Approximate contraction of arbitrary tensor networks with a flexible and efficient density matrix algorithm

Linjian Ma[1], Matthew T. Fishman[2], E. M. Stoudenmire[2], and Edgar Solomonik[1]

[1]Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL 61801, USA
[2]Center for Computational Quantum Physics, Flatiron Institute, New York, New York 10010, USA

Tensor network contractions are widely used in statistical physics, quantum computing, and computer science. We introduce a method to efficiently approximate tensor network contractions using low-rank approximations, where each intermediate tensor generated during the contractions is approximated as a low-rank binary tree tensor network. The proposed algorithm has the flexibility to incorporate a large portion of the environment when performing low-rank approximations, which can lead to high accuracy for a given rank. Here, the environment refers to the remaining set of tensors in the network, and low-rank approximations with larger environments can generally provide higher accuracy. For contracting tensor networks defined on lattices, the proposed algorithm can be viewed as a generalization of the standard boundary-based algorithms. In addition, the algorithm includes a cost-efficient density matrix algorithm for approximating a tensor network with a general graph structure into a tree structure, whose computational cost is asymptotically upper-bounded by that of the standard algorithm that uses canonicalization. Experimental results indicate that the proposed technique outperforms previously proposed approximate tensor network contraction algorithms for multiple problems in terms of both accuracy and efficiency.

1 Introduction

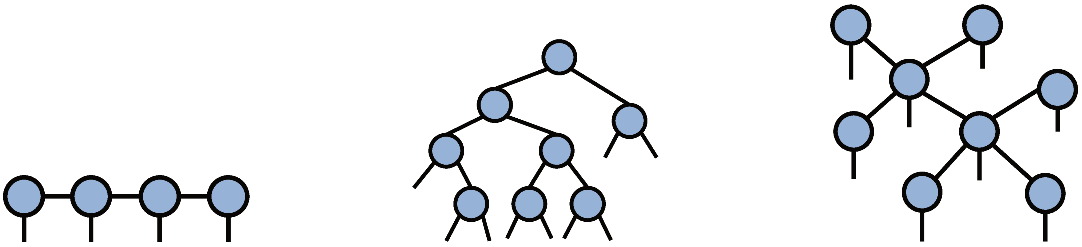
A tensor network [58, 80] uses a set of (small) tensors, where some or all of their modes are contracted according to some pattern, to implicitly represent the structure of high-dimensional tensors that are expensive to form explicitly. Tensor network techniques have been widely used in computational quantum physics [80, 78, 81, 79, 70, 68], where low-rank tensor networks can be used to both represent Hamiltonians and quantum states. These techniques are also applied in multiple other applications, including quantum circuit simulation [61, 29, 62, 49, 83], data mining via tensor methods [37, 15], machine learning [73, 65, 42], and so on.

The *tensor network contraction* operation explicitly evaluates the single tensor represented by a given tensor network, and it has multiple applications. In quantum computing, each quantum circuit execution can be viewed as a tensor network contraction, making this method a useful tool for simulating quantum computers [49, 83, 62, 61]. In statistical physics, tensor network contraction has been used to evaluate the classical partition function of physical models defined on specific graphs [40]. Tensor network contraction has also been used for counting satisfying assignments of constraint satisfaction problems (#CSPs) [38]. In this approach, an arbitrary #CSP formula is transformed into a tensor network, where its full contraction yields the number of satisfying

Linjian Ma: lma16@illinois.edu
Matthew T. Fishman: mfishman@flatironinstitute.org
E. M. Stoudenmire: mstoudenmire@flatironinstitute.org
Edgar Solomonik: solomon2@illinois.edu

assignments of that formula. Tensor network contraction is typically achieved through a sequence of pairwise tensor contractions. This sequence, known as the *contraction path*, is determined by a topological sort of the underlying *contraction tree*. The contraction tree is a rooted binary tree that depicts the complete contraction of the tensor network. In this tree, the leaves correspond to the tensors in the network, and each internal vertex represents the tensor contraction of its two children.

In the general case, contracting tensor networks with arbitrary structure is #P-hard because of the potential production of intermediate tensors with high orders or large modes, leading to significant computational costs for accurate contraction [16, 56, 8]. Nonetheless, in some applications such as many-body physics, it has been observed that tensor networks built on top of specific models can often be approximately contracted with satisfactory accuracy, without incurring large computational costs [57].


(a) MPS

(b) Binary tree tensor network

(c) TTNS

Figure 1: Illustration of the matrix product state (MPS), the (full) binary tree tensor network, and the tree tensor network state (TTNS). MPS is a maximally-unbalanced binary tree tensor network if contracting the tensor at one end with its neighbor. Both MPS and the binary tree tensor network are special cases of TTNS, where each tensor has an order of at most 3.

A common approach to approximately contract a tensor network is to approximate large intermediate tensors as (low-rank) tensor networks, which reduces the memory usage and computational overhead for subsequent contractions. Widely used tensor networks for approximation including the matrix product state (MPS [78], also called tensor train [59]), the binary tree tensor network [70], and the tree tensor network state (TTNS) [55, 54, 22], which are visualized in Fig. 1. For tensor network contractions defined on regular structures, such as projected entangled pair states (PEPS) with 2D lattice structures [79, 78], many efficient approximate contraction algorithms based on MPS approximations [46, 45] have been proposed. However, many of these methods have not been extended to other general tensor network structures.

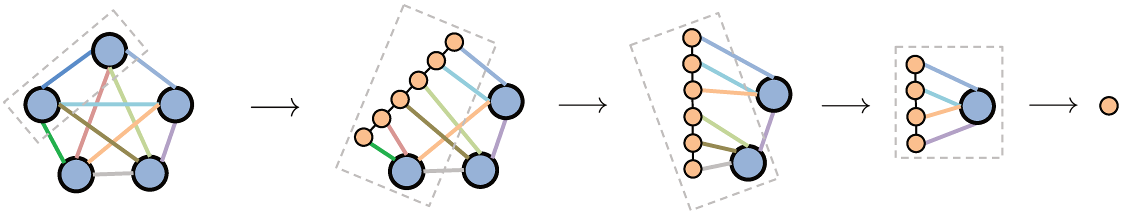


Figure 2: Illustration of the approximate contraction technique used in [35, 61, 14]. Each intermediate is approximated as an MPS, which has an unbalanced binary tree structure. The left diagram is the tensor diagram of the input tensor network. Each dashed box denotes the part of the tensor network that is approximated as an MPS.

Recent works have proposed automated approximation algorithms for contracting tensor networks with more general graph structures [35, 61, 14, 26, 66, 2], and many of these methods employ low-rank approximation/truncation techniques. In [35, 61, 14], each intermediate tensor produced during the contraction is approximated as a binary tree tensor network, and we illustrate this approach in Fig. 2. In particular, [35] approximates each intermediate tensor as a general binary tree tensor network, while the algorithm proposed in [61] called "contracting arbitrary tensor network" (CATN) approximates each intermediate tensor as an MPS. When contracting two MPSs, CATN swaps/permutes the modes that connect both MPSs to the boundaries. Then, it contracts these modes to obtain the output MPS. The adjacent mode swaps are the bottleneck for complexity in CATN. In another algorithm proposed in [14] called "SweepContractor", each intermediate tensor is also approximated as an MPS, and the algorithm leverages an embedding of the tensor network graph into 2D space to find an effective contraction path.

Several factors can significantly impact the efficiency and accuracy of the approximate tensor network contraction process. To begin with, the choice of contraction path plays a crucial role. Ref. [26] demonstrates that selecting different contraction paths using various heuristics can lead to substantial variations in both runtime and accuracy for different problems. Additionally, for both CATN [61] and SweepContractor [14], it is essential to carefully select the binary tree/MPS structures and permutations (i.e., a mapping from tensor modes onto binary tree vertices) [41]. These choices should yield accurate low-rank approximations while enabling efficient subsequent contractions. However, previous works such as [35, 61, 14] have not systematically explored these parts of the design space.

The low-rank truncation algorithm used to reduce the tensor size in approximate contraction is another important factor. Let $\mathbf{M}$ represent the part of the network that requires approximation, and let $\mathbf{E}$ denote the remaining set of tensors in the network, which is commonly referred to as the *environment*. The optimal way to truncate is to minimize the global error by solving $\min_{\mathbf{X}} \|\mathbf{EX} - \mathbf{EM}\|_F$ with the constraint that $\mathbf{X}$ has a specific low-rank tensor network structure, where $\|\cdot\|_F$ denotes the Frobenius norm. Two standard algorithms for solving the low-rank approximation problem are the canonicalization-based algorithm and the density matrix algorithm. In the canonicalization-based algorithm, one first performs a QR decomposition on $\mathbf{E}$, $\mathbf{Q}, \mathbf{R} \leftarrow \texttt{QR}(\mathbf{E})$, then updates $\mathbf{X}$ based on the low-rank approximation of $\mathbf{Q}^T\mathbf{M}$. In the density matrix algorithm, the leading eigenvectors of the density matrix (also called the Gram matrix/normal equations), $\mathbf{M}^T\mathbf{E}^T\mathbf{EM}$, is computed, and $\mathbf{X}$ is computed by projecting $\mathbf{M}$ to the subspace spanned by the leading eigenvectors. Both algorithms have the same output but can have different computational costs.

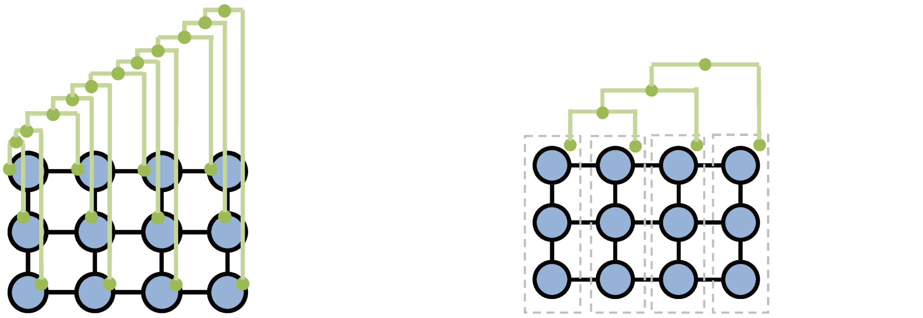
If the environment tensor network $\mathbf{E}$ contains a large number of tensors, minimizing the global error could be computationally expensive. In such cases, one typically resorts to minimizing the local error by solving $\min_{\mathbf{X}} \|\mathbf{X} - \mathbf{M}\|_F$, or by replacing $\mathbf{E}$ with a smaller environment $\hat{\mathbf{E}}$ so the optimization problem is easier to solve.

Achieving a balance between accuracy and efficiency requires favoring different structures and sizes of the environment $\hat{\mathbf{E}}$ for different problems. Hence, it becomes crucial to provide an automated tensor network contraction algorithm with the necessary flexibility to accommodate different environments. This flexibility enables the algorithm to adapt and optimize the contraction process according to the specific requirements of each problem.

In previous studies [61, 14], the selection of environments was implicitly determined by the algorithm. For instance, in the CATN algorithm [61], truncation takes place during adjacent swaps of MPS modes, with the environment consisting of all tensors in the target MPS. Similarly, the SweepContractor algorithm [14] performs truncation while contracting an input MPS with a single tensor, incorporating both the MPS and the tensor into the environment. The method proposed in [26] introduces user-specified environment sizes, and utilizes tree-structured environments $\hat{\mathbf{E}}$ that are constructed by including a spanning tree of tensors around the pair of tensors to be truncated. Ref. [26] demonstrates that including a larger environment leads to more accurate contraction results for multiple problems. In this work, we generalize the strategies presented in the previous works and propose a tensor network contraction algorithm that allows more flexible environment incorporation. For contracting tensor networks defined on lattices, the proposed strategy can be viewed as a generalization of the standard boundary-based algorithms [79].

1.1 Our contributions

We propose a new approach, `partitoned_contract`, for performing approximate contractions of arbitrary tensor networks. We illustrate the approach in Fig. 4. This approach follows the technique used in [35, 61, 14], where each intermediate tensor produced during the contraction is approximated as a binary tree tensor network. Moreover, our approach is composed of the following two novel components.



(a) Complete contraction tree

(b) Contraction tree on the partitioned network

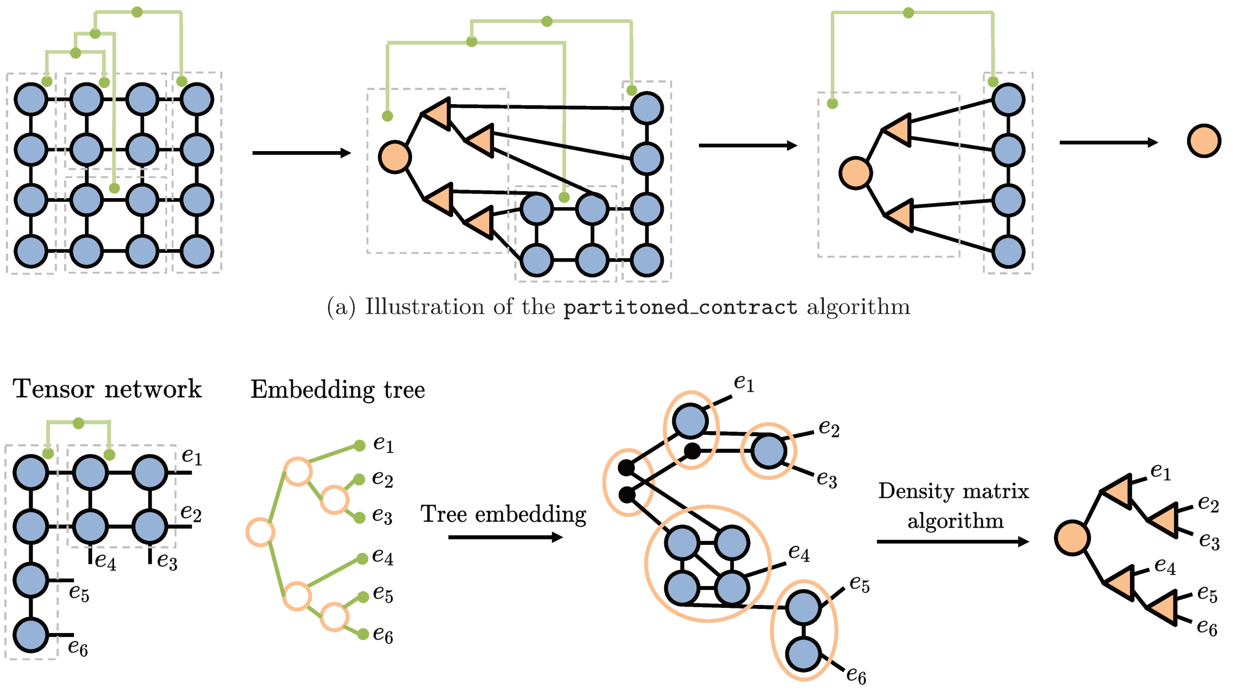
Figure 3: Illustration of different contraction trees. Each blue vertex denotes a tensor, and the green lines and dots denote the binary contraction tree. The contraction tree visualization has been adapted from [26]. In (b), each dotted box denotes a partition of the tensor network. The partial contraction sequence shown in (b) corresponds to a standard left-to-right boundary MPS contraction [79].

First, unlike prior works [35, 61, 14, 26] that contract the tensor network based on a *complete* contraction tree with each leaf corresponding to a tensor in the network, our technique relies on a contraction tree of parts of the tensor network, which is a *partial* contraction tree and each leaf vertex corresponds to a partition. We illustrate complete and partial contraction trees in Fig. 3. In the algorithm, each low-rank approximation considers all tensors in the input partitions as the environment, thus utilizing a larger partition means using a larger environment and can potentially lower the truncation error. In practical applications, one has the option of either utilizing automated graph partitioning libraries like KaHyPar [67] and Metis [36] for partitioning the tensor network, or manually selecting suitable partitions for specific problems. In Section 3.2.2, we will demonstrate how the utilization of the partial contraction tree abstraction enables the straightforward extension of various contraction algorithms designed for 2D grids with different environments, including those that have not been automated in the prior work [35, 61, 14, 26].

Second, we provide a new approach to approximate a given tensor network into a binary tree structure, as depicted in Fig. 4b. This approach is composed of the following three novel components.

- It encompasses a new heuristic for generating binary tree structures and permutations (i.e., a mapping from tensor modes onto binary tree vertices [41]) of intermediate tensor networks. The binary tree structure is also called the embedding tree in Fig. 4b and throughout the paper. Unlike previous studies that relied on arbitrary choices for such structures and permutations, our approach takes into consideration the efficiency of subsequent contractions. This is achieved by ensuring that the embedding tree aligns with a contraction path-generated tree, which imposes constraints on the adjacency relations of binary tree modes. Moreover, we ensure that the selected structure is similar to the given sub-tensor network by solving a graph embedding problem that minimizes the congestion [33, 9, 52, 7, 51], allowing for an accurate approximation with low ranks in the resulting tree tensor network. The details of the algorithm can be found in Section 5.

- It includes a density matrix algorithm to approximate a given tensor network into the target embedding tree. The algorithm uses a sequence of density matrix algorithms for low-rank approximation to output the embedding tree tensor network, and includes all tensors in the



(a) Illustration of the `partitoned_contract` algorithm

(b) Illustration of the process to approximate the contraction of two partitions into a binary tree tensor network

Figure 4: (a) Illustration of the `partitoned_contract` algorithm. The algorithm takes as inputs a tensor network, a partitioning of that tensor network, and a partial contraction tree. The algorithm proceeds by traversing the partial contraction tree and approximately contracting a pair of tensor network partitions into a binary tree tensor network. (b) Illustration of the process to approximate the input tensor network (left diagram) into a binary tree tensor network (right diagram). The embedding tree is a rooted binary tree that represents the output tree structure. The tree embedding step maps a partition of the input tensor network to each non-leaf (orange) vertex in the embedding tree. Finally, the density matrix algorithm (or the canonicalization-base algorithm) approximates the embedded tensor network into a binary tree tensor network. Each black dot in the diagrams represents an identity matrix.

input tensor network as the environment. When compared to the canonicalization-based algorithm that employs the same environment, the density matrix algorithm exhibits the same or lower asymptotic cost, making it more efficient. In particular, the density matrix algorithm exhibits the potential to significantly reduce the asymptotic cost when dealing with large environment sizes. The detail of the algorithm can be found in Section 6.

The tensor network contraction framework proposed in [26] also offers the capability to handle large environments. However, the framework in [26] approximates the environments as trees from the outset by cutting certain bonds in the environment, which ignores certain loop correlations in the environment. In contrast, our density matrix algorithm directly works with the full environment including loops and then approximates the result of the contraction as a tree, which for a given environment should be more accurate but potentially more computationally expensive.

- In scenarios where the mode ordering of the selected tree structure intended for efficient later contractions does not align with the input structure, our approach employs a hybrid algorithm that integrates the density matrix algorithm and a swap-based algorithm to perform the tree approximation. Swap-based algorithms, extensively utilized in MPS-based tensor network contraction algorithms such as when applying long-range gates [72] and in other general approximate contraction algorithms like CATN and SweepContractor, use a sequence of adjacent swaps of MPS modes to permute the ordering of the MPS tensors. Within our algorithm, a sequence of local swap operations are performed using the density matrix algorithm, each time progressively modifying the structure by a small amount to ensure that the overall cost remains manageable. The detail of the algorithm can be found in Section 7.

In Section 8, we assess the performance of the proposed algorithm. Regarding the sub-problem of approximating a general tensor network into a tree tensor network, our experimental results show the superior efficiency of the density matrix algorithm compared to the canonicalization-based algorithm when applied to multiple input tensor network structures. These empirical findings consistently align with our theoretical analysis.

To evaluate the efficacy of our contraction algorithm, we conduct experiments on various tensor network structures. The results demonstrate that by leveraging environments and employing the density matrix algorithm, we achieve significant reductions in overall execution time and improvements in accuracy when dealing with tensor networks defined on lattices and random regular graphs. Notably, our algorithm outperforms both the CATN algorithm proposed in [61] and the SweepContractor proposed in [14] when considering tensor networks defined on lattices representing the classical Ising model. Specifically, our approach achieves an order of magnitude speed-up in execution time while maintaining the same level of accuracy. This improvement in speed demonstrates the efficiency of our approach.

1.2 Organization

This paper is organized as follows. In Sections 2 and 3, we introduce the definitions, the computational cost model, and background for the proposed algorithm. Section 4 provides an overview of the proposed approximate tensor network contraction algorithm, with detailed components discussed in Sections 5 to 7. In Section 8, we present the results of a series of experiments to evaluate the performance of the proposed algorithm.

2 Definitions and the computational cost model

2.1 Tensor network definitions

We introduce the tensor network notation here. The structure of a tensor network can be described by an undirected graph $G = (V, E)$, where each tensor of the tensor network is associated with a vertex in V and each mode of the tensors is associated with an edge in E. We refer to edges with a dangling end (one end not adjacent to any vertex) as uncontracted edges, and those without dangling ends as contracted edges. We use w to denote an edge weight function such that for each edge $e \in E$, $w(e) = \log(s)$ is the natural logarithm of the mode size s associated with an edge e. For an edge set E, we use $w(E) = \sum_{e\in E} w(e)$ to denote the weighted sum of the edge set. The weight $w(E)$ is related to the cost of contracting neighboring tensors along the modes associated with the edge set E, which will be discussed in more detail in the next section.

2.2 The computational cost model

We summarize the computational cost model used throughout the paper. We assume that all tensors in the tensor network are dense. The contraction of two general dense tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, represented as vertices v_a and v_b in $G = (V, E)$, can be cast as a matrix multiplication, and the overall asymptotic cost is

$$\Theta\left(\exp\left(w(E(v_a)) + w(E(v_b)) - w(E(v_a, v_b))\right)\right), \tag{1}$$

where $E(v_a), E(v_b)$ denotes the edges adjacent to v_a, v_b, respectively, and $E(v_a, v_b)$ denotes the edge connecting v_a and v_b. Above we assume the classical matrix multiplication algorithm is used rather than fast algorithms such as Strassen's algorithm [74].

To canonicalize the tree tensor network, a series of QR factorizations is employed. Given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, performing the QR factorization incurs an asymptotic cost of $\Theta(mn{\cdot}\min(m, n))$. In order to reduce the bond size or rank within the tensor network, we utilize low-rank factorization.

Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, low-rank factorization aims to find two matrices, $\mathbf{B} \in \mathbb{R}^{m \times r}$ and $\mathbf{C} \in \mathbb{R}^{r \times n}$, with r being less than the minimum of m and n, while minimizing the Frobenius norm $\|\mathbf{A} - \mathbf{BC}\|_F$. In our cost analysis, we assume the use of the standard low-rank factorization algorithm that employs a rank-revealing QR factorization [28]. The asymptotic cost of this algorithm is $O(mnr)$.

3 Background

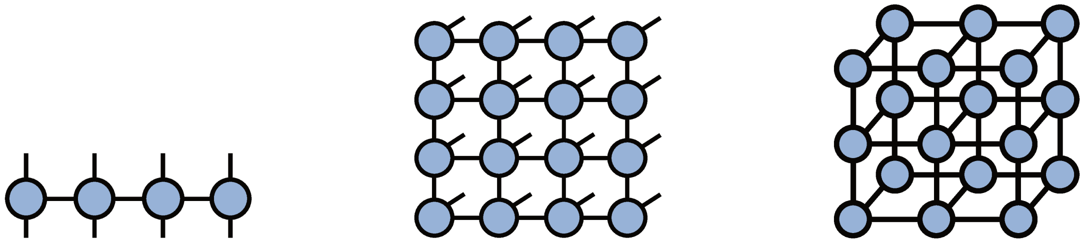
This section offers background for the proposed approach. In Section 3.1, we provide a short survey of several common tensor networks discussed in the paper. In Section 3.2, we review both the canonicalization-based algorithm and the density matrix algorithm for low-rank approximation of tensor networks. This review serves as motivation for the density matrix algorithm explained in detail in Section 6.

We cover additional backgrounds in the appendix. Appendix B.1 covers the standard swap-based algorithm used to permute MPS modes, which serves as a motivation for our algorithm that combines the density matrix algorithm and the swap-based algorithm, as outlined in Section 7. Furthermore, in Appendix B.2, we delve into the definition and heuristics of the graph embedding problem, which is utilized in Section 5 to select an efficient binary tree structure.

3.1 A survey of common tensor network structures

We survey both tree tensor networks and tensor networks defined on lattices. The matrix product state (MPS) [78, 59], a binary tree tensor network [70], and a general tree tensor network state (TTNS) [55, 54, 22] are illustrated in Fig. 1. An MPS is a tensor network with a linear structure, with each tensor having one uncontracted mode. The binary tree tensor network has a rooted binary tree structure, and all non-root vertices have an order of three. In a general TTNS, each tensor can have uncontracted modes, and the network has a general tree structure.

In this work, we focus on discussing both MPS and the binary tree tensor network. These networks are considered as special cases of TTNS, where each tensor has a maximum order of three. This characteristic makes them more memory-efficient compared to more general TTNS, especially when considering a fixed rank r. When the uncontracted mode size s is much smaller than r, each MPS tensor has a size of $O(sr^2)$. This memory requirement is more efficient than that of the general binary tree tensor network, whose tensor size is $O(r^3)$.



(a) MPO

(b) PEPS

(c) 3D lattice tensor network

Figure 5: Illustration of the matrix product operator (MPO), the projected entangled pair states (PEPS), and the $3 \times 3 \times 2$ 3D lattice tensor network.

Fig. 5 and Fig. 3 provide visual representations of other tensor networks, including the matrix product operator (MPO), the projected entangled pair states (PEPS) [79, 78], and a closed tensor network defined on a 3D lattice. In the 2D lattice tensor network, each row is either an MPS or an MPO. In the 3D lattice, each slice is either a PEPS or a PEPO. The PEPO has a similar structure to PEPS, but with the distinction that each tensor has two uncontracted edges.

3.2 The canonicalization-based algorithm and the density matrix algorithm

Let $\mathbf{A} \in \mathbb{R}^{b\times R}, \mathbf{B} \in \mathbb{R}^{R\times c}$ denote two tensors in a tensor network, and let $\mathbf{E} \in \mathbb{R}^{a\times b}$ denote the environment tensor network. The low-rank approximation problem that is widely used in this work can be stated as

$$\min_{\hat{\mathbf{A}}\in\mathbb{R}^{b\times r},\mathbf{V}\in\mathbb{R}^{c\times r}} \left\|\mathbf{EAB} - \mathbf{E}\hat{\mathbf{A}}\mathbf{V}^T\right\|_F, \quad \text{s.t. } \mathbf{V}^T\mathbf{V} = \mathbf{I}, \tag{2}$$

where $r < R$. For the canonicalization-based algorithm, one first performs a QR decomposition on $\mathbf{EA}$ and gets $\mathbf{Q} \in \mathbb{R}^{a\times R}, \mathbf{R} \in \mathbb{R}^{R\times R}$, and then computes the right r leading singular vectors of $\mathbf{RB}$ to obtain $\mathbf{V}$. For the density matrix algorithm, one first computes the Gram matrix (normal equations) $\mathbf{L} = (\mathbf{EAB})^T\mathbf{EAB}$, commonly known as the density matrix in the physics literature (and is at the heart of the original formulation of the density matrix renormalization group (DMRG) algorithm [81]), and then computes the right r leading singular vectors/eigenvectors of $\mathbf{L}$ to obtain $\mathbf{V}$.

For the case where $\mathbf{E}$ is a single matrix, both algorithms yield the same asymptotic cost with the computational cost introduced in Section 2.2. However, when $\mathbf{E}$ takes the form of a tensor network containing a large number of tensors, the density matrix algorithm is more advantageous in terms of simplicity and efficiency. In particular, the density matrix $\mathbf{L} = (\mathbf{EAB})^T\mathbf{EAB}$ can be easily computed using the existing exact tensor network contraction algorithms, while orthogonalizing $\mathbf{EA}$ is usually hard when $\mathbf{E}$ does not have a tree structure. One potential approach for orthogonalizing $\mathbf{EA}$ involves directly performing orthogonalization on the matrix resulting from the contraction of $\mathbf{EA}$, but this method is inefficient when $\mathbf{E}$ is not path-like.

In Section 3.2.1, we review the canonicalization-based algorithm to reduce the mode sizes of tree tensor networks. We will show in Section 6.4 that the cost of the density matrix algorithm is upper-bounded by the canonicalization-based algorithm. In Section 3.2.2, we provide a review of existing algorithms employed in truncating the MPO-MPS contraction, a common tensor network contraction and a special case of our more general algorithm.

3.2.1 The canonicalization-based algorithm for truncating tree tensor networks

We review the canonicalization-based algorithm to truncate a tree tensor network [82]. We first introduce the canonical form in Definition 1. For a given matrix $\mathbf{M}$ that is implicitly represented by a tree tensor network, its canonical form makes the whole tree orthogonal and uses another matrix to store the non-orthogonal part.

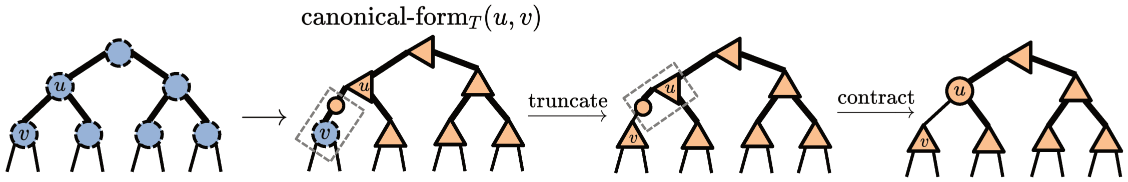


Figure 6: Illustration of truncating the mode represented by the edge (u, v) through canonicalization.

Definition 1 (Canonical form)**.** Consider a tensor network with a tree structure $T = (V, E)$. For a given vertex $u \in V$ and an edge (u, v), let $S \subseteq V$ denote the vertices connected to u when the edge (u, v) is removed from T. $\texttt{canonical\_form}_T(u, v)$ means that all tensors represented by vertices in S are orthogonalized towards the edge (u, v), and a new vertex is added between u and v whose tensor contains the non-orthogonal part. An illustration of $\texttt{canonical\_form}_T(u, v)$ is in Fig. 6.

The canonicalization-based algorithm is shown in Algorithm 1. It proceeds by computing the truncated network through a post-order depth-first search (DFS) traversal of the tree structure. At each vertex v, the algorithm constructs the canonical form around v while truncating the edge connected to v. The resulting orthogonal tensor $\mathbf{U}_v$ is then computed. This iterative process continues until only the root vertex remains, which contains the comprehensive non-orthogonal information of the entire network.

Algorithm 1 The canonicalization-based algorithm for truncating a tree tensor network

```
Input: The tree tensor network $T = (V, E)$, the maximum mode size $\chi$, and the root vertex $r$
$T_r \leftarrow$ a directed tree of $T$ with a root vertex $r$
for each $v \in V \setminus \{r\}$ based on a post-order DFS traversal of $T_r$ do
    $u \leftarrow \texttt{parent}(T_r, v)$
    Change the tree tensor network to $\texttt{canonical\_form}_T(u, v)$ with the non-orthogonal matrix $\mathbf{R}_u$
    $\mathbf{M}_v \leftarrow$ matricization of the tensor at $v$ with the mode connecting $u, v$ combined into the column
    $\mathbf{U}_v\hat{\mathbf{R}}_u \leftarrow$ rank-$\chi$ approximation of $\mathbf{M}_v\mathbf{R}_u$ with $\mathbf{U}_v$ being orthogonal
    Update the tensor at $u$ as $\hat{\mathbf{R}}_u\mathbf{M}_u$
end for
return the tree tensor network that contains all $\mathbf{U}_v$ and the root tensor $\mathbf{M}_r$
```

3.2.2 Existing algorithms for truncating MPO-MPS multiplication

We provide a review of a set of algorithms to truncate the output of MPO-MPS multiplication. These algorithms are widely used in the boundary-based algorithm to approximately contract 2D lattice tensor networks like those surveyed in Section 3.1. The boundary-based contraction algorithm initiates the process with a boundary MPS of the 2D network (e.g., the leftmost MPS in Fig. 3b). At each step, the adjacent MPO is applied to the MPS and the result is approximated as a low-rank MPS. The boundary-based contraction algorithm serves as the basis for motivating the proposed partial contraction tree abstraction and the generalized density matrix algorithm for contracting arbitrary tensor networks.

Previous studies [60, 53] have explored various algorithms for MPO-MPS multiplication. These algorithms include approaches based on canonicalization [72, 60], the density matrix algorithm [53, 24], and the iterative fitting algorithm [79, 72]. In this work, we specifically concentrate on the first two types of algorithms. This choice is driven by the fact that both are one-pass algorithms and theoretical error bounds can be derived for the resulting output of both algorithms. The iterative fitting algorithm could have better scaling and lead to better performance in some cases but the use of that approach within our new algorithm is left for future work.

Algorithms that use canonicalization We review two different canonicalization-based algorithms: the zip-up algorithm [72] and the canonicalization algorithm with full environment [60]. The zip-up algorithm uses a smaller environment compared to the other algorithm, which consider all tensors in the input MPO and MPS when performing truncations. Throughout the analysis we use r to denote the MPS rank, use a to denote the MPO rank, and use s to denote the size of all the other modes. The computational cost comparison between the algorithms is summarized in Table 1.

Algorithm	Asymptotic cost	$s \ll a = \Theta(r)$	$s = \Theta(a) \ll r$
Zip-up	$\Theta(N(s^2a^2r^2 + sar^3))$	$\Theta(N(s^2r^4))$	$\Theta(N(s^2r^3))$
Canonicalization w/ full env	$\Theta(N(s^2a^2r^2 + sa^3r^3))$	$\Theta(N(sr^6))$	$\Theta(N(s^4r^3))$
Density matrix	$\Theta(N(sa^2r^3 + s^2a^3r^2 + s^2ar^3))$	$\Theta(N(s^2r^5))$	$\Theta(N(s^3r^3))$

Table 1: Comparison of asymptotic algorithmic complexity between the zip-up algorithm, the canonicalization-based algorithm that uses the full environment, and the density matrix algorithm. $s = \Theta(a)$ means s is asymptotically bounded by a both above and below.

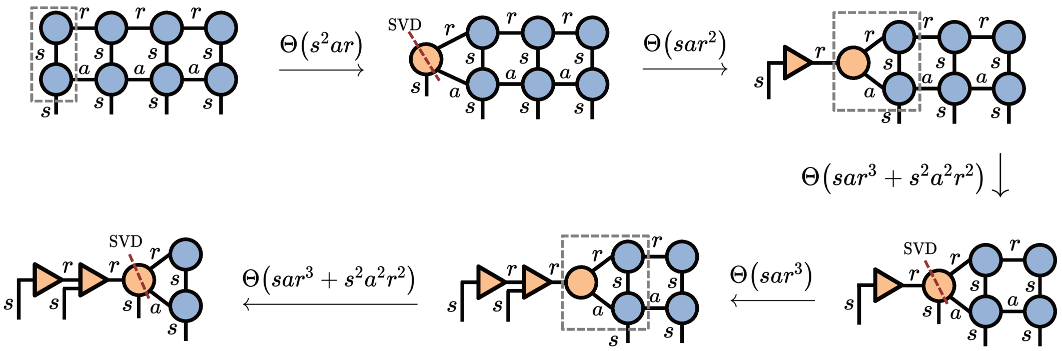


Figure 7: Illustration of the zip-up algorithm. Each dashed block includes the tensors to be contracted at a given step. Each tensor represented by a triangular vertex denotes a tensor with an orthogonality property.

The zip-up algorithm [72, 60] is illustrated in Fig. 7. We also let the output truncated MPS have rank r. The algorithm begins by contracting the leftmost pair of tensors. A truncated singular value decomposition (SVD) is then performed to obtain the left leading singular vectors $\mathbf{U}_1$ and the remaining non-orthogonal component $\mathbf{V}_1$. Next, $\mathbf{V}_1$ is combined with the second leftmost pair of tensors, and another truncated SVD is performed. This process continues until it reaches the right boundary of both the MPO and MPS. When the resulting MPS has an order of N, the algorithm's asymptotic computational cost is $\Theta(N(s^2a^2r^2 + sar^3))$. It should be noted, as depicted in Fig. 7, that the truncation at the ith step employs an environment including all i left MPO and MPS tensors, but not the full environment (all tensors in the MPS and MPO).

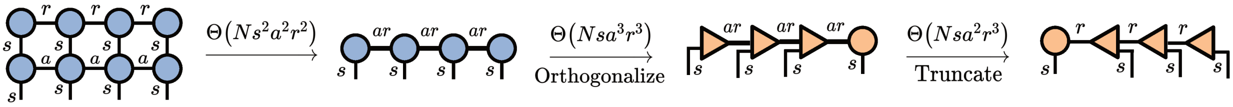


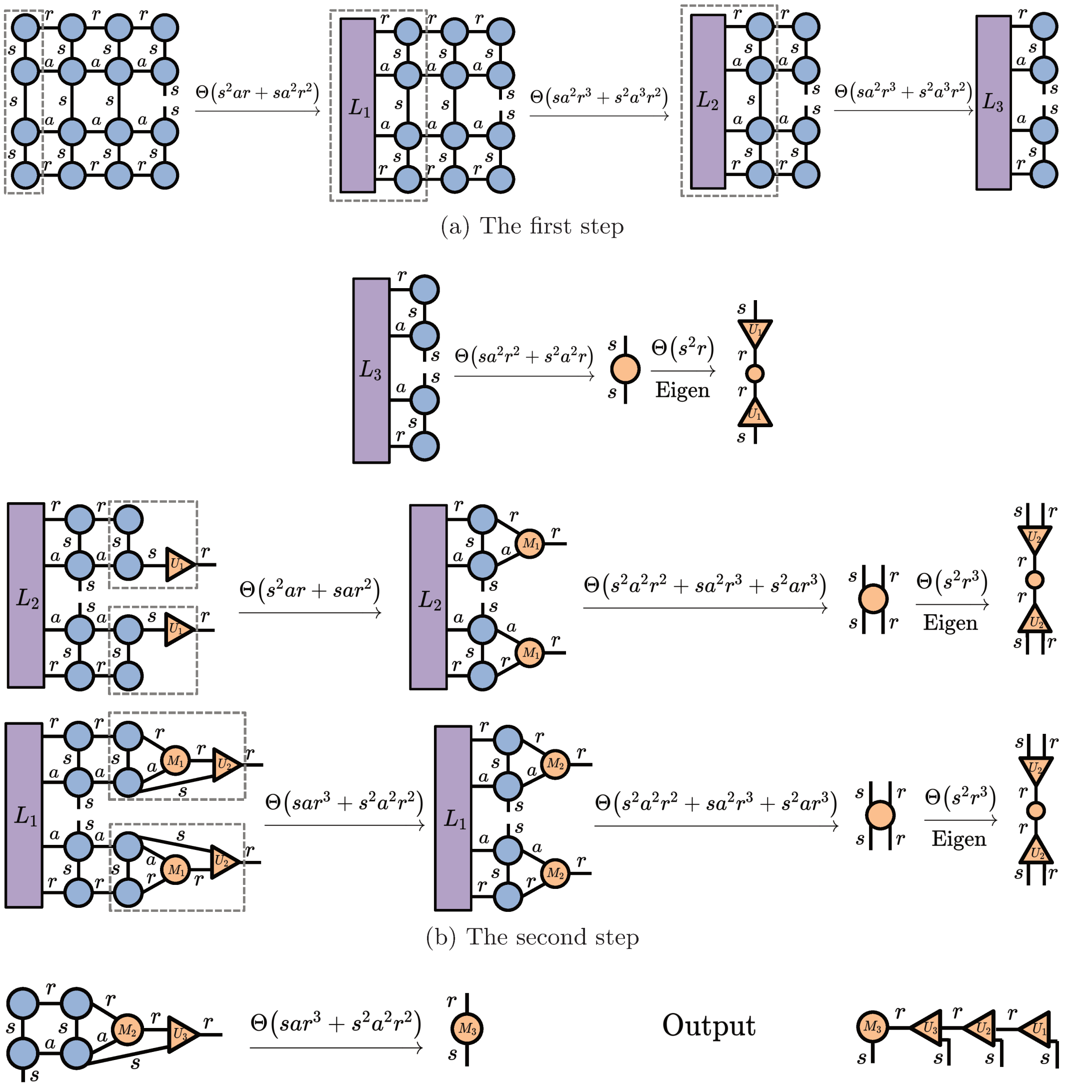
Figure 8: Illustration of the application and truncation algorithm.

The canonicalization-based algorithm that uses the full environment is illustrated in Fig. 8. The algorithm first multiplies the MPS and MPO, resulting in an MPS with a rank of ar. Subsequently, the MPS is truncated via the canonicalization-based algorithm reviewed in Section 3.2.1. When the output MPS has an order N, the algorithm has an asymptotic cost of $\Theta(N(s^2a^2r^2 + sa^3r^3))$, which is $O(a^2)$ times the cost of the zip-up algorithm. However, this algorithm offers better accuracy since each truncation utilizes the full environment. Furthermore, the algorithm maintains a theoretical upper bound on the truncation error [59].

The density matrix algorithm The density matrix algorithm produces an equivalent truncated MPS as the application and truncation algorithm, and we illustrate the algorithm in Fig. 9. The algorithm contains three steps,

1. Computing matrices $\mathbf{L}_i$, as is shown in Fig. 9a. These matrices are computed by sequentially contracting the network from left to right, and intermediates $\mathbf{L}_i$ are saved during the contractions.

2. Performing a sweep of contractions from right to left and use $\mathbf{L}_i$ to compute all the leading singular vectors $\mathbf{U}_i$ for $i \in \{1, \ldots, N-1\}$. Specifically, $\mathbf{L}_N$ is firstly used to compute the density matrix with the last pair of uncontracted modes left open, and truncated eigendecomposition is performed on the density matrix to yield the leading singular vectors $\mathbf{U}_1$. Next, the intermediates $\mathbf{L}_{N-1}$ is utilized to compute the density matrix with the right two uncontracted modes left open. Additionally, the basis of this density matrix is transformed by applying $\mathbf{U}_1$, as shown in Fig. 9b. This process is repeated until $N-1$ tensors $\mathbf{U}_i$ are obtained.



$\Theta(s^2ar + sa^2r^2)$
$\Theta(sa^2r^3 + s^2a^3r^2)$
$\Theta(sa^2r^3 + s^2a^3r^2)$
L_1
L_2
L_3
$\Theta(sa^2r^2 + s^2a^2r)$
$\Theta(s^2r)$
Eigen
U_1
$\Theta(s^2ar + sar^2)$
M_1
$\Theta(s^2a^2r^2 + sa^2r^3 + s^2ar^3)$
$\Theta(s^2r^3)$
Eigen
U_2
$\Theta(sar^3 + s^2a^2r^2)$
M_2
$\Theta(s^2a^2r^2 + sa^2r^3 + s^2ar^3)$
$\Theta(s^2r^3)$
Eigen
$\Theta(sar^3 + s^2a^2r^2)$
M_3
U_3
Output

(a) The first step

(b) The second step

(c) The third step

Figure 9: Illustration of the density matrix algorithm. Each triangular vertex represents a tensor with an orthogonal property.

3. Getting the leftmost matrix $\mathbf{M}_N$ that encompasses all the non-orthogonal information through the contraction depicted in Fig. 9c, and form the output MPS by combining all $\mathbf{U}_i$ and $\mathbf{M}_N$.

When the output MPS has an order N, the density matrix algorithm has an asymptotic cost of $\Theta(N(sa^2r^3+s^2a^3r^2+s^2ar^3))$. In applications arising in statistical physics and quantum computing, the size s is commonly the smallest. As is shown in Table 1, for the case where $s \ll a = \Theta(r)$, the cost of density matrix algorithm is $\Theta(s^2r^5)$, which is $\Theta(r/s)$ better than the canonicalization with full environment algorithm. For the other case, where $s = \Theta(a) \ll r$, the cost of the density matrix algorithm is $\Theta(s^3r^3)$, which is $\Theta(s)$ better than the canonicalization with full environment algorithm.

Automation and generalization of the MPO-MPS multiplication algorithms There is an opportunity to generalize the MPO-MPS multiplication algorithms to arbitrary graphs. In particular, SweepContractor [14] generalizes the MPO-MPS zip-up algorithm, and uses a subroutine that contracts a single tensor with an MPS into a new MPS to contract arbitrary tensor networks. In contrast, our proposed algorithm includes a subroutine that contracts a general tensor network (such as an MPO) with a binary tree tensor network into a binary tree network, allowing the generalizing of all three MPO-MPS multiplication algorithms.

