Probabilistic inference in the era of tensor networks and differential programming

Martin Roa-Villescas [1,*] Xuanzhao Gao [2] Sander Stuijk [1] Henk Corporaal [1] and Jin-Guo Liu [2]
[1] *Eindhoven University of Technology, Eindhoven 5600 MB, Netherlands*
[2] *Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China*

(Received 7 June 2024; accepted 7 August 2024; published 6 September 2024)

Probabilistic inference is a fundamental task in modern machine learning. Recent advances in tensor network (TN) contraction algorithms have enabled the development of better exact inference methods. However, many common inference tasks in probabilistic graphical models (PGMs) still lack corresponding TN-based adaptations. In this paper, we advance the connection between PGMs and TNs by formulating and implementing tensor-based solutions for the following inference tasks: (A) computing the partition function, (B) computing the marginal probability of sets of variables in the model, (C) determining the most likely assignment to a set of variables, (D) the same as (C) but after having marginalized a different set of variables, and (E) generating samples from a learned probability distribution using a generalized method. Our study is motivated by recent technical advances in the fields of quantum circuit simulation, quantum many-body physics, and statistical physics. Through an experimental evaluation, we demonstrate that the integration of these quantum technologies with a series of algorithms introduced in this study significantly improves the performance efficiency of existing methods for solving probabilistic inference tasks.

DOI: 10.1103/PhysRevResearch.6.033261

I. INTRODUCTION

Probabilistic inference is a fundamental component of machine learning. It enables machines to reason, predict, and assist experts in making decisions under uncertain conditions. The main challenge in applying exact inference techniques lies in the explosion of the computational cost as the number of variables involved increases. Unfortunately, modeling real-world problems often demands a high number of variables. Because of this, performing probabilistic inference remains an intractable endeavor in many practical applications.

In the past decades, several methods have been developed to enhance the computational efficiency of exact inference in complex models. Clustering methods, which include the family of junction tree algorithms [1,2], Symbolic probabilistic inference [3–5], weighted model counting [6,7], and differential-based methods [8,9] stand out as popular approaches.

Tensor networks (TNs), widely used in quantum many-body physics and quantum computation [10], are gaining increasing attention in the machine learning community. These networks have been shown to be an exceptionally powerful framework for modeling many-body quantum states [11]. Notable examples of TNs include matrix product states (MPS) [12], tree tensor networks (TTN) [13], multiscale entanglement renormalization ansatz (MERA) [14], and projected entangled pair states (PEPS) [15]. In recent years, they have become increasingly popular for classical benchmarking of quantum computing devices [16–19]. The application of TNs in machine learning includes both supervised [20] and unsupervised learning. In unsupervised learning, the focus has primarily been on generative modeling, where the goal is to learn a model's joint probability distribution from data and generate samples based on it. For instance, Han *et al.* [21] proposed using an MPS network for this purpose, ensuring that the TN topology is constrained to a chain-like structure. Building on this idea, Cheng *et al.* [22] advocated for the use of a TTN over an MPS network, aiming to enhance representational capabilities and to improve efficiency in both training and sampling.

While there have been notable advancements in understanding the theoretical duality between TNs and PGMs [23] and the integration of several TN techniques for generative sampling [21,22], many common probabilistic tasks in PGMs still lack corresponding TN-based adaptations. In this paper, we bridge the gap between PGMs and TNs further by formulating and implementing tensor-based solutions for a series of important probabilistic tasks. Specifically, given evidence for a subset of the variables in the model, we formulate and provide TN-based implementations for computing:

(1) the partition function (PR),
(2) the marginal probability distribution over sets of variables (MAR),
(3) the most probable explanation (most likely assignment) to all variables of the model (MPE),
(4) the maximum marginal *a posteriori* (MMAP), i.e., the most likely assignment to a set of variables after marginalizing a different set, and
(5) samples from the generative model (SAM).

*Contact author: m.roa.villescas@tue.nl

Our paper features a unity-tensor approach for computing marginal probabilities and the most likely assignment, presenting an alternative to existing methods such as the data network approach described in [24]. Positioned at the intersection of tensor networks, differentiable programming, and tropical algebra, this paper uses a combination of these modern numerical tools to reframe and accelerate the probabilistic inference tasks previously described.

Inspired by recent technical progress in the fields of quantum circuit simulation, quantum many-body physics, and statistical physics, our research aims to capitalize on these advancements. We employ hyperoptimized contraction order finding algorithms that have evolved in classical benchmarking of quantum computing devices [16–19]. These algorithms, which optimize both the computation time and runtime memory usage, include local search methods utilizing simulated annealing [25], min-cut-based methods [26], and greedy algorithms. Local search methods, such as simulated annealing, iteratively adjust tensor contraction sequences by applying a series of strategic transformations, aimed at optimizing the balance between computational load and memory usage, thus enhancing overall efficiency. Min-cut-based methods optimize tensor network computations by treating the network as a graph, strategically partitioning it into smaller sections to reduce data transfer and communication costs, thereby decreasing the overall computational load and speeding up processing times. Greedy methods select the most beneficial contractions step-by-step, optimizing immediately without a global configuration view, offering substantial time savings. This study also benefits from the latest advances of tropical tensor networks [27] followed by the introduction of generic tensor networks [28], which allow us to seamlessly devise performant solutions for the different inference tasks described earlier by adjusting the element types of a consistent tensor network. Our implementation also leverages cutting-edge developments commonly found in tensor network libraries, including a highly optimized set of BLAS routines [29,30] and GPU technology.

We present experimental results demonstrating that our tensor-based implementation is highly effective in advancing current methods for solving probabilistic inference tasks. Our library exhibits speedups of three to four orders of magnitude compared to a series of established solvers for the exact inference tasks mentioned above. Furthermore, we present experimental results indicating that by employing a GPU instead of a CPU, our proposed implementation can accelerate the inference of MMAP tasks by up to two orders of magnitude when the problem's computational cost exceeds a certain threshold. The ability of our library to facilitate the seamless use of a GPU instead of a CPU for solving probabilistic inference tasks represents a significant advantage. The source code for the methods described in this paper is available in a Julia package by the name of `TensorInference.jl` [31], licensed under the MIT open-source license.

The remainder of this paper is organized as follows. Section II provides a review of tensor networks, laying the foundational concepts necessary for understanding subsequent discussions. In Sec. III, we delve into the formulation of various probabilistic modeling tasks in terms of tensor network contractions, including the partition function (PR, Sec. III A), the marginal probability (MAR, Sec. III B), the most probable explanation (MPE, Sec. III C), the maximum marginal *a posteriori* (MMAP, Sec. III D), and sampling (SAM, Sec. III E). Section IV presents benchmarks and empirical results to demonstrate the practical implications of our approach. Finally, we conclude the paper in Sec. V, where we discuss the implications, limitations, and potential future directions of our paper.

II. TENSOR NETWORKS

Tensor networks serve as a fundamental tool for modeling and analyzing correlated systems. This section reviews their fundamental concepts.

A tensor is a mathematical object that generalizes scalars, vectors, and matrices. It can have multiple dimensions and is used to represent data in various mathematical and physical contexts. It is formally defined as follows:

Definition II.1 (Tensor). A tensor T associated to a set of discrete variables V is defined as a function that maps each possible instantiation of the variables in its scope $\mathcal{D}_V = \prod_{v\in V} \mathcal{D}_v$ to an element in the set $\mathcal{E}$, where $\mathcal{D}_v$ is the set of all possible values that the variable v can take. The function T_V is given by

$$T_V : \prod_{v\in V} \mathcal{D}_v \to \mathcal{E}. \tag{1}$$

Within the context of probabilistic modeling, the elements in $\mathcal{E}$ are non-negative real numbers, while in other scenarios, they can be of generic types.

Tensors are typically represented as multidimensional arrays, where each dimension is assigned a specific label or name. In probabilistic modeling, these labels correspond to *random variables* (or *variables* for short). The collective set of variables upon which a tensor operates is known as its *scope*. Before introducing the definition of a tensor network, it is important to define the concept of *slicing* (or *indexing*) tensors based on variable assignments. Let T_V be a tensor defined over the set of variables V. Let M be another set of variables with an arbitrary relationship to the set V, i.e., M and V may have all, some, or no elements in common, or one may be a subset of the other. The notation $M = m$ denotes the assignment of specific values denoted by m to the variables in M. The operation of slicing a tensor, denoted as $T_{V|M=m}$, involves evaluating the tensor T_V according to the assignment $M = m$. This operation effectively reduces the dimensions of T_V by constraining it to the subspace where $M = m$. Note that if V and M are disjoint, T_V remains unchanged.

