost the overall efficiency of Fast-PGM. These optimizations can be adapted to all the algorithms developed in Fast-PGM, and the memory and computation optimizations can be applied to both sequential and parallel implementations. We elaborate our optimization techniques in detail in Section 5.

4 System Abstraction

To boost generality and flexibility, we systematically abstract Fast-PGM into four modules and provide four corresponding abstract functions: (i) *initImpFunc()* function for importance function initialization module, (ii) *genSamp()* function for sample generation module, (iii) *accImpScr()* function for importance score accumulation module, and (iv) *updImpFunc()* function for importance function update module.

The general framework of Fast-PGM is shown in Algorithm 1. In the beginning, the importance function $f^0(\mathcal{V})$ is initialized using *initImpFunc()* (Line 2). After that, inside each iteration of the main loop, a sample s_i is generated with an importance score w_{iScr} using *genSamp()* (Line 7). Then, the new importance score w_{iScr} is accumulated using *accScr()* (Line 8). During this process, the main loop in Fast-PGM is divided into multiple stages by the importance function update module. The importance function $f^k(\mathcal{V})$ of stage k is updated every l iterations, where each update corresponds to a new stage (Lines 4-6). The update uses *updImpFunc()* with two types of accumulated scores $\mathbf{w}_{curScr}$ and $\mathbf{w}_{allScr}$ (Line 6). After finishing q iterations, the estimated probability for each variable can be easily obtained by normalization (Line 9).

In what follows, we first briefly describe the above modules. Then, our comprehensive discussion unfolds, placing particular emphasis on the generality and flexibility of Fast-PGM.

4.1 Crucial Modules in Fast-PGM

Here, we briefly describe the four crucial modules with a focus on the distinctions among the algorithms. We omit highly algorithm-specific details and suggest interested readers refer to the respective algorithm papers [15, 16, 21, 26, 47, 59, 60] for a more comprehensive understanding.

Importance Function Initialization. The first module initializes the initial importance function $f^0(\mathcal{V})$. The importance function here is a probability distribution over PGMs that should be easy to sample from. The initial importance function $f^0(\mathcal{V})$ can be decomposed into multiple local CPDs (cf. Equation 1). Thus, each importance function $f^k(\mathcal{V})$ of stage k can be used and updated by maintaining a tabular CPD (i.e. conditional probability table (CPT)) of each variable, which is defined as *importance conditional probability table* (ICPT) and has the same structure as that of the CPT. We let $P^k(V_j|\mathbf{C}(V_j))$ denote the ICPT of variable V_j in stage k. In PLS, LW, SIS, SISv1 and AIS-BN, $f^0(\mathcal{V})$ is based on the CPTs, with detailed formulations available in Appendix A.3.1. EPIS-BN is different. It uses loopy belief propagation [39] to pre-compute its $f^0(\mathcal{V})$. Moreover, this module includes two heuristics: (i) heuristic U: initializing the ICPTs of the conditioning set of evidence to uniform distribution, and (ii) heuristic S: adjusting very small probabilities controlled by θ. EPIS-BN uses heuristic S, while AIS-BN uses both heuristics.

Sample Generation. The second module generates a sample s_i and computes its importance score w_{iScr}. According to Section 2.3, w_{iScr} is the product of the scores for all variables:

$$w_{iScr} = \frac{P(\mathcal{V}\backslash\mathcal{E}, \mathcal{E}=\mathbf{e})}{f^k(\mathcal{V})} = \frac{\prod_{j=0}^{n-1} P(V_j|\mathbf{C}(V_j))\Big|_{\mathcal{E}=\mathbf{e}}}{\prod_{j=0}^{n-1} P^k(V_j|\mathbf{C}(V_j))}. \quad (5)$$

Based on this idea, this module iteratively handles each variable V_j. The variable handling ordering differs for BNs and MRFs. In BNs, we typically use a topological ordering, while in MRFs, a sampling process similar to Gibbs sampling is commonly employed, following a random order. Detailed pseudo-code and descriptions are provided in Appendix A.3.2. Take the process on BNs as an example, there are four steps for handling V_j. Firstly, the instantiation $\mathbf{c}$ of its parents $\mathbf{C}(V_j)$ is obtained from the current instantiated sample s_i. The topological ordering ensures the variables in $\mathbf{C}(V_j)$ are always instantiated before the sampling of V_j. Secondly, we obtain a weight vector that contains the probabilities of V_j's possible states given the instantiation $\mathbf{C}(V_j) = \mathbf{c}$. It is obtained by reducing the ICPT $P^k(V_j|\mathbf{C}(V_j))$ based on $\mathbf{C}(V_j) = \mathbf{c}$. Thirdly, a value of V_j is randomly picked from its state space based on the weight vector, to be $s_i[j]$. Finally, the score of $V_j = s_i[j]$ is computed and multiplied into w_{iScr}. PLS handles all the variables using the same process above, without distinguishing evidence and non-evidence variables, while other algorithms simplify the handling of evidence variables by instantiating each evidence variable to its observed state.

Importance Score Accumulation. This module accumulates the newly computed w_{iScr} to the **scrArr** for each variable V_j. The **scrArr** for V_j is a r_{V_j}-size array, where r_{V_j} is the number of possible states of V_j. Based on the instantiated value of V_j in s_i, the product $v^k \cdot w_{iScr}$ is added to the corresponding entry of the **scrArr** of V_j, where v^k is the weight of the samples in stage k. The weight v^k is used only in AIS-BN, while being 1 for the other algorithms. Moreover, for SIS, SISv1 and AIS-BN, an additional process of updating $\mathbf{w}_{curScr}$ or $\mathbf{w}_{allScr}$ is required. $\mathbf{w}_{curScr}$ is the accumulated score for the samples of the current stage while $\mathbf{w}_{allScr}$ is the accumulated score for all the samples available till now. They have the same structure as CPT. The product $v^k \cdot w_{iScr}$ is added to $\mathbf{w}_{curScr}$ and $\mathbf{w}_{allScr}$ for each V_j based on the instantiation of V_j and $\mathbf{C}(V_j)$ in s_i.

Importance Function Update. The last module updates the importance function periodically according to an updating interval l. Among the algorithms developed in Fast-PGM, PLS, LW, and EPIS-BN never update the importance function. SIS and AIS-BN update the importance function using $\mathbf{w}_{curScr}$, while SISv1 updates using $\mathbf{w}_{allScr}$. Moreover, a learning rate $\eta(k)$ is also required in AIS-BN. Detailed updating functions are available in Appendix A.3.3.

4.2 Discussion on Generality and Flexibility

Through identifying the four crucial modules, Fast-PGM develops a framework for importance sampling-based approximate inference on PGMs. We show the overview of Fast-PGM in Figure 2. It takes the PGM and evidence as input, and outputs the posterior distribution of all the non-evidence variables, computed by a selected inference algorithm. The design of Fast-PGM enables good generality and flexibility. We highlight the following key features (F1, F2, and F3).

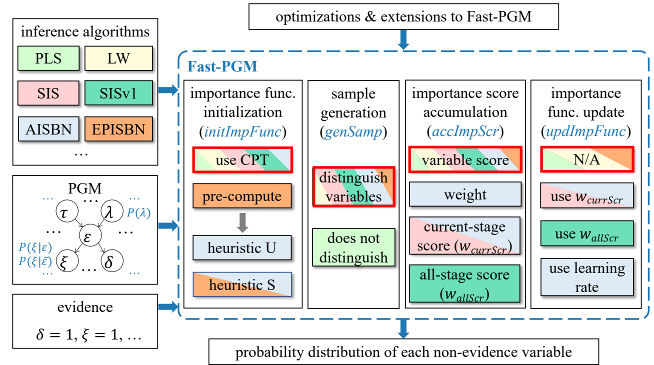


Figure 2: The overview of Fast-PGM with four crucial modules. Various functionalities are supported inside each module and we color-code the different functionalities utilized by the different algorithms. The functionalities with red rectangular boxes represent the default implementations of the abstract functions of the corresponding modules.

F1: Versatile Modules and Functionalities Support. As discussed in Section 4.1, distinctions among the inference algorithms are evident. Fast-PGM supports various functionalities inside each module, so that different algorithms can use different functionalities, as indicated by the colored code in Figure 2. For instance, to implement EPIS-BN, we use pre-computation and heuristic S in the first module, distinguish variables in the second module, accumulate the variable scores in the third module, and leave the last module inactive.

F2: Quick New Algorithm Implementation. Fast-PGM equips the abstract function of each module with a useful default implementation, illustrated in Figure 2 by the red rectangular boxes. It is worth noting that the underlying data structures are well-implemented with optimizations and are readily available to use. Therefore, implementing new importance sampling-based algorithms in Fast-PGM is quick, as we only need to focus on customizing and refining the aspects that deviate from the default implementations. For example, implementing AIS-BN based on the building blocks provided in Fast-PGM only needs less than 20 lines of code, as shown in Listing 1. If we consider AIS-BN as a completely new algorithm, we only need to implement the "updByCurrScr" function with an additional 5 lines of code. However, implementing AIS-BN from scratch needs at least 400 lines.

F3: Ease of Optimization and Extension. With the rich interfaces provided, Fast-PGM allows users to easily implement their ideas by optimizing or extending any step of the general framework. For example, some functionalities[1] require hyperparameters that may be highly problem-dependent. Thus, users can optimize the hyperparameters with our user-friendly interfaces according to their specific requirements. For another example, EPIS-BN uses the pre-computation function-

[1] Here, the functionalities that require hyperparameters include "heuristic U", "heuristic S", "weight", and "use learning rate".

```
class AISBN: public PGMSys {
protected:
  virtual void initImpFunc(Network *n,Evidence *e)
       override{
    PGMSys::initImpFunc(n,e); //call default impl.
    heurU(n,e); //call the heuristic U func.
    heurS(n); //call the heuristic S func.
  }
  virtual void accImpScr(Network *n, int *s, double &scr,
       double w) override{
    double w=getW(); //call the weight func.
    PGMSys::accImpScr(n,s,scr,w);//call default impl.
    accCurrScr(n,scr); //call the w_currScr func.
  }
  virtual void updImpFunc(int k) override{
    double eta=getEta(k); //call learning rate func.
    updByCurrScr(eta); //call using w_currScr func
  }
}
```

Listing 1: Implementing AIS-BN with Fast-PGM. Underlying data structures are omitted for the sake of readability.

ality for computing the initial importance function through loopy belief propagation, which is an optimization-based approximate inference algorithm. One also has the flexibility to integrate alternative algorithms to compute the initial importance function.