The analysis and observations above suggest that the density matrix algorithm has greater efficiency compared to the canonicalization-based algorithm. As a result, we generalize the density matrix algorithm for the MPO-MPS multiplication and implement one that is able to approximate a general tensor network into a tree tensor network. Generalization of the density matrix algorithm to trees presents two challenges. Firstly, determining how to efficiently perform memoization (accelerating the contraction by caching partial contraction results) to reduce costs becomes less straightforward. In order to address this issue, we have introduced a strategy that utilizes graph partitioning in Section 6. Secondly, selecting an appropriate output tree structure that enhances the efficiency of the approximation poses a challenge. For the MPO-MPS multiplication, it is evident that the MPS ordering consistent with the input MPS and MPO would yield favorable results. In Section 5, we propose algorithms to select efficient tree structures for general graphs.

4 The proposed tensor network contraction algorithm

In this section, we present the proposed approximate tensor network contraction algorithm.

4.1 Definitions

We use $G[S] = (S, E_S)$ to denote a sub tensor network defined on $S \subseteq V$, where E_S contains all edges in E adjacent to any $v \in S$. For two disjoint subsets of V denoted as X, Y, we let $E(X, Y)$ denote the set of edges connecting X, Y. We let $E(X)$ denote the set of uncontracted edges of $G[X]$.

For the tensor network represented by $G = (V, E)$, we use $\mathcal{V} = \{V_1, \ldots, V_N\}$ to denote a graph partitioning that partitions V into $V_1, \ldots, V_N$. A *contraction tree* of the partitioned network is a directed binary tree showing how vertex subsets in $\mathcal{V}$ are contracted, and it is denoted $T^{(\mathcal{V})}$. Each leaf of $T^{(\mathcal{V})}$ is a vertex subset in $\mathcal{V}$, and each non-leaf vertex in $T^{(\mathcal{V})}$ can be represented by a subset of the vertices, $W_1 \cup W_2$, where its two children are represented by W_1 and W_2, respectively.

For a given vertex in the contraction tree $T^{(\mathcal{V})}$ that is represented by $V' \subset V$, $\mathsf{path}(T^{(\mathcal{V})}, V')$ denotes a sub-contraction path of $T^{(\mathcal{V})}$. This sub-contraction path is a subgraph of $T^{(\mathcal{V})}$ that contains all vertices in $T^{(\mathcal{V})}$ that are ancestors of V' as well as the children of these ancestors. To illustrate, we provide an example of the sub-contraction path of V_4 in Fig. 10.

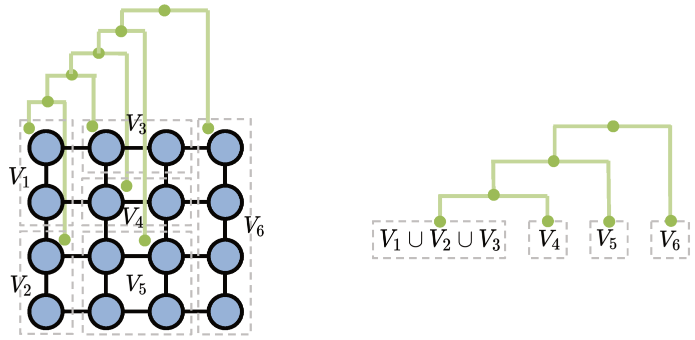


Figure 10: Illustration of the sub-contraction path. The left diagram denotes the graph partitioning and the contraction tree $T^{(\mathcal{V})}$, and the right diagram denotes the sub-contraction path $\mathsf{path}(T^{(\mathcal{V})}, V_4)$.

4.2 An overview of the algorithm

In this section we present an overview of the algorithm. The algorithm takes as inputs a tensor network, a partitioning of that tensor network, a partial contraction tree of the partitioned tensor network, an ansatz for the structure of intermediate network contractions (for example, an MPS or a comb tree), and parameters for performing intermediate approximate tensor network contractions which tune the level of accuracy of the method. The algorithm proceeds by traversing the partial contraction tree and approximately contracting pairs of tensor network partitions specified by the contraction tree. The pair of tensor network partitions are approximately contracted using a specified algorithm such as the density matrix algorithm, resulting in a tensor network with the specified structure, such as an MPS or comb tree structure. The algorithm proceeds until all partitions are contracted. An example of the algorithm is shown in Fig. 4.

Algorithm 2 `partitioned_contract`: approximate tensor network contraction based on its given partition

```
1:  Input: The tensor network $\mathfrak{T}$ with graph $G = (V, E)$, its partition $\mathcal{V} = \{V_1, \dots, V_N\}$ and its
    contraction path $T^{(\mathcal{V})}$, ansatz $A$, maximum bond size $\chi$, and swap batch size $r$ // The ansatz
    $A$ can be either “MPS” or “Comb”
2:  tn ← a mapping that maps each vertex set to its approximated tensor network
3:  $\mathcal{E} \leftarrow \{E(V_i, V_j) : i, j \in \{1, \dots, N\}\}$ // The set where each element is an edge subset connecting two
    different partitions
4:  // Lines 5-9: construct edge/edgeset linear orderings that define the embedding tree
5:  $\left\{\sigma^{(E')} : E' \in \mathcal{E}\right\}$ ← selecting an ordering for each edgeset in $\mathcal{E}$ via recursive bisection
6:  for each contraction $(U_s, W_s) \in T^{(\mathcal{V})}$ do
7:      $\mathcal{E}_s$ ← The subset of $\mathcal{E}$ that is adjacent to the sub tensor network with vertices $U_s \cup W_s$
8:      $\sigma^{(\mathcal{E}_s)}$ ← embedding_tree_ordering$\left(G[U_s \cup W_s], \texttt{path}\left(T^{\mathcal{V}}, U_s \cup W_s\right), \mathcal{E}_s\right)$ // Select an order-
        ing for edgesets in $\mathcal{E}_s$
9:  end for
10: for each contraction $(U_s, W_s) \in T^{(\mathcal{V})}$ do
11:     tn$(U_s \cup W_s)$ ← approx_tensor_network$\left(\texttt{tn}(U_s) \cup \texttt{tn}(V_s), \sigma^{(\mathcal{E}_s)}, \{\sigma^{(E')} : E' \in \mathcal{E}_s\}, \chi, r, A\right)$ //
        Approximate the input tensor network tn$(U_s)\cup$tn$(V_s)$ as a binary tree tensor network $\mathfrak{X}_s$, Algorithm 6
12: end for
13: return the final approximated tensor network, tn$(V)$
```

Pseudocode providing more details of steps of the algorithm is presented in Algorithm 2. Definitions of all notations are summarized in Table 2. The algorithm takes as input the tensor network partition $\mathcal{V}$ and its contraction path $T^{(\mathcal{V})}$. During each contraction step along the path, all tensors within the input partitions are treated as the environment. Consequently, larger partitions typically lead to higher approximation accuracy, but at the cost of increased computational complexity.

When two tensor network partitions are contracted, an embedding tree is first constructed which specifies the structure of the network that will result from the contraction. The embedding tree is a full binary tree where each leaf vertex is associated with a dangling edge/mode of the subnetwork made from composing the two partitions that are being contracted. Furthermore, each non-leaf vertex in the embedding tree corresponds to a tensor within the resulting binary tree tensor network. All tensors within this network have an order of three, except for the tensor located at the root vertex. An example of such an embedding tree is illustrated in the second left diagram of Fig. 4b.

The selection of the embedding tree is guided by an analysis of the structure of the input tensor network graph G, its partitioning, and the contraction path. This analysis aims to identify a tree structure that optimizes the efficiency of both the current contraction and any subsequent contractions involving the contracted output. The determination of each embedding tree structure occurs in lines 5-9. Note that the generation of the embedding tree only depends on the tensor network graph structure, rather than the actual tensor data. The relationship between the embedding tree and the orderings of the edges is further explained in Section 4.3.

While our method introduces a novel perspective on the construction of the embedding tree, it is worth noting that alternative approaches have been proposed for determining tree structures based on various other heuristics. For instance, Seitz et al. [69] propose a method specifically for determining a tree structure based on a quantum circuit. Other studies, such as those by Nakatani and Chan [55], Murg et al. [54], Szalay et al. [75], and Ferrari et al. [23], explore different heuristics for tree construction, leveraging factors like entanglement or interaction strength.

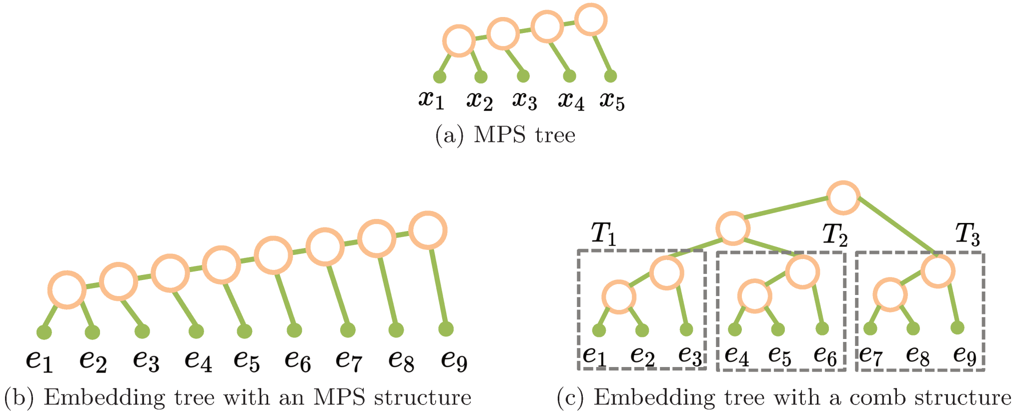
After selecting an embedding tree, we proceed to embed the tensor network comprising two partitions into the embedding tree and truncate it to ensure that the maximum bond size remains below χ. This process is performed in lines 10-12. In-depth explanations of the hybrid algorithm, which combines the density matrix algorithm and the swap-based algorithm to obtain the approximated binary tree tensor network, can be found in Section 6 and Section 7. This hybrid algorithm involves multiple iterations of the density matrix algorithm, each progressively modifying the structure of the tensor network to a degree controlled by the swap batch size r. The choice of r allows the user to find a balance between accuracy and computational cost for specific problem instances.

4.3 Determination of the embedding tree

We explain the embedding tree structure used in Algorithm 2. As is defined in Section 4.2, an embedding tree is a rooted full binary tree, with each leaf vertex representing an uncontracted edge in the tensor network.

Let $\mathcal{E} = \{E(V_i, V_j) : i, j \in \{1, \dots, N\}\}$, so that each element in $\mathcal{E}$ is an edge subset connecting two different partitions. For a specific contraction (U_s, W_s) defined in Table 2, we let $\mathcal{E}_s$ be the subset of $\mathcal{E}$ that is adjacent to the tensor network represented by $U_s \cup W_s$. We design the embedding tree structure so that the leaves that represent each $E_i \in \mathcal{E}_s$ are in close proximity to one another. This arrangement is advantageous because all edges within each E_i are always contracted together in the same contraction. Placing them close to each other simplifies the contraction process and eliminates the need for unnecessary permutation of modes.

Two structures we use for the embedding tree are the MPS (maximally-unbalanced full binary tree) and the comb [5, 13]. The comb tensor network is a tree tensor network arranged in a linear chain with branches. Both structures are based on a linear orderings $\sigma^{(\mathcal{E}_s)}$ for $\mathcal{E}_s$ and linear orderings $\sigma^{(E')}$ for $E' \in \mathcal{E}_s$, and they are generated in lines 5-9 of Algorithm 2. We formally define the embedding tree with an MPS and a comb structure in Appendix C. We visualize both the embedding tree with an MPS structure and with a comb structure in Fig. 11.


x_1 x_2 x_3 x_4 x_5
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9
T_1
T_2
T_3
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9

(a) MPS tree

(b) Embedding tree with an MPS structure

(c) Embedding tree with a comb structure

Figure 11: (a) Visualization of the MPS tree defined on σ^S with $\sigma^S(x_i) = i$. (b)(c) Visualization of the embedding tree with an MPS structure and a comb structure. The input orderings are $\sigma^{(\mathcal{E}_i)} = (E_1, E_2, E_3)$ with $E_1 = \{e_1, e_2, e_3\}$, $E_2 = \{e_4, e_5, e_6\}$, and $E_3 = \{e_7, e_8, e_9\}$. $\sigma^{(E_1)}, \sigma^{(E_2)}, \sigma^{(E_3)}$ are defined so that in $\hat{\sigma} = \sigma^{(E_1)} \oplus \sigma^{(E_2)} \oplus \sigma^{(E_3)}$, $\hat{\sigma}(e_i) = i$.

In comparison to the MPS structure, the comb structure has a smaller diameter, representing the maximum distance between any two vertices. However, the comb structure also has a larger maximum tensor size of $\Theta(\chi^3)$, where χ is the maximum bond size. This is larger than the maximum tensor size of the MPS structure, which is $\Theta(s\chi^2)$, where s represents the uncontracted mode size and is typically much smaller than χ. In Section 8, we conduct experimental comparisons between the performance of the MPS structure and the comb structure.

Various heuristics can be used to obtain the linear ordering $\sigma^{(E')}$ for each $E' \in \mathcal{E}$. In this work, we utilize the recursive bisection algorithm described in Appendix B.2, on a partition of the input graph G that is connected to E'. The recursive bisection algorithm is a heuristic that aims to minimize congestion in the linear ordering. By applying this algorithm, we obtain an ordering that results in the embedding tree tensor network having low ranks. The algorithm for selecting the ordering $\sigma^{(\mathcal{E}_s)}$ is explained in detail in Section 5.

5 The algorithm to select the edge subset ordering of the embedding tree

For a given contraction (U_s, W_s), we detail the algorithm to select the linear ordering $\sigma^{(\mathcal{E}_s)}$ for the intermediate tensor network $G_s = (V_s, E_s)$, where $V_s = U_s \cup W_s$. $\sigma^{(\mathcal{E}_s)}$ is generated based on both G_s and the sub-contraction path $T = \mathsf{path}\left(T^{(\mathcal{V})}, V_s\right)$ defined at Section 4.1, where $T^{(\mathcal{V})}$ is the contraction tree over the partition $\mathcal{V}$.

The ordering $\sigma^{(\mathcal{E}_s)}$ is chosen with two objectives. Firstly, it is designed to satisfy a specific adjacency relation that greatly facilitates efficient subsequent contractions. This adjacency relation ensures that for each of the subsequent contractions (U_k, W_k), the contracted edges between U_k and W_k are adjacent in both input tensor networks $\mathtt{tn}(U_k)$ and $\mathtt{tn}(V_k)$. The adjacency of these contracted edges results in a lower cost for the contraction, compared to the scenario where the contracted edges are not adjacent. This adjacency relation is described by the *constraint tree* for $\mathcal{E}_s$, $T^{(\mathcal{E}_s)}$. Each leaf vertex in the constraint tree represents an edge set in $\mathcal{E}_s$, and each non-leaf vertex has at least 2 children and indicates the edge subsets represented by the children are adjacent. Each non-leaf vertex also denotes whether the children's vertices are ordered or not. We show an example of the constraint tree in the bottom right diagram of Fig. 25. In Appendix D, a detailed explanation is provided on how to select the constraint tree.

Secondly, the resulting binary tree structure should be similar to the tensor network G_s in order to keep the ranks of the resulting tree tensor network low. In Section 5.1, we detail the algorithm to find the ordering not only consistent with the constraint tree, but also to minimize the cost of permutation (Kendall-Tau distance between the chosen ordering and another reference ordering whose corresponding line structure is similar to G_s).

5.1 Determination of the edge set ordering based on the constraint tree

We provide an explanation of the algorithm that determines the ordering for the set of elements $\mathcal{E}_s$, denoted as $\sigma^{(\mathcal{E}_s)}$. This ordering is not only constrained by the constraint tree $T^{(\mathcal{E}_s)}$ but also aims to reflect the structure of the input graph G_s. The algorithm is presented in Algorithm 3. To begin with, in Line 2, we generate a reference ordering denoted as τ for the set of elements $\mathcal{E}_s$. This reference ordering is generated using recursive bisection and represents a linear structure that is close to the structure of G_s. Subsequently, the algorithm proceeds to construct the output ordering by employing a post-order DFS traversal of the constraint tree $T^{(\mathcal{E}_s)}$. This traversal strategy ensures that the ordering takes into account the constraints imposed by the tree structure. In Appendix E, we prove that the output ordering of Algorithm 3 minimizes the Kendall-Tau distance with the reference ordering under the adjacency constraint.

6 The density matrix algorithm for tree approximations

We present a density matrix algorithm to approximate an arbitrary tensor network into a tree tensor network. The standard approach involves embedding the input tensor network into an embedding tree and explicitly forming the untruncated tree tensor network, then truncating the resulting tree tensor network using the canonicalization-based algorithm. However, this can lead

Algorithm 3 `linear_ordering_under_constraint_tree`: Algorithm to get the edge set ordering that minimizes the Kendall-Tau distance with the reference ordering under the adjacency constraint

```
Input: the edge set $\mathcal{E}_s$, the constraint tree $T^{(\mathcal{E}_s)}$, the tensor network graph $G_s = (V_s, E_s)$
$\tau \leftarrow$ linear_ordering$(\mathcal{E}_s, G_s)$ // Ordering generated via recursive bisection
$f \leftarrow$ a mapping that maps each vertex in $T^{(\mathcal{E}_s)}$ to its edge set ordering
for each leaf vertex $v$ that represents $E_i$ in $T^{(\mathcal{E}_s)}$ do
    $f(v) \leftarrow$ the ordering that contains the single edge set $E_i$
end for
for each non-leaf vertex $v$ that represents $\hat{\mathcal{E}}_i$ based on a post-order DFS traversal of $T^{(\mathcal{E}_s)}$ do
    $u_1, \dots, u_{n_v} \leftarrow$ children of $v$
    if $v$ is labeled as ordered then
        $\sigma_1 \leftarrow f(u_1) \oplus f(u_2) \oplus \cdots \oplus f(u_{n_v})$ // Concatenate all $f(u_i)$ in order
        $\mathcal{S} \leftarrow \{\sigma_1, \texttt{reverse}(\sigma_1)\}$
    else
        $\mathcal{S} \leftarrow$ a set of all permutations of $\{f(u_1), f(u_2), \dots, f(u_{n_v})\}$
    end if
    $\tau_v \leftarrow$ a partial ordering of $\tau$ over the subset $\hat{\mathcal{E}}_i$
    $f(v) \leftarrow \arg\min_{\sigma \in \mathcal{S}} d_{\mathrm{KT}}(\sigma, \tau_v)$ // Minimize the Kendall-Tau distance defiend at Definition 3
end for
return $f\left(\texttt{root}\left(T^{(\mathcal{E}_s)}\right)\right)$
```

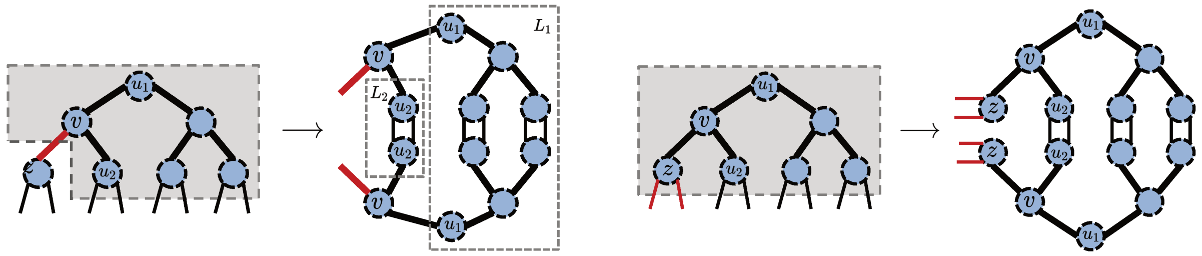
to tree tensor networks with large ranks, resulting in expensive canonicalization and low-rank approximation processes.

Our proposed density matrix algorithm builds upon the density matrix algorithm originally designed for MPO-MPS multiplication, which is discussed in Section 3.2.2. Given a tree embedding of the input tensor network, our algorithm eliminates the need to explicitly construct the untruncated tree tensor network. It offers the advantage of forming a low-rank tree tensor network without requiring the generation of large intermediate tensors. Specifically, we show in Section 6.4 that the asymptotic computational cost of the algorithm is upper-bounded by the cost of the canonicalization-based algorithm, and we show in Section 8 that for many input tensor networks, the proposed algorithm substantially reduces the overall execution time.

6.1 Definitions

Within the algorithm, we use $\texttt{density\_matrix}_T(v)$ and $\texttt{density\_matrix}_T(v, z)$ introduced in Definition 2. For a given embedding tree $T = (V_T, E_T)$ with each vertex in T representing a partition of the tensor network embedded to that vertex, we use the notation $\texttt{density\_matrix}_T(v)$ to calculate the density matrix of vertex v on top of the embedding tree T, with the open edges of the matrix being the uncontracted edges incident to v. Moreover, $\texttt{density\_matrix}_T(v, z)$ calculates the density matrix of vertex v with the open edge of the matrix being $E_T(v, z)$. We show an illustration in Fig. 12.

Definition 2 (Density matrix)**.** Consider a given embedding tree $T = (V_T, E_T)$ with each $T(v)$ for $v \in V_T$ representing a sub tensor network, and let $T(S) = \cup_{v \in S} T(v)$. For a given vertex $v \in V_T$, and a set of edges $\tilde{E}_v \subset E_T$ that is adjacent to v, let $S \subseteq V_T$ denote the vertices connected to v when $\tilde{E}_v$ is removed from T. Let $\mathbf{T}_{(\tilde{E}_v)}$ denote the matricization of the tensor network $T(S)$ with all modes defined by $\tilde{E}_v$ are combined into the matrix row. Then the density matrix defined on $T, v, \tilde{E}_v$, denoted as $\texttt{density\_matrix}_T\left(v, \tilde{E}_v\right)$, equals $\mathbf{T}_{(\tilde{E}_v)}\mathbf{T}^T_{(\tilde{E}_v)}$. For simplicity, we let $\texttt{density\_matrix}_T(v)$ denote the density matrix of v when $\tilde{E}_v = E_T(v, *)$ is the uncontracted edge set incident on v, and we let $\texttt{density\_matrix}_T(v, u)$ denote the density matrix of v when $\tilde{E}_v = E_T(u, v)$.



(a) $\texttt{density\_matrix}_T\,(v,z)$ (b) $\texttt{density\_matrix}_T\,(z)$

Figure 12: Visualization of $\texttt{density\_matrix}_T\,(v,z)$ and $\texttt{density\_matrix}_T\,(z)$. In the left diagrams of (a)(b), the tree structure is the embedding tree T, and each vertex represents a partition of the network embedded in that vertex. The open edge of the density matrix is marked in red. The dashed boxes denote the tensor networks squared in the density matrices ($T(S)$ in Definition 2). The right diagrams visualizes the density matrices. In (a), $L_1 = \texttt{density\_matrix}_T\,(u_1,v)$ and $L_2 = \texttt{density\_matrix}_T\,(u_2,v)$ can be cached and reused when computing the density matrix.