We now turn our attention to the formal definition of a *tensor network*.

Definition II.2 (Tensor network [28,32,33]). A tensor network is a mathematical framework for defining multilinear maps, which can be represented by a triple $\mathcal{N} = (\Lambda, \mathcal{T}, V_0)$, where

(i) Λ is the set of variables present in the network $\mathcal{N}$.

(ii) $\mathcal{T} = \{T_{V_k}\}_{k=1}^{K}$ is the set of input tensors, where each tensor T_{V_k} is associated with the labels V_k.

(iii) V_0 specifies the labels of the output tensor.

Specifically, each tensor $T_{V_k} \in \mathcal{T}$ is labeled by a set of variables $V_k \subseteq \Lambda$, where the cardinality $|V_k|$ equals the rank

of T_{V_k}. The multilinear map, or the *contraction*, applied to this triple is defined as

$$T_{V_0} = \mathtt{con}(\Lambda, \mathcal{T}, V_0) \stackrel{\text{def}}{=} \sum_{m \in \mathcal{D}_{\Lambda \setminus V_0}} \prod_{T_V \in \mathcal{T}} T_{V|M=m}, \tag{2}$$

where $M = \Lambda \setminus V_0$.

For instance, matrix multiplication can be described as the contraction of a tensor network given by

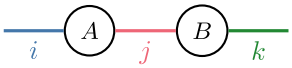
$$(AB)_{\{i,k\}} = \mathtt{con}(\{i, j, k\}, \{A_{\{i,j\}}, B_{\{j,k\}}\}, \{i, k\}), \tag{3}$$

where matrices A and B are input tensors containing the variable sets $\{i, j\}, \{j, k\}$, respectively, which are subsets of $\Lambda = \{i, j, k\}$. The output tensor comprises variables $\{i, k\}$ and the summation runs over variables $\Lambda \setminus \{i, k\} = \{j\}$. The contraction corresponds to

$$(AB)_{\{i,k\}} = \sum_j A_{\{i,j\}} B_{\{j,k\}}. \tag{4}$$

Definition II.2 introduces a minor generalization of the standard tensor network definition commonly used in physics. It allows a label to appear more than twice across the tensors in the network, deviating from the conventional practice of restricting each label to two appearances. This generalized form, while maintaining the same level of representational power, has been demonstrated to potentially reduce the network's treewidth [28], a metric that measures its connectivity.

Diagrammatically, a tensor network can be represented as an *open hypergraph*, where each tensor is mapped to a vertex and each variable is mapped to a hyperedge. Two vertices are connected by the same hyperedge if and only if they share a common variable. The diagrammatic representation of the matrix multiplication shown in Eq. (4) is given as follows:

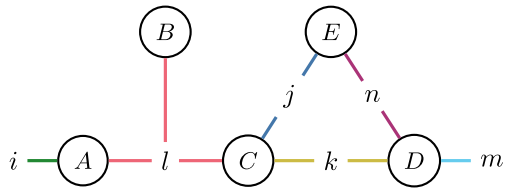


Here, we use different colors to denote different hyperedges. Hyperedges for i and k are left open to denote variables of the output tensor.

A somewhat more elaborate example of this is as follows:

$$\begin{aligned} &\mathtt{con}(\{i, j, k, l, m, n\}, \\ &\quad \{A_{\{i,l\}}, B_{\{l\}}, C_{\{k,j,l\}}, D_{\{k,m,n\}}, E_{\{j,n\}}\}, \\ &\quad \{i, m\}) \\ &= \sum_{j,k,l,n} A_{\{i,l\}} B_{\{l\}} C_{\{k,j,l\}} D_{\{k,m,n\}} E_{\{j,n\}}, \end{aligned} \tag{5}$$

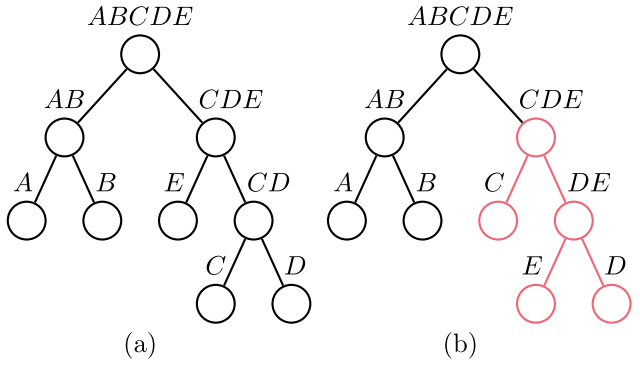
which is graphically represented by the following open hypergraph:



Note that the variable l is shared by three tensors, making regular edges, which by definition connect two nodes, insufficient for its representation. This motivates the need for hyperedges, which can connect a single variable to any number of nodes.

We would now like to stress an important property of tensor networks, namely the *contraction order*. While the summations in Eq. (5) (over j, k, l, and n) can be carried out in any order without affecting the contraction result, the order in which these summations are performed significantly impacts the computational cost required to contract the network. Finding the optimal order of variables to be contracted in a tensor network is crucial for overall efficiency. To minimize the computational cost of a TN contraction, one must optimize over the different possible orderings of pairwise contractions and find the optimal case [33]. This problem is NP hard. However, several efficient heuristic contraction order finding algorithms [25,26] have been developed by the community. Given a contraction order, each pairwise contraction can be further decomposed into a series of BLAS operations, which are highly optimized for modern hardware [34].

To illustrate this point, consider the two contraction orders specified below for evaluating Eq. (5) using binary trees:



These binary trees specify the order of pairwise contractions between tensors, starting from the bottom (leaves) and moving to the top (root). Each leaf represents an initial tensor, and each internal node results from contracting two child tensors and summing out any indices not needed for later operations. Note that directly evaluating Eq. (5) requires $O(n^6)$ time, where n is the dimension of each variable. In contrast, both contraction orders illustrated above reduce the time complexity to $O(n^4)$. However, there is an important difference between these two orders, highlighted in red. Specifically, contraction order (b) is preferred over (a) because of its lower *space complexity*, which is determined by the highest rank among all intermediate tensors. For example, the intermediate tensor CD in order (a) has a rank of 4, whereas tensor DE in order (b) has a rank of 3. Lower space complexity helps to reduce the memory usage bottleneck in tensor network computations.

As a final remark, it is worth noting that any contraction order yields correct results because of the *associative* and *commutative* properties of addition and multiplication, as discussed in [28]. These properties hold for many common element types, including real numbers, complex numbers, and tropical semirings. The complexity of the contraction process is independent of the specific tensor element type, provided that it adheres to these associative and commutative properties.

III. TENSOR NETWORKS FOR PROBABILISTIC MODELING

Probabilistic graphical models (PGMs) are a class of models that use graphs to represent complex dependencies between random variables and reason about them, with Bayesian networks, Markov random fields, and factor graphs being among the most prevalent examples. While tensor networks and PGMs share a conceptual foundation in representing multivariate relationships graphically, they have traditionally evolved in parallel within distinct fields of study. Despite their different origins, both frameworks exhibit remarkable similarities in structure and functionality. They are both used to decompose high-dimensional objects into a network or graph of simpler, interconnected components. The aim of this section is to reformulate probabilistic modeling in terms of tensor networks, thereby allowing the field of probabilistic modeling to leverage the remarkable developments achieved in tensor network modeling in recent years.

It is worth noting that there may not be a tensor network that can fully capture the conditional independence assumptions encoded by a specific Bayesian network. This limitation arises because tensor networks use undirected edges, whereas Bayesian networks use directed edges. Although every Bayesian network can be converted into an undirected graph through a process called *moralization*, this can result in the loss of independence information. Moralization involves removing the direction from the edges and connecting nodes that have common children. Independence information is lost as new edges are added. However, if no new edges are added, then the transformed graph maintains all the original independence assumptions.

The analyses and discussions in this section are framed within the context of a probabilistic model. This model is characterized by a set of variables Λ with a corresponding joint *probability mass function*$p(\Lambda)$. Furthermore, $p(\Lambda)$ is represented as a product of tensors in $\mathcal{T}$, where each tensor $T_V \in \mathcal{T}$ is associated with a subset of variables V in Λ. Within Λ, we distinguish three disjoint subsets of variables: the *query* variables Q, representing our variables of interest; the *evidence* variables E, which denote observed variables; and the *nuisance* variables M, which include the remaining variables.

In what follows, we present the formulation of prevalent probabilistic inference tasks through the application of tensor network methodologies. All the tasks presented below take into account given evidence for a subset of the variables in the model. These tasks include:

(1) Calculating the *partition function* (PR), also referred to as the *probability of evidence* (Sec. III A).

