TOWARDS EFFICIENT ALGORITHMS AND SYSTEMS FOR TENSOR DECOMPOSITIONS AND TENSOR NETWORKS

BY

LINJIAN MA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Doctoral Committee:

Associate Professor Edgar Solomonik, Chair and Director of Research
Professor Chandra Chekuri
Professor Luke Olson
Dr. Miles Stoudenmire, Flatiron Institute

ABSTRACT

Tensors, which are multidimensional arrays generalizing vectors and matrices, play an important role across various domains, including signal processing, machine learning, and computational physics. Despite their widespread applications, computing with high-dimensional tensors poses a challenge known as the “curse of dimensionality”. Tensor decomposition provides a pathway to solving this challenge by representing or approximating high-dimensional tensors in the form of tensor networks. These networks consist of interconnected small tensors, forming a specific graph structure that represents their contraction relationships. This thesis introduces computationally efficient numerical algorithms and computer systems for both tensor decompositions and problems involving tensor networks. On the algorithmic side, we present novel inexact solvers that not only outperform standard methods in terms of computational costs but also offer theoretical guarantees on the accuracy of approximations. On the system side, we develop libraries that efficiently automate the algorithmic development for tensor networks, thereby enhancing their practical usability.

In the first part of the thesis, we explore how alternating minimization, the most common iterative algorithm for tensor decompositions, can be made more efficient and can be done in an automated manner. We propose an inexact solver named pairwise perturbation, which uses the fact that the tensor decomposition outputs change little when the algorithm approaches convergence, and uses the previously-computed normal equations with perturbative corrections to approximate the exact normal equations to reduce the asymptotic cost. Moreover, a major challenge to efficient alternating minimization is making use of the shared sub-structure of tensor contractions computed at each iteration. We introduce an automatic differentiation system named AutoHOOT, which incorporates tensor algebra-specific transformations and includes algorithms to automatically amortize shared sub-structure of tensor contractions across subproblems in each iteration of alternating minimization.

Each subproblem in an iteration of alternating minimization is a linear least squares problem with the left-hand-side matrix being tall and skinny with a specific tensor network structure. Such problems motivate the second part of the thesis, which includes novel sketching algorithms for both tensor decompositions and tensor networks. Sketching involves employing random matrices, also known as embeddings, to project data onto low-dimensional spaces, thereby reducing the computational cost of subsequent operations. In the context of data with a tensor network structure, we present efficient algorithms that utilize tensor network-structured embeddings to sketch the data. Moreover, we provide theoretical bounds on the accuracy of sketching achieved through these algorithms. The proposed sketching techniques are used to accelerate various problems involving tensor networks, including tensor decompositions.

The third part includes algorithms to approximate the output of tensor network contraction, which explicitly evaluates the single tensor represented by a given tensor network and is widely used in statistical physics, quantum computing, and computer science. We introduce methods to efficiently approximate tensor network contractions using low-rank approximations, where each intermediate tensor generated during the contractions is approximated as a low-rank tree tensor network. We introduce CATN-GO, an algorithm that uses graph theory to analyze the tensor network structure and generates contraction paths (a rooted binary tree showing how the tensor network is contracted) that yield both the minimum number of approximations and the minimum computational cost, which improves both efficiency and accuracy. In addition, we introduce another algorithm named Partitioned Contract, which has the flexibility to incorporate a large portion of the tensor network when performing low-rank approximations to reduce the truncation error, and includes a cost-efficient algorithm to approximate any tensor network into a tree structure.

In the fourth part, we consider applications of tensor decompositions in quantum computing. We use canonical polyadic (CP) tensor decomposition to simulate and analyze quantum algorithms. We successfully simulate multiple quantum algorithms including the Grover's search, quantum Fourier transform, and quantum phase estimation using low-rank CP decomposition, and we analyze the entanglement properties of specific quantum states using the CP decomposition rank.

ACKNOWLEDGMENTS

During my journey as a Ph.D. student, I feel incredibly fortunate to have been a part of CS@Illinois and to have collaborated with many talented and inspiring individuals. Their support and guidance have been instrumental in the completion of this dissertation.

First and foremost, I am deeply grateful to my advisor, Professor Edgar Solomonik, for his support and mentorship over the years. Professor Edgar Solomonik gave me a lot of freedom to pursue my interests, and also offered invaluable insights and guidance for each of my projects, which played a critical role in improving the quality of every publication. I have gained valuable knowledge in various domains from Professor Edgar Solomonik, including the theoretical analysis of algorithms and academic writing skills.

I would like to extend my appreciation to my committee members, Professor Chandra Chekuri, Professor Luke Olson, and Dr. Miles Stoudenmile. Professor Chandra Chekuri's courses on randomized algorithms and approximation algorithms have significantly contributed to my Ph.D. work. Professor Luke Olson served as my program of study advisor and qualifying exam committee member and provided valuable guidance, and I also learned extensively from him during my time as his teaching assistant for numerical analysis. I am also thankful for Dr. Miles Stoudenmile' guidance in the last two years, and his expertise in computational physics has been instrumental to my growth.