For a tensor network $G=(V,E)$, we also define $\mathsf{cut}_G(X,Y)=\sum_{e\in E(X,Y)} w(e)$, where $E(X,Y)$ denotes the set of edges connecting two disjoint vertex subsets X,Y. For two vertices $u,v\in V$, we define the minimum cut between u,v in G as

$$\mathsf{mincut}_G(u,v)=\min_{\substack{A,B\subset V\\ u\in A,v\in B}} \mathsf{cut}_G(A,B).$$

Let E_1,E_2 be the two different subsets of the uncontracted edges of G, we define $\mathsf{mincut}_G(E_1,E_2)$ as the mincut between two new vertices a,b on the graph that contains both G and a,b, where a,b are adjacent to E_1,E_2, respectively.

6.2 The density matrix algorithm

The density matrix algorithm is summarized in Algorithm 4. The algorithm involves computing the output network by performing a post-order DFS traversal of the embedding tree. During the traversal, at each vertex v, the corresponding tensor $\mathbf{U}_v$ is computed. Subsequently, vertex v is removed from the embedding tree. This process continues iteratively until only the root vertex remains, whose tensor encapsulates all the non-orthogonal information of the network. A visualization of the algorithm is shown in Fig. 13.

In Algorithm 4, we initially construct an embedding ϕ utilizing the recursive bisection technique outlined in Algorithm 5. This embedding assigns a tensor network partition to each vertex in the embedding tree and serves as a guide for the memoization strategy. As is reviewed in Appendix B.2, recursive bisection is a standard heuristic to find embeddings with low congestion. It is worth noting that Algorithm 5 may produce an embedding in which there exists a vertex in the embedding tree whose corresponding tensor network partition is empty. In such cases, we can address this problem by introducing identity matrices into the input graph. This adjustment ensures that the resulting tensor network remains equivalent while guaranteeing the non-emptiness of each partition.

For computing $\mathbf{U}_v$ at each vertex $v\in V_T$, Algorithm 4 incorporates two subroutines that handle two distinct cases efficiently. In the algorithm, we let $\mathbf{M}_v$ denote the matricized contraction output of the partition at v, $T(v)$, that combines all uncontracted modes into the matrix row. In addition, let $\mathbf{L}_v = \texttt{density\_matrix}_{T'}(v)$ and $\mathbf{L}_u = \texttt{density\_matrix}_{T'}(u,v)$.

Since $\mathbf{L}_v = \mathbf{M}_v\mathbf{L}_u\mathbf{M}_v^T$, if the number of rows in $\mathbf{L}_v$ is smaller than the number of rows in $\mathbf{L}_u$, in Lines 10-11 we compute $\mathbf{L}_v$ then obtain its singular vectors, which is the most efficient approach. Conversely, if the number of rows in $\mathbf{L}_v$ exceeds the number of rows in $\mathbf{L}_u$, it implies that $\mathbf{L}_v$ is not full rank. In such cases, we use an subroutine called QR-SVD [62] instead in Lines 13-18. we first use QR factorization to orthogonalize $\mathbf{M}_v$ and yield $\mathbf{Q}_v\mathbf{R}_v$, and subsequently calculate the leading singular vectors of $\mathbf{R}_v\mathbf{L}_u\mathbf{R}_v^T$, which yields an implicit representation of the singular vectors of $\mathbf{L}_v$.

QR-SVD avoids the generation of the large density matrix $\mathbf{L}_v$, thus having a better asymptotic cost. In Section 6.4, we demonstrate that Algorithm 4 provides a guarantee that its asymptotic computational cost remains upper-bounded by that of the canonicalization-based algorithm.

Algorithm 4 `density_matrix_alg`: The density matrix algorithm for tree approximation

```
Input: The tensor network $G = (V, E)$, its embedding tree $T = (V_T, E_T)$, and maximum bond size $\chi$
$\phi \leftarrow$ tree_embedding$(G, T)$ // Constructed based on Algorithm 5
$r \leftarrow$ root vertex in $T$
$T' \leftarrow$ a tree with the same structure as $T$ and $T'(v)$ for $v \in V_T$ denotes all tensors embedded to $v$ in $\phi$
for each $v \in V_T \setminus \{r\}$ based on a post-order DFS traversal of $T$ do
    $A_v \leftarrow$ uncontracted_edges$(T', v)$
    $B_v \leftarrow$ contracted_edges$(T', v)$
    $u \leftarrow$ parent$(T', v)$
    if $w(A_v) = O(w(B_v))$ then
        $\mathbf{L}_v \leftarrow$ density_matrix$_{T'}(v)$ // Defined in Definition 2
        $\mathbf{U}_v \leftarrow$ leading_eigenvectors$(\mathbf{L}_v, \chi)$
    else
        // Perform QR-SVD [62] to reduce the asymptotic cost
        $\mathbf{L}_u \leftarrow$ density_matrix$_{T'}(u, v)$
        $\mathbf{M}_v \leftarrow$ the matricized contraction output of $T(v)$ with $A_v$ combined into row
        $\mathbf{Q}_v, \mathbf{R}_v \leftarrow$ QR$(\mathbf{M}_v)$
        $\hat{\mathbf{U}}_v \leftarrow$ leading_singular_vectors$(\mathbf{R}_v \mathbf{L}_u \mathbf{R}_v^T, \chi)$
        $\mathbf{U}_v \leftarrow \mathbf{Q}_v \hat{\mathbf{U}}_v$
    end if
    Add both $T'(v)$ and a vertex that represents $\mathbf{U}_v^T$ to $T'(u)$, and remove $v$ from $T'$
end for
$\mathbf{M}_r \leftarrow$ contraction output of $T(r)$
return the tree tensor network that contains all $\mathbf{U}_v$ and the root tensor $\mathbf{M}_r$
```

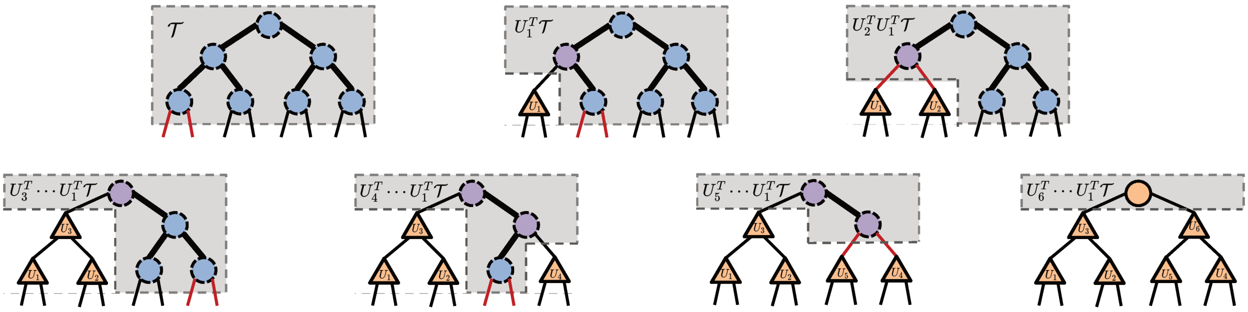

$\mathcal{T}$
$U_1^T\mathcal{T}$
$U_2^TU_1^T\mathcal{T}$
$U_3^T \cdots U_1^T\mathcal{T}$
$U_4^T \cdots U_1^T\mathcal{T}$
$U_5^T \cdots U_1^T\mathcal{T}$
$U_6^T \cdots U_1^T\mathcal{T}$
U_1
U_2
U_3
U_4
U_5
U_6

Figure 13: Visualization of the density matrix algorithm. The tree structure in each diagram is the embedding tree. Each dashed circle represents a partition of the tensor network, and each solid circle/rectangle represents a tensor. Blue, purple, and orange vertices represent the input tensor network, intermediate tensors generated during the algorithm, and the output tensors, respectively. The input tensor network is represented by the top left diagram, and the output one is represented by the bottom right diagram. In each diagram, the network included in the dashed box has a structure of T' in Algorithm 4 and is used to compute the density matrix, and red edges denote the open edges of the density matrix.

Algorithm 5 `tree_embedding`: embedding a graph into the embedding tree via recursive bisection

```
Input: The source graph $G = (V, E)$, the embedding tree $T = (V_T, E_T)$
if $|V_T| = 1$ then
    return an embedding that mapping all $v \in V$ to the vertex in $V_T$
end if
$\phi \leftarrow$ an empty embedding function
$r \leftarrow$ root vertex in $T$
$E_L, E_R \leftarrow$ open edges represented by the left leaves and right leaves $r$, respectively
$S_L, S_R \leftarrow$ bipartition of $V$ such that $\mathsf{cut}_G(S_L, S_R) = \mathsf{mincut}_G(E_L, E_R)$
$\phi_L \leftarrow$ tree_embedding $(G[S_L], \texttt{left\_child\_tree}(T))$
$E'_L \leftarrow E(S_L, S_R)$
$S'_L, S'_R \leftarrow$ bipartition of $S_R$ such that $\mathsf{cut}_G(S'_L, S'_R) = \mathsf{mincut}_G(E'_L, E_R)$
For each $v \in S'_L$, let $\phi(v) = r$
$\phi_R \leftarrow$ tree_embedding $(G[S'_R], \texttt{right\_child\_tree}(T))$
return the combination of $\phi, \phi_L, \phi_R$
```

6.3 The density matrix algorithm with memoization

As can be seen from Fig. 13, there are many shared tensor network parts across density matrices. We present a memoization strategy that generalizes the memoization strategy for the density matrix algorithm of the MPO-MPS multiplication to reduce the computational cost. The strategy is used in Lines 10, 14 of Algorithm 4.

The memoization strategy uses the following recursive relation for $\texttt{density\_matrix}_T\,(v)$ and $\texttt{density\_matrix}_T\,(v, z)$,

$$\begin{aligned} \texttt{density\_matrix}_T\,(v) &= \mathbf{M}^{(v)}_{E_T(v,*)} \left(\bigotimes_{u \in N(v)} \texttt{density\_matrix}_T(u, v) \right) \mathbf{M}^{(v)T}_{E_T(v,*)}, \\ \texttt{density\_matrix}_T\,(v, z) &= \mathbf{M}^{(v)}_{E_T(v,z)} \left(\bigotimes_{u \in N(v) \setminus \{z\}} \texttt{density\_matrix}_T(u, v) \right) \mathbf{M}^{(v)T}_{E_T(v,z)}, \end{aligned} \tag{3}$$

where $N(v)$ denotes the set of vertices adjacent to v, $\otimes$ denotes a Kronecker product, and $\mathbf{M}^{(v)}_{E_T(v,*)}$ denote a matricization of the tensor network represented by v, $T(v)$. In this matricization, all uncontracted modes incident on v are combined into the row. $\mathbf{M}^{(v)}_{E_T(v,z)}$ denote a matricization of $T(v)$ where the mode represented by the edge $E_T(v, z)$ is the matrix row.

To compute the density matrix $\texttt{density\_matrix}_T\,(v, z)$, we first compute the density matrices for its neighboring vertices $u \in N(v) \setminus \{z\}$, then contract the target network that contains the density matrices as well as the tensor network $T(v)$ following (3). The contraction cost of the above target tensor network is dependent on the selected contraction path, and in practice one can either choose the optimal contraction path that minimizes the contraction cost or select it based on multiple heuristics [27]. Note that one way to contract the target network is to contract $T(v)$ into a tensor first and then contract it with the density matrices, but it may not yield the optimal cost. If the terms $\texttt{density\_matrix}_T(u, v)$ have already been computed when generating other density matrices, we will cache and reuse them here. We illustrate such strategy in Fig. 12a. The same strategy is used to compute $\texttt{density\_matrix}_T\,(v)$.

In Algorithm 4, the computation of each density matrix occurs only once. Considering that the embedding tree T is limited to being a rooted binary tree, there are at most three density matrices to be calculated for each vertex v in the embedding tree. Below we bound the asymptotic computational cost of the density matrix algorithm using memoization, and we show that for a given embedding ϕ, the cost will be upper-bounded by the algorithm that uses canonicalization, justifying the efficiency of the algorithm.

6.4 Computational cost analysis

We compare the asymptotic computational costs of the density matrix algorithm and the baseline algorithm that utilizes canonicalization, as discussed in Section 3.2.1. In Theorem 6.1, we establish that the density matrix algorithm can be more efficient in approximating a general tensor network as an embedding tree. The cost of the density matrix algorithm is upper-bounded by that of the canonicalization-based algorithm. This efficiency arises from the fact that the density matrix algorithm does not need to explicitly contract the partition embedded in each tree vertex into a tensor.

Theorem 6.1. *Consider a given tensor network $G = (V, E)$, an embedding tree $T = (V_T, E_T)$, and an embedding ϕ that embeds G into T. Let $\sigma : V_T \to \{1, \dots, |V_T|\}$ be a post-order DFS traversal of T that shows the the tensor update ordering. Assuming that changing a tree tensor network into its canonical form will not change any bond size of the network, the asymptotic cost of the density matrix algorithm (Algorithm 4) is upper-bounded by that of the canonicalization-based algorithm (Algorithm 1) if both algorithms use the same embedding ϕ, the same update ordering σ, and the same maximum bond size χ.*

Proof. For the contraction of each density matrix at vertex v in the density matrix algorithm, a valid contraction path can be obtained by contracting the partition at v into a tensor first, then contracting it with other density matrices based on (3). The cost of this contraction path is an upper bound of the contraction cost of this density matrix, assuming the optimal contraction path is selected.

Therefore, the overall cost of the density matrix algorithm, assuming the optimal contraction path is used during the contraction of each density matrix, is upper-bounded by the case where each partition embedded into every vertex $v \in V_T$ is contracted into a tensor $\mathbf{M}_v$ prior to conducting the depth-first search (DFS) traversal. This transforms the tensor network into an untruncated tree tensor network. According to Lemma F.3 in Appendix F, both the density matrix algorithm and the canonicalization-based algorithm exhibit the same asymptotic cost when truncating a tree tensor network. By examining this particular case, we establish that the upper bound of the density matrix algorithm matches the asymptotic cost described in Algorithm 1. This finishes the proof. □

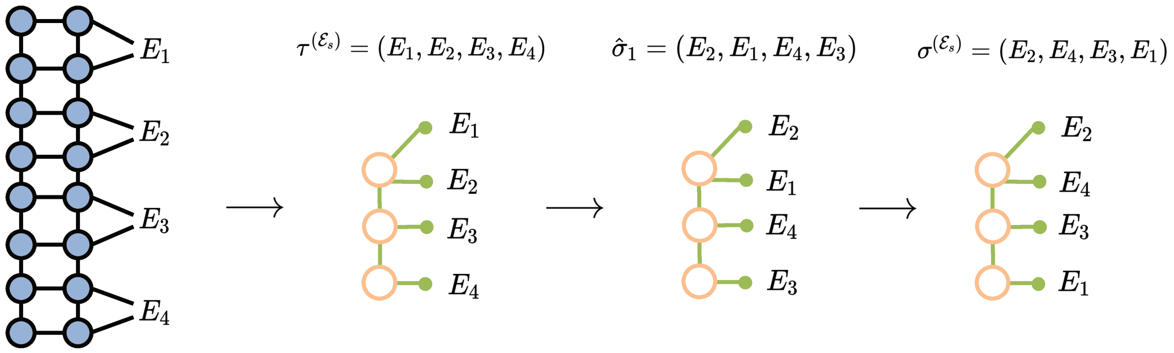
7 The algorithm to approximate an input tensor network into an embedding tree

We introduce a hybrid algorithm that combines the density matrix algorithm with the swap-based algorithm to approximate an input tensor network $G_s = (V_s, E_s)$ into an embedding tree. This hybrid algorithm offers a compromise between accuracy and computational cost by performing multiple iterations of the density matrix algorithm. Each iteration incrementally modifies the structure of the tensor network by a small degree, ensuring that the overall computational cost remains manageable. While this approach may sacrifice a certain degree of approximation accuracy, it provides a balanced solution that achieves a reasonable trade-off between accuracy and computational efficiency compared to the pure density matrix algorithm.

We present the algorithm in Algorithm 6, and an illustration is shown in Fig. 14. In this algorithm, we denote the edge set ordering in the embedding tree as $\sigma^{(\mathcal{E}_s)}$, and the reference edge set ordering of G_s as $\tau^{(\mathcal{E}_s)}$. We measure the structural difference between G_s and the embedding tree using the Kendall-Tau distance in Definition 3, defined as $d = d_{\mathrm{KT}}\left(\tau^{(\mathcal{E}_s)}, \sigma^{(\mathcal{E}_s)}\right)$. The algorithm utilizes a parameter r to control the extent of structural modifications made by each density matrix algorithm iteration. The number of density matrix algorithms performed is determined by $\lceil d/r \rceil$. Users can choose different values of r depending on the specific problem. By selecting a larger value

Algorithm 6 `approx_tensor_network`: approximate a tensor network into an embedding tree

```
Input: The tensor network $\mathfrak{T}$ with graph $G_s = (V_s, E_s)$, the edge set ordering $\sigma^{(\mathcal{E}_s)}$, the edge orderings $\{\sigma^{(E')} : E' \in \mathcal{E}_s\}$, the maximum bond size $\chi$, the swap batch size $r$, and ansatz $A$ // The ansatz $A$ can be either "MPS" or "Comb"
$\tau^{(\mathcal{E}_s)} \leftarrow$ linear_ordering$(\mathcal{E}_s, G_s)$ // Ordering generated via recursive bisection
$d \leftarrow d_{\mathrm{KT}}\left(\tau^{(\mathcal{E}_s)}, \sigma^{(\mathcal{E}_s)}\right)$ // Number of adjacent edge set swaps needed to change $\tau^{(\mathcal{E}_s)}$ to $\sigma^{(\mathcal{E}_s)}$
$n \leftarrow \lceil d/r \rceil$ // The number of density matrix algorithms to be performed
$\hat{\sigma}_1 \ldots, \hat{\sigma}_n \leftarrow n$ equally-spaced inverval orderings that separate $\tau^{(\mathcal{E}_s)}$ and $\sigma^{(\mathcal{E}_s)}$
$\mathfrak{X}_0 \leftarrow \mathfrak{T}$
for $i \in \{1, \ldots, n\}$ do
    $T \leftarrow$ embedding_tree$\left(\hat{\sigma}_i, \{\sigma^{(E')} : E' \in \mathcal{E}\}, A\right)$ // construct the embedding tree based on Definition 5 and Definition 6
    $\mathfrak{X}_i \leftarrow$ density_matrix_alg$(\mathfrak{X}_{i-1}, T, \chi)$ // Algorithm 4
end for
return the output tensor network $\mathfrak{X}_n$
```



E_1
E_2
E_3
E_4
$\tau^{(\mathcal{E}_s)} = (E_1, E_2, E_3, E_4)$
$\hat{\sigma}_1 = (E_2, E_1, E_4, E_3)$
$\sigma^{(\mathcal{E}_s)} = (E_2, E_4, E_3, E_1)$
E_1
E_2
E_3
E_4
E_2
E_1
E_4
E_3
E_2
E_4
E_3
E_1

Figure 14: Illustration of Algorithm 6 with the swap batch size being $r = 2$. The left diagram denotes the input tensor network G_s as well as the set of edge subsets $\mathcal{E}_s = \{E_1, E_2, E_3, E_4\}$. The second leftmost diagram shows the ordering $\tau^{(\mathcal{E}_s)}$ that is generated based on analyzing the graph structure of G_s. Since 4 swaps are needed to change $\tau^{(\mathcal{E}_s)}$ to $\sigma^{(\mathcal{E}_s)}$, two density matrix algorithms are performed, one with the embedding tree generated by the ordering $\hat{\sigma}_1$ and the other with the embedding tree generated by the ordering $\sigma^{(\mathcal{E}_s)}$.

of r, the behavior of the algorithm closely resembles that of the pure density matrix algorithm. On the other hand, a smaller value of r generally leads to improved computational efficiency while sacrificing some approximation accuracy.

In summary, the hybrid approach within the proposed `partitioned_contract` algorithm effectively balances accuracy and computational cost for specific problem instances.

8 Experimental results

In this section, we present the results of a series of experiments to evaluate the performance of the proposed approach. All experiments were executed on an Intel Core i7 2.9 GHz Quad-Core machine.

In Section 8.1, we introduce our implementations, the tensor networks and models tested in our experiments. In Section 8.2, we conducted a detailed comparison between the proposed density matrix algorithm for tree approximation and the canonicalization-based algorithm. Across all experiments, the density matrix algorithm consistently demonstrated either lower or the same asymptotic cost. In particular, we achieved a remarkable 4.9X speedup with the density matrix algorithm compared to the canonicalization-based algorithm when approximating an MPO-MPS multiplication into an MPS.

In Section 8.3, we justify the `partitioned_contract` algorithm presented in Algorithm 2. We justify our embedding tree selection algorithm and explore the impact of the environment size on accuracy and efficiency across multiple problems. Additionally, we conduct a comprehensive comparison between the MPS and the comb ansatz. Furthermore, we evaluate `partitioned_contract`, the CATN algorithm [61][1], SweepContractor [14][2], and hyperoptimized approximate contraction [26][3], in contracting tensor networks defined on lattices and random regular graphs as well as tensor networks from random quantum circuit simulation. We demonstrate a 9.2X speed-up while maintaining the same level of accuracy when contracting tensor networks defined on 3D lattices using the Ising model.

8.1 Implementations, tested tensor networks, and the evaluation

The proposed algorithms in the paper are being developed at `https://github.com/ITensor/ITensorNetworks.jl`. ITensorNetworks.jl is a publicly available Julia [6] package built for manipulating tensor networks of arbitrary geometry, and is built on top of ITensors.jl [24]. The library also provides an interface to OMEinsumContractionOrders.jl[4], which implements multiple heuristics introduced in [27, 44] to generate efficient contraction paths for exact tensor network contractions. For all the results presented in this work, we use the Simulated Annealing bipartition + Greedy algorithm (SABipartite) [44] to generate contraction paths for exact tensor network contractions.

Our experiments consider three types of tensor networks: those generated from random tensors, the classical Ising model, and random quantum circuits.

For those generated from random tensors, each element within the tensors is an i.i.d. variable uniformly distributed in the range of $[\alpha, 1]$, where $\alpha \in [-1, 0]$. These particular tensor networks have been utilized in previous research [26] as benchmarks for evaluating contraction algorithms. For specific structures like random regular graphs and 3D lattices, the approximate contraction of the tensor network becomes more challenging as α approaches the value of -1 [12].