(2) Computing the marginal probability distribution over sets of variables (MAR) (Sec. III B).

(3) Finding the most likely assignment to all variables, formally referred to as the *most probable explanation* (MPE) (Sec. III C).

(4) Finding the most likely assignment to a set of query variables after marginalizing out the remaining variables, also known as the *maximum marginal a posteriori* (MMAP) estimate (Sec. III D).

(5) Generating samples from the learned distribution (SAM), also known as *generative modeling* (Sec. III E).

For more information about these tasks, refer to the website of the UAI 2022 Probabilistic Inference Competition [35].

A. Partition function (PR)

The partition function is a central concept in statistical mechanics and probabilistic graphical models. In statistical mechanics, it sums over all possible states of a system, weighted by their energy, to derive key thermodynamic quantities. In probabilistic models, it not only normalizes the joint probability distribution, ensuring that the probabilities of all possible outcomes sum to one, but also facilitates model comparison by providing a measure of how well each model explains the observed data.

Suppose we are given some evidence e observed over a set of variables $E \subseteq \Lambda$. The partition function is calculated by summing the joint distribution p over all possible values of the variables $M \subseteq \Lambda$ that are not in E, i.e., $E \cap M = \emptyset$. Thus, the partition function corresponds to

$$p(E = e) = \sum_{m \in \mathcal{D}_{\Lambda \setminus E}} p(E = e, M = m). \tag{6}$$

Let us denote the set of tensors associated with the variables in Λ as $\mathcal{T} = \{T_V\}$, where T_V is a tensor associated with the variables $V \subseteq \Lambda$. The partition function can be expressed as the tensor network contraction given by

$$p(E = e) = \sum_{m \in \mathcal{D}_{\Lambda \setminus E}} \prod_{T_V \in \mathcal{T}} T_{V|E=e,M=m}, \tag{7}$$

where $\mathcal{T}_{E=e} = \{T_{V|E=e} \mid T_V \in \mathcal{T}\}$ is the set of tensors sliced over the fixed values of the evidence variables. We can express this operation more succinctly using the definition of a tensor contraction shown in Eq. (2), which results in

$$p(E = e) = \texttt{con}(\Lambda \setminus E, \mathcal{T}_{E=e}, \emptyset). \tag{8}$$

Here, since evidence variables are fixed, the contraction is performed over the remaining variables in $\Lambda \setminus E$. Correspondingly, the tensors in $\mathcal{T}_{E=e}$ are sliced according to the evidence. To obtain the marginal probability, all remaining variables are marginalized, leaving the output tensor with an empty label set $\emptyset$.

B. Marginal probability (MAR)

The marginal probability (MAR) task involves computing the conditional probability distribution for the set of query variables Q, based on known information about the evidence variables E, i.e., $p(Q|E = e)$. This process requires marginalizing out nuisance variables M from the joint distribution $p(\Lambda)$, effectively averaging their impact within the joint probability distribution. Such averaging is crucial as it accounts for the indirect effect of these variables on the resulting marginal probabilities, thereby enabling predictions and informed decision-making with limited information. In what follows, we introduce a novel tensor-based algorithm for efficiently computing marginal probability distributions over multiple sets of query variables, which demonstrates

improved performance over traditional approaches, such as the junction tree algorithm [1].

Given some evidence $E = e$, the marginal probability query computes the conditional distribution over the query variables $Q \subseteq \Lambda$. This is denoted as $p(Q = q \mid E = e)$, where $q \in \mathcal{D}_Q$ and it is ensured that $E \cap Q = \emptyset$. Following the *law of total probability*, the marginal probability can be obtained as follows:

$$p(Q|E = e) = \frac{p(Q, E = e)}{p(E = e)}. \tag{9}$$

The numerator $p(Q, E = e)$ corresponds to the joint marginal probability of configurations, which is given by

$$p(Q, E = e) = \sum_{m \in \mathcal{D}_{\Lambda \setminus (Q,E)}} p(Q, E = e, M = m), \tag{10}$$

or, equivalently, by the following tensor network contraction:

$$p(Q, E = e) = \texttt{con}(\Lambda \setminus E, \mathcal{T}_{E=e}, Q). \tag{11}$$

The denominator $p(E = e)$ in Eq. (9) corresponds to the partition function, calculated according to Eq. (8).

Consider the scenario where we want to obtain the marginal probabilities for multiple sets of query variables. For simplicity, we consider the sets of single variables $Q_i \in \mathcal{Q}$, where $\mathcal{Q} = \{\{q_i\} \mid q_i \in \Lambda \setminus E\}$. Using the above strategy would require contracting $O(|\mathcal{Q}|)$ different tensor networks, which is inefficient. In the following, we present an automatic differentiation [36] based approach to obtain the marginal probabilities for all sets of variables in $\mathcal{Q}$ by contracting the tensor network only once. The proposed algorithm reduces the problem of finding marginal probability distributions to the problem of finding the gradients of introduced auxiliary tensors, which can be efficiently handled by differential programming. The differentiation rules for tensor network contraction can be represented as the contraction of the tensor network shown in Theorem III.1.

Theorem III.1 (Tensor network differentiation). Let $(\Lambda, \mathcal{T}, \emptyset)$ be a tensor network with scalar output. The gradient of the tensor network contraction with respect to $T_V \in \mathcal{T}$ is

$$\frac{\partial \texttt{con}(\Lambda, \mathcal{T}, \emptyset)}{\partial T_V} = \texttt{con}(\Lambda, \mathcal{T} \setminus \{T_V\}, V). \tag{12}$$

That is, the gradient corresponds to the contraction of the tensor network with the tensor T_V removed and the output label set to V.

The proof of Theorem III.1 is given in Sec. A. The algorithm to obtain the marginal probabilities for all sets of variables in $\mathcal{Q}$ is summarized as follows:

(1) Add a unity tensor $\mathbb{1}_{Q_i}$ to the tensor network for each variable set $Q_i \in \mathcal{Q}$. A unity tensor is defined as a tensor with all elements equal to one. The augmented tensor network is represented as follows:

$$\mathcal{T}_{\text{aug}} \leftarrow \mathcal{T} \cup \{\mathbb{1}_{Q_i} | Q_i \in \mathcal{Q}\}. \tag{13}$$

The introduction of unity tensors does not change the contraction result of a tensor network.

(2) *Forward pass*: Contract the augmented tensor network to obtain

$$p(E = e) = \texttt{con}(\Lambda \setminus E, (\mathcal{T}_{\text{aug}})_{E=e}, \emptyset). \tag{14}$$

In practice, the tensor network is contracted according to a given pairwise contraction order of tensors, caching intermediate results for later use. As detailed in Sec. II, this order can be specified using a binary tree, which we will refer to as a binary contraction tree.

(3) *Backward pass*: Compute the gradients of the introduced unity tensors by back propagating the contraction process in Step 2. During back-propagation, the cached intermediate results from Step 2 are used. The resulting gradients are

$$\mathcal{G} = \left\{ \frac{\partial p(E = e)}{\partial \mathbb{1}_{Q_i}} \,\middle|\, Q_i \in \mathcal{Q} \right\}. \tag{15}$$

Each gradient tensor $\partial p(E = e)/\partial \mathbb{1}_{Q_i}$ corresponds to a joint probability $p(Q_i, E = e)$. Dividing this gradient tensor by the partition function $p(E = e)$ yields the marginal probability $p(Q_i|E = e)$.

In Step 1, we augment the tensor network by adding a rank 1 unity tensor for each variable in $\mathcal{T}$. These tensors, being vectors, can be absorbed into existing tensors of an optimized contraction tree, thereby not considerably affecting the overall computing time. However, the computational cost may increase significantly when unity tensors for joint marginal probabilities of multiple variables are introduced.

The caching of intermediate contraction results in Step 2 is automatically managed by a differential programming framework. These cached results are then utilized in the back-propagation step. While this caching does not significantly increase the computing time, it does lead to greater memory usage. Practically, the added memory cost is typically just a few times greater than the forward pass's peak memory. This occurs because of the program's nonlinear nature, often constrained by a handful of intensive contraction steps. Step 3 follows from the observation that for any $Q_i \in \mathcal{Q}$, the following holds:

$$p(E = e) = \sum_{q \in \mathcal{D}_{Q_i}} p(E = e, Q_i = q) \mathbb{1}_{L|Q_i=q}. \tag{16}$$

Using Theorem III.1, differentiating $p(E = e)$ with respect to $\mathbb{1}_{Q_i}$ is equivalent to removing the unity tensor $\mathbb{1}_{Q_i}$ from the tensor network and setting the output label to Q_i, the result of which corresponds to the joint probability $p(Q_i, E = e)$.