I would like to thank my collaborators, Jiayu Ye, Chao Yang, Matt Fishman, Cameron Ibrahim, Ilya Safro, Louis Schatzki, and Eric Chitambar, all of whom have made significant contributions to multiple Chapters of this thesis. In particular, I have learned a lot on system development from Jiayu and Matt, on graph algorithms from Cameron and Ilya, on numerical linear algebra from Chao, and on quantum information from Louis and Eric. Additionally, I would like to express my gratitude to the group members from the Laboratory for Parallel Numerical Algorithms, Edward Hutter, Navjot Singh, Yuchen Pang, Mina Sun, and Raghavendra Kanakagiri, for their assistance and insightful discussions on various topics.

Lastly, I would like to thank my wife Zhenan Shao, my mother Shengli Zhu, my father Xixiang Ma, and my friends for their unconditional support throughout the years.

TABLE OF CONTENTS

Chapter 1: INTRODUCTION

Tensors are multidimensional arrays that generalize the vector and matrix concepts. Formally-speaking, an N-way or Nth-order tensor is an element of the tensor product of N vector spaces. A scalar, vector, and matrix correspond to tensors of order zero, one, and two, respectively.

Tensors arise naturally in many areas, including signal processing, machine learning, and computational quantum physics. In signal processing, tensors can be used to represent multi-dimensional data such as images or audio signals. For example, the pixels of an image can be represented as a three-dimensional tensor, with the dimensions corresponding to the width, height, and color channels of the image. Tensors can also be used to analyze complex signals such as electroencephalography (EEG) recordings [1], where the tensor dimensions correspond to the time, frequency, and spatial location of the electrodes. In machine learning, tensors are building blocks of deep neural network structures, and they are also widely used in Bayesian networks [2] to represent the conditional probabilities in the network. In quantum physics, tensors are used to represent wave functions and Hamiltonians [3]. In quantum computing, tensors are used to represent the quantum state of the qubits, and quantum gates typically represented as unitary matrices are used to perform operations such as rotations, entanglement, and measurement when acting on the quantum states.

One of the key challenges in working with tensors is called the "curse of dimensionality", where tensors with large dimensionality can have an extremely large number of components, making it difficult to analyze and extract meaningful information from them. In this thesis, we look into the theory and applications of *tensor networks* and *tensor decomposition*, which are two powerful techniques for addressing this challenge. In particular, we develop efficient algorithms and computer systems for tensor decompositions and multiple other problems involving tensor networks. A tensor network [4] employs a collection of small tensors, where some or all of their dimensions are contracted according to some pattern, to implicitly represent a high-dimensional tensor, and tensor decomposition [5] extends classical matrix factorization algorithms to multi-way data and represents/approximates high-dimensional tensors as specific tensor network structures.

1.1 TENSOR DECOMPOSITION AND ITS APPLICATIONS

Three of the most important and popular tensor decomposition types are canonical polyadic (CP) [6]–[9], Tucker [10], and the matrix product state (MPS) [4], [11] (also known as tensor train [12]) decompositions. Both CP and Tucker decompositions generalize the concept of the singular value decomposition (SVD) from matrices to general tensors in different ways. Tensor train decomposition is another decomposition format that is commonly used in computational quantum physics.

CP decomposition, also known as the PARAFAC (parallel factor) decomposition, decomposes a tensor into a sum of rank-one tensors. Mathematically, the elementwise expression of CP tensor decomposition of a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{s_1 \times s_2 \times \cdots \times s_N}$ with rank R is given by

$$x_{i_1 i_2 \ldots i_N} = \sum_{r=1}^{R} a^{(1)}_{i_1 r} a^{(2)}_{i_2 r} \cdots a^{(N)}_{i_N r}, \tag{1.1}$$

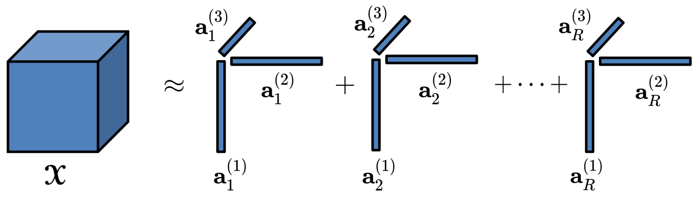
where $x_{i_1 i_2 \ldots i_N}$ is an element of $\boldsymbol{\mathcal{X}}$, and $a^{(j)}_{i_j r}$ is an element of the jth factor matrix $\mathbf{A}^{(j)}$. The minimum R for the exact CP decomposition is called the *tensor rank* [13], [14]. The CP decomposition is symmetric and requires a small number of parameters when R is low. However, the computation of the tensor rank is NP-hard [15].

Tucker decomposition decomposes a tensor into a core tensor multiplied by a set of factor matrices along each mode. Mathematically, Tucker decomposition of a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{s_1 \times s_2 \times \cdots \times s_N}$ into a core tensor $\boldsymbol{\mathcal{C}}$ with size $R_1 \times R_2 \times \cdots \times R_N$ and factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{s_n \times R_n}$ for $n \in \{1, \ldots, N\}$ is given by

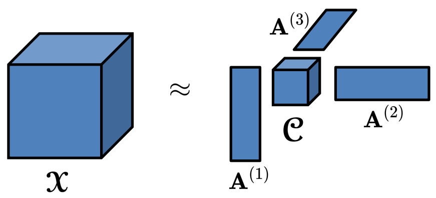
$$x_{i_1 i_2 \ldots i_N} = \sum_{r_1=1}^{R_1} \cdots \sum_{r_N=1}^{R_N} c_{r_1 r_2 \ldots r_N} a^{(1)}_{i_1 r_1} a^{(2)}_{i_2 r_2} \cdots a^{(N)}_{i_N r_N}. \tag{1.2}$$

Different from CP decomposition, computing the minimum R_i for $i \in \{1, \ldots, N\}$ can be easily achieved through N singular value decompositions on different matricizations of $\boldsymbol{\mathcal{X}}$, where each matricization reorders the elements in $\boldsymbol{\mathcal{X}}$ and forms a matrix. However, the number of parameters used in Tucker decomposition is exponential in N, which makes it suitable for small tensor order. Illustrations of CP and Tucker decompositions are given in Fig. 1.1.