For a tensor network defined on a graph $G = (V, E)$ using the ferromagnetic Ising model, the contraction output, denoted as Z and referred to as the partition function, can be expressed as follows,

$$Z = \sum_{\sigma_i, \sigma_j \in \{-1, 1\}} \prod_{(i,j) \in E} \exp(\beta \sigma_i \sigma_j).$$

In the tensor network, the tensor $\mathfrak{T}^{(v)}$ defined at each $v \in V$ has an elementwise expression of

$$t^{(v)}_{E(v)} = \sum_i \prod_{e \in E(v)} W_{i,e},$$

where

$$W = \frac{1}{\sqrt{2}} \begin{bmatrix} \sqrt{\cosh(\beta)} + \sqrt{\sinh(\beta)} & \sqrt{\cosh(\beta)} - \sqrt{\sinh(\beta)} \\ \sqrt{\cosh(\beta)} - \sqrt{\sinh(\beta)} & \sqrt{\cosh(\beta)} + \sqrt{\sinh(\beta)} \end{bmatrix}$$

and β is an input parameter to the model. We show the relation between the relative error of $\ln Z$ and the running time of `partitioned_contract` and the baselines in Section 8.3. The quantity $\ln Z$ is an important measure that is proportional to the free energy of the system.

We also explore the simulation of a 2D random quantum circuit, denoted as $|\psi\rangle$, as detailed in [62, 10, 4]. The initial quantum state $|0, \ldots, 0\rangle$ is organized into a 6×6 grid, and we apply six layers of random circuit gates to this initial state. Each layer contains random one-qubit rotations

[1] We use the CATN implementation at `https://github.com/panzhang83/catn`.

[2] We use the SweepContractor implementation at `https://github.com/chubbc/SweepContractor.jl`.

[3] We use the hyperoptimized approximate contraction implementation at quimb [25] (`https://github.com/jcmgray/quimb`).

[4] The library is implemented at `https://github.com/TensorBFS/OMEinsumContractionOrders.jl`.

on top of each qubit and a sequence of two-qubit controlled-X gates, and the two-qubit gates are structured in a brick-layer pattern. The specific configuration of each 2-qubit layer is detailed in Fig. 15. In Section 8.3, we approximately contract the tensor network $\langle\psi|\psi\rangle$, and measure the absolute error of the quantity.

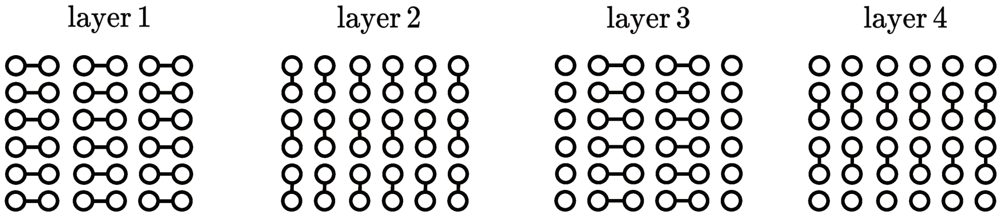
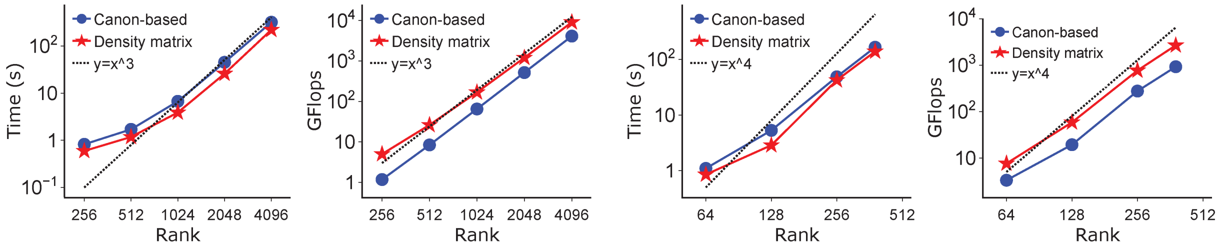


Figure 15: The arrangement of the initial four layers of quantum gates. Each circle denotes a qubit, and each line denotes a two-qubit gate. The subsequent layers follow the same pattern.

To evaluate and compare the efficiencies of various algorithms, we measure both the execution time and the required number of GFlops (giga floating-point operations). The GFlops calculations encompass tensor contractions, QR factorization, and low-rank approximations, as outlined in the model detailed in Section 2.2. It is worth noting that in our reported results, the execution time excludes the graph analysis part, which involves graph embedding and computing the contraction sequence of given tensor networks. This part remains independent of the tensor network ranks and is negligible when the ranks are high.

8.2 Comparison between the density matrix algorithm and the canonicalization-based algorithm

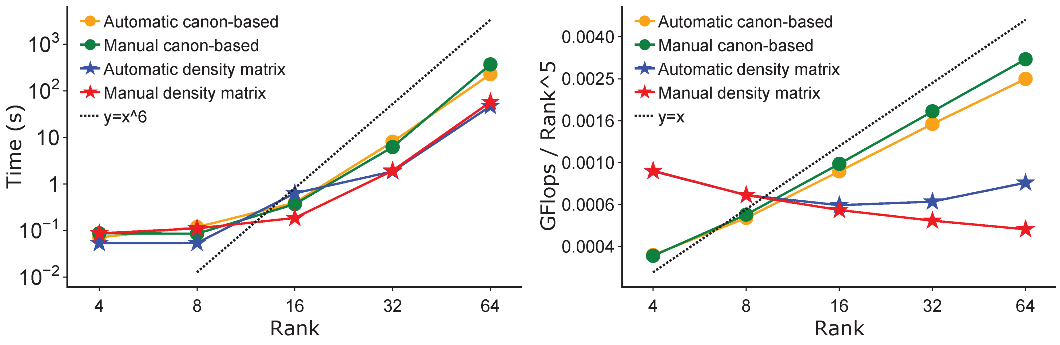


(a) MPS, $\chi = 100$ (b) MPS, $\chi = 100$ (c) BBT, $\chi = 50$ (d) BBT, $\chi = 50$

Figure 16: Performance comparison between the density matrix algorithm and the canonicalization-based algorithm in truncating a binary tree tensor network. In (a)(b), the input networks are MPSs with different ranks. In (c)(d), the inputs are balanced binary tree (BBT) tensor networks with different ranks. The number of uncontracted modes is fixed to be 30 for all input tensor networks.

We conduct an efficiency comparison between the density matrix algorithm and the canonicalization based algorithm to approximate an input tensor network into a binary tree tensor network. Our evaluation covers scenarios where the input tensor network structure matches the output structure, as well as cases where the input network has a general non-tree structure. In both instances, the density matrix algorithm has equal or superior asymptotic cost compared to the canonicalization-based algorithm.

In Fig. 16, we conduct a performance comparison of truncating both MPSs and balanced binary tree tensor networks. Let R denote the rank of the input MPS and the balanced binary tree, the analytical asymptotic cost for truncating an MPS is $\Theta(R^3)$, whereas for truncating a balanced binary tree is $\Theta(R^4)$. As depicted in the results, the scaling behavior of both algorithms aligns with the analytical predictions. Despite the density matrix algorithm incurring a constant overhead in terms of GFlops, we observe that it exhibits slightly faster performance. This advantage can be attributed to the fact that the majority of the density matrix algorithm's execution time is spent on tensor contractions, which are practically faster compared to matrix factorizations, even though both operations have a similar computational complexity. This property makes the density matrix algorithm more favorable on GPUs due to their ability to efficiently run tensor contractions in parallel.

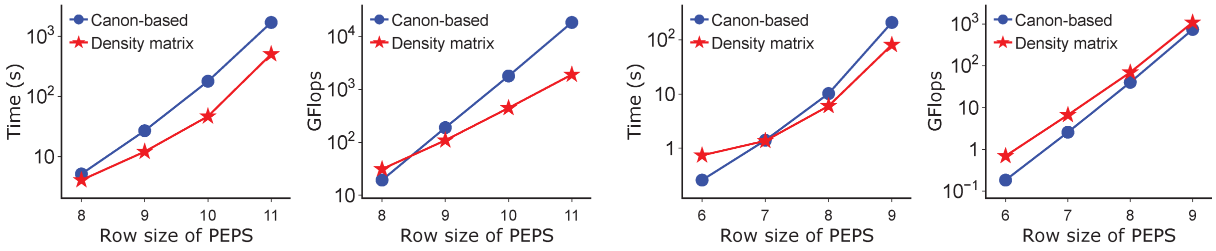


(a)

(b)

Figure 17: Performance comparison between the density matrix algorithm and the canonicalization-based algorithm in approximating the MPO-MPS multiplication into a low-rank MPS. The order of the input MPS and MPO is fixed to be 40. Both the input MPS and MPO have the same rank χ, and the output MPS rank is also upper-bounded by χ. The manual algorithms are those reviewed in Section 3.2.2 that use a manually-determined memoization strategy.

In Fig. 17, we compare the performance of truncating the multiplication of an MPS and an MPO. Our experiments encompass both the canonicalization-based algorithm and the density matrix algorithm, alongside the reference algorithms reviewed in Section 3.2.2. In the reference algorithms, the memoization strategy is determined and implemented manually rather than automatically. For the canonicalization-based algorithm, the asymptotic cost is $\Theta(R^6)$, where R represents the input rank of both MPS and MPO. On the contrary, the density matrix algorithm exhibits an asymptotic cost of $\Theta(R^5)$. As shown in Fig. 17b, the scaling behavior of both algorithms aligns with our analysis. The density matrix algorithm outperforms the canonicalization-based algorithm and has a remarkable 4.9X execution time speedup when the input rank is 64. Furthermore, our algorithm, equipped with the automatically-chosen memoization strategy, performs similarly to the reference algorithms, thereby confirming the efficacy of our approach.



(a) MPS, $\chi = 250$

(b) MPS, $\chi = 250$

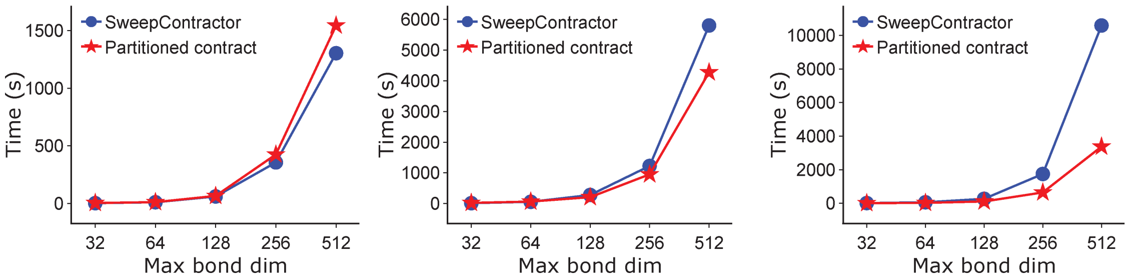
(c) Comb, $\chi = 50$

(d) Comb, $\chi = 50$

Figure 18: Performance comparison between the density matrix algorithm and the canonicalization-based algorithm in approximating a PEPS with rank 2 into a binary tree tensor network. The column size of the PEPS equals the row size. In (a)(b), the embedding tree structure is an MPS, and the MPS site ordering is chosen based on the row- or column-wise traversal of the 2D coordinates of the PEPS tensors. In (c)(d), the embedding tree structure is a comb, and each edge subset in the comb is a row of the PEPS.

In Fig. 18, we compare the performance of approximating PEPS on square grids into MPS and comb binary tree structures. Both structures are defined in Section 4.3. As can be seen, the density matrix algorithm outperforms the canonicalization-based algorithm when the number of rows and columns of the PEPS is large. The inefficiency of the canonicalization-based algorithm is due to the fact that there exists some partition embedded in one vertex of the MPS/comb whose contraction yields a large-sized tensor. The density matrix algorithm avoids the explicit formation of such tensors and thus is more efficient. In later sections we will discuss the relative merits of using MPS or comb tree structures for intermediate networks.

8.3 Benchmark of the partitioned_contract algorithm



(a) 2D lattice, rank=16 (b) 3D lattice, rank=4 (c) Random regular graph, rank=8

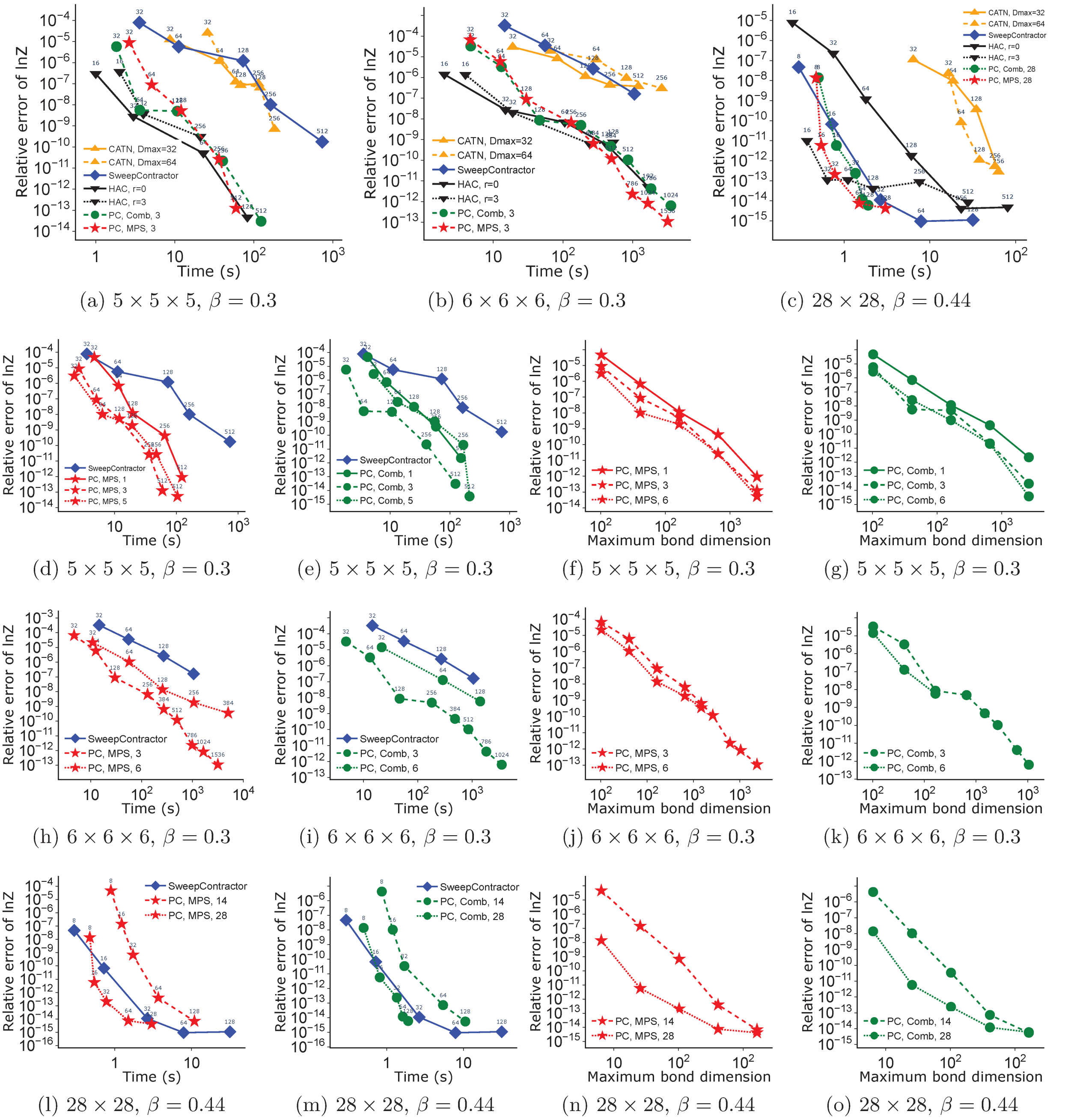
Figure 19: Performance comparison between partitioned_contract and SweepContractor under the same contraction path. The swap batch size is set to 1 for all experiments in partitioned_contract. In (a), the row and the column size of the 2D lattice is 8. In (b), each mode in the 3D lattice has a size of 5. In (c), the random regular graph has 100 vertices, each with a degree of 3.

Impact of the embedding tree on contraction efficiency In this section we present results that justify our embedding tree selection algorithm in partitioned_contract. In Fig. 19, we compare partitioned_contract with SweepContractor for tensor networks defined on three different structures using tensor networks with random tensor elements that was introduced in Section 8.1. For all the experiments, partitioned_contract uses the MPS ansatz, and both algorithms use the same maximally-unbalanced contraction tree where each partition only contains one tensor. Consequently, the only distinction between the two algorithms lies in the usage of different embedding trees for each contraction between an MPS and a tensor.

As can be seen, both algorithms have a similar performance when contracting a 2D grid, while partitioned_contract significantly outperforms SweepContractor for the other two graph structures. This difference in performance arises from the fact that different embedding trees result in varying numbers of adjacent swaps of MPS modes. For tensor networks defined on 3D lattice and random regular graphs, our algorithm generates embedding trees that lead to substantially fewer adjacent swaps. Note that the partitioned_contract algorithm achieves higher approximation accuracy on these two graphs, as fewer swaps imply reduced truncations, contributing to improved accuracy in the results.

Impact of the environment size on contraction accuracy and efficiency We explore the impact of the environment size on the accuracy and efficiency of contracting tensor networks defined on 2D and 3D lattices as well as random regular graphs, and the results are shown in Fig. 20, Fig. 21, and Fig. 22.

In both lattices and random regular graphs, we employ the maximally-unbalanced partial contraction path for the contraction process. This path initiates from one partition and progressively combines the previously-contracted section with a new partition following a linear sequence of the partitions. For 3D lattices, each partition represents either a portion of a fiber or an entire fiber of the lattice. The contraction path is determined through a row- or column-wise traversal of the 2D array resulting from partitioning the 3D lattice into fibers. Regarding random regular graphs, we draw inspiration from [34] to construct the contraction path using a linear ordering of vertices. We achieve this by first employing recursive bisection to generate the linear ordering of all the



(a) $5 \times 5 \times 5$, $\beta = 0.3$

(b) $6 \times 6 \times 6$, $\beta = 0.3$

(c) 28×28, $\beta = 0.44$

(d) $5 \times 5 \times 5$, $\beta = 0.3$

(e) $5 \times 5 \times 5$, $\beta = 0.3$

(f) $5 \times 5 \times 5$, $\beta = 0.3$

(g) $5 \times 5 \times 5$, $\beta = 0.3$

(h) $6 \times 6 \times 6$, $\beta = 0.3$

(i) $6 \times 6 \times 6$, $\beta = 0.3$

(j) $6 \times 6 \times 6$, $\beta = 0.3$

(k) $6 \times 6 \times 6$, $\beta = 0.3$

(l) 28×28, $\beta = 0.44$

(m) 28×28, $\beta = 0.44$

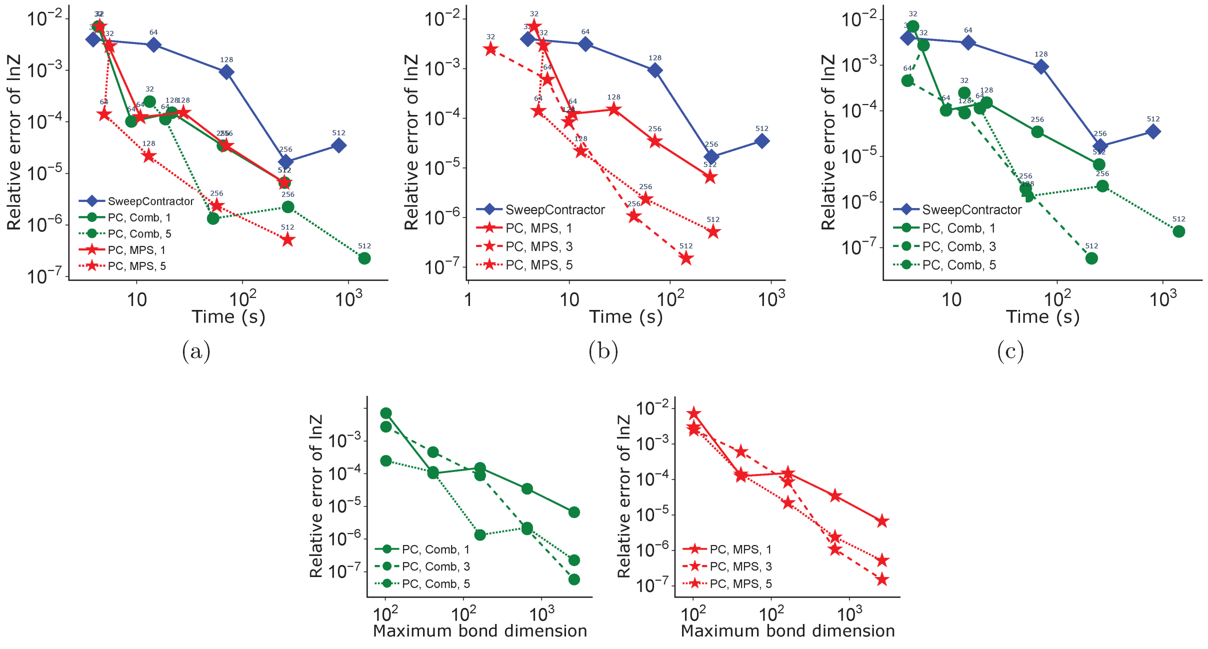
(n) 28×28, $\beta = 0.44$

(o) 28×28, $\beta = 0.44$

Figure 20: Performance comparison between `partitioned_contract`, SweepContractor [14], CATN [61], and hyperoptimized approximate contraction (HAC) [26] in contracting lattices based on the Ising model. The swap batch size is fixed to be 32 for all experiments. In the legends, "PC" denotes `partitioned_contract`, MPS/Comb denotes the embedding tree ansatz, and the values (1, 3, 5) denote the size of each partition. In (a)-(f), the number shown on top of each point is the maximum bond size χ. In CATN, "Dmax" is an additional input parameter of the algorithm that controls the size of the MPS uncontracted modes. In HAC, r denotes the distance of the spanning trees used in canonicalization.

vertices. Then, we sequentially include a partition consisting of a specified number of tensors into the contraction path, following the order of traversal in the vertex ordering.