Corollary III.1. Let $\mathcal{P}$ be a program to contract a tensor network $(\Lambda, \mathcal{T}, \emptyset)$ using a binary contraction tree. The time required to differentiate $\mathcal{P}$ using reverse-mode automatic differentiation is three times that required to evaluate $\mathcal{P}$.

Proof. Since the program $\mathcal{P}$ is decomposed into a series of pairwise tensor contractions, to explain the overall factor of three, it suffices to show that for any pairwise tensor contraction, the computation time for backward-propagating gradients is twice that of the forward pass. Given a pairwise tensor contraction, $\texttt{con}(\Lambda, \{A_{V_a}, B_{V_b}\}, V_c)$, where $\Lambda = V_a \cup V_b \cup V_c$, its computational cost is $\prod_{v \in \Lambda} |\mathcal{D}_v|$, where $|\cdot|$ denotes the cardinality of a set. The backward rule for pairwise tensor contraction is also a tensor contraction. Let the adjoint of the output tensor be $\overline{C} \equiv \frac{\partial \mathcal{L}}{\partial C}$, where $\mathcal{L}$ is a scalar loss function, the explicit form of which does not need to be known. As shown in Sec. A, the backward rule for tensor

contraction is

$$\begin{aligned}\overline{A}_{V_a} &= \texttt{con}(\Lambda, \{\overline{C}_{V_c}, B_{V_b}\}, V_a),\\ \overline{B}_{V_b} &= \texttt{con}(\Lambda, \{A_{V_a}, \overline{C}_{V_c}\}, V_b).\end{aligned} \tag{17}$$

Since the above tensor networks share the same set of unique variables, their computing time is roughly equal to that of the forward computation. Consequently, the reverse-mode automatic differentiation for a tensor network is approximately three times more costly than computing only the forward pass, thus proving the theorem. ■

C. Most probable explanation (MPE)

Consider the probabilistic model given by the tensor network $p(\Lambda) = (\Lambda, \mathcal{T}, \Lambda)$, and suppose we are given evidence $E = e$, where $E \subseteq \Lambda$. The objective of the most probable explanation (MPE) estimate is to determine the most likely assignment q for the variables $Q \in \Lambda\backslash E$. Mathematically, this can be expressed as

$$\text{MPE}(E = e) = \arg\max_{q \in \mathcal{D}_{\Lambda\backslash E}} p(Q = q, E = e),$$

where the goal is not only to find the most likely assignment $Q = q^*$ but also to calculate its corresponding probability. In the subsequent discussion, we will transition the tensor elements from real positive numbers to max-plus numbers and reformulate the configuration extraction problem within the context of differential programming.

1. Tropical tensor networks

Tropical algebra, a nonstandard algebraic system, diverges from classical algebra by replacing the standard operations of addition and multiplication with different binary operations. In the following discussion, we focus on the max-plus tropical algebra, a variant where the operations are *maximum* for addition and *plus* for multiplication.

Definition III.1 (Tropical tensor network). A *tropical tensor network* [27] is a tensor network with max-plus tropical numbers as its tensor elements. Given two max-plus tropical numbers $a, b \in \mathbb{R} \cup \{-\infty\}$, their addition and multiplication operations are defined as

$$\begin{aligned}a \oplus b &= \max(a, b),\\ a \odot b &= a + b.\end{aligned} \tag{18}$$

Correspondingly, the *zero* element (or the additive identity) is mapped to $-\infty$, and the *one* element (or the multiplicative identity) is mapped to 0. Following from Eqs. (18) and (II.2), the *tropical contraction* applied to a tropical tensor network $(\Lambda, \mathcal{T}, V_0)$ is defined as

$$\texttt{tcon}(\Lambda, \mathcal{T}, V_0) = \max_{q \in \mathcal{D}_{\Lambda\backslash V_0}} \sum_{T_V \in \mathcal{T}} T_{V|Q=q}. \tag{19}$$

The max operation runs over all possible configurations over the set of variables absent in the output tensor.

Tropical tensor networks have been effectively employed in previous studies to determine both the ground-state energy and its degeneracy across various statistical physics models [27,28]. In this paper, we extend their application to include a broader range of tensor network topologies, utilizing them for probabilistic inference tasks. Given some evidence $E = e$, let us denote the MPE as $Q = q^* \in \mathcal{D}_{\Lambda\backslash E}$. The log probability of this MPE estimate can be computed as follows:

$$\begin{aligned}\log p(Q = q^*, E = e) &= \max_{q \in \mathcal{D}_{\Lambda\backslash E}} \log p(Q = q, E = e)\\ &= \max_{q \in \mathcal{D}_{\Lambda\backslash E}} \log \prod_{T_V \in \mathcal{T}} T_{V|(Q=q, E=e)}\\ &= \max_{q \in \mathcal{D}_{\Lambda\backslash E}} \sum_{T_V \in \mathcal{T}} \log T_{V|(Q=q, E=e)}\\ &= \texttt{tcon}(\Lambda\backslash E, \log(\mathcal{T})_{E=e}, \emptyset),\end{aligned} \tag{20}$$

where $\log(\mathcal{T}) \equiv \{\log(T_V) \mid T_V \in \mathcal{T}\}$ represents the application of the logarithm operation to each tensor in the set $\mathcal{T}$. The logarithm operation applied to a tensor is defined as taking the logarithm of each element within the tensor.

2. The most probable configuration

In the context of the MPE estimate, the primary interest often lies not in acquiring the log-probability of the MPE, calculated according to Eq. (20), but rather in obtaining the configuration of $Q = q^*$ itself. The algorithm for this purpose is summarized as follows:

(1) For each variable $v \in \Lambda$, add a unity tensor $\mathbb{1}_v$ that is associated with v to the tensor network. The augmented tensor network is given by

$$\mathcal{T}_{\text{aug}} \leftarrow \mathcal{T} \cup \{\mathbb{1}_{\{v\}} \mid v \in \Lambda\backslash E\}. \tag{21}$$

Once more, we emphasize that introducing unity tensors does not change the contraction result of the tensor network.

(2) Evaluate the log-probability of the MPE, given by

$$\log p(Q = q^*, E = e) = \texttt{tcon}(\Lambda\backslash E, \log(\mathcal{T}_{\text{aug}})_{E=e}, \emptyset), \tag{22}$$

where $Q = q^* \in \mathcal{D}_{\Lambda\backslash E}$ is the MPE. Intermediate contraction results are cached for future use.

(3) Backpropagate through the contraction process outlined in Step 2 to obtain the gradients for each log-unity vector,

$$\mathcal{G} = \left\{ \frac{\partial \log(p(Q = q^*, E = e))}{\partial \log(\mathbb{I}_{\{v\}})} \,\middle|\, v \in \Lambda \right\}. \tag{23}$$

Following a specific convention, we ensure that for each $G_v \in \mathcal{G}$, there exists exactly one nonzero entry, denoted as $G_v(q_v^*) = 1$. This unique entry q_v^* corresponds to the assignment of variable v in the MPE solution.

Steps 1 and 2 of this approach mirror their counterparts in the algorithm detailed in Sec. III B for computing marginal probabilities, with the notable distinction that tensor elements are now represented as tropical numbers. Step 3 follows from the observation that, although the introduced log-unity tensors (or zero tensors) do not affect the contraction result of the tropical tensor network, differentiating the contraction result with respect to these tensors yields a gradient signal at $Q = q^*$.

3. Back-propagation in tropical tensor networks

Obtaining the MPE configuration using back-propagation through a tropical tensor network is a nontrivial task, especially when there are multiple configurations with the same

maximum probability. In such cases, the gradient signal must be designed to keep only one of the configurations. To achieve this, we use a Boolean mask to represent the gradient signal of a tensor, with its elements being either 0 or 1. In the following, instead of deriving the exact backpropagation rule, we present a backward rule that only works for Boolean gradients, which is sufficient for the MPE task.