CP and Tucker decompositions are commonly used in various fields, including data analytics [5], [16], machine learning [17]–[23], and quantum chemistry [24], [25]. In data analytics, CP decomposition is often utilized to extract meaningful features from tensor data. By decomposing the data into multiple rank-1 factors, CP decomposition can reveal hidden patterns that are difficult to identify using other methods. Tucker decomposition, on the other hand, is commonly used for dimensionality reduction and data compression. In machine learning, CP decomposition is often used to speed up the estimation of parameters



(a) CP decomposition of an order 3 tensor



(b) Tucker decomposition of an order 3 tensor

Figure 1.1: Visualization of CP and Tucker decompositions.

in latent variable models [23]. Both CP and Tucker decompositions have also been used to accelerate neural network training, by approximating specific tensor kernels as low-rank decompositions [21]. In quantum chemistry, CP decomposition has been applied to the Cholesky factorization of the order four two-electron integral tensors, enabling the compression of the operator and speeding up post-Hartree-Fork calculations [26]. Tucker decomposition has been used to design efficient wave function approximations for special molecular systems [27].

The MPS decomposition decomposes a tensor into a chain-like structure. Mathematically, the MPS decomposition of a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{s_1 \times s_2 \times \cdots \times s_N}$ into N core tensors $\boldsymbol{\mathcal{A}}^{(1)} \in \mathbb{R}^{s_1 \times R_1}$, $\boldsymbol{\mathcal{A}}^{(i)} \in \mathbb{R}^{R_{i-1} \times s_i \times R_i}$ for $i \in \{2, \dots, N-1\}$, and $\boldsymbol{\mathcal{A}}^{(N)} \in \mathbb{R}^{R_{N-1} \times s_{N-1}}$, is given by

$$x_{i_1 i_2 \dots i_N} = \sum_{r_1=1}^{R_1} \cdots \sum_{r_{N-1}=1}^{R_{N-1}} a^{(1)}_{i_1 r_1} a^{(2)}_{r_1 i_2 r_2} \cdots a^{(N)}_{r_{N-1} i_N}. \tag{1.3}$$

Similar to Tucker decomposition, finding the minimum R_i to achieve the exact decomposition is easy and can be achieved through a sequence of SVDs. In addition, each $\boldsymbol{\mathcal{A}}^{(i)}$ is an order three tensor, making the format efficient for tensors with high order. Applications of the MPS decomposition are discussed in Section 1.2.

To seek accurate CP, Tucker, or the MPS decomposition of a given tensor $\boldsymbol{\mathcal{X}}$ under a fixed decomposition rank, one commonly solves the optimization problem that minimizes the Frobenius norm of the residual tensor, $\min_{\hat{\boldsymbol{\mathcal{X}}}} \left\| \boldsymbol{\mathcal{X}} - \hat{\boldsymbol{\mathcal{X}}} \right\|_F$, where $\hat{\boldsymbol{\mathcal{X}}}$ is the approximated tensor of $\boldsymbol{\mathcal{X}}$ represented in the tensor decomposition format. For all the three decomposition formats, this problem is generally NP-hard [28].

One widely-used numerical algorithm to optimize tensor decompositions is alternating least squares (ALS). The ALS algorithm consists of iterations, and each iteration is also called a sweep in this dissertation. Each ALS iteration solves multiple linear least squares subproblems, and each subproblem updates part of the variables while keeping other variables fixed. For CP decomposition of the order N tensor, each sweep of the commonly-used ALS algorithm (CP-ALS) contains N subproblems, and the ith subproblem optimizes the ith factor matrix $\mathbf{A}^{(i)}$ while keeping other factor matrices fixed. The common ALS method for Tucker decomposition [5], [29], [30] is called *higher-order orthogonal iteration* (HOOI). Each HOOI sweep contains N subproblems, and the ith subproblem proceeds by updating ith factor matrix $\mathbf{A}^{(i)}$ along with the core tensor $\mathfrak{C}$. For the MPS decomposition, each sweep contains either $N-1$ or $N-2$ subproblems, and the ith subproblem optimizes one or two neighboring tensors in the MPS [31]. For CP-ALS, Tucker HOOI and ALS for the MPS, each subproblem is quadratic, which allows for the minima of the subproblem to be found directly and the decomposition residual decreases monotonically. In Part I and Part II of the dissertation, we propose multiple techniques to accelerate ALS.