When contracting tensor networks defiend on lattices, the results presented in Figs. 20 and 21 reveal that employing a larger parition size (3 or 5 for the $5 \times 5 \times 5$ 3D grid and 14 for the 28×28 2D grid) leads to both faster and more accurate contractions when compared to the base condition where each partition contains only one tensor. The improved efficiency arises from using larger partitions, which reduces the number of executions of the density matrix algorithm,



(a) (b) (c)

(d) (e)

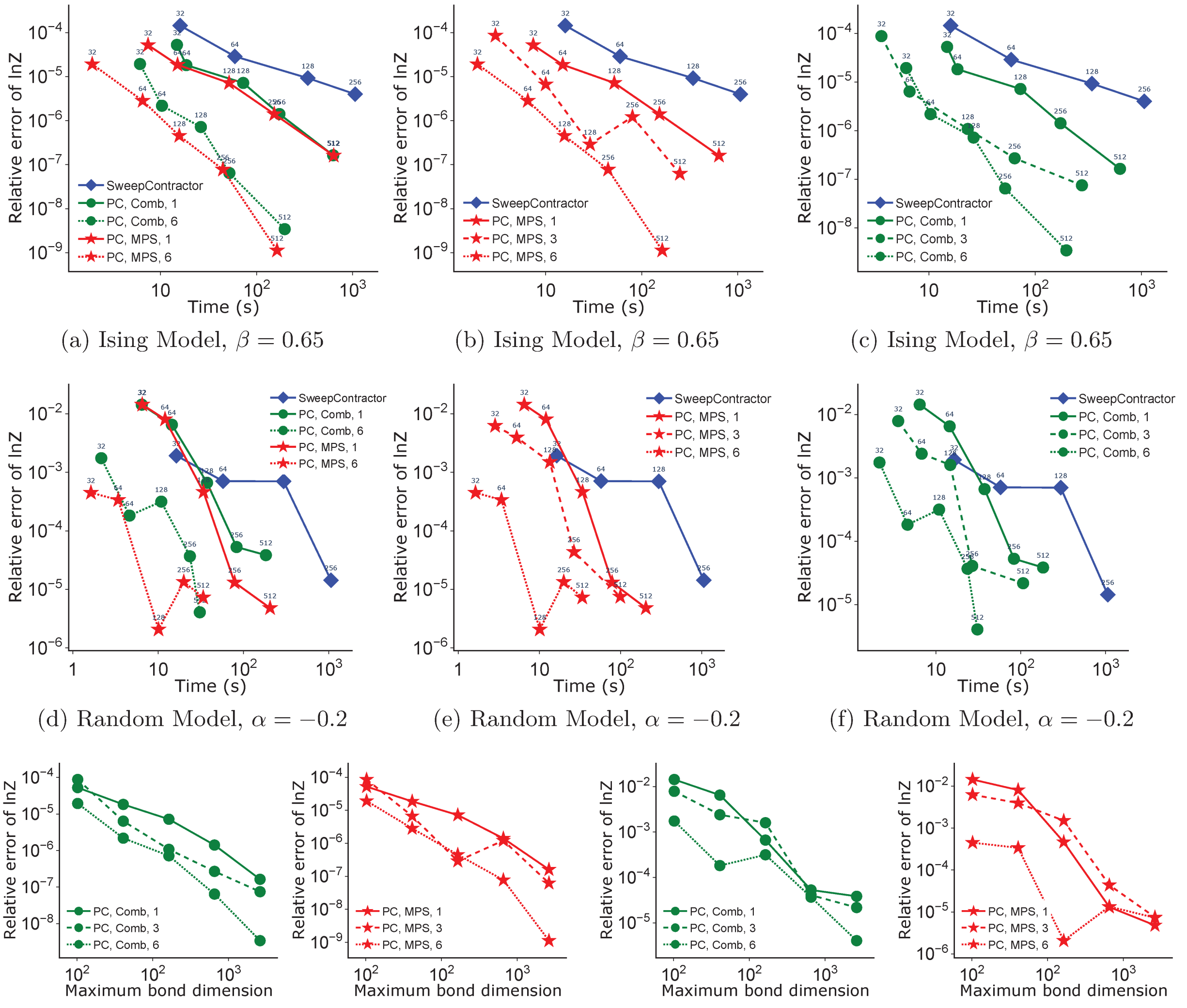
Figure 21: Performance comparison between partitioned_contract and SweepContractor [14] in contracting $5 \times 5 \times 5$ random tensor networks defined on 3D lattices with $\alpha = -0.4$. The swap batch size is fixed to be 32 for all experiments. In the legends, "PC" denotes partitioned_contract, MPS/Comb denotes the embedding tree ansatz, and the values (1, 3, 5) denote the size of each partition. In (a)-(c), the number shown on top of each point is the maximum bond size χ.

offsetting any overhead from using larger environments. Regarding accuracy, we can see from Figs. 20f, 20g, 20j, 20k, 20n and 20o that under the same maximum contracted bond size, utilizing a larger partition size generally yields lower relative errors for both MPS and comb structures. This observation validates the efficacy of the environment in enhancing accuracy. See the next section for a comparison of the MPS and comb structures.

Regarding random regular graphs, the results displayed in Figs. 22b, 22c and 22e to 22j indicate that using a partition size of 6 results in the best combination of efficiency and accuracy. To summarize, employing a larger partition leads to a larger environment size, generally reducing the contraction error under the same rank. However, when it comes to efficiency, the optimal partition size depends on the specific problem. Factors such as the number of executions of the density matrix algorithm to be performed and the cost of forming the density matrix under different environment sizes need to be taken into consideration to determine the most suitable partition size.

Comparison between the MPS and the comb structure In this section we discuss the relative merits of using the MPS and comb tree ansatzes. When contracting lattices, the results in Figs. 20a to 20c and 21a demonstrate that both MPS and comb structures exhibit similar performance when the maximum bond size is small. However, as the maximum bond size increases, using the comb ansatz becomes slower in comparison to MPS. On the other hand, when contracting random regular graphs, the results in Figs. 22a and 22d reveal that both structures display similar levels of accuracy and efficiency. In summary, both MPS and comb binary tree structures perform similarly in terms of accuracy. However, the efficiency of the comb structure may lag behind MPS, particularly when dealing with a large rank. This disparity in performance is attributed to the presence of large tensors with size χ^3 in the comb ansatz.

Comparison among partitioned_contract and the baselines Here we compare our proposed partitioned_contract algorithm, along with the CATN algorithm [61], SweepContrac-



(a) Ising Model, $\beta = 0.65$

(b) Ising Model, $\beta = 0.65$

(c) Ising Model, $\beta = 0.65$

(d) Random Model, $\alpha = -0.2$

(e) Random Model, $\alpha = -0.2$

(f) Random Model, $\alpha = -0.2$

(g) Ising, $\beta = 0.3$

(h) Ising, $\beta = 0.3$

(i) Random, $\alpha = -0.4$

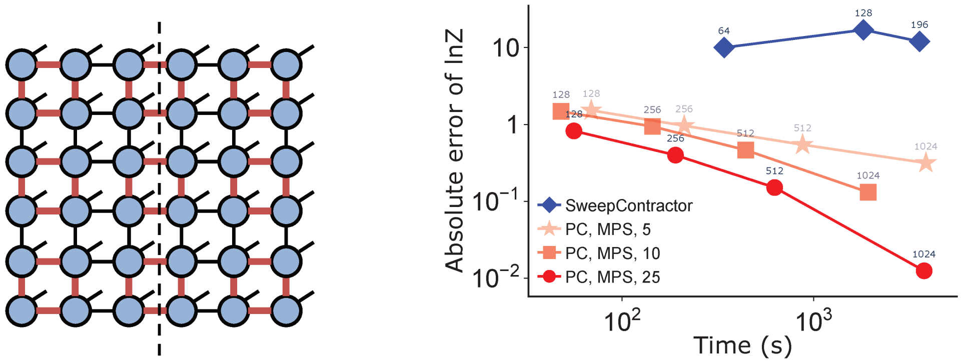
(j) Random, $\alpha = -0.4$

Figure 22: Performance comparison between `partitioned_contract` and SweepContractor [14] in contracting random regular graphs with degree 3 and 220 vertices. The swap batch size is fixed to be 32 for all experiments. In the legends, "PC" denotes `partitioned_contract`, MPS/Comb denotes the embedding tree ansatz, and the number (1, 3, 6) denotes the size of each partition. In (a)-(f), the number shown on top of each point is the maximum contracted bond size χ.

tor [14], and hyper-optimizated approximate contraction (HAC) [26], on contracting tensor networks defined on lattices and random regular graphs.

As demonstrated in Figs. 20a to 20c, 21a and 22a, our algorithm consistently outperforms CATN in terms of efficiency across all levels of relative error. When compared to SweepContractor, our algorithm shows superior efficiency for 3D lattices and random graphs, while both methods perform similarly on 2D grids. This is expected, as SweepContractor is specifically optimized for planar graphs. Notably, when contracting a 3D lattice tensor network based on the Ising model, `partitioned_contract` achieves a 9.2X speed-up compared to both CATN and SweepContractor when reaching a relative error of less than 10^{-9}. Similarly, when contracting a tensor network with a random regular graph structure based on the Ising model, `partitioned_contract` achieves a 52.4X speed-up compared to SweepContractor when achieving a relative error of less than 10^{-5}.

As shown from the results in Figs. 20a to 20c, HAC is a more efficient baseline compared to CATN and SweepContractor in contracting lattices based on the Ising model. On both $5 \times 5 \times 5$ and 28×28 grids, both HAC and `partitioned_contract` can reach pretty high accuracy with similar efficiency. On the $6 \times 6 \times 6$ grid, the results indicate that `partitioned_contract` can reach



(a) Tensor diagram of $|\psi\rangle$

(b) Benchmark results

Figure 23: An illustration and benchmark results for approximately contracting a 2D random quantum circuit tensor network with 6 layers of gates. (a) The tensor diagram visualization of the PEPS $|\psi\rangle$ that is the output of the random quantum circuit simulation with 6 layers of gates. Each black edge denotes a mode with size 2, and each thick red edge denotes a mode with size 4. The dashed line denotes a graph cut with the cut mode size being 2^{12}. (b) Performance comparison between `partitioned_contract` and SweepContractor [14] in contracting the tensor network that represents $\langle\psi|\psi\rangle$. The swap batch size is fixed to be 8 for all experiments. In the legends, "PC" denotes `partitioned_contract`, and the number (5, 10, 25) denotes the size of each partition. The number shown on top of each point is the maximum contracted bond size χ.

higher accuracy and is more efficient when the relative error is smaller than 10^{-10}. This can be attributed to HAC generating high-order intermediate tensors, which slows it down and exceeds memory capacity when tested with a rank of 256. In contrast, `partitioned_contract` performs better due to its use of the MPS ansatz, making it more memory-efficient and effective at higher ranks.

In terms of memory usage, let s denote the size of dimensions in the input graph, $\chi \geq s$ the maximum bond dimension, and $D_{\max} \geq s$ an additional parameter controlling the size of uncontracted MPS modes in CATN. The largest intermediate tensor generated in `partitioned_contract` has a size of $s\chi^2$ for the MPS ansatz and χ^3 for the comb ansatz. For CATN, the largest tensor size is $D_{\max}r^2$. SweedContractor and `partitioned_contract` with the MPS ansatz share a similar memory footprint of $s\chi^2$. For HAC, the largest tensor size can be χ^d where d is the maximum order of the intermediate tensors. To summarize, `partitioned_contract` with the MPS ansatz has memory usage that is better than other algorithms as a function of bond dimension.

We also benchmark our algorithm against SweepContractor [14] in contracting the random quantum circuit tensor network outlined in Section 8.1. The output state of the circuit can be represented as a PEPS that is visualized in Fig. 23a. When accurately transforming $|\psi\rangle$ into an MPS, the maximum bond size will be at least $2^{12} = 4096$, as inferred from the graph cut in Fig. 23a. As illustrated in Fig. 23b, our method manages to approximate the tensor network contraction with a bond size of $2^{10} = 1024$, achieving an absolute error of 10^{-2}. In contrast, SweepContractor fails to converge within the same amount of time. These significant speed improvements clearly demonstrate the efficiency of our approach over the compared algorithms.

9 Conclusion

In this work we introduced an efficient algorithm called `partitioned_contract` to contract tensor networks with arbitrary structures. The algorithm has the flexibility to incorporate a large portion of the environment when performing low-rank approximations, and includes a cost-efficient density matrix algorithm for approximating a general tensor network into a tree structure whose computational cost is asymptotically upper-bounded by that of the standard algorithm that uses canonicalization. Experimental results indicate that the proposed technique outperforms previously proposed approximate tensor network contraction algorithms for multiple problems in terms of both accuracy and efficiency.

We emphasize several potential future directions that need further exploration and investigation. Firstly, the partitioned_contract algorithm assumes that both a partitioning of the input tensor network and a contraction path over these partitions are provided. There remains an opportunity to explore efficient methods for finding optimal partitionings and contraction paths for `partitioned_contract`, which could further improve its performance. Additionally, there is scope for investigating how the canonicalization-based algorithm for tree approximation can be accelerated. One possibility is to leverage tensor network sketching techniques [48, 47, 63, 1] to speed up randomized SVD [30], which may enhance the efficiency of the tree approximation process. Another is to use variational or fitting algorithms for approximately contracting network partitions, which can have better scaling than density matrix and canonicalization-based algorithms at the expense of potentially requiring multiple iterations to converge [79, 72]. Finally, integrating the proposed algorithm into automatic differentiation libraries [50, 25, 43] could be highly beneficial. This integration would enable the algorithm to be used in gradient-based optimization algorithms for tensor networks, thereby expanding its utility in various optimization tasks.

Acknowledgments

M.F. and M.S. are grateful for ongoing support through the Flatiron Institute, a division of the Simons Foundation. L.M. acknowledges support through the Flatiron Institute internship program through which this work was initiated. The work of L.M. and E.S. was also supported by the US National Science Foundation (NSF) grant #1942995 and the Department of Energy (DOE) Advanced Scientific Computing Research program via award DE-SC0023483.

References

[1] Thomas D. Ahle, Michael Kapralov, Jakob B. T. Knudsen, Rasmus Pagh, Ameya Velingker, David P. Woodruff, and Amir Zandieh. *Oblivious Sketching of High-Degree Polynomial Kernels*, page 141–160. Society for Industrial and Applied Mathematics, January 2020. ISBN 9781611975994. DOI: 10.1137/1.9781611975994.9. URL `https://doi.org/10.1137/1.9781611975994.9`.

[2] R. Alkabetz and I. Arad. Tensor networks contraction and the belief propagation algorithm. *Physical Review Research*, 3(2), April 2021. ISSN 2643-1564. DOI: 10.1103/physrevresearch.3.023073. URL `https://doi.org/10.1103/PhysRevResearch.3.023073`.

[3] Sanjeev Arora, Satish Rao, and Umesh Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of the ACM*, 56(2):1–37, April 2009. ISSN 1557-735X. DOI: 10.1145/1502793.1502794. URL `https://doi.org/10.1145/1502793.1502794`.

[4] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, October 2019. ISSN 1476-4687. DOI: 10.1038/s41586-019-1666-5. URL `https://doi.org/10.1038/s41586-019-1666-5`.

[5] Daniel Bauernfeind, Manuel Zingl, Robert Triebl, Markus Aichhorn, and Hans Gerd Evertz. Fork tensor-product states: Efficient multiorbital real-time DMFT solver. *Physical Review X*, 7(3), July 2017. ISSN 2160-3308. DOI: 10.1103/physrevx.7.031013. URL `https://doi.org/10.1103/PhysRevX.7.031013`.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, January 2017. ISSN 1095-7200. DOI: 10.1137/141000671. URL `https://doi.org/10.1137/141000671`.

[7] S.L. Bezrukov, J.D. Chavez, L.H. Harper, M. Röttger, and U.-P. Schroeder. The congestion of n-cube layout on a rectangular grid. *Discrete Mathematics*, 213(1–3):13–19, February 2000.

ISSN 0012-365X. DOI: 10.1016/s0012-365x(99)00162-4. URL https://doi.org/10.1016/S0012-365X(99)00162-4.

[8] Jacob D. Biamonte, Jason Morton, and Jacob Turner. Tensor network contractions for #SAT. *Journal of Statistical Physics*, 160(5):1389–1404, June 2015. ISSN 1572-9613. DOI: 10.1007/s10955-015-1276-z. URL https://doi.org/10.1007/s10955-015-1276-z.

[9] Dan Bienstock. On embedding graphs in trees. *Journal of Combinatorial Theory, Series B*, 49(1):103–136, June 1990. ISSN 0095-8956. DOI: 10.1016/0095-8956(90)90066-9. URL https://doi.org/10.1016/0095-8956(90)90066-9.

[10] Sergio Boixo, Sergei V. Isakov, Vadim N. Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J. Bremner, John M. Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595–600, April 2018. ISSN 1745-2481. DOI: 10.1038/s41567-018-0124-x. URL https://doi.org/10.1038/s41567-018-0124-x.

[11] Moses Charikar, Mohammad Taghi Hajiaghayi, Howard Karloff, and Satish Rao. ℓ_2^2 spreading metrics for vertex ordering problems. *Algorithmica*, 56:577–604, 2010.

[12] Jielun Chen, Jiaqing Jiang, Dominik Hangleiter, and Norbert Schuch. Sign problem in tensor network contraction. 2024. DOI: 10.48550/arXiv.2404.19023. URL https://doi.org/10.48550/arXiv.2404.19023.

[13] Natalia Chepiga and Steven R. White. Comb tensor networks. *Physical Review B*, 99(23), June 2019. ISSN 2469-9969. DOI: 10.1103/physrevb.99.235426. URL https://doi.org/10.1103/PhysRevB.99.235426.

[14] Christopher T Chubb. General tensor network decoding of 2D pauli codes. 2021. DOI: 10.48550/arXiv.2101.04125. URL https://doi.org/10.48550/arXiv.2101.04125.

[15] Andrzej Cichocki. Tensor networks for big data analytics and large-scale optimization problems. 2014. DOI: 10.48550/ARXIV.1407.3124. URL https://doi.org/10.48550/ARXIV.1407.3124.

[16] Carsten Damm, Markus Holzer, and Pierre McKenzie. The complexity of tensor calculus. *computational complexity*, 11(1):54–89, 2002. DOI: 10.1109/ccc.2000.856737. URL https://doi.org/10.1109/CCC.2000.856737.

[17] Nikhil R. Devanur, Subhash A. Khot, Rishi Saket, and Nisheeth K. Vishnoi. Integrality gaps for sparsest cut and minimum linear arrangement problems. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of Computing*, STOC06, page 537–546. ACM, May 2006. DOI: 10.1145/1132516.1132594. URL https://doi.org/10.1145/1132516.1132594.

[18] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16. ACM, August 2016. DOI: 10.1145/2939672.2939862. URL https://doi.org/10.1145/2939672.2939862.

[19] Josep Díaz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356, September 2002. ISSN 1557-7341. DOI: 10.1145/568522.568523. URL https://doi.org/10.1145/568522.568523.

[20] Guy Even, Joseph Seffi Naor, Satish Rao, and Baruch Schieber. Divide-and-conquer approximation algorithms via spreading metrics. *Journal of the ACM*, 47(4):585–616, July 2000. ISSN 1557-735X. DOI: 10.1145/347476.347478. URL https://doi.org/10.1145/347476.347478.

[21] Uriel Feige and James R. Lee. An improved approximation ratio for the minimum linear arrangement problem. *Information Processing Letters*, 101(1):26–29, January 2007. ISSN 0020-0190. DOI: 10.1016/j.ipl.2006.07.009. URL https://doi.org/10.1016/j.ipl.2006.07.009.

[22] Timo Felser, Simone Notarnicola, and Simone Montangero. Efficient tensor network ansatz for high-dimensional quantum many-body problems. *Physical Review Letters*, 126(17), April 2021. ISSN 1079-7114. DOI: 10.1103/physrevlett.126.170603. URL https://doi.org/10.1103/PhysRevLett.126.170603.

[23] Giovanni Ferrari, Giuseppe Magnifico, and Simone Montangero. Adaptive-weighted tree tensor networks for disordered quantum many-body systems. *Physical Review B*, 105(21), June 2022. ISSN 2469-9969. DOI: 10.1103/physrevb.105.214201. URL https://doi.org/10.1103/PhysRevB.105.214201.

[24] Matthew Fishman, Steven White, and Edwin Stoudenmire. The ITensor software library for tensor network calculations. *SciPost Physics Codebases*, August 2022. DOI: 10.21468/scipost-physcodeb.4. URL https://doi.org/10.21468/SciPostPhysCodeb.4.

[25] Johnnie Gray. quimb: A Python package for quantum information and many-body calculations. *Journal of Open Source Software*, 3(29):819, September 2018. ISSN 2475-9066. DOI: 10.21105/joss.00819. URL https://doi.org/10.21105/joss.00819.

[26] Johnnie Gray and Garnet Kin-Lic Chan. Hyperoptimized approximate contraction of tensor networks with arbitrary geometry. *Physical Review X*, 14(1), January 2024. ISSN 2160-3308. DOI: 10.1103/physrevx.14.011009. URL https://doi.org/10.1103/PhysRevX.14.011009.

[27] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, March 2021. ISSN 2521-327X. DOI: 10.22331/q-2021-03-15-410. URL https://doi.org/10.22331/q-2021-03-15-410.

[28] Ming Gu and Stanley C. Eisenstat. Efficient algorithms for computing a strong rank-revealing QR factorization. *SIAM Journal on Scientific Computing*, 17(4):848–869, July 1996. ISSN 1095-7197. DOI: 10.1137/0917055. URL https://doi.org/10.1137/0917055.

[29] Chu Guo, Yong Liu, Min Xiong, Shichuan Xue, Xiang Fu, Anqi Huang, Xiaogang Qiang, Ping Xu, Junhua Liu, Shenggen Zheng, He-Liang Huang, Mingtang Deng, Dario Poletti, Wan-Su Bao, and Junjie Wu. General-purpose quantum circuit simulator with projected entangled-pair states and the quantum supremacy frontier. *Physical Review Letters*, 123 (19), November 2019. ISSN 1079-7114. DOI: 10.1103/physrevlett.123.190501. URL https://doi.org/10.1103/PhysRevLett.123.190501.

[30] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, January 2011. ISSN 1095-7200. DOI: 10.1137/090771806. URL https://doi.org/10.1137/090771806.

[31] M.D. Hansen. Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems. In *30th Annual Symposium on Foundations of Computer Science*, page 604–609. IEEE, 1989. DOI: 10.1109/sfcs.1989.63542. URL https://doi.org/10.1109/SFCS.1989.63542.