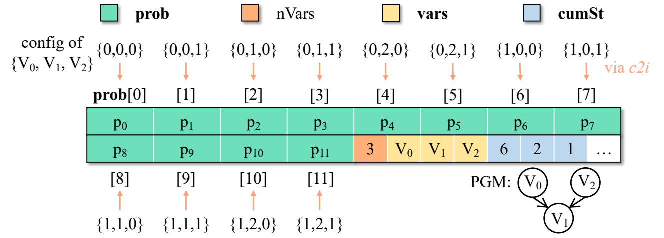
To summarize, Fast-PGM provides a general framework, with each module equipped with useful default implementation and supporting various functionalities. The system abstraction of Fast-PGM facilitates optimizations and extensions to any module or functionality, enabling users to effortlessly customize and implement their own inference algorithms based on their specific needs.

5 Implementation Details

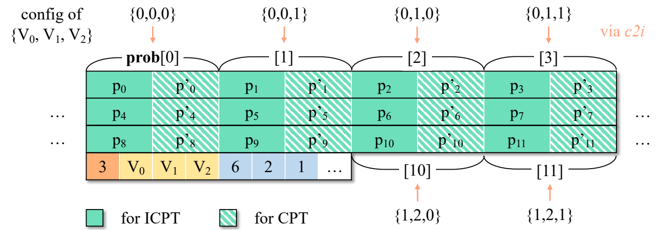
This section focuses on the another primary objective of Fast-PGM: high efficiency. Since the importance function initialization only performs once and the importance function update is also performed infrequently, the modules of sample generation and importance score accumulation take up most of the execution time, becoming our main focus in optimizing Fast-PGM. Here, we elaborate the implementation details and our optimization techniques from three aspects: memory management, computation simplification, and parallelization.

5.1 Memory Management

The core of the underlying implementation of Fast-PGM lies in manipulating the importance functions $f^k(\mathcal{V})$. This translates into the manipulation of the ICPT $P^k(V_j|\mathbf{C}(V_j))$ of each variable within the context of PGM. However, managing large ICPTs and dealing with irregular memory access patterns pose significant challenges in memory management. In this section,



(a) Data structure of the ICPT before data fusion



(b) Data structure of the ICPT and CPT after data fusion

Figure 3: Probability table data structure before and after using the data fusion strategy for variables V_1.

we present these challenges (C1, C2, and C3) and propose solutions (S1, S2, and S3) to alleviate them.

5.1.1 Basic data structure

C1: Large ICPTs. The size of the ICPTs in a PGM can be remarkably large. For example, suppose that V_j and all the variables in its conditioning set $\mathbf{C}(V_j)$ have r possible states, then the number of entries in its ICPT is $r^{|\mathbf{C}(V_j)|+1}$. In highly connected complex problems, the variables may have many neighbors and a large number of possible states, leading to extremely large ICPTs. For each of the $r^{|\mathbf{C}(V_j)|+1}$ entries in the ICPT of V_j, the naive implementation stores all the $|\mathbf{C}(V_j)|+1$ states of $\{V_j\} \cup \mathbf{C}(V_j)$, as well as the corresponding probability values for the state configurations, resulting in substantial memory consumption.

S1: Avoiding Storing States. We optimize the data storage via a *probability table* data structure to avoid storing the state configurations for each entry in the ICPTs. Specifically, the probability table of V_j includes:

- **vars**: an array containing V_j and $\mathbf{C}(V_j)$;
- *nVars*: the size of **vars**, which equals $|\mathbf{C}(V_j)|+1$;
- **prob**: the main array, storing the probability values;
- **cumSt**: a helper array storing the cumulative number of states starting from the rightmost element of **vars**.

Figure 3a shows an example of the probability table data structure of variable V_1 with two parents V_0 and V_2 in a BN, where V_1 has three possible states (i.e., $r_{V_1} = 3$) and the other two variables have two possible states (i.e., $r_{V_0} = r_{V_2} = 2$). In this case, **vars** is $\{V_0, V_1, V_2\}$, and **cumSt** is $\{r_{V_2} \cdot r_{V_1}, r_{V_2}, 1\} =$

```
int ProbabilityTable::c2i(int *config) {
  int idx=0;
  for(int i=0;i<this->nVars;++i)
    idx+=config[i]*this->cumSt[i]; // use cumSt
  return idx;
}
```

Listing 2: The *c2i* method that is used to transform from a state configuration to an index of **prob**.

$\{6,2,1\}$, which is constructed iteratively starting from the rightmost element of **vars**. The main array **prob** maintains the probabilities, where each entry corresponds to a state configuration. We avoid storing the states of all the related variables, because each state configuration (e.g., $config = \{1,0,1\}$) can be easily transformed to and from a table index (e.g., $idx = 7$) through the auxiliary array **cumSt**. For example, Listing 2 describes the *c2i* function that is able to transform a configuration into an index.

Furthermore, the data structure offers the additional benefit of reducing computational complexity. Specifically, the time complexity of the key operation of getting weight vectors is reduced from an exponential growth rate (i.e. exponential to *nVars*) to a linear growth rate (i.e. proportional to *nVars*). The details are discussed in Section 5.2.

5.1.2 Data fusion

C2: Irregular Memory Access Caused by the Stochastic Nature of the Sampling Process. As discussed in Section 4.1, we need to get the weight vector frequently when generating a sample. For variable V_j, the weight vector is obtained from V_j's ICPT, by getting the probabilities of the configurations that are consistent with the instantiation **c** of its conditioning set $\mathbf{C}(V_j)$. However, the inherent randomness of **c** introduced by sampling leads to irregular access to ICPTs. Moreover, we need to frequently find the desired probability from CPTs to compute the importance score. This causes a similar demand for frequent and irregular access to CPTs, which further exacerbates the memory issue.

S2: Fusing ICPT and CPT. Fast-PGM relieves the burden of memory access by fusing ICPT and CPT. It stems from our two practical findings. Firstly, the ICPT and CPT for the same variable have the same structure, because they both correspond to different state configurations of the variable and its conditioning set. Secondly, in the process of generating a sample and computing its importance score, the probabilities at the same position of the paired ICPT and CPT are frequently accessed together. Building on the two observations, we propose to fuse the ICPT and CPT of each variable into a unified structure and integrate the combined structure into the probability table data structure. ICPT and CPT and then share the *nVars*, **vars**, and **cumSt**, as illustrated in Figure 3b. When accessing a certain probability in the ICPT, the

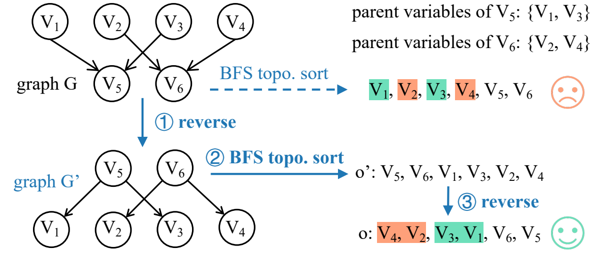


Figure 4: The data reordering strategy to maximize the proximity of each variable's parents.

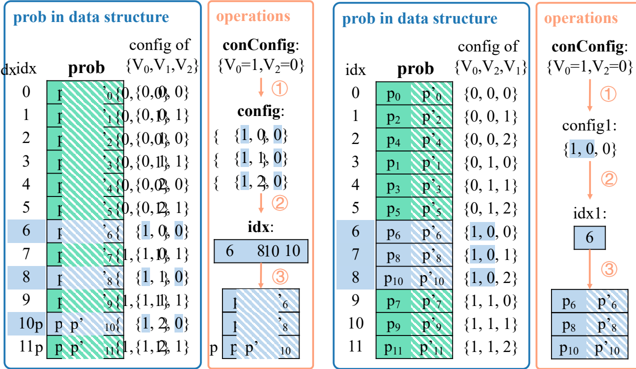
related probability in the CPT is readily available in contiguous memory locations, reducing the time for data fetching and minimizing cache misses. Data fusion capitalizes on the frequent simultaneous access property of ICPTs and CPTs to enhance data locality and improve memory access patterns.

5.1.3 Data Reordering

C3: Irregular Memory Access Caused by the Graphical Structure. To get the weight vector of each variable V_j, we must pick the instantiation **c** of its conditioning set $\mathbf{C}(V_j)$ from the current instantiated sample that contains the instantiations of all the variables. However, the inherent graphical nature of PGMs often leads to non-contiguous memory layouts for $\mathbf{C}(V_j)$, resulting in irregular memory access patterns that incur additional memory latency.

S3: Reordering Variables in Memory. We propose a data reordering strategy to maximize the proximity of each variable's parent variables in memory on BNs. This involves three steps. Firstly, we reverse the direction of all the edges in the underlying graph $\mathbb{G}$ of the BN to create a modified graph $\mathbb{G}'$. Secondly, we apply a breadth-first search (BFS)-based topological sorting algorithm on $\mathbb{G}'$ to get the topological ordering o' of $\mathbb{G}'$. Finally, we get the reverse ordering o of o'. Figure 4 illustrates the process through an example where $\{V_1,V_3\}$ is the parent set of V_5 and $\{V_2,V_4\}$ is the parent set of V_6. The resulting ordering o is a possible topological ordering of $\mathbb{G}$, which can be used as the variable sampling ordering. At the same time, it allows the parent variables of each variable to be positioned as close as possible in memory. As shown in Figure 4, variables V_1 and V_3 can be stored contiguously in memory, as well as V_2 and V_4. By storing the variable instantiations according to o, we aim to promote data locality and thus mitigate irregular memory access issues associated with BNs as much as possible. Due to space limitation, we provide more discussion of data reordering in Appendix A.4.