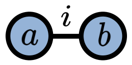
1.2 TENSOR NETWORK PROBLEMS AND THEIR APPLICATIONS

A tensor network is a network of tensors with a specific contraction relation among these tensors. The structure of a tensor network can be described by an undirected hypergraph $G=(V,E)$, also called a tensor diagram [3]. In this diagram, each vertex $v\in V$ represents a tensor $\mathfrak{T}^{(v)}$, with each hyperedge adjacent to it corresponding to the index of one tensor dimension of $\mathfrak{T}^{(v)}$. Let $E(v)=\{e_1,\ldots,e_{n_v}\}$ denote the set of edges incident on v, the $(e_1,e_2,\ldots,e_{n_v})$th element of $\mathfrak{T}$ is presented by $t^{(v)}_{E(v)}$. Hyperedges $e\in E$ may be adjacent to one or more vertices, and those with a dangling end are called uncontracted hyperedges, while those without are called contracted hyperedges. The dimensions represented by contracted hyperedges are summed over in the tensor contraction expression. The tensor $\mathfrak{X}$ represented by the tensor network is expressed as

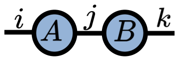
$$x_{E\backslash\hat{E}} = \sum_{e\in\hat{E}}\prod_{v\in V} t^{(v)}_{E(v)}, \tag{1.4}$$

where $\hat{E}$ denotes the set of contracted hyperedges. Tensor diagrams are commonly used to visualize various tensor operations, and we illustrate them in Fig. 1.2. Note that CP, Tucker, and the MPS decompositions can be easily visualized using tensor diagrams.

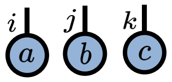
Tensor networks have been originally used in computational quantum physics [32]–[37], where low-rank tensor networks can be used efficiently and accurately to represent quantum states and operators based on the area law. Recently, tensor networks are also widely used in simulating quantum computers [38]–[41], neural networks [42], and kernel-based statistical learning [43]–[46]. Below we introduce commonly-discussed tensor network problems and



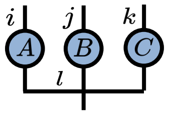
(a) Inner product: $x = \sum_i a_i b_i$



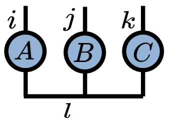
(b) Matmul: $x_{ik} = \sum_j a_{ij} b_{jk}$



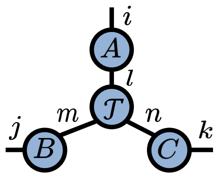
(c) Kronecker product: $x_{ijk} = a_i b_j c_k$



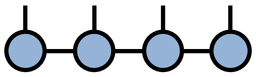
(d) Khatri-Rao product: $x_{ijkl} = a_{il} b_{jl} c_{kl}$



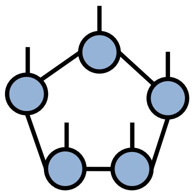
(e) CP decomposition: $x_{ijk} = \sum_l a_{il} b_{jl} c_{kl}$



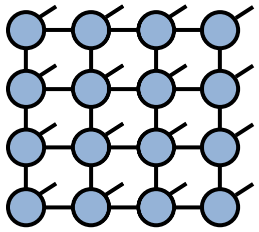
(f) Tucker decomposition: $x_{ijk} = \sum_{l,m,n} a_{il} b_{jm} c_{kn} t_{lmn}$



(g) Matrix product state (MPS)



(h) Tensor ring



(i) Projected entangled pair states

Figure 1.2: Tensor diagram representation of widely used tensor networks.

their applications.

Low-rank tensor network approximation The objective of low-rank tensor network approximation is to generate a low-rank tensor network that approximates a given input tensor network. When the input is a single tensor, this process is known as tensor decomposition.

Low-rank tensor network approximations are widely utilized in both computational quantum physics and the simulation of quantum algorithms [39], [40], [47], [48]. One common problem involves a tensor network structure for a Hamiltonian/operator matrix $\mathbf{H}$ and a quantum state vector $\mathbf{x}$, where the goal is to find another quantum state $\mathbf{y}$ with a specific (low-rank) tensor network structure that approximates $\mathbf{Hx}$. This involves solving the minimization problem $\min_{\mathbf{y}} \|\mathbf{Hx} - \mathbf{y}\|_F$. In computational quantum physics, this subproblem is crucial for obtaining the ground state quantum state and energy of the Hamiltonian $\mathbf{H}$. In the simulation of quantum algorithms, $\mathbf{x}$ represents the initial quantum state of the algorithm, while $\mathbf{H} = \mathbf{U}_1 \ldots \mathbf{U}_n$ is a multiplication of multiple quantum gates (unitaries) $\mathbf{U}_i$. The output $\mathbf{y}$ is then used to approximate the quantum state after applying the gates in $\mathbf{H}$.

Low-rank approximation problems have been widely studied in the context of the matrix product state (MPS) [4], [11], also known as tensor train [12], as well as the matrix product operator (MPO). An MPS is a tensor network with a path structure illustrated in Fig. 1.2g, where each tensor has one uncontracted dimension. It is commonly used to represent wave functions and quantum states $\mathbf{x}$. On the other hand, an MPO is also a tensor network with a linear structure, where each tensor has two uncontracted dimensions, and can be used to represent the Hamiltonian of specific quantum systems $\mathbf{H}$. Various algorithms have been proposed to approximate the MPO-MPS multiplication as another low-rank MPS [47], as illustrated in Fig. 1.3.

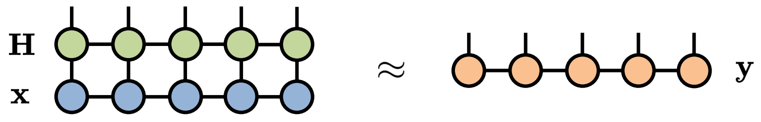


Figure 1.3: Illustration of low-rank tensor network approximation involving an MPS and an MPO.