[32] L. H. Harper. Optimal assignments of numbers to vertices. *Journal of the Society for Industrial and Applied Mathematics*, 12(1):131–135, March 1964. ISSN 2168-3484. DOI: 10.1137/0112012. URL https://doi.org/10.1137/0112012.

[33] Stephen W. Hruska. On tree congestion of graphs. *Discrete Mathematics*, 308(10):1801–1809, May 2008. ISSN 0012-365X. DOI: 10.1016/j.disc.2007.04.030. URL https://doi.org/10.1016/j.disc.2007.04.030.

[34] Cameron Ibrahim, Danylo Lykov, Zichang He, Yuri Alexeev, and Ilya Safro. Constructing optimal contraction trees for tensor network quantum circuit simulation. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, page 1–8. IEEE, September 2022. DOI: 10.1109/hpec55821.2022.9926353. URL https://doi.org/10.1109/HPEC55821.2022.9926353.

[35] Adam Jermyn. Automatic contraction of unstructured tensor networks. *SciPost Physics*, 8 (1), January 2020. ISSN 2542-4653. DOI: 10.21468/scipostphys.8.1.005. URL https://doi.org/10.21468/SciPostPhys.8.1.005.

[36] George Karypis. METIS: Unstructured graph partitioning and sparse matrix ordering system. *Technical report*, 1997.

[37] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, August 2009. ISSN 1095-7200. DOI: 10.1137/07070111x. URL https://doi.org/10.1137/07070111X.

[38] Stefanos Kourtis, Claudio Chamon, Eduardo Mucciolo, and Andrei Ruckenstein. Fast counting with tensor networks. *SciPost Physics*, 7(5), November 2019. ISSN 2542-4653. DOI: 10.21468/scipostphys.7.5.060. URL https://doi.org/10.21468/SciPostPhys.7.5.060.

[39] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *29th Annual Symposium on Foundations of Computer Science*. IEEE, 1988. DOI: 10.1109/sfcs.1988.21958. URL https://doi.org/10.1109/SFCS.1988.21958.

[40] Michael Levin and Cody P. Nave. Tensor renormalization group approach to two-dimensional classical lattice models. *Physical Review Letters*, 99(12), September 2007. ISSN 1079-7114. DOI: 10.1103/physrevlett.99.120601. URL https://doi.org/10.1103/PhysRevLett.99.120601.

[41] Chao Li, Junhua Zeng, Zerui Tao, and Qibin Zhao. Permutation search of tensor network structures via local sampling. In *International Conference on Machine Learning*, pages 13106–13124. PMLR, 2022.

[42] Jingling Li, Yanchao Sun, Jiahao Su, Taiji Suzuki, and Furong Huang. Understanding generalization in deep learning via tensor methods. In *International Conference on Artificial Intelligence and Statistics*, pages 504–515. PMLR, 2020.

[43] Hai-Jun Liao, Jin-Guo Liu, Lei Wang, and Tao Xiang. Differentiable programming tensor networks. *Physical Review X*, 9(3), September 2019. ISSN 2160-3308. DOI: 10.1103/physrevx.9.031041. URL https://doi.org/10.1103/PhysRevX.9.031041.

[44] Jin-Guo Liu, Xun Gao, Madelyn Cain, Mikhail D. Lukin, and Sheng-Tao Wang. Computing solution space properties of combinatorial optimization problems via generic tensor networks. *SIAM Journal on Scientific Computing*, 45(3):A1239–A1270, June 2023. ISSN 1095-7197. DOI: 10.1137/22m1501787. URL https://doi.org/10.1137/22M1501787.

[45] Michael Lubasch, J. Ignacio Cirac, and Mari-Carmen Bañuls. Algorithms for finite projected entangled pair states. *Physical Review B*, 90(6), August 2014. ISSN 1550-235X. DOI: 10.1103/physrevb.90.064425. URL https://doi.org/10.1103/PhysRevB.90.064425.

[46] Michael Lubasch, J Ignacio Cirac, and Mari-Carmen Bañuls. Unifying projected entangled pair state contractions. *New Journal of Physics*, 16(3):033014, March 2014. ISSN 1367-2630. DOI: 10.1088/1367-2630/16/3/033014. URL https://doi.org/10.1088/1367-2630/16/3/033014.

[47] Linjian Ma and Edgar Solomonik. Fast and accurate randomized algorithms for low-rank tensor decompositions. *Advances in Neural Information Processing Systems*, 34:24299–24312, 2021.

[48] Linjian Ma and Edgar Solomonik. Cost-efficient gaussian tensor network embeddings for tensor-structured inputs. In *Advances in Neural Information Processing Systems*, volume 35, pages 38980–38993, 2022.

[49] Linjian Ma and Chao Yang. Low rank approximation in simulations of quantum algorithms. *Journal of Computational Science*, 59:101561, March 2022. ISSN 1877-7503. DOI: 10.1016/j.jocs.2022.101561. URL https://doi.org/10.1016/j.jocs.2022.101561.

[50] Linjian Ma, Jiayu Ye, and Edgar Solomonik. AutoHOOT: Automatic high-order optimization for tensors. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 125–137. ACM, September 2020. DOI: 10.1145/3410463.3414647. URL https://doi.org/10.1145/3410463.3414647.

[51] Paul Manuel, Indra Rajasingh, Bharati Rajan, and Helda Mercy. Exact wirelength of hypercubes on a grid. *Discrete Applied Mathematics*, 157(7):1486–1495, April 2009. ISSN 0166-218X. DOI: 10.1016/j.dam.2008.09.013. URL https://doi.org/10.1016/j.dam.2008.09.013.

[52] Akira Matsubayashi. Separator-based graph embedding into multidimensional grids with small edge-congestion. *Discrete Applied Mathematics*, 185:119–137, April 2015. ISSN 0166-218X. DOI: 10.1016/j.dam.2014.11.024. URL https://doi.org/10.1016/j.dam.2014.11.024.

[53] Tensornetwork.org contributors. Density matrix algorithm. *tensornetwork.org*, 2021.

[54] V. Murg, F. Verstraete, R. Schneider, P. R. Nagy, and Ö. Legeza. Tree tensor network state with variable tensor order: An efficient multireference method for strongly correlated systems. *Journal of Chemical Theory and Computation*, 11(3):1027–1036, February 2015. ISSN 1549-9626. DOI: 10.1021/ct501187j. URL https://doi.org/10.1021/ct501187j.

[55] Naoki Nakatani and Garnet Kin-Lic Chan. Efficient tree tensor network states (TTNS) for quantum chemistry: Generalizations of the density matrix renormalization group algorithm. *The Journal of Chemical Physics*, 138(13), April 2013. ISSN 1089-7690. DOI: 10.1063/1.4798639. URL https://doi.org/10.1063/1.4798639.

[56] Bryan O'Gorman. Parameterization of tensor network contraction. 2019. DOI: 10.48550/arXiv.1906.00013. URL https://doi.org/10.48550/arXiv.1906.00013.

[57] Román Orús. Tensor networks for complex quantum systems. *Nature Reviews Physics*, 1 (9):538–550, August 2019. ISSN 2522-5820. DOI: 10.1038/s42254-019-0086-7. URL https://doi.org/10.1038/s42254-019-0086-7.

[58] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, October 2014. ISSN 0003-4916. DOI: 10.1016/j.aop.2014.06.013. URL https://doi.org/10.1016/j.aop.2014.06.013.

[59] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5): 2295–2317, January 2011. ISSN 1095-7197. DOI: 10.1137/090752286. URL https://doi.org/10.1137/090752286.

[60] Sebastian Paeckel, Thomas Köhler, Andreas Swoboda, Salvatore R. Manmana, Ulrich Schollwöck, and Claudius Hubig. Time-evolution methods for matrix-product states. *Annals of Physics*, 411:167998, December 2019. ISSN 0003-4916. DOI: 10.1016/j.aop.2019.167998. URL https://doi.org/10.1016/j.aop.2019.167998.

[61] Feng Pan, Pengfei Zhou, Sujie Li, and Pan Zhang. Contracting arbitrary tensor networks: General approximate algorithm and applications in graphical models and quantum circuit simulations. *Physical Review Letters*, 125(6), August 2020. ISSN 1079-7114. DOI: 10.1103/physrevlett.125.060503. URL https://doi.org/10.1103/PhysRevLett.125.060503.

[62] Yuchen Pang, Tianyi Hao, Annika Dugad, Yiqing Zhou, and Edgar Solomonik. Efficient 2D tensor network simulation of quantum systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020. DOI: 10.5555/3433701.3433719. URL https://dl.acm.org/doi/10.5555/3433701.3433719.

[63] Beheshteh Rakhshan and Guillaume Rabusseau. Tensorized random projections. In *International Conference on Artificial Intelligence and Statistics*, pages 3306–3316. PMLR, 2020.

[64] Satish Rao and Andréa W. Richa. New approximation techniques for some linear ordering problems. *SIAM Journal on Computing*, 34(2):388–404, January 2005. ISSN 1095-7111. DOI: 10.1137/s0097539702413197. URL https://doi.org/10.1137/S0097539702413197.

[65] J A Reyes and E M Stoudenmire. Multi-scale tensor network architecture for machine learning. *Machine Learning: Science and Technology*, 2(3):035036, July 2021. ISSN 2632-2153. DOI: 10.1088/2632-2153/abffe8. URL https://doi.org/10.1088/2632-2153/abffe8.

[66] Subhayan Sahu and Brian Swingle. Efficient tensor network simulation of quantum many-body physics on sparse graphs. 2022. DOI: 10.48550/arXiv.2206.04701. URL https://doi.org/10.48550/arXiv.2206.04701.

[67] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics*, 27:1–39, December 2022. ISSN 1084-6654. DOI: 10.1145/3529090. URL https://doi.org/10.1145/3529090.

[68] U. Schollwöck. The density-matrix renormalization group. *Reviews of Modern Physics*, 77 (1):259–315, April 2005. ISSN 1539-0756. DOI: 10.1103/revmodphys.77.259. URL https://doi.org/10.1103/RevModPhys.77.259.

[69] Philipp Seitz, Ismael Medina, Esther Cruz, Qunsheng Huang, and Christian B. Mendl. Simulating quantum circuits using tree tensor networks. *Quantum*, 7:964, March 2023.

ISSN 2521-327X. DOI: 10.22331/q-2023-03-30-964. URL https://doi.org/10.22331/q-2023-03-30-964.

[70] Y.-Y. Shi, L.-M. Duan, and G. Vidal. Classical simulation of quantum many-body systems with a tree tensor network. *Physical Review A*, 74(2), August 2006. ISSN 1094-1622. DOI: 10.1103/physreva.74.022320. URL https://doi.org/10.1103/PhysRevA.74.022320.

[71] Horst D. Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, September 1997. ISSN 1095-7197. DOI: 10.1137/s1064827593255135. URL https://doi.org/10.1137/S1064827593255135.

[72] E M Stoudenmire and Steven R White. Minimally entangled typical thermal state algorithms. *New Journal of Physics*, 12(5):055026, may 2010. DOI: 10.1088/1367-2630/12/5/055026. URL https://doi.org/10.1088/1367-2630/12/5/055026.

[73] Edwin Stoudenmire and David J Schwab. Supervised learning with tensor networks. *Advances in Neural Information Processing Systems*, 29, 2016. DOI: 10.5555/3157382.3157634. URL https://dl.acm.org/doi/10.5555/3157382.3157634.

[74] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969. ISSN 0945-3245. DOI: 10.1007/bf02165411. URL https://doi.org/10.1007/BF02165411.

[75] Szilárd Szalay, Max Pfeffer, Valentin Murg, Gergely Barcza, Frank Verstraete, Reinhold Schneider, and Ors Legeza. Tensor product methods and entanglement optimization for ab initio quantum chemistry. *International Journal of Quantum Chemistry*, 115(19):1342–1391, 2015. DOI: 10.1002/qua.24898. URL https://doi.org/10.1002/qua.24898.

[76] Dimitrios M. Thilikos, Maria Serna, and Hans L. Bodlaender. Cutwidth I: A linear time fixed parameter algorithm. *Journal of Algorithms*, 56(1):1–24, July 2005. ISSN 0196-6774. DOI: 10.1016/j.jalgor.2004.12.001. URL https://doi.org/10.1016/j.jalgor.2004.12.001.

[77] Vijay V Vazirani. *Approximation algorithms*, volume 1. Springer, 2001.

[78] F. Verstraete, V. Murg, and J.I. Cirac. Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems. *Advances in Physics*, 57(2):143–224, March 2008. ISSN 1460-6976. DOI: 10.1080/14789940801912366. URL https://doi.org/10.1080/14789940801912366.

[79] Frank Verstraete and J Ignacio Cirac. Renormalization algorithms for quantum-many body systems in two and higher dimensions. 2004. DOI: 10.48550/ARXIV.COND-MAT/0407066. URL https://doi.org/10.48550/ARXIV.COND-MAT/0407066.

[80] Guifré Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical Review Letters*, 91(14), October 2003. ISSN 1079-7114. DOI: 10.1103/physrevlett.91.147902. URL https://doi.org/10.1103/PhysRevLett.91.147902.

[81] Steven R. White. Density matrix formulation for quantum renormalization groups. *Physical Review Letters*, 69(19):2863–2866, November 1992. ISSN 0031-9007. DOI: 10.1103/physrevlett.69.2863. URL https://doi.org/10.1103/PhysRevLett.69.2863.

[82] Yifan Zhang and Edgar Solomonik. On stability of tensor networks and canonical forms. 2020. DOI: 10.48550/arXiv.2001.01191. URL https://doi.org/10.48550/arXiv.2001.01191.

[83] Yiqing Zhou, E. Miles Stoudenmire, and Xavier Waintal. What limits the simulation of quantum computers? *Physical Review X*, 10(4), November 2020. ISSN 2160-3308. DOI: 10.1103/physrevx.10.041038. URL https://doi.org/10.1103/PhysRevX.10.041038.

A Notations

B Additional background

B.1 The swap-based algorithm to reorder MPS modes

In the MPS-based automated tensor network contraction algorithms including CATN and Sweep-Contractor, an important step is to reorder the sites in an MPS. The reordering changes the

Notations	Meanings
$G = (V, E)$	The input tensor network graph
$\mathcal{V} = \{V_1, \ldots, V_N\}$	A partitioning of G
$T^{(\mathcal{V})}$	A contraction tree over $\mathcal{V}$
$G[U]$	A subgraph of G that only contains vertices U
$\mathcal{E} = \{E(V_i, V_j) : i, j \in \{1, \ldots, N\}\}$	The set where each element in $\mathcal{E}$ is an edge subset connecting two different partitions
(U_s, W_s)	The contraction s where the first tensor is the contraction output of vertices U_s and the second tensor is the contraction output of vertices W_s
$\mathcal{E}_s = \{E' \cap E(U_s \cup W_s) : E' \in \mathcal{E}\}$	The subset of $\mathcal{E}$ that is adjacent to the tensor network represented by $U_s \cup W_s$
$\sigma^{(\mathcal{E}_s)}$	A linear ordering of the elements in $\mathcal{E}_s$
$T = \mathsf{path}\left(T^{(\mathcal{V})}, V_s\right)$	The sub-contraction path over V_s
$T^{(\mathcal{E}_s)}$	Constraint tree detailed in Appendix D

Table 2: Notation used throughout the paper.

adjacency relation in the MPS, and is used so that subsequent contractions can be performed with lower cost. The reordering is commonly performed via a sequence of adjacent site swappings. For a given MPS whose sites are denoted as a set S and its input ordering is denoted as an injective mapping $\sigma : S \to \{1, \ldots, |S|\}$, changing it to a different ordering τ requires at least $d_{\text{KT}}(\sigma, \tau)$ number of swaps, where d_{KT} denotes the Kendall-Tau distance defined in Definition 3.

Definition 3. Let σ, τ be two orderings over S. The Kendall Tau distance between σ, τ is the number of pairs that are ordered differently in σ, τ, and is also the number of pairwise adjacent transpositions needed to transform σ into τ (or vise versa),

$$d_{\text{KT}}(\sigma, \tau) = \sum_{(c,c') \in S} \left| \sigma(c, c') - \tau(c, c') \right|, \tag{4}$$

where $\sigma(c, c') := \mathbb{1}\left(\sigma(c) < \sigma(c')\right)$ indicates if c is ahead of c' in σ.

We illustrate the standard algorithm to swap adjacent MPS sites via a contraction and a low-rank approximation in Fig. 24. The algorithm first contracts two sites into a single tensor and subsequently performs a low-rank approximation to split the tensor into two parts. When the uncontracted modes have sizes x and y, and the MPS ranks are a, c, and b, the contraction step has an asymptotic cost of $\Theta(abcxy)$, resulting in a tensor with a size of $abxy$. Without truncation, the output rank of the low-rank approximation operation would be the minimum among ay, bx, cxy. In practice, it is common to set an upper bound γ for the MPS ranks, which limits the asymptotic cost of the approximation operation to $O(abxy \min(ay, bx, cxy, \gamma))$ when using the cost model in Section 2.2. To reduce the truncation error, canonicalization is commonly performed on the MPS to orthogonalize all other sites.

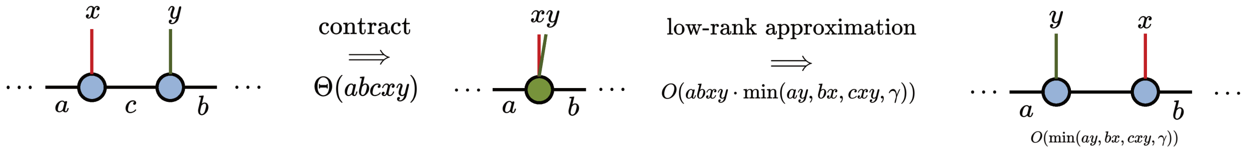


Figure 24: Illustration of the swap operation and the asymptotic computational cost.

B.2 Background on embedding an source graph into a target graph

Our proposed algorithm uses heuristics from the graph embedding problem. A graph embedding of a source graph $G_s = (V_s, E_s)$ into a target graph $G_t = (V_t, E_t)$ is a map from vertices of the input graph onto vertices of the output graph, $\phi : V_s \to V_t$, and each edge connecting u, v of G_s is mapped onto a path connecting $\phi(u), \phi(v)$ of G_t. For each edge $e \in E_t$, we let $\texttt{congestion}(e)$ denote the number of times e is used as a corresponding path of some edge in G_s. We look at the problem of finding the graph embedding that minimizes the congestion [33, 9, 52, 7, 51]. This metric is used since when embedding a tensor network into another graph, low congestion implies that the embedded tensor network has low ranks as well as low memory usage.

For the case where G_t is a line graph and ϕ is an injective mapping, finding ϕ that minimizes the congestion is the widely-discussed linear ordering problem. When the objective is to minimize $\max_{e \in E_t} \texttt{congestion}(e)$, the problem has been called the minimum cut linear arrangement problem, and the congestion is also called cutwidth in the previous work [76]. When the objective is to minimize $\sum_{e \in E_t} \texttt{congestion}(e)$, the problem has been called the minimum linear arrangement problem [32, 19], and multiple approximation algorithms with bounded complexity have been proposed [20, 64, 21, 11, 17].

Recursive bisection is a simple yet effective divide-and-conquer heuristic widely adopted in both linear ordering problems [18, 31] and balanced graph partitioning [67, 71]. For the linear ordering problem, the algorithm proceeds via first applying an approximate 1/3-balanced cut to separate V_s into two parts S and $V_s \setminus S$, then placing all vertices of S before all vertices not in S, and then recursing on both S and $V_t \setminus S$. Let n denote the number of vertices in the graph, it is known that if one has a γ-approximation algorithm for minimum 1/3-balanced cut, then both the minimum cut linear arrangement and the minimum linear arrangement problem admit an approximation of $O(\gamma \log n)$ [31, 77]. The approximation factor for the 1/3-balanced cut is improved from $\gamma = O(\log n)$ [39] to $\gamma = O(\sqrt{\log n})$ [3], making the approximation factor of the recursive bisection $O(\log^{1.5} n)$.

In Sections 5 and 6, we use recursive bisection as a heuristic for other embedding problems where ϕ is not necessarily injective, and G_t is a general binary tree rather than a line graph.

C Embedding tree definitions

We formally define the embedding tree with an MPS and a comb structure in Definition 5 and Definition 6. Both definitions are based on the MPS tree, which is defined in Definition 4.

Definition 4 (MPS tree)**.** Consider a set S with a linear ordering σ^S. Let $x_i \in S$ denote the element with $\sigma^S(x_i) = i$. The MPS tree defined on σ^S is a full binary tree with the elements of S serving as the tree's leaf nodes. The MPS tree contains $|S| - 1$ non-leaf nodes, where the first non-leaf node is connected to x_1 and x_2, and the ith non-leaf node for $i \in \{2, \dots, |S| - 1\}$ is connected to the $i - 1$th non-leaf node and x_{i+1}. An example is shown in Fig. 11a.

Definition 5 (Embedding tree with an MPS structure)**.** Consider orderings $\sigma^{(\mathcal{E}_s)}$ and $\sigma^{(E')}$ for $E' \in \mathcal{E}_s$. Let $n_s = |\mathcal{E}_s|$, and let E_i denote the edgeset with $\sigma^{(\mathcal{E}_s)}(E_i) = i$. The MPS embedding tree based on $\sigma^{(\mathcal{E}_s)}$, $\{\sigma^{(E')}, E' \in \mathcal{E}_s\}$ is the MPS tree defined on the ordering $\sigma^{(E_1)} \oplus \cdots \oplus \sigma^{(E_{n_s})}$, where we use $\sigma^{S_1} \oplus \sigma^{S_2}$ to denote the concatenation of two orderings σ^{S_1} and σ^{S_2}, so that each $x \in S_1$ is mapped to $\sigma^{S_1}(x)$ and each $x \in S_2$ is mapped to $\sigma^{S_2}(x) + |S_1|$.

Definition 6 (Embedding tree with a comb structure)**.** Consider orderings $\sigma^{(\mathcal{E}_s)}$ and $\sigma^{(E')}$ for $E' \in \mathcal{E}_s$. Let $n_s = |\mathcal{E}_s|$, and let E_i denote the edgeset with $\sigma^{(\mathcal{E}_s)}(E_i) = i$. Let T_i denote the MPS tree on top of $\sigma^{(E_i)}$ and let r_i denote the root node of T_i. The comb embedding tree based on $\sigma^{(\mathcal{E}_s)}$, $\{\sigma^{(E')}, E' \in \mathcal{E}_s\}$ contains all T_i for $i \in \{1, \ldots, n_s\}$ and another MPS tree $\hat{T}$ used to connect all T_i. The MPS tree $\hat{T}$ connects all r_i and is defined on top of the ordering $\hat{\sigma} : \{r_1, \ldots, r_{n_s}\} \to \{1, \ldots, n_s\}$, where $\hat{\sigma}(r_i) = i$.