Theorem III.2. Given a pairwise contraction of two tropical tensors $\texttt{tcon}(\Lambda, \{A_{V_a}, B_{V_b}\}, V_c)$, where $\Lambda = V_a \cup V_b \cup V_c$, the backward rule, used for computing masks for nonzero gradients, is defined as follows:

$$\begin{aligned}\overline{A}_{V_a} &= \delta\big(A_{V_a}, \texttt{tcon}\big(\Lambda, \big\{C_{V_c}^{-1}\overline{C}_{V_c}, B_{V_b}\big\}, V_a\big)^{-1}\big),\\ \overline{B}_{V_b} &= \delta\big(B_{V_b}, \texttt{tcon}\big(\Lambda, \big\{A_{V_a}, C_{V_c}^{-1}\overline{C}_{V_c}\big\}, V_b\big)^{-1}\big).\end{aligned} \tag{24}$$

Proof. This rule is provable by reducing the tensor network contraction to tropical matrix multiplication $C = AB$. The backpropagation rule for tropical matrix multiplication has been previously derived in [28]. Here, we revisit the main results for completeness. We require the gradients to be either 0 or 1. This binary nature aligns with the representation of configurations using one hot vectors. Consequently, a Boolean mask can effectively be employed to extract or represent any given configuration in this context. The gradient mask for C is denoted as $\overline{C}$. The back-propagation rule for these gradient masks is

$$\overline{A}_{ij} = \delta\big(A_{ij}, ((C^{\circ -1} \circ \overline{C})B^{\mathsf{T}})_{ij}^{\circ -1}\big), \tag{25}$$

where δ is the Dirac delta function, returning one for equal arguments and zero otherwise. The notation $\circ$ signifies the element-wise product, and $^{\circ -1}$ indicates the element-wise inverse. Boolean *false* is equated with the tropical zero $(-\infty)$, and Boolean *true* is the tropical one (0). ■

In Eq. (24), the right-hand side primarily involves tropical tensor contractions, which can be efficiently handled using fast tropical BLAS routines [30]. Notably, the backward rule's computing time mirrors that of the forward pass.

D. Maximum marginal a posteriori (MMAP)

Tensor networks are equally applicable in the context of computing *maximum marginal a posteriori* (MMAP) estimations. This task involves computing the most likely assignment for the query variables, after marginalizing out the remaining variables. Consider a scenario with evidence e observed over variables $E \subseteq \Lambda$ and query variables $Q \subseteq \Lambda$, such that $E \cap Q = \emptyset$. Mathematically, the MMAP solution is given by

$$\begin{aligned}&\text{MMAP}(Q \mid E = e)\\ &\quad = \arg\max_{q \in \mathcal{D}_Q} \sum_{m \in \mathcal{D}_{\Lambda\setminus(Q,E)}} p(Q = q, M = m, E = e).\end{aligned} \tag{26}$$

Upon closer examination of Eq. (26), it is clear that the equation combines elements of both max-sum and sum-product networks. To optimize the computation process, we utilize a routine that divides the computation into two separate phases: conventional tensor network contraction and tropical tensor network contraction. The process to compute MMAP solutions using tensor networks is described as follows:

(1) Find a partition $\hat{\mathcal{S}}$ of $\mathcal{T}$ such that, for each marginalized variable $v \in \Lambda\setminus(Q \cup E)$, there exists an $\mathcal{S}_i \in \hat{\mathcal{S}}$ that contains all tensors associated with it.

(2) For each $\mathcal{S}_i \in \hat{\mathcal{S}}$, marginalize the variables in $\Lambda\setminus(Q \cup E)$ by contracting the tensor network

$$S_{\Lambda_i \cap Q} = \texttt{con}(\Lambda_i, (\mathcal{S}_i)_{E=e}, \Lambda_i \cap Q), \tag{27}$$

where Λ_i is the set of variables involved in $\mathcal{S}_i$.

(3) Solve the MPE problem on the probability model specified by tensor network $(Q, \{S_{\Lambda_1 \cap Q}, \ldots, S_{\Lambda_{|\hat{\mathcal{S}}|} \cap Q}\}, \emptyset)$, the result corresponds to the solution of the MMAP problem.

In Step 1, The sets in $\hat{\mathcal{S}}$ can be constructed by first choosing a marginalized variable $v \in \Lambda\setminus(Q \cup E)$, and then greedily including tensors containing v into the set.

E. Sampling the generative model (SAM)

In the following discussion, we examine the use of tensor networks for generating samples from learned distributions in the context of probabilistic modeling. This section aims to connect the contributions of this paper with other studies in the domain of tensor-based generative modeling, notably those by Han *et al.* [21] and Cheng *et al.* [22]. We introduce a generic framework for unbiased variable sampling that generalizes the sampling algorithms of these references.

Generating a sample from the generative distribution is closely related to the method for computing the partition function, as detailed in Sec. III A and summarized here: $p(E = e) = \texttt{con}(\Lambda\setminus E, \mathcal{T}_{E=e}, \emptyset)$, where E represents the set of evidence variables, and Λ denotes the set of all variables in the model. This computation involves summing the joint distribution p over all unobserved variables, which requires selecting a variable elimination order—also referred to as the contraction order—for all unobserved variables. Given a tensor network contraction that calculates the partition function, a sample can be generated by back-tracing this variable elimination process. This mechanism will be detailed next.

Let S_i be the set of variables eliminated in the i-th pairwise tensor contraction, where $i = 1, 2, \ldots, K$ and $\bigcup_{i=1}^{K} S_i = \Lambda\setminus E$. Samples for each variable elimination group S_i are generated in the reverse order of the variable elimination sequence. This process involves generating a sample for the variables in S_i, conditioned on the samples for the variables eliminated in subsequent steps, as specified by the following conditional probability distribution:

$$\boldsymbol{s}_{S_i} \sim p(S_i | S_{i+1} = \boldsymbol{s}_{S_{i+1}}, \ldots, S_K = \boldsymbol{s}_{S_K}). \tag{28}$$

The resulting sample is composed of the combined samples from each variable elimination group S_i, given by $\boldsymbol{s}_{S_1}, \boldsymbol{s}_{S_2}, \ldots, \boldsymbol{s}_{S_K}$. To further explore the generation of a sample for a variable elimination group S_i, as described in Eq. (28), we refer to Fig. 1, which illustrates a single tensor pairwise tensor contraction. Note that our algorithm addresses the general case where a subset of the variables are to be sampled, which are indicated in Fig. 1 by edges marked with slashes, while the rest are marginalized. Let us denote this pairwise tensor contraction as $C_Z = \texttt{con}(X \cup Y, \{A_X, B_Y\}, Z)$, where A_X and B_Y are the operand tensors that generate C_Z. This contraction can be conceptually visualized as dividing the tensor network into three parts: The first part comprises the

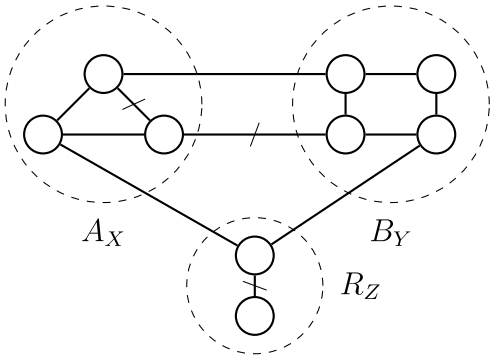


FIG. 1. A pairwise contraction $A_X, B_Y \to C_Z$ can be conceptually visualized as dividing the tensor network into three parts: the tensors generating A_X, the tensors generating B_Y, and the remaining tensors R_Z. Each part is indicated by a dashed circle. Variables to be sampled are denoted by a line with a slash, while the rest are marginalized.

tensors that generate A_X, the second those that generate B_Y, and the third consists of the remaining tensors, referred to as the environment tensors R_Z. Each part is denoted by a dashed circle in Fig. 1. The set of variables eliminated in this step is $S = X \cup Y \backslash Z$. We consider the general case where only a subset of variables, $S' \subseteq S$, is to be sampled. The marginal probability for S' can be obtained by contracting the tensor network,

$$p(S') = \texttt{con}(X \cup Y, \{A_X, B_Y, R_Z\}, S'), \tag{29}$$

where A_X and B_Y have been computed and cached during the forward pass, while R_Z, the environment tensor, is updated during the backward pass. After obtaining a sample $\boldsymbol{s}_{S'}$ for S', the environment tensor R_Z, along with the tensors in A and B, needs to be updated to reflect the new condition. The algorithm for generating an unbiased sample from the generative model is summarized as follows:

Let the tensor network under consideration be $(\Lambda, \mathcal{T}, \emptyset)$, and the set of variables to be sampled $S_{\text{tot}} \subseteq \Lambda$.

(1) Contract the tensor network to obtain the partition function $\mathcal{Z}$ and the cache of intermediate results in a binary tree $\mathcal{C}$. The children of each tensor in the binary tree are the operand tensors generating the tensor under consideration.