As in the case of tensor decomposition, alternating least squares (ALS) is commonly used to optimize the problem. When the output tensor network has a tree structure, such as Tucker decomposition (Fig. 1.2f) or an MPS (Fig. 1.2g), the hierarchical singular value decomposition (SVD) algorithm can be efficiently used. Hierarchical SVD is an one-sweep algorithm that updates each tensor in the output tensor network only once. The approach uses SVD to successively decompose vertices (tensors) in the network into a pair of vertices (tensors) connected by a single edge, and it has near-optimal approximation error when the output network has a tree structure [12]. For Tucker decomposition, this algorithm is also called *higher-order SVD* (HOSVD) [49]–[51]. For the MPS (tensor train), this algorithm is usually called TT-SVD [12]. In Part II, we propose efficient sketching-based algorithms to accelerate both ALS and hierarchical SVD.

In many cases, it is desirable to approximate a *part* of the tensor network as a low-rank network. Let $\mathbf{M}$ represent the part of the network that requires approximation, and let $\mathbf{E}$ denote the remaining set of tensors, which is commonly referred to as the *environment*. The optimal way to obtain the low-rank tensor network is by minimizing the global error, which can be achieved by solving $\min_{\mathbf{X}} \|\mathbf{EX} - \mathbf{EM}\|_F$ with the constraint that $\mathbf{X}$ has a specific low-rank tensor network structure. However, 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. In Chapter 8, we explore how such techniques can be automated and how different environment sizes impact the accuracy of the proposed approximate tensor network contraction algorithm.

Tensor network eigenvalue problem The tensor network eigenvalue problem aims to identify the extreme eigenvector $\mathbf{x}$ and eigenvalue of a Hamiltonian $\mathbf{H}$ that has a tensor network structure. Specifically, the problem is to minimize the ratio $\frac{\mathbf{x}^T\mathbf{H}\mathbf{x}}{\mathbf{x}^T\mathbf{x}}$, subject to the constraint that $\mathbf{x}$ has a particular tensor network structure.

When the Hamiltonian $\mathbf{H}$ is an MPO and $\mathbf{x}$ is an MPS, a highly effective algorithm for solving this problem is the *density matrix renormalization group* (DMRG) [37]. DMRG is an alternating minimization algorithm that computes the minimum of the objective with respect to one or two neighboring tensors in the MPS $\mathbf{x}$ during each local step. This process is repeated in sweeps of local steps until the results converge. In Chapter 4, we propose an automatic differentiation system that automates the implementation of the DMRG algorithm and allows for faster experimentation and prototyping.

Tensor network contraction Tensor network contraction explicitly evaluates the single tensor represented by a given tensor network. When each tensor in the network is dense, 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.

Tensor network contraction has found diverse applications in different fields of research. For instance, in quantum computing, each quantum algorithm can be viewed as a tensor network contraction, making this method a useful tool for simulating quantum computers [38]–[41]. In statistical physics, tensor network contraction has been used to evaluate the classical partition function of physical models defined on specific graphs [52]. Tensor network contraction has also been used for counting satisfying assignments of constraint satisfaction problems (#CSPs) [53]. In this approach, an arbitrary #CSP formula is transformed into a tensor network, where its full contraction yields the number of satisfying assignments of that formula.

Contracting tensor networks with arbitrary structure is #P-hard in the general case [54]–[56], even when the network represents a scalar. The reason for this is that during the contraction of general tensor networks, intermediate tensors with high orders or large dimension sizes can emerge, leading to a substantial computational cost for precise contraction.

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 exponential costs [57]. In Part III of this dissertation, we propose algorithms for performing *approximate* contractions of arbitrary tensor networks. We demonstrate that the techniques we propose improve both the accuracy and efficiency relative to prior approximate contraction approaches, and we also provide theoretical analysis to show the optimality of the proposed techniques.

1.3 PREVIOUS WORK

We provide a review of previous research on the development of efficient algorithms to solve problems introduced in Section 1.1 and Section 1.2. To start, we examine the dimension tree [58], [59] for alternating minimization, which is an amortization strategy that can enhance the leading order cost of tensor decomposition and tensor network eigenvalue problems. Next, we review parallelization strategies for CP decomposition. In recent years, there has been a surge in the use of sketching techniques [60], which utilize randomization to accelerate numerical linear algebra in tensor problems, and we explore this approach. Finally, we provide an overview of algorithms developed for approximate tensor network contractions.

Dimension tree for alternating minimization In each sweep of alternating minimization, many terms necessary to form the subproblems have many equivalent intermediates, and properly amortizing them can greatly save the cost. The widely used caching strategy is called dimension tree for alternating least squares (ALS) of both CP and Tucker decompositions, and below we illustrate the technique on top of CP decomposition.