D Determination of the constraint tree based on the contraction path

The constraint tree $T^{(\mathcal{E}_s)}$ is constructed based on the sub-contraction path T. The tree is constructed bottom-up by connecting subsets of edges involved in the contraction path. This construction is based on the assumption that ordering edges to make earlier rather than later contractions efficient is more important.

Specifically, we let $U_1, \ldots, U_n$ be the n partitions contracted with V_s in order in the path T, let $\mathcal{E}$ be the edge partitions defined in Line 3 of Algorithm 2, and let $\mathcal{E}(U_i) = \{\bar{E} \cap E(U_i) : \bar{E} \in \mathcal{E}\}$ be the subset of $\mathcal{E}$ incident on U_i. For each contraction with U_i, we use $\hat{\mathcal{E}}_i$ to denote the subset of $\mathcal{E}_s$ that we want to be connected in $T^{(\mathcal{E}_s)}$ based on the contraction. In particular, $\hat{\mathcal{E}}_1 = (\mathcal{E}_s \cap \mathcal{E}(U_1))$ contains all contracted edges $E(V_s, U_1)$. For each $i \in \{2, \ldots, n\}$, we want $(\mathcal{E}_s \cap \mathcal{E}(U_i))$ along with some $\hat{\mathcal{E}}_j, j < i$ to be adjacent. Formally speaking, for each $i \in \{1, \ldots, n\}$, we define

$$\hat{\mathcal{E}}_i = (\mathcal{E}_s \cap \mathcal{E}(U_i)) \bigcup_{j \in S_i} \hat{\mathcal{E}}_j, \tag{5}$$

where $S_i \subseteq \{1, \ldots, i-1\}$ is a subset of indices ahead of i such that for each $j \in S_i$, U_j is adjacent to U_i. In Fig. 25, we use an example to illustrate the constraint tree construction algorithm, and each $\hat{\mathcal{E}}_i$ is also shown in the figure.

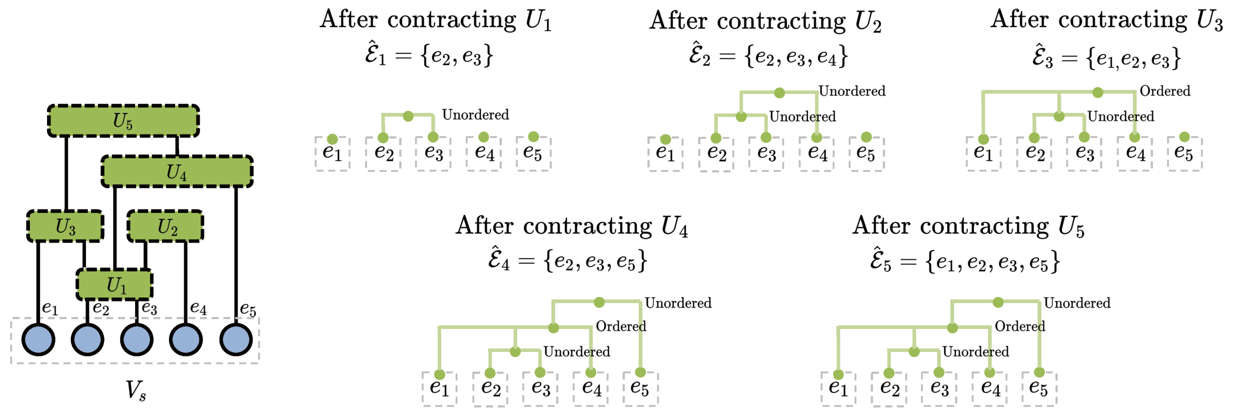

After contracting U_1
$\hat{\mathcal{E}}_1 = \{e_2, e_3\}$
After contracting U_2
$\hat{\mathcal{E}}_2 = \{e_2, e_3, e_4\}$
After contracting U_3
$\hat{\mathcal{E}}_3 = \{e_1, e_2, e_3\}$
After contracting U_4
$\hat{\mathcal{E}}_4 = \{e_2, e_3, e_5\}$
After contracting U_5
$\hat{\mathcal{E}}_5 = \{e_1, e_2, e_3, e_5\}$
U_5
U_4
U_3
U_2
U_1
e_1
e_2
e_3
e_4
e_5
V_s
Unordered
Ordered

Figure 25: Illustration of the algorithm to construct the constraint tree. The constraint tree is built on top of the uncontracted edgesets of V_s, $\mathcal{E}_s = \{\{e_1\}, \{e_2\}, \{e_3\}, \{e_4\}, \{e_5\}\}$. The partitions $U_1, \ldots, U_5$ are contracted with V_s in order. For the ith contraction, we show the value of $\hat{\mathcal{E}}_i$ and show the constraint tree after that contraction step.

In the algorithm, $T^{(\mathcal{E}_s)}$ is initialized to be a disconnected graph with vertices $\mathcal{E}_s$. For the ith contraction that contracts U_i, the algorithm updates the $T^{(\mathcal{E}_s)}$ so that the leaves $\hat{\mathcal{E}}_i$ will be connected. The rules are as follows.

1. If $\hat{\mathcal{E}}_i$ are already connected in $T^{(\mathcal{E}_s)}$, we just keep the constraint tree unchanged. For example, in Fig. 25 the constraint tree is unchanged after we consider the fifth contraction, since $\hat{\mathcal{E}}_5$ is already connected.

2. If $\hat{\mathcal{E}}_i$ is the union of multiple connected leaf subsets, then a vertex is added to $T^{(\mathcal{E}_s)}$ whose children are the root vertices of these connected leaf subsets. In addition, this new vertex is

labeled as “unordered”. In Fig. 25, the constraint trees after both the first and the second contraction belong to this case.

3. If $\hat{\mathcal{E}}_i$ is a subset of the union of multiple connected leaves subsets $\bar{\mathcal{E}}$, then there are cases where $\hat{\mathcal{E}}_i$ cannot be adjacent in the tree. For this case, a vertex is added to $T^{(\mathcal{E}_s)}$ whose children are the root vertices of $\bar{\mathcal{E}}$ and the vertex is labeled as “unordered”. In Fig. 25, the constraint trees after the fourth contraction belongs to this case. For the other cases, we can reorder the constraint tree and label some vertices as “ordered” to add the adjacency constraints. In Fig. 25, the constraint trees after the third contraction belongs to this case.

E Optimality of the edge set ordering algorithm

In this section, we prove that the output ordering of Algorithm 3 minimizes the Kendall-Tau distance with the reference ordering under the adjacency constraint.

Let $\mathcal{P}\left(T^{(\mathcal{E}_s)}\right)$ represents the set of orderings of the leaves of $T^{(\mathcal{E}_s)}$ constrained by $T^{(\mathcal{E}_s)}$. Each ordering in this set must adhere to all the adjacency relations specified by $T^{(\mathcal{E}_s)}$. In Theorem E.2, we establish that the output ordering produced by Algorithm 3 aims to minimize the Kendall-Tau distance, as defined in Definition 3, between itself and the reference ordering τ,

$$\sigma^{(\mathcal{E}_s)} = \arg \min_{\sigma \in \mathcal{P}\left(T^{(\mathcal{E}_s)}\right)} d_{\mathrm{KT}}\left(\sigma, \tau\right). \tag{6}$$

Before the presentation of Theorem E.2, we first present Lemma E.1 that is used in the proof of the theorem. The lemma can be easily proved based on the definition of Kendall-Tau distance in Definition 3.

Lemma E.1. *Consider an ordering $\tau^{(C)}$ over a set $C = C_1 \cup C_2$, and let $\tau^{(C_1)}$, $\tau^{(C_2)}$ denote the restrictions of the ordering $\tau^{(C)}$ to the subset C_1, C_2, respectively. Consider another two orderings $\sigma^{(C_1)}$, $\sigma^{(C_2)}$ over C_1, C_2, respectively. Then, we have*

$$d_{KT}\left(\tau^{(C)}, \sigma^{(C_1)} \oplus \sigma^{(C_2)}\right) = d_{KT}\left(\tau^{(C)}, \tau^{(C_1)} \oplus \tau^{(C_2)}\right) + d_{KT}\left(\tau^{(C_1)}, \sigma^{(C_1)}\right) + d_{KT}\left(\tau^{(C_2)}, \sigma^{(C_2)}\right), \tag{7}$$

where $\tau^{(C_1)} \oplus \tau^{(C_2)}$ denotes the concatenation of $\tau^{(C_1)}$, $\tau^{(C_2)}$.

Theorem E.2. *Given a reference ordering τ and a guide tree $T^{(\mathcal{E}_s)}$, the output ordering of Algorithm 3 is an optimal solution of the optimization problem, $\min_{\sigma \in \mathcal{P}\left(T^{(\mathcal{E}_s)}\right)} d_{KT}\left(\sigma, \tau\right)$.*

Proof. For each vertex v in the constraint tree $T^{(\mathcal{E}_s)}$, we let $\texttt{subtree}\left(v, T^{(\mathcal{E}_s)}\right)$ denote the subtree in the constraint tree where the root vertex is v. In addition, as is defined in Line 15 of Algorithm 3, we use τ_v to denote the restriction of the ordering τ to the subset represented by the leaves of $\texttt{subtree}\left(v, T^{(\mathcal{E}_s)}\right)$. Below we prove that for each $v \in T^{(\mathcal{E}_s)}$,

$$f(v) = \arg \min_{\sigma \in \mathcal{P}\left(\texttt{subtree}\left(v, T^{(\mathcal{E}_s)}\right)\right)} d_{\mathrm{KT}}\left(\sigma, \tau_v\right), \tag{8}$$

where $f(v)$ is defined in Line 16 of Algorithm 3. Since we output $f\left(r\right)$ with $r = \texttt{root}\left(T^{(\mathcal{E}_s)}\right)$, and $\texttt{subtree}\left(r, T^{(\mathcal{E}_s)}\right) = T^{(\mathcal{E}_s)}$, the output ordering satisfies $f(r) = \arg\min_{\sigma \in \mathcal{P}\left(T^{(\mathcal{E}_s)}\right)} d_{\mathrm{KT}}\left(\sigma, \tau\right)$. This finishes the proof.

For the base cases where v is one of the leaf vertices, (8) holds since the set to be ordered only contains one element thus the ordering is unique.

Now consider the case where v is a non-leaf vertex. In the analysis we assume v has two children, u_1 and u_2. Note that the analysis can be easily generalized to the case of more than 2 children for both “unordered” and “ordered” labels.

Assume (8) holds for its children, u_1 and u_2. Consider the case where

$$d_{\mathrm{KT}}\left(f(u_1) \oplus f(u_2), \tau_v\right) < d_{\mathrm{KT}}\left(f(u_2) \oplus f(u_1), \tau_v\right), \tag{9}$$

so that Line 16 sets $f(v)$ as $f(u_1) \oplus f(u_2)$. We then have

$$\begin{aligned} d_{\mathrm{KT}}\left(f(v), \tau_v\right) &= d_{\mathrm{KT}}\left(f(u_1) \oplus f(u_2), \tau_v\right) \\ &\overset{\text{Lemma E.1}}{=} d_{\mathrm{KT}}\left(\tau_v, \tau_{u_1} \oplus \tau_{u_2}\right) + d_{\mathrm{KT}}\left(f(u_1), \tau_{u_1}\right) + d_{\mathrm{KT}}\left(f(u_2), \tau_{u_2}\right). \end{aligned} \tag{10}$$

The first term in (10) reaches the minimum since (9) holds. Moreover, the last two terms also reach the minimum since (8) holds for u_1 and u_2. These conditions imply $f(v)$ satisfies (8). Similar analysis can be applied for the case where $d_{\mathrm{KT}}\left(f(u_1) \oplus f(u_2), \tau_v\right) > d_{\mathrm{KT}}\left(f(u_2) \oplus f(u_1), \tau_v\right)$. This along with the based cases finish the proof.

□

F Lemmas in computational cost analysis

This section provides lemmas for Theorem 6.1, which shows that the asymptotic cost of the density matrix algorithm is upper-bounded by that of the canonicalization-based algorithm. We demonstrate in Lemma F.3 that when the input tensor network has a tree structure, both the density matrix algorithm and the canonicalization-based algorithm exhibit the same asymptotic cost for truncating the bond sizes in the tree tensor network.

The Lemma F.1 and Lemma F.2 below are used to prove Lemma F.3.

Lemma F.1. *Consider a tensor network with a tree structure $T = (V_T, E_T)$. Assuming that changing a tree tensor network into the canonical form will not change any bond size of the network. For two adjacent vertices z, v, forming `canonical_form`$_T(v, z)$ has the same asymptotic cost as forming `density_matrix`$_T(v, z)$.*

Proof. For each edge set $E' \subseteq E_T$, we let $s(E') = \exp(w(E'))$ denote the bond size of E'. We also let $\mathbf{M}_v$ denote the tensor at each vertex $v \in V_T$.

For the pair of adjacent vertices v, z, assume that `canonical_form`$_T(u, v)$ already exist for all $u \in N(v) \backslash \{z\}$. Let $\mathbf{R}_u$ denote the non-orthogonal core of `canonical_form`$_T(u, v)$. To construct the form `canonical_form`$_T(v, z)$, we first contract $\mathbf{M}_v$ with $\mathbf{R}_u$ for each $u \in N(v) \setminus \{z\}$, which yields a cost of $\Theta\left(\sum_{u \in N(v) \backslash \{z\}} s(E_T(v)) s(E_T(u, v))\right)$, and then use a QR decomposition to orthogonalize the tensor at v, which yields a cost of $\Theta\left(s(E_T(v)) s(E_T(v, z))\right)$. These steps make the overall cost

$$\Theta\left(\sum_{u \in N(v)} s(E_T(v)) s(E_T(u, v))\right). \tag{11}$$

We now consider the computation of `density_matrix`$_T(v, z)$ under the assumption that for all $u \in N(v) \setminus \{z\}$, $\mathbf{L}_u =$ `density_matrix`$_T(u, v)$ already exist. Below we consider the three different cases,

- when $N(v) \setminus \{z\} = \emptyset$, the computation involves the contraction $\mathbf{M}_v \mathbf{M}_v^T$,
- when $N(v) \setminus \{z\} = \{u\}$, the computation involves the contraction $(\mathbf{M}_v \mathbf{L}_u) \mathbf{M}_v^T$,
- when $N(v) \setminus \{z\} = \{u_1, u_2\}$, the computation involves the contraction $\mathbf{M}_v (\mathbf{L}_{u_1} \otimes \mathbf{L}_{u_2}) \mathbf{M}_v^T$, which can be efficiently computed by performing the contractions $\mathbf{M}_v$ with $\mathbf{L}_{u_1}$ and $\mathbf{M}_v$ with $\mathbf{L}_{u_2}$ first, and then contracting the outputs.

For all the cases above, the overall cost is $\Theta\left(\sum_{u \in N(v)} s(E_T(v)) s(E_T(u, v))\right)$, which equals the cost of the canonical form. Since both `canonical_form`$_T(v, z)$ and `density_matrix`$_T(v, z)$ have the same recursive relation, computing `canonical_form`$_T(u, v)$ has the same cost as that of the `density_matrix`$_T(u, v)$ for $u \in N(v) \setminus \{z\}$. This finishes the proof.

□

Lemma F.2. *Consider a tensor network with a tree structure $T = (V_T, E_T)$, where each $z \in V_T$ represents a tensor $\mathbf{M}_z$. Let $v \in V_T$ be a leaf vertex that represents $\mathbf{M}_v \in \mathbb{R}^{a_v \times b_v}$, where a_v denotes the size of the uncontracted modes and b_v denotes the size of the contracted modes incident on v, and let $u = \texttt{parent}(T, v)$. Given that $\texttt{density\_matrix}_T(u, v)$ has been computed, computing the orthogonal matrix $\mathbf{U}_v$ (Line 11 or 18 of Algorithm 4) has a cost of $\Theta\left(a_v b_v^2\right)$.*

Proof. For the case where $a_v = O\left(b_v\right)$, the algorithm first computes $\mathbf{L}_v = \texttt{density\_matrix}_T(v)$ with a cost of $\Theta\left(a_v b_v^2 + a_v^2 b_v\right)$, and then computes $\mathbf{U}_v$ via a low-rank factorization on $\mathbf{L}_v \in \mathbb{R}^{a_v \times a_v}$ with the maximum rank being $r = O\left(a_v\right)$, which costs $\Theta\left(a_v^2 r\right)$. The overall cost is $\Theta\left(a_v b_v^2 + a_v^2 b_v + a_v^2 r\right) = \Theta\left(a_v b_v^2\right)$.

For the case where $a_v = \Omega\left(b_v\right)$, the algorithm first performs a QR decomposition of $\mathbf{M}_v$ into $\mathbf{U}_v \in \mathbb{R}^{a_v \times b_v}, \mathbf{R}_v \in \mathbb{R}^{b_v \times b_v}$ with a cost of $\Theta\left(a_v b_v^2\right)$, then computes the leading singular vectors of $\mathbf{R}_v \mathbf{L}_u$ that is denoted $\hat{\mathbf{U}}_v \in \mathbb{R}^{b_v \times r}$, which costs $\Theta\left(b_v^3\right)$. Finally, $\mathbf{U}_v$ is updated as the product $\mathbf{U}_v \hat{\mathbf{U}}_v$ with a cost of $\Theta(a_v b_v r)$. Overall the cost is $\Theta\left(a_v b_v^2 + b_v^3 + a_v b_v r\right) = \Theta\left(a_v b_v^2\right)$. This finishes the proof. □

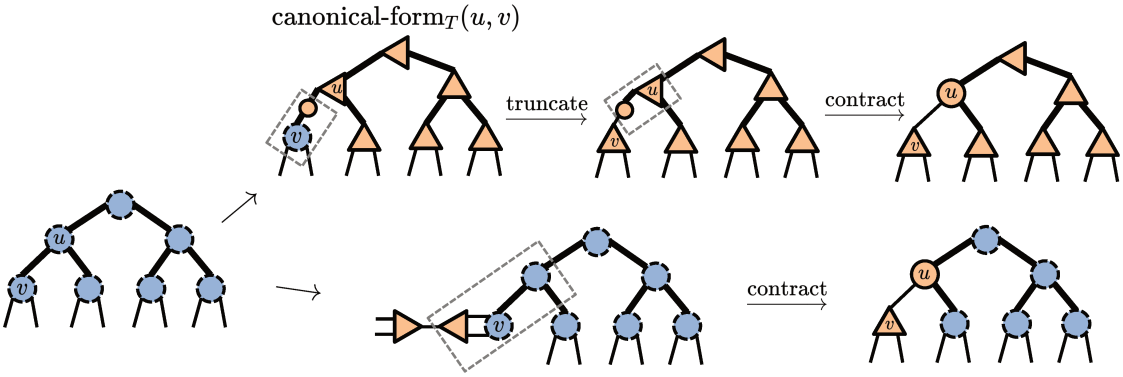


Figure 26: Illustration of the difference between the canonicalization-based algorithm and the density matrix algorithm. The upper path denotes truncating the edge (u, v) using canonicalization, and the lower path uses the density matrix algorithm. In the lower path, the orthogonal matrix is calculated as the leading singular vectors/eigenvectors of the density matrix $\texttt{density\_matrix}_T(z)$.

Lemma F.3. *Consider a given tree tensor network $T = (V_T, E_T)$. Let $\sigma : V_T \to \{1, \ldots, |V_T|\}$ be a post-order DFS traversal of T that shows the the tensor update ordering. Assuming that changing a tree tensor network into its canonical form will not change any bond size of the network, the asymptotic cost of the density matrix algorithm (Algorithm 4) for truncating the modes in T is the same as that of the canonicalization-based algorithm (Algorithm 1) if both algorithms use the same update ordering σ, and the same maximum bond size χ.*

Proof. Consider the step to update the tensor at a given vertex $v \in V_T$. Let $\mathbf{M}_v \in \mathbb{R}^{a_v \times b_v}$, where a_v denote the size of the uncontracted modes and b_v denote the size of the contracted modes of $\mathbf{M}_v$. Also let $r = \min\left(a_v, b_v, \chi\right)$ and $u = \texttt{parent}(T, v)$. We break down the cost of Algorithm 4 and Algorithm 1 into 3 parts, and show that for each of the three parts, the costs of the two algorithms are asymptotically equal.

In Algorithm 1, the steps include 1) forming $\texttt{canonical\_form}_T(u, v)$, 2) multiplying $\mathbf{M}_v$ with $\mathbf{R}_u \in \mathbb{R}^{b_v \times b_v}$, the non-orthogonal core of the canonical form, and 3) performing a rank-χ approximation to get $\mathbf{U}_v \in \mathbb{R}^{a_v \times r}, \hat{\mathbf{R}}_u \in \mathbb{R}^{r \times b_v}$, and 3) multiplying $\hat{\mathbf{R}}_u$ with $\mathbf{M}_u$.

In Algorithm 4 with each partition contracted into a tensor, the steps include 1) forming the density matrix $\texttt{density\_matrix}_T(u, v)$, 2) using $\texttt{density\_matrix}_T(u, v)$ and $\mathbf{M}_v$ to compute $\mathbf{U}_v \in \mathbb{R}^{a_v \times r}$ and $\mathbf{M}_v = \mathbf{U}_v^T \mathbf{M}_v$, and 3) multiplying $\mathbf{M}_v \in \mathbb{R}^{r \times b_v}$ with $\mathbf{M}_u$.

The comparison between the two algorithms is visualized in Fig. 26. It can be seen that the third step of both algorithms have the same asymptotic cost. For the first step, we show in Lemma F.1 that both algorithms have the same asymptotic cost. For the second step, the canonicalization-based algorithm yields a cost of $\Theta\left(a_v b_v^2 + a_v b_v r\right) = \Theta\left(a_v b_v^2\right)$ using the cost model in Section 2.2. In addition, we show in Lemma F.2 that the cost to compute $\mathbf{U}_v$ in the density matrix algorithm under the assumption that each partition is contracted into a tensor is also $\Theta\left(a_v b_v^2\right)$. Since the multiplication $\mathbf{U}_v^T \mathbf{M}_v$ costs $\Theta\left(a_v b_v r\right) = O\left(a_v b_v^2\right)$, the cost equals the cost of the canonicalization-based algorithm, thus finishing the proof. □