(2) Call the recursive sampling function in Algorithm 1 to generate a sample for the variables in S_{tot} as follows:

$$\boldsymbol{s}_{S_{\text{tot}}} \leftarrow \text{SAMPLERECURSIVE}(C_\emptyset, R_\emptyset, \mathcal{C}, S_{\text{tot}}),$$

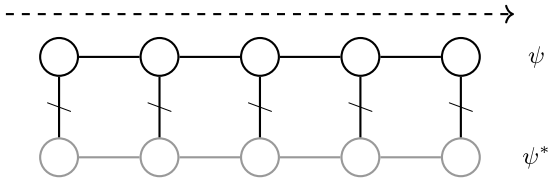
where $C_\emptyset = \mathcal{Z}$ is a zero-ranked tensor and $R_\emptyset = 1$ is the environment tensor over an empty set of variables $\emptyset$.

In Algorithm 1, the recursive function SAMPLERECURSIVE is designed to generate samples for a subtree rooted at the tensor C_Z, with the aid of an environmental tensor R_Z and a cache $\mathcal{C}$. Initially, the algorithm checks if there are child tensors for C_Z in the cache (Line 1 to 3) and determines which variables S' need to be sampled (Line 4). It then computes the marginal probability $p(S')$ for these variables by performing the tensor network contraction given in Eq. (29) (Line 5) (which accounts for variables previously sampled) and subsequently samples a set of values $\boldsymbol{s}_{S'}$ from this distribution (Line 6). The UPDATECACHE function adjusts the cache by slicing tensors according to these sampled values, which may require some recomputation to keep the cache valid (Line 7). The algorithm then recursively invokes SAMPLERECURSIVE for the children of C_Z, using updated environmental tensors (Line 8 and 9). Finally, it aggregates and returns these new samples with those from the current set S', producing a sample set for the current subtree (Line 10).

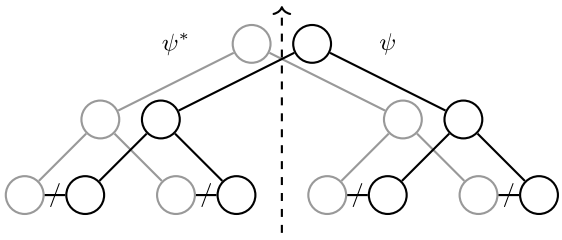
ALGORITHM 1. Recursive Sampling.

Input: Target tensor C_Z, cache $\mathcal{C}$, environment tensor R_Z, and variables S_{tot} to be sampled.
Output: A sample set for the variables in S_{tot}.
SAMPLERECURSIVE $(C_Z, R_Z, \mathcal{C}, S_{\text{tot}})$
1 **if** children $(C_Z, \mathcal{C}) = \emptyset$ **then**
2 **return** $\emptyset$;
3 $A_X, B_Y \leftarrow$ children$(C_Z, \mathcal{C})$;
4 $S' \leftarrow (X \cap Y \setminus Z) \cap S_{\text{tot}}$;
5 $p(S') \leftarrow \texttt{con}(X \cup Y, \{A_X, B_Y, R_Z\}, S')$;
6 $\boldsymbol{s}_{S'} \sim p(S')$;
7 $\mathcal{C} \leftarrow$ UPDATECACHE$(\mathcal{C}, \boldsymbol{s}_{S'})$;
8 $\boldsymbol{s}_1 \leftarrow$ SAMPLERECURSIVE$(A_X, \texttt{con}(Y \cup Z, \{B_Y, R_Z\}, X),)$ $\mathcal{C}, S_{\text{tot}})$;
9 $\boldsymbol{s}_2 \leftarrow$ SAMPLERECURSIVE$(B_Y, \texttt{con}(X \cup Z, \{A_X, R_Z\}, Y),)$ $\mathcal{C}, S_{\text{tot}})$;
10 **return** $\boldsymbol{s}_1 \cup \boldsymbol{s}_2 \cup \boldsymbol{s}_{S'}$;

This sampling algorithm is a natural generalization of those used in the quantum-inspired probabilistic models, such as the matrix product state ansatz [21] and the tree tensor network ansatz [22]. In quantum-inspired models, the involved tensors are complex-valued and probabilities are represented



(a) Probabilistic interpretation of a matrix product state (MPS) tensor network.



(b) Probabilistic interpretation of a tree tensor network (TTN).

FIG. 2. Probabilistic interpretation of popular tensor networks. Dashed arrows denote the variable elimination order. Edges with slashes correspond to the variables of interest. The set of gray tensors is the complex conjugate of the black tensors.

TABLE I. The inference tasks supported by the libraries used in the benchmark. See Sec. III for descriptions of these tasks.

	PR	MAR	MPE	MMAP
TensorInference.jl	✓	✓	✓	✓
Merlin	✓	✓	✓	✓
libDAI	✓	✓	✓	×
JunctionTrees.jl	×	✓	×	×

using Born's rule. Born's rule corresponds to contracting the complex tensor network with its conjugate, as demonstrated in Fig. 2. For example, in Fig. 2(a), the contraction order of a matrix product state ansatz is from left to right, as indicated by the dashed line. The variables are sampled in the reverse order of elimination, i.e., from right to left. In a quantum-inspired ansatz, only the "physical" variables (edges with slashes) are sampled, while the "virtual" variables (black edges) are marginalized out. The tree tensor network ansatz, shown in Fig. 2(b), is similar to the matrix product state ansatz but features a different tensor network structure.

IV. PERFORMANCE BENCHMARKS

This section presents a series of performance benchmarks comparing the runtime of our tensor-based probabilistic inference library, namely *TensorInference.jl* [31], against that of other established solvers for probabilistic inference. We have selected two open-source libraries written in C++ for this purpose, namely the *Merlin* [37] and *libDAI* [38] solvers. Their positive results in past UAI inference competitions [39,40] make them representative examples of standard practices in the field. Additionally, we have included *JunctionTrees.jl* [41], an open-source library written in Julia and the predecessor of *TensorInference.jl*. The inference tasks supported by the libraries used in the benchmark are summarized in Table I.

In these experiments, we used the UAI 2014 inference competition's benchmark suite, which comprises problem sets from various domains, including computer vision, signal processing, and medical diagnosis. These benchmark problems serve as a standardized testbed for algorithms dealing with uncertainty in AI. For the PR, MAR, and MPE tasks, we used the UAI 2014 *MAR* problem sets, as they are suitable for exact inference tasks. On the other hand, we used the UAI 2014 *MMAP* problem sets for the MMAP task, as these contain specific sets of query variables required for such task. However, since the *MMAP* problem sets were designed for approximate algorithms, we were unable to solve some of these problems using our exact inference methods. For the CPU experiments, we conducted benchmarks for all four tasks. For the GPU experiments, we focused only on benchmarking the MMAP task, since the problems of the other tasks in the UAI 2014 benchmark suite are either too large for exact inference or too small to benefit from GPU acceleration. The CPU experiments were conducted on an AMD Ryzen Threadripper PRO 3995WX 64-Cores Processor operating at 3.7 GHz and equipped with 256 GiB of RAM. The GPU experiments were conducted on an NVIDIA Quadro RTX 8000 with 48 GiB of VRAM.

It should be noted that the UAI 2014 benchmark problems, arising from diverse domains, present unique challenges because of their varied topologies and dependency structures. This structural variety has motivated our decision to adopt a generic tensor network (TN) language for their representation instead of attempting to fit each benchmark problem into standard TN frameworks such as MPS, TTN, or MERA. Reducing these problems to standard models may require a computational cost that exponentially increases with the problem size, which is not feasible for large-scale problems.

The benchmark results, conducted on a CPU, are presented in Fig. 3, where each subfigure displays the results for each of the considered inference tasks. The benchmark problems are arranged along the x axis in ascending order of the network's *space complexity*. This metric is defined as the logarithm base 2 of the number of elements in the largest tensor encountered during contraction with a given optimized contraction order. A common pattern observed among these four benchmark results is that, as the complexity of the problem increases, our TN-based implementation progressively outperforms the reference libraries. The improvement is attributed to the tensor network contraction order algorithm, which simultaneously reduces the space, time, and read-write complexities, and is further enhanced by our use of advanced BLAS routines. The graphs feature a fitted linear curve in log space to underscore this exponential improvement. However, for other less complex problems (those with space complexities smaller than 10), our library generally performs slower than the reference libraries. The reason is that the hyper-optimized contraction order-finding algorithms in our library incur a cost that becomes non-negligible for small-sized problems.

Figure 4 demonstrates the speedups achieved by executing our tensor-based method on a GPU versus on a CPU across different problem sizes for the MMAP task. The results indicate that for large problem sizes, the GPU-based implementation can improve *TensorInference.jl*'s performance by one to two orders of magnitude. However, for tasks with small problem sizes, the overhead associated with transferring data between the CPU and GPU, along with the time to launch GPU kernels, outweighs the advantages of using the GPU, resulting in decreased performance. This finding aligns with the observation that when space complexity is high, a few steps of tensor contraction operations become the most time-consuming parts, and GPUs are especially effective in accelerating these operations.