Note that data reordering is designed only for BNs. In MRFs, the interdependencies between variables make it challenging to ensure the proximity of one variable's Markov blanket without affecting the proximity of another variable's Markov blanket.



(a) Before computation simplification (b) After computation simplification

Figure 5: Table reduction operations on the probability table data structure for variable V_1. The **prob** array in the probability table is also provided for convenient cross-referencing.

5.2 Computation Simplification

The most time-consuming operation is to get the weight vector of variables in the sample generation module. This is done by reducing its ICPT based on the instantiation of its conditioning set. Here, we first analyze the computational complexity, then present our optimization to reduce the compleixty.

5.2.1 Computational Complexity

To reduce an ICPT, the traversal of the state configurations in the ICPT is required. The size of ICPT for V_j is equal to the number of possible states of $\{V_j\} \cup \mathbf{C}(V_j)$, which is exponential in $|\mathbf{C}(V_j)|+1$. More precisely, the size is $r_{V_j} \times \prod_{V_i \in \mathbf{C}(V_j)} r_{V_i}$, where r_{V_j} is the number of possible states of V_j. For example, considering the simplified scenarios that V_j and all the variables in its conditioning set $\mathbf{C}(V_j)$ have r possible states, then the number of entries in its ICPT is $r^{|\mathbf{C}(V_j)|+1}$.

For each entry of the ICPT to be reduced, we need to compare the state configuration with the instantiation $\mathbf{c}$ of $\mathbf{C}(V_j)$. Additionally, the algorithm generates q samples, each involving sampling all the non-evidence variables. Therefore, the overall complexity is calculated as

$$O(q \times \sum_{V_j \notin \mathcal{E}} (|\mathbf{C}(V_j)| \times r_{V_j} \times \prod_{V_i \in \mathbf{C}(V_j)} r_{V_i})), \tag{6}$$

where $\mathcal{E}$ is the set of evidence variables and V_j denotes each of the non-evidence variables. The complexity of the simplified scenarios mentioned earlier is $O(q \times \sum_{V_j \notin \mathcal{E}}(|\mathbf{C}(V_j)| \times r^{|\mathbf{C}(V_j)|+1}))$.

5.2.2 Table Reorganization Optimization

This ICPT reduction operation can be performed efficiently with the help of the probability table data structure, as illustrated in Figure 5a. For the example described in Figure 3[2], assume that the conditioning set instantiation of V_1 is $\{V_0 = 1, V_2 = 0\}$, then the entries that conflict with the instantiation are ignored and we get $\{p_6, p_8, p_{10}\}$ as the weight vector of V_1. It is processed by first constructing r_{V_1} configurations based on the conditioning set configuration **conConfig** and location of V_1 in **vars**. Next, the desired indexes **idx** are obtained by performing the configuration-to-index mapping $c2i$. Finally, we can fetch the probabilities from **prob** according to **idx**. Since the complexity of $c2i$ is $O(nVars) = O(|\mathbf{C}(V_j)|+1)$, as shown in Listing 2, we can compute the overall time complexity, which is reduced to $O(q \times \sum_{V_j \notin \mathcal{E}}(r_{V_j} \times (|\mathbf{C}(V_j)|+1)))$.

To further alleviate the computational burden, we carefully re-organize the probability table data structure in a more efficient way. Specifically, we constrain the ordering of **vars** as follows: for the probability table of variable V_j, V_j itself is the rightmost element in **vars**. Figure 5b shows the optimized process of the table reduction operation in three steps. In the first step, only the first configuration $config1$ is required to be constructed. Next, the first index $idx1$ is obtained via the $c2i$ method. Finally, we can fetch values from the contiguous memory of **prob** starting from $idx1$. In this way, the overall complexity is further reduced to $O(q \times \sum_{V_j \notin \mathcal{E}}(|\mathbf{C}(V_j)|+1))$, which is particularly beneficial for complex PGMs with a large number of possible states for each variable.

The optimized probability table data structure benefits all the importance sampling-based algorithms under the general framework in Fast-PGM. For the algorithms that need to update the importance function, such as SIS and AIS-BN, the required current-stage score $\mathbf{w}_{curScr}$ and all-state score $\mathbf{w}_{allScr}$ also have the same structure as that of CPT and ICPT. Thus, they also benefit from the optimized probability table data structure. In fact, most of the inference algorithms on PGMs can take advantage of the probability table data structure, since both the exact and approximate inference algorithms require computations on the input CPTs, which can be stored via the probability table data structure. Moreover, the overhead of re-organizing the probability table is negligible, because the probability tables of all the variables are constructed only once in the importance function initialization module.

5.3 Parallelization

Here, we aim to parallelize the importance sampling framework. From coarse-grained to fine-grained, we identify three granularities of parallelism, including case-level, sample-level, and variable-level. In what follows, we first analyze the shortcomings of accelerating Fast-PGM using case-level and variable-level parallelism, then propose to exploit sample-level parallelism to improve the efficiency of Fast-PGM.

[2]Example: V_1 has two variables in its conditioning set: V_0 and V_2. Moreover, $r_{V_1} = 3$, $r_{V_0} = r_{V_2} = 2$.

5.3.1 Case-level and Variable-level Parallelism

The most natural scheme is to parallelize different test cases, which is a coarse-grained parallelism. In many scenarios, dealing with a large number of test cases is common, and these test cases are independent of each other. Therefore, we can simply dedicate these test cases to different threads equally. However, the major limitation of case-level parallelism is load unbalancing. Although different test cases are used for the same PGM, they often involve different evidence variables. As analyzed in Equation 6, such difference in evidence variables $\mathcal{E}$ greatly influences the computational complexity, leading to highly different workloads across different test cases.

Another scheme is variable-level parallelism, a fine-grained scheme that parallelizes the sampling process of different variables. Since the sampling of different variables is inherently sequential, this method requires first partitioning the PGM into independent and parallelizable subgraphs. This corresponds to another research topic: graph partitioning [10]. However, graph partitioning relies on the PGM structure. Not all graphs are easy to partition, particularly those with high connectivity. This reliance also diverges from our objective of developing a unified framework. Another limitation of variable-level parallelism is the relatively small workload associated with each variable operation. We typically need a larger amount of workload for each thread to amortize the overhead of parallel computing like frequent thread creation. Additionally, graph partitioning introduces extra overhead.

5.3.2 Sample-level Parallelism

To overcome the limitations, Fast-PGM exploits sample-level parallelism. The main loop (cf. Algorithm 1 Lines 3-8) of the framework is divided into multiple stages by the importance function update module, where the size of each stage k depends on the updating interval l. Different samples generated inside the same stage use the same importance function, and the workloads for different samples are the same. Thus, the importance sampling-based algorithms under the general framework are well-suited for sample-level parallelism.

Specifically, there are always l samples to be generated inside each stage except for the last stage, which may have more than l samples. Inside each stage, the samples to be generated are dedicated to the parallel threads equally. Each thread independently generates a number of samples and computes an importance score w_{iScr} for each sample, similar to the sequential implementation. The difference is that each thread maintains private $\mathbf{w}_{curScr}$, $\mathbf{w}_{allScr}$ and **scrArr**. The importance score w_{iScr} is accumulated to these private vectors inside each thread. After all the threads finish their tasks of each stage, the partial results in the private vectors of all the threads are reduced to get the final results of $\mathbf{w}_{curScr}$, $\mathbf{w}_{allScr}$ and **scrArr**. This process is even simpler for some algorithms, such as PLS, LW and EPIS-BN: since they never update their importance functions, they contain only one stage. In other words, all the desired q samples are in the same stage and are generated using the same importance function $f^0(\mathcal{V})$. The q samples can be parallelized directly, and the multiple threads only require to be synchronized once. We also consider the hybrid sample-level and variable-level parallelization scheme, but it faces some challenges such as communication overhead and additional computation costs. Detailed discussion is provided in Appendix A.5 due to space limitation.

Table 1: Information of reference PGMs.

PGMs	# nodes	# edges	# parameters
Alarm [8]	37	46	509
Hailfinder [1]	56	66	2656
Pathfinder [25]	109	195	77155
Pigs [48]	441	592	5618
Munin2 [5]	1003	1244	69431
Munin4 [5]	1038	1388	80352

Moreover, PGMs are relatively small in memory. Most PGMs contain no more than hundreds of variables. For example, a BN with hundreds of variables is considered a large network [15, 28, 46, 59]. Therefore, the model and data can be stored in a single machine easily without being distributed to multiple machines.

6 Experimental Evaluation

We perform experiments to evaluate the effectiveness of our proposed techniques and compare with existing methods.

6.1 Experimental Setup

Baselines and platform. We implemented Fast-PGM using C++ for importance sampling-based approximate inference on PGMs. We compared the performance of Fast-PGM with the existing work SMILE [49] and BNJ [12]. SMILE and BNJ are both sequential implementations. All experiments were conducted on a Linux machine with a 16-core 3.2GHz Intel(R) Core(TM) i9-12900K CPU and 128GB main memory.
Data sets. Our experiments were tested on BNs as exemplary instances of PGMs. We tested Fast-PGM on six real-world BNs as shown in Table 1. They have been widely used for comparative studies, and the last four are considered as large-scale networks in the community [15, 29, 46]. We generated 1,000 test cases from each network. Following the common settings [15, 59, 60], each test case has 20 observed variables.
Parameter settings. The default number of samples q is 4,000 for *Alarm* and *Hailfinder*, and 40,000 for the other larger networks. For the tunable parameters in AIS-BN[3] and EPIS-BN[4],

[3] Parameter settings for AIS-BN: $\eta(k) = a(\frac{b}{a})^{k/k_{max}}$, where $a = 0.4$, $b = 0.14$, k_{max} is the times of importance function updating and is no more than 10; $v^k = 1$ for the last updating interval and $v^k = 0$ otherwise; $\theta = 0.4$.