During each sweep of CP decomposition with ALS, the most computationally expensive step is the right-hand-side constructions for each normal equation in the linear least squares subproblem. This operation, known as the matricized tensor-times Khatri-Rao product (MTTKRP), involves multiple tensor-times-matrix and tensor-times-vector operations. For an order N tensor with modes of dimension s and the CP rank R, the leading order cost of the MTTKRP operation[1] is $2s^NR$. If we naively compute N MTTKRPs in each ALS sweep, the cost would be $2Ns^NR$. To address this issue, the dimension tree data structure partitions the mode indices of the order N tensor hierarchically and constructs the intermediate tensors accordingly [58], [59], [62]. This technique reduces the leading order cost to $4s^NR$. In Fig. 1.4, we illustrate the tensor diagram of the MTTKRP operation and the dimension

[1]In the complexity analysis throughout the thesis, we assume the classical matrix multiplications rather than fast algorithms such as Strassen's algorithm [61] are employed.

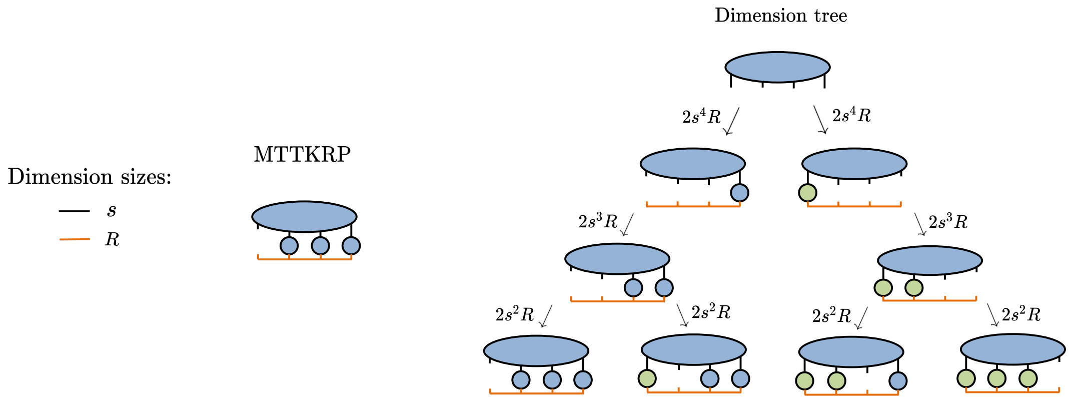


Figure 1.4: Tensor diagram representation of the MTTKRP operation and the dimension tree used in CP decomposition with ALS. In the dimension tree illustration, the matrices denoted by blue vertices represent the factor matrices generated before the current ALS sweep, while the green vertices represent the factor matrices generated during the current ALS sweep.

tree. In Chapter 2, the dimension tree strategy is efficiently applied on a newly-introduced approximation algorithm, pairwise perturbation. In Chapter 3, we introduce a novel dimension tree algorithm that offers superior leading order cost compared to existing dimension tree algorithms. Additionally, we propose an automatic differentiation system to automate the implementation of dimension trees for arbitrary tensor decomposition in Chapter 4.

Parallelization strategies for CP decomposition Parallelization strategies for CP decomposition have been developed both for dense tensors on GPUs [63] and distributed memory systems [62], [64], as well as for sparse tensors on shared memory systems [65], [66] and distributed memory systems [67]–[69]. Communication lower bounds for a single dense MTTKRP computation have been derived in [70], [71].

To parallelize CP decomposition with ALS, it is crucial to parallelize the MTTKRP operation. While one naive approach under a distributed parallel setting is to parallelize each tensor-times-matrix and tensor-times-vector operation using libraries like ScaLAPACK [72], this method results in large data movement across processors and inefficient communication. In recent studies [62], [73], a communication-efficient distributed parallel algorithm is proposed for the low-rank CP decomposition, where the input data tensor is distributed across a multidimensional processor grid, and only small factor matrices are communicated for MTTKRP calculations. In Chapter 3, we introduce a new communication-efficient CP-ALS algorithm that utilizes this strategy, and show how it can be applied to efficiently parallelize pairwise perturbation.

Sketching for tensor decompositions and tensor networks Sketching techniques, which randomly project high-dimensional data onto lower dimensional spaces while still preserving relevant information in the data [74], have been widely used in numerical linear algebra, including for regression, low-rank approximation, and matrix multiplication [60]. One key step of sketching algorithms is to design an embedding matrix $\mathbf{S} \in \mathbb{R}^{m \times n}$ with $m \ll n$ for specific inputs $\mathbf{x} \in \mathbb{R}^n$, such that the projected vector norm is $(1 \pm \epsilon)$-close to the input vector, $\|\mathbf{Sx}\|_2 = (1 \pm \epsilon)\|\mathbf{x}\|_2$, with probability at least $1 - \delta$, and the multiplication $\mathbf{Sx}$ can be computationally efficient. $\mathbf{S}$ is commonly chosen as a random matrix with each element being an i.i.d. Gaussian variable when $\mathbf{x}$ is dense, or a random sparse matrix when $\mathbf{x}$ is sparse, etc.

There has been a recent interest in designing embeddings $\mathbf{S}$ that can efficiently compute $\mathbf{Sx}$ for specific tensor network structures of $\mathbf{x}$. This is particularly useful in the development of sketched ALS algorithms, which involve solving the sketched linear least squares subproblem $\min_{\mathbf{A}} \|\mathbf{SPA} - \mathbf{SX}\|_F$ instead of the regular linear least squares subproblem $\min_{\mathbf{A}} \|\mathbf{PA} - \mathbf{X}\|_F$. The matrix $\mathbf{P}$ has a Khatri-Rao product (illustrated in Fig. 1.2d) tensor network structure for CP decomposition, and a Kronecker product structure (illustrated in Fig. 1.2c) for Tucker decomposition. In cases where the input tensor matricization $\mathbf{X}$ is sparse, it is also desirable for $\mathbf{S}$ to be sparse, which allows for efficient computation of $\mathbf{SX}$.