V. CONCLUSIONS

We have formulated a series of prevalent probabilistic inference tasks in terms of tensor network contractions and provided their corresponding implementations. Our proposed formulation streamlines analog formulations encountered in classical Bayesian inference methods, including the family of junction tree algorithms, by abstracting notions based on message passing and by leveraging established tools such as differential programming frameworks. We have shown how adjusting the algebraic system of a tensor network can be used to solve different probabilistic inference tasks. We introduced the unity-tensor approach to efficiently compute the marginal probabilities of multiple variables using automatic

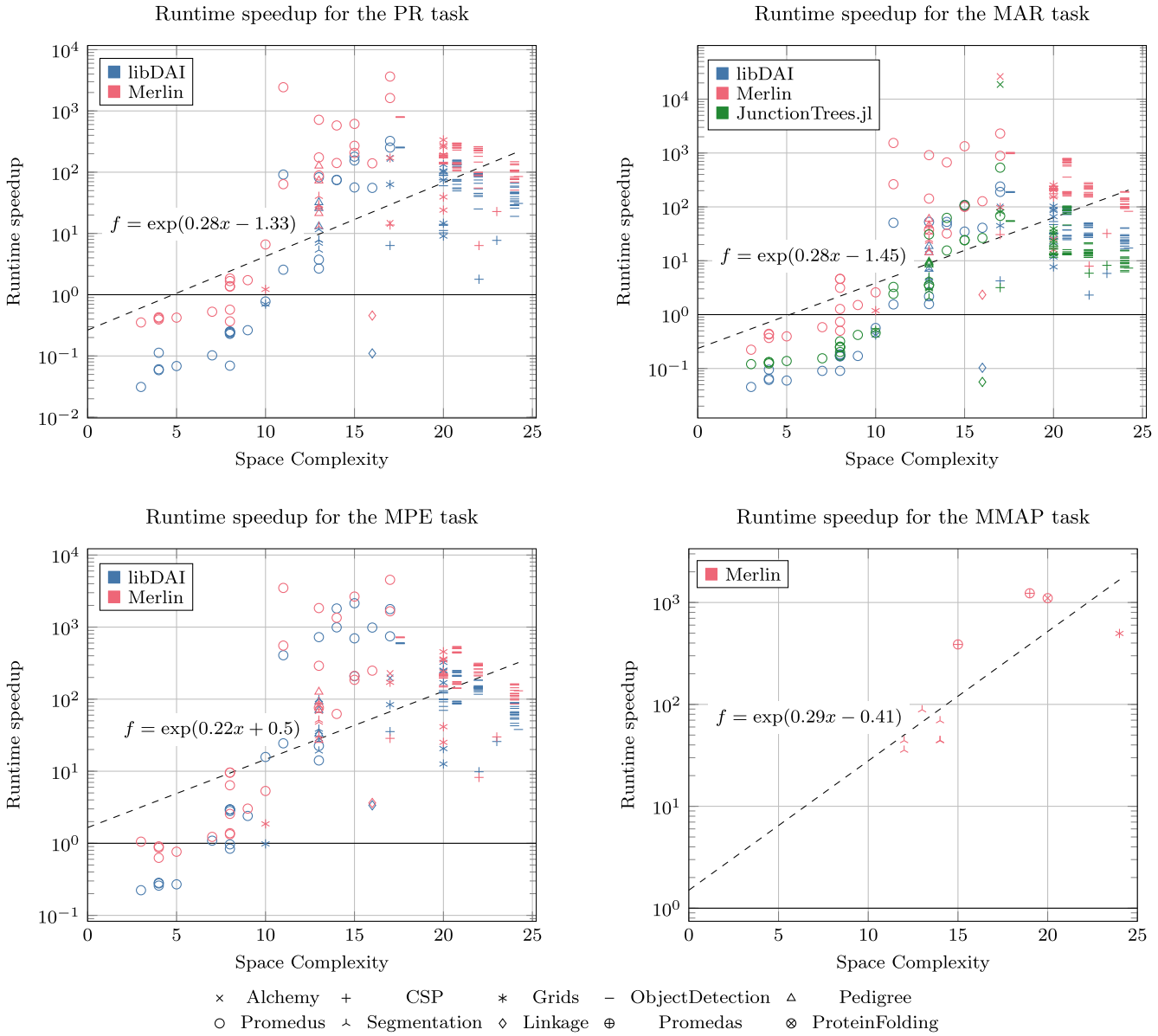


FIG. 3. Runtime speedup achieved by our tensor-based library, *TensorInference.jl*, across four different probabilistic inference tasks, relative to *Merlin* [37], *libDAI* [38] and *JunctionTrees.jl* [41]. The experiments were conducted on a CPU using the UAI 2014 inference competition benchmark problems.

differentiation. We also demonstrated how tropical tensor network representations can be employed for computing the most likely assignment of variables. Additionally, we unified the previously developed sampling algorithms for chain and tree tensor networks.

As a product of this research, we have provided an implementation of our proposed methods in the form of a Julia package, namely `TensorInference.jl` [31]. Our library integrates the latest developments in tensor network contraction order finding algorithms from quantum computing into probabilistic inference. Moreover, our tensor contraction implementation naturally compiles to BLAS functions, enabling us to fully utilize the computational power of hardware such as CPUs, GPUs, and TPUs, although the latter was not tested in this study.

We conducted a comparative evaluation against three other open-source libraries for probabilistic inference. Our method demonstrated substantial speedups in runtime performance compared to the reference libraries across various probabilistic tasks. Notably, the improvements became more pronounced as the model complexity increased. These results underscore the potential of our method in broadening the tractability spectrum of exact inference for increasingly complex models.

As a future direction, we plan to apply the ideas presented in this paper to further improve the computational efficiency of tensor-based quantum error correction (QEC) algorithms [42]. This process can be formulated as an MPE problem, where the goal is to find the most likely error pattern given the observed syndrome (or evidence).

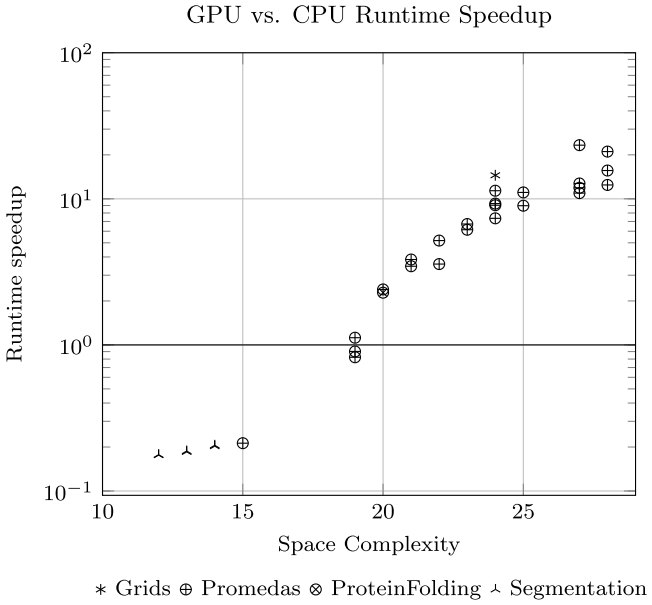


FIG. 4. *TensorInference.jl*'s runtime speedup on a GPU for the MMAP task, relative to CPU performance, benchmarked on the UAI 2014 inference competition problems.

ACKNOWLEDGMENTS

This work is partially funded by the Netherlands Organization for Scientific Research (P15-06 Project 2) and the Guangzhou Municipal Science and Technology Project (No. 2023A03J0003 and No. 2024A04J4304). The authors thank Madelyn Cain and Pan Zhang for valuable advice, and Zhong-Yi Ni for insightful discussions on quantum error correction. We acknowledge the use of AI tools like Grammarly and ChatGPT for sentence rephrasing and grammar checks.