[4] Parameter settings for EPIS-BN: $d = 2$; $\theta = 0.006$ for variables with

Table 2: Execution time comparison of Fast-PGM ("Ours") with SMILE ("S") and BNJ ("B") on 1000 test cases. Speedups of Fast-PGM over SMILE and BNJ are also reported. "N/A" means that the library does not support the corresponding algorithm.

PGM	PLS					LW					SIS				
	Time (sec)			Speedup		Time (sec)			Speedup		Time (sec)			Speedup	
	S	B	Ours	S	B	S	B	Ours	S	BNJ	S	B	Ours	S	B
Alarm	0.39	1.59	0.13	3.0	12.0	0.42	1.50	0.11	3.7	13.2	N/A	3.0	0.14	N/A	20.7
Hailfinder	0.76	5.7	0.26	3.0	22.1	0.66	2.7	0.21	3.1	13.1	N/A	15.8	0.26	N/A	60.0
Pathfinder	5.0	76.0	1.1	4.4	66.8	6.1	88.0	1.0	5.8	84.2	N/A	1.1k	1.3	N/A	850
Pigs	21.4	1.2k	1.4	15.0	841	14.6	1.2k	1.5	10.1	854	N/A	4.9k	1.8	N/A	2.8k
Munin2	68.4	6.2k	4.5	15.1	1.4k	48.9	6.0k	4.5	10.8	1.3k	N/A	27k	5.9	N/A	4.5k
Munin4	67.8	6.1k	5.1	13.4	1.2k	47.7	7.6k	5.2	9.2	1.5k	N/A	32k	6.9	N/A	4.6k

PGM	SISv1					AIS-BN					EPIS-BN				
	Time (sec)			Speedup		Time (sec)			Speedup		Time (sec)			Speedup	
	S	B	Ours	S	B	S	B	Ours	S	B	S	B	Ours	S	B
Alarm	0.46	N/A	0.15	3.2	N/A	0.77	10.2	0.15	5.2	69.3	0.65	N/A	0.14	4.5	N/A
Hailfinder	0.83	N/A	0.27	3.1	N/A	1.3	27.7	0.27	4.9	104	0.83	N/A	0.26	3.2	N/A
Pathfinder	13.0	N/A	1.3	10.0	N/A	11.5	2.7k	1.3	8.8	2.1k	6.0	N/A	1.3	4.5	N/A
Pigs	30.0	N/A	1.8	16.8	N/A	32.0	6.9k	1.8	17.9	3.9k	17.8	N/A	1.8	10.2	N/A
Munin2	94.2	N/A	6.1	15.4	N/A	121	47k	6.2	19.7	7.4k	56.3	N/A	5.8	9.7	N/A
Munin4	63.9	N/A	6.8	9.3	N/A	119	55k	6.9	17.2	7.9k	62.9	N/A	6.6	9.6	N/A

we used the default settings following the suggestions in the original papers [15, 59, 60]. For the algorithms that require a learning process, i.e., SIS, SISv1 and AIS-BN, we varied the updating interval l to achieve better accuracy. Specifically, large or complex networks often require a larger l. The settings of l used in our experiments are: 2,500 for *Alarm*, *Hailfinder* and *Pathfinder*; 50,000 for *Pigs*, *Munin2* and *Munin4*.

6.2 Overall Comparison

In this section, we first compare Fast-PGM with existing works for the overall execution time, then investigate the impact of our proposed optimizations, and finally study the speedup obtained by multi-thread parallelization.

6.2.1 Execution Time Comparison with Existing Work

We compare the overall efficiency of Fast-PGM with the existing implementations SMILE and BNJ. Among the importance sampling-based algorithms, SMILE implements five of them: PLS, LW, SISv1, AIS-BN and EPIS-BN; BNJ has implementations of four: PLS, LW, SIS and AIS-BN. Therefore, we compare the implementations of these algorithms with the corresponding implementations in Fast-PGM. The number of samples q in this experiment is set to 1000. For the last four networks, we varied the number of threads t from 1 to 16 for Fast-PGM and chose the one with the shortest execution time.

Our experimental results are summarized in Table 2, where "N/A" means the algorithm is not implemented in the library.

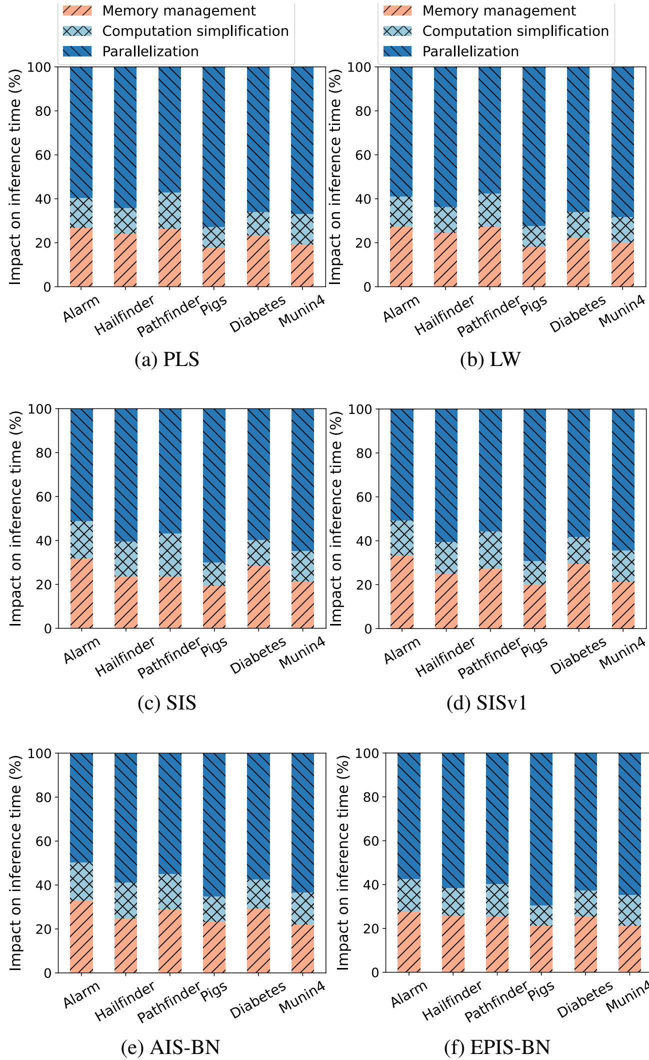
fewer than 5 possible states, $\theta = 0.001$ for variables with the number of states between 5 and 8, and otherwise, $\theta = 0.0005$.

As can be seen from the "Speedup" columns, Fast-PGM is 3 to 20 times faster than the state-of-the-art implementation SMILE. When compared to BNJ, Fast-PGM can often achieve three orders of magnitude significant speedup. The speedups are mainly due to our careful optimizations on memory, computation and parallelization aspects for improving the efficiency of Fast-PGM. The experiments on the relatively small networks *Alarm* and *Hailfinder* were conducted under one single thread, since the optimizations on the sequential version of Fast-PGM already lead to a short execution time (i.e., less than 0.5s). For the other larger networks, Fast-PGM always achieves its shortest execution time when $t = 16$.

Meanwhile, since both SMILE and BNJ do not support acceleration using multi-thread techniques, we also conduct experiments to compare SMILE and BNJ with the sequential version of Fast-PGM. We find that sequential Fast-PGM still outperforms SMILE, achieving up to 5 times speedup, and can achieve two to three orders of magnitude speedup than BNJ. Due to space limitation, we provide the detailed experimental results in Appendix A.6.

6.2.2 Impact of Individual Optimizations

Here, we study the impact of individual optimizations on the overall efficiency. The optimizations include (i) memory management: includes data structure, data fusion and data reordering, to avoid frequent finding operations and improve data locality; (ii) computation simplification: re-organizes the data structure to simplify key computations; (iii) parallelization: sample-level parallelism for the most expensive modules. To investigate the impacts of the optimizations, we



(a) PLS

(b) LW

(c) SIS

(d) SISv1

(e) AIS-BN

(f) EPIS-BN

Figure 6: Impact of the individual optimizations on the overall efficiency improvement of Fast-PGM.

successively switched off parallelization, computation simplification, and memory management. Hence, the contribution of each optimization to the overall efficiency improvement can be observed, as shown in Figure 6.

On average of the six algorithms, 25% of the improvement originates from memory management, 14% from computation simplification, and 61% from parallelization. Among them, AIS-BN, SIS and SISv1 need to learn an importance function (i.e., learning-based algorithms), and we observe that 41% of their improvement originates from memory management and computation simplification, and 59% from parallelization. On the other hand, for EPIS-BN, PLS and LW that use a fixed importance function (i.e., non-learning-based algorithms), 37% of the improvement originates from memory management and computation simplification, while 63% from parallelization. To conclude, the learning-based algorithms benefit more from memory management and computation simplification compared to the non-learning-based algorithms. There are two main reasons for this. Firstly, the learning-based algorithms maintain additional $\mathbf{w}_{curScr}$ or $\mathbf{w}_{allScr}$ for each variable in the PGMs. Since $\mathbf{w}_{curScr}$ and $\mathbf{w}_{allScr}$ are integrated into the probability table data structure, the computations on them can take advantage of memory management and computation simplification on the probability table data structure. Secondly, parallelizing non-learning-based algorithms requires less synchronization. Thus, they benefit more from parallelization.

It is worth noting that although parallelization contributes more, memory management and computation simplification play a critical role in the efficiency improvement. For example, running the 1,000 test cases on *Munin4* using EPIS-BN with 40,000 samples takes 9,808 seconds for the naive implementation. The execution time is significantly reduced to 2,099 seconds after careful memory management and computation simplification optimizations. They bring 4.7 times speedup. On the basis of the two optimizations, parallelization is able to further reduce the time to 263 seconds.