Multiple sketching algorithms have been proposed to both Tucker and CP decompositions in several previous works. For Tucker decomposition, methods introduced in [75]–[78] accelerate the traditional HOSVD/HOOI via random projection, where factor matrices are updated based on performing SVD on the matricization of the randomly projected input tensor. However, these methods are computationally inefficient for large sparse tensors. Becker and Malik [79] compute Tucker decomposition via a sketched ALS scheme where in each optimization subproblem, one of the factor matrices or the core tensor is updated. They also solve each sketched linear least squares subproblem via TensorSketch [43], which is a random embedding with a Khatri-Rao product structure. For CP decomposition, Battaglino et al. [80] and Jin et al. [81] introduce a randomized algorithm based on Kronecker fast Johnson-Lindenstrauss Transform (KFJLT) to accelerate CP-ALS. However, KFJLT is effective only for the decomposition of dense tensors. Larsen and Kolda [82] propose to sketch the Khatri-Rao product using an approximate leverage score sampling scheme.

Chapter 5 presents a novel sketching algorithm for accelerating CP-ALS and Tucker-HOOI in scenarios where the input tensor is sparse and the CP and Tucker ranks are low. Additionally, Chapter 6 proposes an efficient algorithm for sketching arbitrary tensor network data using embeddings comprising Gaussian random tensors.

Approximate tensor network contraction algorithms 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 dimensions, leading to significant computational costs for accurate contraction [54]–[56]. To mitigate this issue, a common approach is to represent or approximate large intermediate tensors as (low-rank) tensor networks, which reduces the memory usage and computational overhead for downstream contractions. Common tensor networks used for approximation include the matrix product states (MPS) and the tree tensor networks (TTN) [36].

Efficient approximate contraction algorithms based on MPSs have been proposed for tensor network contractions defined on regular structures such as the Projected Entangled Pair States (PEPS) [33], [35], [83], [84], which has a 2D lattice structure. However, these methods are not easily extendable to other general tensor network structures.

Recent works have proposed approximation algorithms for contracting tensor networks with more general graph structures. For example, [85] approximates each intermediate tensor produced during the contraction path as a binary tree tensor network, while [41] approximates each intermediate tensor as an MPS. In [86], each intermediate tensor is also approximated as an MPS, but the system is designed for the specific unbalanced contraction paths and only targets the approximate contraction of tensor networks defined on planar graphs. Another approach proposed in [87] is to perform low-rank approximation on the remaining tensor network after contractions, rather than on the intermediate tensors. The experimental results demonstrate that this framework is more efficient and accurate than [41]. In Chapter 7 and Chapter 8, we propose multiple new approximate contraction algorithms.

1.4 THESIS GOALS AND CONTRIBUTIONS

In this thesis, we present both efficient algorithms and systems for tensor decompositions and tensor networks. On the algorithmic side, we develop efficient inexact solvers that have lower costs than standard algorithms. As to tensor decompositions, these algorithms have theoretical guarantee on the decomposition accuracy. On the system side, we develop libraries that are both efficient and versatile for tensor network applications. Additionally, these libraries have the ability to automate the algorithmic development for tensor networks, further enhancing their practical utility.

The thesis contains four parts. Part I includes a new inexact solver and an automatic differentiation computer system for accelerating alternating minimization of tensor decompositions. Each subproblem of alternating minimization is a linear least squares problem with the left-hand-side matrix containing a specific tensor network structure. Such problems motivate Part II, which includes our contributions to fast sketching algorithms for tensor decompositions and tensor networks. We change the focus from tensor decompositions to tensor network contractions in Part III, where we introduce efficient algorithms to approximately contract tensor networks. Part IV of the thesis includes applications of tensor decompositions in quantum computing.

Accelerating alternating minimization of tensor decompositions In Part I, we propose a novel inexact solver and an automatic differentiation system for the alternating minimization of tensor decompositions.

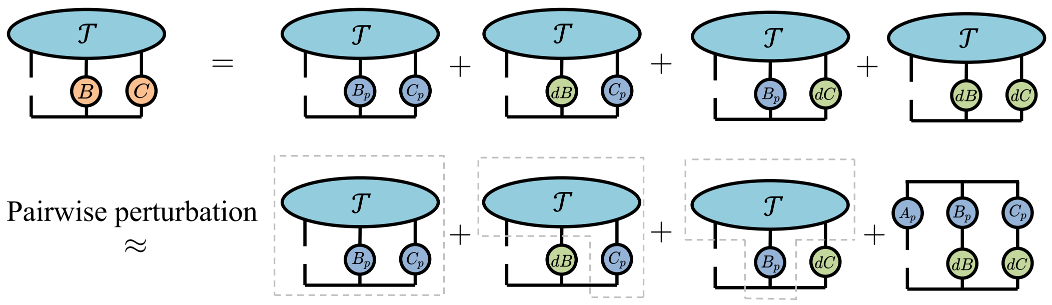


Figure 1.5: Illustration of how pairwise perturbation is used to calculate an approximated MTTKRP between an order three tensor $\boldsymbol{\mathcal{T}}$ and two matrices $\mathbf{B}$ and $\mathbf{C}$. The factor matrices $\mathbf{B}_p$ and $\mathbf{C}_p$ are computed at a previous ALS iteration, and the current factor matrices are obtained by adding small perturbations $d\mathbf{B}$ and $d\mathbf{C}$ to them. The upper part of the figure illustrates the new MTTKRP as a sum of the MTTKRP at the previous ALS iteration and perturbation terms. Pairwise perturbation is used to approximate the last perturbation term, and amortization of the terms shown in dashed blocks save the computational cost.