APPENDIX: BACKWARD RULE FOR TENSOR CONTRACTION

In this Appendix, we will derive Eq. (17), which is the backward rule for a pairwise tensor contraction, denoted by $\texttt{con}(\Lambda, \{A_{V_a}, B_{V_b}\}, V_c)$. Let $\mathcal{L}$ be a loss function of interest, where its differential form is given by

$$\begin{aligned}\delta\mathcal{L} &= \texttt{con}(V_a, \{\delta A_{V_a}, \overline{A}_{V_a}\}, \emptyset) + \texttt{con}(V_b, \{\delta B_{V_b}, \overline{B}_{V_b}\}, \emptyset) \\ &= \texttt{con}(V_c, \{\delta C_{V_c}, \overline{C}_{V_c}\}, \emptyset). \end{aligned} \tag{A1}$$

The goal is to find $\overline{A}_{V_a}$ and $\overline{B}_{V_b}$ given $\overline{C}_{V_c}$. This can be achieved by using the differential form of tensor contraction, which states that

$$\delta C = \texttt{con}(\Lambda, \{\delta A_{V_a}, B_{V_b}\}, V_c) + \texttt{con}(\Lambda, \{A_{V_a}, \delta B_{V_b}\}, V_c). \tag{A2}$$

By inserting this result into Eq. (A1), we obtain

$$\begin{aligned}\delta\mathcal{L} = &\ \texttt{con}(V_a, \{\delta A_{V_a}, \overline{A}_{V_a}\}, \emptyset) \\ &+ \texttt{con}(V_b, \{\delta B_{V_b}, \overline{B}_{V_b}\}, \emptyset) \\ = &\ \texttt{con}(\Lambda, \{\delta A_{V_a}, B_{V_b}, \overline{C}_{V_c}\}, \emptyset) \\ &+ \texttt{con}(\Lambda, \{A_{V_a}, \delta B_{V_b}, \overline{C}_{V_c}\}, \emptyset). \end{aligned} \tag{A3}$$

Since δA_{V_a} and δB_{V_b} are arbitrary, the above equation immediately implies Eq. (17).

[1] S. L. Lauritzen and D. J. Spiegelhalter, Local computations with probabilities on graphical structures and their application to expert systems, J. R. Stat. Soc. Ser. B **50**, 157 (1988).

[2] F. Jensen, S. Lauritzen, and K. Olesen, Bayesian updating in causal probabilistic networks by local computations, Comput. Stat. Quarterly **4**, 269 (1990).

[3] R. D. Shachter, B. D'Ambrosio, and B. A. Del Favero, Symbolic probabilistic inference in belief networks, in *Proceedings of the Eighth National Conference on Artificial Intelligence–Volume 1*, AAAI'90 (AAAI Press, Washington, DC, 1990), pp. 126–131.

[4] Z. Li and B. D'Ambrosio, Efficient inference in Bayes networks as a combinatorial optimization problem, Int. J. Approx. Reason. **11**, 55 (1994).

[5] T. Gehr, S. Misailovic, and M. Vechev, PSI: Exact symbolic inference for probabilistic programs, in *Computer Aided Verification*, edited by S. Chaudhuri and A. Farzan (Springer International Publishing, Cham, 2016), pp. 62–83.

[6] M. Chavira and A. Darwiche, On probabilistic inference by weighted model counting, Artif. Intell. **172**, 772 (2008).

[7] S. Holtzen, G. Van den Broeck, and T. Millstein, Scaling exact inference for discrete probabilistic programs, Proc. ACM Program. Lang. **4**, 1 (2020).

[8] A. Darwiche, A differential approach to inference in Bayesian networks, J. ACM **50**, 280 (2003).

[9] A. Darwiche, An advance on variable elimination with applications to tensor-based computation, in *ECAI 2020-24th European Conference on Artificial Intelligence*, edited by G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, Frontiers in Artificial Intelligence and Applications Vol. 325 (IOS Press, Amsterdam, 2020), pp. 2559–2568.

[10] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition* (Cambridge University Press, Cambridge, 2010).

[11] R. Orús, Advances on tensor network theory: Symmetries, fermions, entanglement, and holography, Eur. Phys. J. B **87**, 280 (2014).

[12] D. Perez-Garcia, F. Verstraete, M. M. Wolf, and J. I. Cirac, Matrix product state representations, Quantum Inf. Comput. **7**, 401 (2007).

[13] Y.-Y. Shi, L.-M. Duan, and G. Vidal, Classical simulation of quantum many-body systems with a tree tensor network, Phys. Rev. A **74**, 022320 (2006).

[14] G. Vidal, Entanglement renormalization, Phys. Rev. Lett. **99**, 220405 (2007).

[15] F. Verstraete and J. I. Cirac, Renormalization algorithms for quantum-many body systems in two and higher dimensions, arXiv:cond-mat/0407066.

[16] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell *et al.*,

Quantum supremacy using a programmable superconducting processor, Nature (London) **574**, 505 (2019).
[17] I. L. Markov and Y. Shi, Simulating quantum computation by contracting tensor networks, SIAM J. Comput. **38**, 963 (2008).
[18] F. Pan and P. Zhang, Simulation of quantum circuits using the big-batch tensor network method, Phys. Rev. Lett. **128**, 030501 (2022).
[19] X. Gao, M. Kalinowski, C.-N. Chou, M. D. Lukin, B. Barak, and S. Choi, Limitations of linear cross-entropy as a measure for quantum advantage, PRX Quantum **5**, 010334 (2024).
[20] E. Stoudenmire and D. J. Schwab, Supervised learning with tensor networks, in *Advances in Neural Information Processing Systems*, edited by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Curran Associates, Red Hook, New York, 2016), Vol. 29.
[21] Z.-Y. Han, J. Wang, H. Fan, L. Wang, and P. Zhang, Unsupervised generative modeling using matrix product states, Phys. Rev. X **8**, 031012 (2018).
[22] S. Cheng, L. Wang, T. Xiang, and P. Zhang, Tree tensor networks for generative modeling, Phys. Rev. B **99**, 155131 (2019).
[23] E. Robeva and A. Seigal, Duality of graphical models and tensor networks, Inf. Inference: A J. IMA **8**, 273 (2019).
[24] A. A. Klishin and G. van Anders, When does entropy promote local organization? Soft Matter **16**, 6523 (2020).
[25] G. Kalachev, P. Panteleev, and M.-H. Yung, Multi-tensor contraction for XEB verification of quantum circuits, arXiv:2108.05665.
[26] J. Gray and S. Kourtis, Hyper-optimized tensor network contraction, Quantum **5**, 410 (2021).
[27] J. G. Liu, L. Wang, and P. Zhang, Tropical tensor network for ground states of spin glasses, Phys. Rev. Lett. **126**, 090506 (2021).
[28] J. G. Liu, X. Gao, M. Cain, M. D. Lukin, and S. T. Wang, Computing solution space properties of combinatorial optimization problems via generic tensor networks, SIAM J. Sci. Comput. **45**, A1239 (2023).
[29] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, An updated set of basic linear algebra subprograms (BLAS), ACM Trans. Math. Software **28**, 135 (2002).
[30] J. G. Liu and C. Elrod, TropicalGEMM.jl: A Julia package for high-performance tropical matrix multiplication, https://github.com/TensorBFS/TropicalGEMM.jl (2023).
[31] M. Roa-Villescas and J. G. Liu, TensorInference: A Julia package for tensor-based probabilistic inference, J. Open Source Software **8**, 5700 (2023).
[32] J. I. Cirac, D. Pé rez-García, N. Schuch, and F. Verstraete, Matrix product states and projected entangled pair states: Concepts, symmetries, theorems, Rev. Mod. Phys. **93**, 045003 (2021).
[33] R. Orús, A practical introduction to tensor networks: Matrix product states and projected entangled pair states, Ann. Phys. (NY) **349**, 117 (2014).
[34] M. Roa-Villescas, J. G. Liu, P. W. Wijnings, S. Stuijk, and H. Corporaal, Scaling probabilistic inference through message contraction optimization, in *Proceedings of the 2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)* (IEEE, Piscataway, NJ, 2023), pp. 123–130.
[35] R. Dechter, UAI 2022 probabilistic inference competition (2022), accessed: 2024-02-11.
[36] H. J. Liao, J. G. Liu, L. Wang, and T. Xiang, Differentiable programming tensor networks, Phys. Rev. X **9**, 031041 (2019).
[37] R. Marinescu, Merlin: An extensible C++ library for probabilistic inference in graphical models, https://github.com/radum2275/merlin (2022).
[38] J. M. Mooij, libDAI: A free and open source C++ library for discrete approximate inference in graphical models, J. Mach. Learn. Res. **11**, 2169 (2010).
[39] Summary of the 2010 UAI approximate inference challenge (2010), accessed: 2021-08-21.
[40] V. Gogate, UAI 2014 Probabilistic inference competition (2014), accessed: 2021-08-21.
[41] M. Roa-Villescas, P. W. Wijnings, S. Stuijk, and H. Corporaal, Partial evaluation in junction trees, in *Proceedings of the 2022 25th Euromicro Conference on Digital System Design (DSD)* (IEEE, Piscataway, NJ, 2022), pp. 429–437.
[42] A. J. Ferris and D. Poulin, Tensor networks and quantum error correction, Phys. Rev. Lett. **113**, 030501 (2014).