To evaluate the impact of the proposed optimizations on the inference results, we compare the probability results of the algorithms before and after applying the optimizations using the same test cases. In this experiment, we also fixed the random seed for each run to avoid the variance caused by the sampling processes. Our experiments show that the results with and without our optimizations are identical for all the implemented approximate inference algorithms. The reason is that our optimizations on memory, computation, and parallelization do not involve any approximation or alteration of the algorithms themselves.

6.2.3 Comparison with Theoretical Speedup

We compare the speedups of the parallel version of Fast-PGM to the sequential version with the theoretical speedups. The theoretical speedups are computed by Amdahl's Law [3], which gives the theoretical speedup of a parallel program by

$$Speedup(t) = \frac{1}{(1-rp)+\frac{rp}{t}},$$

where t is the number of parallel threads, rp is the ratio of the parallel part of the program, and $Speedup(t)$ means the theoretical speedup under t threads. We can easily obtain rp from one sequential run of the program.

Figure 7 shows the theoretical speedups and the speedups of the parallel version of Fast-PGM on the four larger models. We can observe that the speedups of Fast-PGM approach the theoretical speedups, especially for the largest model *Munin4*. Figure 7 also provides the ratios of the parallel and sequential parts of the program respectively. As shown in Figure 7b, *Pigs* has the largest rp, which is 99%. Therefore, it has the largest theoretical speedup as well as the practical speedup of

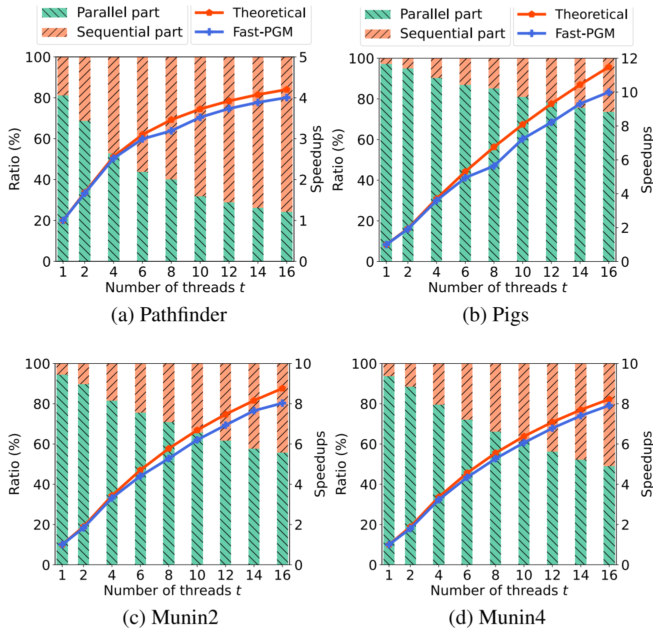


Figure 7: Theoretical and practical speedups of Fast-PGM under different number of threads. The ratios of parallel part and sequential part of the program are also provided.

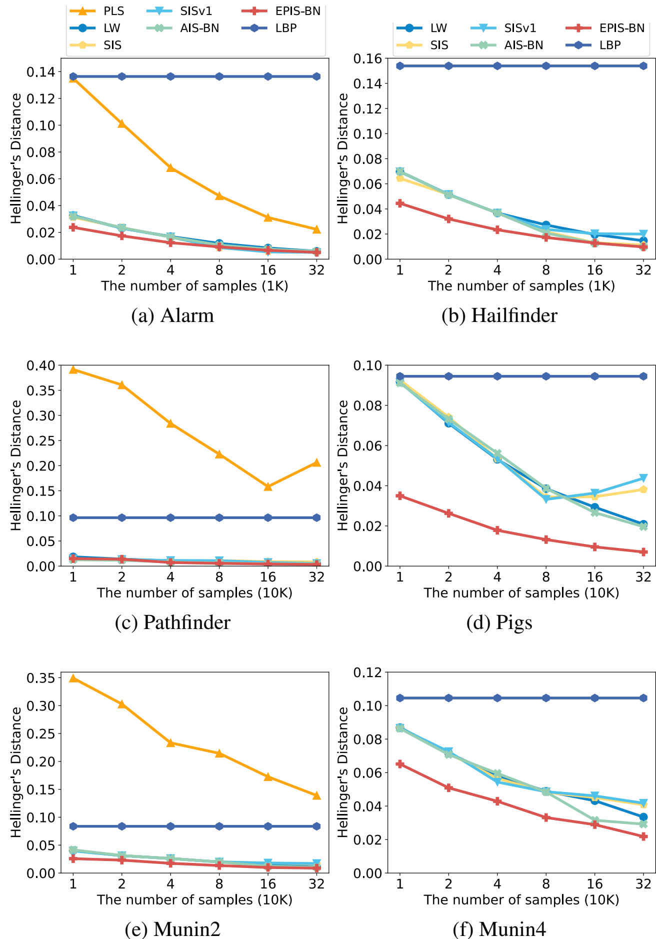


Figure 8: Convergence curves of the sampling-based algorithms measured by Hellinger's distance.

Fast-PGM among the networks. On the other hand, Fast-PGM has a relatively small speedup on *Pathfinder* due to the small ratio of its parallel part (i.e., $rp = 81\%$).

6.3 Effectiveness of Algorithms in Fast-PGM

We first compare the accuracy of the importance sampling algorithms in Fast-PGM, and then discuss the efficiency comparison among different algorithms.

6.3.1 Accuracy of Approximation

The accuracy of an approximate algorithm is inherently determined by the algorithm itself. For the sake of completeness, we here provide the accuracy results of the six algorithms implemented in Fast-PGM. The accuracy of approximation was measured by computing the probability distance between exact and approximate solutions. The distance metric we used is Hellinger's distance [31], consistent with the choice of EPIS-BN [59, 60]. Hellinger's distance $H(F_1, F_2)$ between two distributions F_1 and F_2, which have probabilities $P_1(x_{ij})$ and $P_2(x_{ij})$ for state j where $j = 1, 2, \ldots r_i$ for variable i respectively, is defined as:

$$H(F_1, F_2) = \sqrt{\frac{\sum_{X_i \in \mathcal{V} \setminus \mathcal{E}} \sum_{j=1}^{r_i} \left(\sqrt{P_1(x_{ij})} - \sqrt{P_2(x_{ij})}\right)^2}{\sum_{X_i \in \mathcal{V} \setminus \mathcal{E}} r_i}},$$

where $\mathcal{V}$ is the variables in the PGM, $\mathcal{E}$ is the evidence variables, and r_i is the number of states for variable i. We obtained the exact probability distribution of each variable using the exact inference method junction tree algorithm [33]. Fast-PGM also supports the Mean Square Error (MSE) metric, and other distance metrics can be easily extended. A major advantage of Hellinger's distance is that it can handle zero probabilities, which are common in PGMs. It weighs small absolute probability differences near 0 much more heavily than those near 1, where the probability differences near 0 are indeed more important.

Figure 8 shows Hellinger's distance with varying number of samples of the six importance sampling-based algorithms. The number of samples q ranges from 1,000 to 32,000 for *Alarm* and *Hailfinder*, since they can obtain good accuracy under this setting. For other larger networks, we varied q from 10,000 to 320,000 to achieve reasonably good accuracy. We also report the results of loopy belief propagation [39] after 100 iterations as a reference. The reported Hellinger's distance is averaged over the 1,000 test cases.

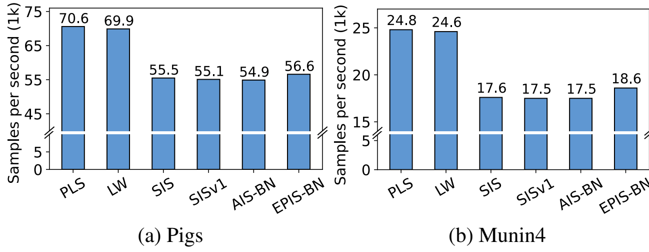


Figure 9: Comparisons of execution time with the number of samples generated on the sampling-based algorithms.

From Figure 8, we have the following four key observations. Firstly, EPIS-BN is the best, which is consistent with the findings in [59,60]. The propagation length d in EPIS-BN was set to only 2, which is already sufficient for EPIS-BN to yield very good results. In comparison, loop belief propagation with $d = 100$ is at least one order of magnitude worse than EPIS-BN. Secondly, PLS performance is the worst due to its poor sample efficiency. It discards many inconsistent samples, and the ratio of the valid samples is close to the likelihood of evidence. Therefore, PLS does not work on cases with extremely unlikely evidence, such as on *Hailfinder*, *Pigs* and *Munin4*. Due to this reason, we omit the results of PLS. Thirdly, among the algorithms that are required to update the importance function, AIS-BN is often better than SIS and SISv1. These algorithms try to gradually learn a better importance function, and the quality of the generated samples impacts the quality of the learned function. AIS-BN introduces sample weight and learning rate, which can weight the samples generated in the later stage more heavily. Fourthly, the default settings of the parameters in the heuristics may not be suitable for all the case, since the parameters are highly problem-dependent. After fine-tuning the parameters, the accuracy of EPIS-BN and AIS-BN could be further improved.

6.3.2 Efficiency Comparison Among Algorithms

We briefly discuss the comparisons of execution time with the number of samples here. Figure 9 shows the average number of samples generated per second per test case for the sequential run. We used the experiments on *Pigs* and *Munin4* as examples and similar observations can be made from other tested networks. We have the following three key observations. Firstly, PLS and LW can generate more samples per second compared to the other algorithms. The time saved comes from the fact that they do not need to compute the importance scores when instantiating the non-evidence variables because they are always 1. Due to its simplicity, LW becomes one of the most popular sampling algorithms for approximate inference. It often approaches or even outperforms some more sophisticated algorithms by generating more samples in the same amount of time. Secondly, SIS, SISv1 and AIS-BN generate fewer samples per second because of their learning overhead. For example, learning the importance function spends the three algorithms about an extra 1.07 seconds per test case for the experiment on *Munin4* using 32,0000 samples. Thirdly, EPIS-BN can generate slightly more samples than SIS, SISv1 and AIS-BN, since the overhead of EPIS-BN running loopy belief propagation is less than that of these learning-based algorithms.

7 Conclusion and Future Work

In this paper, we have proposed an efficient inference system, namely Fast-PGM, for importance sampling-based approximate inference on PGMs. Fast-PGM provides the implementations of mainstream importance sampling-based algorithms, together with various functionalities and rich interfaces that facilitate fast and easy optimizations, extensions and customization. Moreover, Fast-PGM is powered by a series of optimizations, including the data structure design, data fusion and reordering optimizations, computation simplification, and parallelization techniques. We highlight that our optimizations are easily applicable to various other PGM-related topics like structure learning or exact inference. Furthermore, our optimizations can inspire acceleration for a broader class of graph-based and probability-based algorithms.

Extensive experimental results showed that (i) Fast-PGM is 3 to 20 times faster than SMILE, and significantly outperforms another solution BNJ; (ii) Fast-PGM is effective regarding the accuracy of the approximate inference algorithms; (iii) parallelization notably improves the efficiency, where the practical speedups approach to the theoretical ones.

While this paper provides a strong foundation, we see many challenges that demand future improvements. Future work could extend Fast-PGM to distributed computing environments to improve its scalability and incorporate additional inference methods to broaden its applicability.

Acknowledgments

We sincerely thank the anonymous USENIX ATC'24 reviewers for their constructive and valuable comments. This research was funded by ARC Grant number DP190102443. Professor Ajmal Mian is the recipient of an Australian Research Council Future Fellowship Award (project number FT210100268) funded by the Australian Government.

Availability

Fast-PGM source code is freely available at https://github.com/jjiantong/FastPGM to facilitate further research and validation. We use publicly available BNs from https://www.bnlearn.com/bnrepository/ to generate the testing cases.

References

[1] Bruce Abramson, John Brown, Ward Edwards, Allan Murphy, and Robert L Winkler. Hailfinder: A bayesian system for forecasting severe weather. *International Journal of Forecasting*, 12(1):57–71, 1996.

[2] Srinivas M Aji and Robert J McEliece. The generalized distributive law. *IEEE transactions on Information Theory*, 46(2):325–343, 2000.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[4] Md Tanjin Amin, Faisal Khan, Salim Ahmed, and Syed Imtiaz. A data-driven Bayesian network learning method for process fault diagnosis. *Process Safety and Environmental Protection*, 150:110–122, 2021.

[5] S Andreassen, FV Jensen, SK Andersen, B Falck, U Kjrul, M Woldbye, AR Srensen, A Rosenfalck, and F Jensen. Computer-aided electromyography and expert systems, 1989.

[6] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, et al. Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020.

[7] Khaled Bayoudh, Raja Knani, Fayçal Hamdaoui, and Abdellatif Mtibaa. A survey on deep multimodal learning for computer vision: advances, trends, applications, and datasets. *The Visual Computer*, pages 1–32, 2021.

[8] Ingo A Beinlich, Henri Jacques Suermondt, R Martin Chavez, and Gregory F Cooper. The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks. In *European Conference on Artificial Intelligence in Medicine*, pages 247–256. Springer, 1989.

[9] Vaishak Belle and Ioannis Papantonis. Principles and practice of explainable machine learning. *Frontiers in big Data*, page 39, 2021.

[10] Charles-Edmond Bichot and Patrick Siarry. *Graph partitioning*. John Wiley & Sons, 2013.

[11] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518):859–877, 2017.

[12] BNJ library. `https://bnj.sourceforge.net/`, 2004. [Online].

[13] Giovanni Briganti, Marco Scutari, and Richard J McNally. A tutorial on Bayesian networks for psychopathology researchers. *Psychological methods*, 2022.

[14] Rommel Carvalho, KB Laskey, Paulo Costa, Marcelo Ladeira, Laécio Santos, and Shou Matsumoto. UnBBayes: modeling uncertainty for plausible reasoning in the semantic web. *Semantic web*, pages 953–978, 2010.

[15] Jian Cheng and Marek J Druzdzel. AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large Bayesian networks. *Journal of Artificial Intelligence Research*, 13:155–188, 2000.

[16] Steve B Cousins, William Chen, and Mark E Frisse. A tutorial introduction to stochastic simulation algorithms for belief networks. *Artificial Intelligence in Medicine*, 5(4):315–340, 1993.

[17] Zhiyong Cui, Longfei Lin, Ziyuan Pu, and Yinhai Wang. Graph Markov network for traffic forecasting with missing data. *Transportation Research Part C: Emerging Technologies*, 117:102671, 2020.

[18] Paul Dagum and Michael Luby. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial intelligence*, 60(1):141–153, 1993.

[19] Alta de Waal and Johan W Joubert. Explainable Bayesian networks applied to transport vulnerability. *Expert Systems with Applications*, 209:118348, 2022.

[20] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[21] Robert Fung and Kuo-Chu Chang. Weighing and integrating evidence for stochastic simulation in bayesian networks. In *Machine Intelligence and Pattern Recognition*, volume 10, pages 209–219. Elsevier, 1990.

[22] Alan Garvey and Victor Lesser. A survey of research in deliberative real-time artificial intelligence. *Real-Time Systems*, 6(3):317–347, 1994.

[23] Juan L Garzon, Óscar Ferreira, AC Zózimo, CJEM Fortes, AM Ferreira, LV Pinheiro, and MT Reis. Development of a Bayesian networks-based early warning system for wave-induced flooding. *International Journal of Disaster Risk Reduction*, 96:103931, 2023.

[24] Justin Grimmer. An introduction to Bayesian inference via variational approximations. *Political Analysis*, 19(1):32–47, 2011.

[25] David Earl Heckerman, Eric J Horvitz, and Bharat N Nathwani. Toward normative expert systems: Part i the pathfinder project. *Methods of information in medicine*, 31(02):90–105, 1992.

[26] Max Henrion. Propagating uncertainty in bayesian networks by probabilistic logic sampling. In *Machine Intelligence and Pattern Recognition*, volume 5, pages 149–163. Elsevier, 1988.

[27] Ariel Jaimovich, Ofer Meshi, Ian McGraw, and Gal Elidan. Fastinf: An efficient approximate inference library. *The Journal of Machine Learning Research*, 11:1733–1736, 2010.

[28] Jiantong Jiang, Zeyi Wen, Atif Mansoor, and Ajmal Mian. Fast parallel exact inference on Bayesian networks. In *ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 425–426, 2023.

[29] Jiantong Jiang, Zeyi Wen, and Ajmal Mian. Fast parallel Bayesian network structure learning. In *International Parallel and Distributed Processing Symposium*, pages 617–627. IEEE, 2022.

[30] Jiantong Jiang, Zeyi Wen, Zeke Wang, Bingsheng He, and Jian Chen. Parallel and distributed structured SVM training. *IEEE Transactions on Parallel and Distributed Systems*, 33:1084–1096, 2022.

[31] George Kokolakis and Photis Nanopoulos. Bayesian multivariate micro-aggregation under the Hellinger's distance criterion. *Research in official statistics*, 4(1):117–126, 2001.

[32] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[33] Steffen L Lauritzen and David J Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society: Series B (Methodological)*, 50(2):157–194, 1988.

[34] Huanhuan Li, Xujie Ren, and Zaili Yang. Data-driven Bayesian network for risk analysis of global maritime accidents. *Reliability Engineering & System Safety*, 230:108938, 2023.

[35] Andrew McCallum, Karl Schultz, and Sameer Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs. *Advances in Neural Information Processing Systems*, 22, 2009.

[36] Bojan Mihaljević, Concha Bielza, and Pedro Larrañaga. Bayesian networks for interpretable machine learning and optimization. *Neurocomputing*, 456:648–665, 2021.

[37] Joris M Mooij. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *The Journal of Machine Learning Research*, 11:2169–2173, 2010.

[38] Kevin Murphy et al. The bayes net toolbox for matlab. *Computing science and statistics*, 33(2):1024–1034, 2001.

[39] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Conference in Uncertainty in Artificial Intelligence*, 1999.

[40] Nihar Ranjan Nayak, Sumit Kumar, Deepak Gupta, Ashish Suri, Mohd Naved, and Mukesh Soni. Network mining techniques to analyze the risk of the occupational accident via Bayesian network. *International Journal of System Assurance Engineering and Management*, 13(1):633–641, 2022.

[41] PNL library. https://sourceforge.net/projects/openpnl/, 2013. [Online].

[42] Nico Potyka, Xiang Yin, and Francesca Toni. Explaining random forests using bipolar argumentation and Markov networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 9453–9460, 2023.

[43] Reuven Y Rubinstein and Dirk P Kroese. *Simulation and the Monte Carlo method*. John Wiley & Sons, 2016.

[44] Antonio Salmerón, Andrés Cano, and Serafín Moral. Importance sampling in Bayesian networks using probability trees. *Computational statistics & data analysis*, 34(4):387–413, 2000.

[45] Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. Formal verification of higher-order probabilistic programs: reasoning about approximation, convergence, Bayesian inference, and optimization. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[46] Marco Scutari. Bayesian network constraint-based structure learning algorithms: Parallel and optimised implementations in the bnlearn R package. *arXiv preprint arXiv:1406.7648*, 2014.

[47] Ross D Shachter and Mark A Peot. Simulation approaches to general probabilistic inference on belief networks. In *Machine Intelligence and Pattern Recognition*, volume 10, pages 221–231. Elsevier, 1990.

[48] Claus Skaanning. Blocking gibbs sampling for inference in large and complex bayesian networks with applications in genetics. 1997.

[49] SMILE library. https://www.bayesfusion.com/smile/, 2022. [Online].

[50] Minh Vu and My T Thai. PGM-Explainer: Probabilistic graphical model explanations for graph neural networks. *Advances in neural information processing systems*, 33:12225–12235, 2020.

[51] Martin J Wainwright, Tommi S Jaakkola, and Alan S Willsky. Tree-based reparameterization framework for analysis of sum-product and related algorithms. *IEEE Transactions on information theory*, 49(5):1120–1146, 2003.

[52] Chuanyuan Wang, Shiyu Xu, Duanchen Sun, and Zhi-Ping Liu. ActivePPI: quantifying protein–protein interaction network activity with Markov random fields. *Bioinformatics*, 39(9):btad567, 2023.

[53] Yangyang Xu, Zengmao Wang, and Xiaoping Zhang. Leveraging spatial residual attention and temporal Markov networks for video action understanding. *Neural Networks*, 169:378–387, 2024.

[54] Peiyu Yang, Naveed Akhtar, Jiantong Jiang, and Ajmal Mian. Backdoor-based explainable ai benchmark for high fidelity evaluation of attribution methods. *arXiv preprint arXiv:2405.02344*, 2024.

[55] Peiyu Yang, Naveed Akhtar, Zeyi Wen, and Ajmal Mian. Local path integration for attribution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 3173–3180, 2023.

[56] Peiyu Yang, Naveed Akhtar, Zeyi Wen, Mubarak Shah, and Ajmal Mian. Re-calibrating feature attributions for model interpretation. In *International Conference on Learning Representations*, 2022.

[57] Peiyu Yang, Zeyi Wen, and Ajmal Mian. Multi-grained interpre table network for image recognition. In *International Conference on Pattern Recognition*, pages 3815–3821. IEEE, 2022.

[58] Jonathan S Yedidia, William Freeman, and Yair Weiss. Generalized belief propagation. *Advances in neural information processing systems*, 13, 2000.

[59] Changhe Yuan and Marek J. Druzdzel. An importance sampling algorithm based on evidence pre-propagation. In *Conference in Uncertainty in Artificial Intelligence*, pages 624–631, 2003.

[60] Changhe Yuan and Marek J. Druzdzel. Importance sampling algorithms for Bayesian networks: Principles and performance. *Mathematical and Computer Modelling*, 43(9-10):1189–1207, 2006.

[61] Changhe Yuan and Marek J Druzdzel. Theoretical analysis and practical insights on importance sampling in bayesian networks. *International Journal of Approximate Reasoning*, 46(2):320–333, 2007.

[62] Huanzhou Zhu, Bo Zhao, Gang Chen, Weifeng Chen, Yijie Chen, Liang Shi, Yaodong Yang, Peter Pietzuch, and Lei Chen. MSRL: Distributed reinforcement learning with dataflow fragments. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 977–993, 2023.

A Appendix

A.1 Notation

Table 3 summarizes the notations and abbreviations used throughout the paper. For the terminologies related to the graph, capital letters, such as V_j, Y_j, denote random variables. Capital letters in math calligraphy, such as $\mathcal{V}$, $\mathcal{E}$, denote sets of variables. Lower case letters, such as v_j, denote the particular instantiations of variables. Bold lower case letters, such as $\mathbf{p}$, $\mathbf{e}$, denote configurations of sets of variables.

A.2 Feature Comparison of PGM Inference Libraries

Table 4 provides a feature comparison of Fast-PGM with various other PGM libraries that also support inference. Compared with other libraries, Fast-PGM supports more importance sampling-based approximate inference methods. Exact inference methods and other types of approximate inference methods (i.e., optimization-based methods) are also supported. Moreover, it is worth noting that although our focus in this paper is on PGM inference, Fast-PGM supports structure learning and parameter learning as well.

Table 3: The notations and abbreviations used in this paper.

Notation	Definition.
$\mathcal{A}$	Edges in a PGM.
$\mathbf{C}(V_j)$	Conditioning set of V_j in a PGM, encompassing the parents in a BN and the Markov blankets in an MRF.
d	Propagation length, used in LBP and EPIS-BN.
cumSt	A helper array in probability table data structure.
$c2i$	A function in probability table data structure that maps a configuration to an index.
$\mathcal{E}$, $\mathbf{e}$	Evidence variables and observations in a PGM.
$f^k(\mathcal{V})$	Importance function for stage k.
$\mathbb{G}$, $\mathbb{G}'$	Structure of a PGM; modified graph after reversing the direction of all edges in BN's $\mathbb{G}$.
I	An integral.
l	Updating interval.
n	Number of variables in a PGM.
$nVars$	Size of **vars** in probability table data structure.
o, o'	Topological ordering, and reverse topological ordering of a BN.
$\mathbb{P}$	Parameters of a PGM.
$P(V_j \vert \mathbf{C}(V_j))$	Conditional probability table (CPT) of V_j.
$P^k(V_j \vert \mathbf{C}(V_j))$	Importance conditional probability table (ICPT) of V_j in stage k.
prob	Main array in probability table data structure, storing the ICPT and CPT.
q	Desired number of samples.
r_{V_j}	Number of possible states of V_j.
rp	Ratio of the parallel part of the program.
s_i	A sample generated from the importance sampling process.
srcArr	Score arrays; each variable owns one and uses it to estimate the posterior distribution.
t	Number of parallel threads.
tmpW	Weight vector used in the sample generation module.
$\mathcal{V}$, $\mathcal{V}_c$	Random variables in a PGM; random variables associated with factor c in a PGM.
vars	An array in probability table data structure, containing the variable and its parents.
v^k	Weight of samples in stage k.
$\mathbf{w}_{allScr}$, $\mathbf{w}_{curScr}$	Accumulated score for all the available samples till now, and that for the samples of the current stage.
w_{iScr}	Importance score of s_i.
$\mathcal{Y}$	Variables of interest in a PGM.
$\eta(k)$	Learning rate for stage k.
θ	Probability threshold used in the heuristic strategy.
ϕ_f	Potential function corresponding to factor f in an MRF.

Table 4: Feature comparison of various PGM inference libraries. FastInf is marked as lack of full open-source availability because the provided link in the FastInf paper is no longer accessible.

Library	Fast-PGM	SMILE	BNJ	FastInf	libDAI	Factorie	PNL	BNT	UnBBayes
Language	C++	C++	Java	C++	C++	Scala	C++	Matlab	Java
Fully open-source	✓	✗	✓	✗	✓	✓	✓	✓	✓
Graphical user interface	✗	✓	✓	✗	✗	✗	✗	✗	✓
# of importance sampling-based inference methods	6	5	4	0	0	0	1	1	1
Other stochastic sampling inference support	✓	✓	✓	✓	✓	✓	✓	✓	✓
Optimization-based inference support	✓	✓	✓	✓	✓	✓	✓	✓	✗
Exact inference support	✓	✓	✓	✓	✓	✗	✓	✓	✗
Parameter learning support	✓	✓	✗	✓	✓	✓	✓	✓	✓
Structure learning support	✓	✓	✓	✗	✗	✗	✓	✓	✓

A.3 Further Details of The Crucial Modules

In Section 4.1, we briefly describe the four crucial modules identified in Fast-PGM. Here we provide further details.

A.3.1 Importance Function Initialization Module

This module is to initialize the initial importance function $f^0(\mathcal{V})$, which is a probability distribution that can be decomposed into independent ICPTs of each variable. In LW, SIS, SISv1 and AIS-BN, the initial importance function is

$$f^0(\mathcal{V}) = P(\mathcal{V}\backslash\mathcal{E})\Big|_{\mathcal{E}=\mathbf{e}} = \prod_{V_j\notin\mathcal{E}} P(V_j|\mathbf{C}(V_j))\Big|_{\mathcal{E}=\mathbf{e}},$$

where an ICPT $P^0(V_j|\mathbf{C}(V_j)) = P(V_j|\mathbf{C}(V_j))\Big|_{\mathcal{E}=\mathbf{e}}$ is maintained for each non-evidence variable V_j. PLS simply uses the joint probability distribution $P(\mathcal{V})$ as $f^0(\mathcal{V})$, without considering the evidence.

A.3.2 Sample Generation Module

This module is to generate one sample s_i and computes its importance score w_{iScr}. The basis of this module can be found in Equation 5, where w_{iScr} can be calculated as the product of the scores for all the variables. Here, we first take BNs as an example to describe the process of this module.

Algorithm 2 shows the pseudo-code of this module on BNs. In BNs, this module iteratively handles every variable following the topological ordering o. Inside the main loop, there are four steps for handling variable V_j:

1. The instantiation $\mathbf{c}$ of V_j's parents $\mathbf{C}(V_j)$ is obtained from the current instantiated sample s_i (Line 3).
2. A weight vector **tmpW** is obtained by reducing V_j's ICPT $P^k(V_j|\mathbf{C}(V_j))$ based on the instantiation $\mathbf{C}(V_j) = \mathbf{c}$ (Line 5). The resulting **tmpW** contains the probabilities of V_j's possible states and thus has a size of r_{V_j}, which is the number of possible states of V_j.
3. A random value is generated from V_j's state space based on **tmpW** (Line 6). *randByW* is a random generator that can generate a random value by weights.
4. The score of $V_j = s_i[j]$ can be computed and multiplied into w_{iScr} according to Equation 5.

For evidence variables, the process is the same but can be

Algorithm 2: sample generation module in BNs

input : topological ordering o of the BN, prior probability for each variable V_j $P(V_j|\mathbf{C}(V_j))\Big|_{\mathcal{E}=\mathbf{e}}$, ICPT for each variable V_j $P^k(V_j|\mathbf{C}(V_j))$
output : sample s_i, score w_{iScr}

```
1  w_iScr ← 1.0
2  for j ← 1 to n according to o do
3      p ← sampRed(s_i) // get instantiation of parents
4      if V_j ∉ E then
           /* for non-evidence variables */
5          tmpW ← tabRed(P^k(V_j|C(V_j)), p)
6          s_i[j] ← randByW(tmpW)
7          w_iScr ← w_iScr · P(V_j=s_i[j]|C(V_j)=p)|_{E=e} / P^0(V_j=s_i[j]|C(V_j)=p)
8      else
           /* for evidence variables */
9          s_i[j] ← v_j // v_j is the observed state of V_j
10         w_iScr ← w_iScr · P(V_j = s_i[j]|C(V_j) = p)|_{E=e}
```

simplified. The reason is that the ICPT $P^k(E_j|\mathbf{C}(E_j))$ of an evidence variable E_j with observed state e_j and r_{E_j} possible states can be viewed as a 0-1 vector of size r_{E_j}, where the value is 1 for the entry e_j and are 0 for the others. Therefore, the ICPT is exactly **tmpW**, and thus we do not need to reduce the ICPT. Meanwhile, according to the values of the ICPT, we will get e_j every time without randomness. To conclude, if V_j is an evidence variable, it is directly instantiated to its observed state v_j (Line 9), and the computation of V_j's score is also simplified since the denominator is 1 (Line 10).

The sample generation module on MRFs is similar to that on BNs. However, the variable sampling ordering differs. Unlike directed acyclic graphs where a topological ordering can be found as the sampling ordering, MRFs may contain cycles and lack a natural topological ordering. To generate samples on MRFs, we employ a sampling process similar to Gibbs sampling. In this process, each sample is still generated by iteratively sampling each variable. When sampling a variable V_j, all other variables are fixed, and the instantiation $\mathbf{c}$ of $\mathbf{C}(V_j)$ conditions the sampling of V_j. The order in which variables are sampled is not critical, and a common practice is to use a random or a fixed order to ensure the exploration of the entire search space.

A.3.3 Importance Function Update Module

This module updates the importance function periodically according to an updating interval l. SIS updates the importance function using $\mathbf{w}_{curScr}$:

$$P^{k+1}(V_j|\mathbf{C}(V_j)) \propto P^k(V_j|\mathbf{C}(X_j)) + \mathbf{w}_{curScr},$$

while SISv1 updates the importance function using $\mathbf{w}_{allScr}$:

$$P^k(V_j|\mathbf{Par}(V_j)) \propto P^0(V_j|\mathbf{Par}(X_j)) + k \cdot \mathbf{w}_{allScr}.$$

AIS-BN is a bit more complicated. The update is performed using $\mathbf{w}_{curScr}$ like SIS and a learning rate:

$$P^{k+1}(V_j|\mathbf{Par}(V_j)) = P^k(V_j|\mathbf{Par}(X_j)) + \eta(k)(\mathbf{w}_{curScr} - P^k(V_j|\mathbf{Par}(X_j))),$$

where $\eta(k)$ is the learning rate from 0 to 1. In AIS-BN, $\eta(k)$ is set to be:

$$\eta(k) = a(\frac{b}{a})^{k/k_{max}},$$

where $a = 0.4$, $b = 0.14$, k_{max} is the times of importance function updating.

A.4 Data Reordering in Fast-PGM

The purpose of data reordering in Fast-PGM is to let the parent variables of each variable be positioned as close as possible in memory. Note that this technique is only designed for BNs.

Our strategy is to store the variables according to the topological ordering o obtained by the following three steps.

Algorithm 3: BFS-based topological sorting

input : underlying graph $\mathbb{G}$ of the BN (n: #variables, $np[j]$: #parents for variable V_j, $par[j][k]$: the k-th parent for variable V_j)
output : topological ordering o of $\mathbb{G}$

```
1  res, in ← empty arrays size n, queue ← empty queue
2  for j ← 1 to n do
3      in[j] ← np[j] // store the in-degree
       /* enqueue variables with an in-degree of 0     */
4      if in[j] == 0 then
5          queue ← enqueue j
6  while queue is not empty do
7      v ← dequeue from queue // dequeue a variable
8      res ← append v /* append the dequeued variable
           */
9      for k ← 1 to np[v] do
           /* update each parent's in-degree           */
10         in[k] ← in[k] − 1
11         if in[k] == 0 then
               /* enqueue if its in-degree becomes 0   */
12             queue ← enqueue k
```

1. Reverse the direction of edges in the underlying graph $\mathbb{G}$ of the BN to create a modified DAG $\mathbb{G}'$.

2. Apply a breadth-first search (BFS)-based topological sorting algorithm on $\mathbb{G}'$ to get the topological ordering o' of $\mathbb{G}'$, as shown in Algorithm 3.

3. Get the topological ordering o of $\mathbb{G}$ as the reverse ordering of o'.

Figure 4 in Section 5.1.3 illustrates a toy example BN with six variables. In this example, variables V_5 and V_6 have multiple parents, namely $\{V_1, V_3\}$ and $\{V_2, V_4\}$ respectively. By using a BFS-based topological sorting algorithm, we obtain the ordering V_1, V_2, V_3, V_4, V_5, V_6. It can be observed that variables V_1 and V_3, as well as V_2 and V_4, are not stored contiguously in memory. By applying the proposed strategy, we can get an ordering V_4, V_2, V_3, V_1, V_6, V_5, as depicted in Figure 4. Consequently, V_1 and V_3 can be stored contiguously in memory, as well as variables V_2 and V_4.

The main concept behind this strategy is to make the children of each variable as close as possible in the modified graph $\mathbb{G}'$ by BFS-based topological sorting algorithm. Thus, we can also optimize the proximity of each variable's parents in the original graph $\mathbb{G}$. Although we cannot guarantee that all parents of each variable will be stored in contiguous memory, our strategy strives to shorten the distance between them. Therefore, this approach can help promote data locality and mitigate irregular memory access issues associated with BNs.

Table 5: Execution time comparison of the *sequential* version of Fast-PGM ("Ours") with SMILE ("S") and BNJ ("B") on 1000 test cases. Speedups of the sequential version of Fast-PGM over SMILE and BNJ are also reported. The desired number of samples is set to 1000 in this experiment. "N/A" in the "S" and"B" columns means that SMILE and BNJ do not implement the corresponding algorithm.

PGM	PLS					LW					SIS				
	Time (sec)			Speedup		Time (sec)			Speedup		Time (sec)			Speedup	
	S	B	Ours	S	B	S	B	Ours	S	B	S	B	Ours	S	B
Alarm	0.39	1.59	0.13	3.0	12.0	0.42	1.50	0.11	3.7	13.2	N/A	3.0	0.14	N/A	20.7
Hailfinder	0.76	5.7	0.26	3.0	22.1	0.66	2.7	0.21	3.1	13.1	N/A	15.8	0.26	N/A	60.0
Pathfinder	5.0	76.0	4.7	1.1	16.1	6.1	88.0	4.2	1.4	20.8	N/A	1.1k	5.2	N/A	210
Pigs	21.4	1.2k	14.3	1.5	84.1	14.6	1.2k	14.4	1.0	83.8	N/A	4.9k	17.7	N/A	279
Munin2	68.4	6.2k	36.8	1.9	168	48.9	6.0k	36.7	1.3	165	N/A	27k	47.1	N/A	565
Munin4	67.8	6.1k	40.8	1.7	149	47.7	7.6k	41.9	1.1	182	N/A	32k	54.4	N/A	580

PGM	SISv1					AIS-BN					EPIS-BN				
	Time (sec)			Speedup		Time (sec)			Speedup		Time (sec)			Speedup	
	S	B	Ours	S	B	S	B	Ours	S	B	S	B	Ours	S	B
Alarm	0.46	N/A	0.15	3.2	N/A	0.77	10.2	0.15	5.2	69.3	0.65	N/A	0.14	4.5	N/A
Hailfinder	0.83	N/A	0.27	3.1	N/A	1.3	27.7	0.27	4.9	104	0.83	N/A	0.26	3.2	N/A
Pathfinder	13.0	N/A	5.2	2.5	N/A	11.5	2.7k	5.3	2.2	513	6.0	N/A	5.4	1.1	N/A
Pigs	30.0	N/A	17.7	1.7	N/A	32.0	6.9k	17.7	1.8	393	17.8	N/A	17.5	1.0	N/A
Munin2	94.2	N/A	49.0	1.9	N/A	121	47k	48.6	2.5	939	56.3	N/A	46.5	1.2	N/A
Munin4	63.9	N/A	54.0	1.2	N/A	119	55k	54.4	2.2	1.0k	62.9	N/A	52.0	1.2	N/A

A.5 Discussion on Hybrid Parallelism

Besides the parallelization schemes discussed in Section 5.3, we also consider the hybrid sample-level and variable-level parallelism, including three different hybrid ways: nested, flattened, and pipeline, but each approach has its core drawbacks as follows:

- **Nested parallelism** increases communication/synchronization overhead at variable-level and introduces contention among parallel tasks at different levels.
- **Flattened parallelism** introduces additional memory and computational overhead due to the high complexity of the inference algorithms.
- **Pipeline parallelism** faces pipeline throughput and efficiency degradation, since different variables can have highly different workloads.

A.6 Extended Experiments on Execution Time Comparison with Existing Work

In Section 6.2.1, we compare our proposed Fast-PGM with the existing work SMILE and BNJ regarding execution time. Since both SMILE and BNJ are sequential implementations that do not support acceleration using multi-thread techniques, here we provide more results on the comparison between the two implementations and the *sequential* version of Fast-PGM.

Experimental results are summarized in Table 5. For ease of comparison, we also list the execution time of SMILE and BNJ, which is the same as in Table 2. As can be seen from the "Speedup" columns of the tables, the sequential version of Fast-PGM still outperforms the state-of-the-art implementation SMILE, achieving up to 5.2 times speedup. When compared to BNJ, the sequential version of Fast-PGM can achieve two to three orders of magnitude significant speedup. The speedups are mainly due to our proposed optimizations on memory and computation. These optimizations can be applied to both sequential and parallel implementations.