In Chapter 2, we introduce the new *pairwise perturbation* algorithm that reduces the asymptotic computational cost of both CP-ALS and Tucker HOOI for dense tensors. Pairwise perturbation uses perturbative corrections to the alternating least squares subproblems rather than recomputing the tensor contractions, and an illustration of using it to compute approximated MTTKRP for CP-ALS is shown in Fig. 1.5. This approximation is accurate when the factor matrices are changing little across iterations, which occurs when alternating least squares approaches convergence. We provide a theoretical analysis to bound the approximation error, and our numerical experiments demonstrate that the proposed pairwise perturbation algorithms are easy to control and converge to minima that are as good as ALS. The experimental results show improvements of up to 3.1 with respect to state-of-the-art ALS approaches for various model tensor problems and real datasets.

In Chapter 3, we propose two new parallel algorithms for CP-ALS. We introduce a communication-efficient approach to parallelize pairwise perturbation. We also propose a new caching strategy called *multi-sweep dimension tree* (MSDT) for CP-ALS, which requires the contraction between an order N input tensor and the first-contracted input matrix once every $(N-1)/N$ sweeps. This algorithm reduces the leading order computational cost by a factor of $2(N-1)/N$ relative to the best previously known approach. Our benchmark results show that the per-sweep time achieves 1.25X speed-up for MSDT and 1.94X speed-up for pairwise perturbation compared to the state-of-art dimension trees running on 1024 processors on the Stampede2 supercomputer.

In Chapter 4, we introduce AutoHOOT, an automatic differentiation computer system that targets tensor network applications. In particular, AutoHOOT incorporates tensor algebra-specific transformations, and includes algorithms to automatically generate dimension tree implementations for alternating minimization. Experimental results demonstrate that AutoHOOT outperforms existing automatic differentiation software on both CPUs and GPUs for tensor decomposition and tensor network applications. Additionally, AutoHOOT's performance is comparable to that of other tensor computation libraries that use manually written kernels.

Sketching for tensor decompositions and tensor networks In Part II, we propose novel sketching algorithms for both tensor decompositions and tensor networks.

In Chapter 5, we propose a sketched high-order orthogonal iteration (HOOI) algorithm for low-rank Tucker decomposition of large and sparse tensors. In this algorithm, sketching is directly applied on each *rank-constrained* linear least squares problem $\min_{\mathbf{X},\text{rank}(\mathbf{X})\leq R} \|\mathbf{A}\mathbf{X} - \mathbf{B}\|_F$, with the left-hand-side matrix $\mathbf{A}$ composed of orthonormal columns. Theoretical sketch size upper bounds are provided to achieve $O(\epsilon)$-relative error for each subproblem with two sketching techniques, TensorSketch and leverage score sampling. Experimental results show that this new sketched HOOI algorithm yields decomposition accuracy comparable to the standard HOOI algorithm. In addition, across tested datasets, the new scheme achieves up to 22.0% relative decomposition residual improvement compared to the state-of-the-art sketched Tucker algorithm [79], where a sequence of sketched unconstrained linear least squares problems are solved. This sketched HOOI algorithm is further used to accelerate CP decomposition, by using randomized Tucker compression followed by CP decomposition of the Tucker core tensor. Experimental results show that this algorithm not only converges faster, but also yields more accurate CP decompositions.

In Chapter 6, we propose a cost-efficient algorithm to use Gaussian tensor network embeddings to sketch arbitrary tensor networks, and the embedding structure is shown in Fig. 1.6. Assuming tensor contractions are performed with classical dense matrix multiplication algorithms, this algorithm achieves optimal asymptotic cost in multiple regimes and

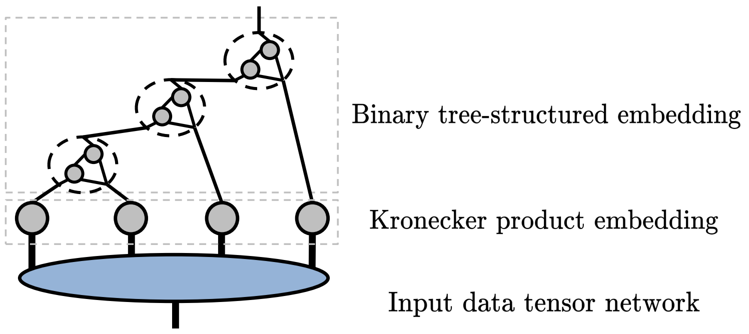

Binary tree-structured embedding
Kronecker product embedding
Input data tensor network

Figure 1.6: Illustration of the proposed cost-efficient embedding. The embedding has a Kronecker part and a binary tree part. Each vertex in the binary tree part contains a pair of tensors.

yields lower computational costs than existing work [44] when sketching a Kronecker product. The proposed algorithm is then used to sketch ALS for CP decomposition and hierarchical SVD for tensor train rounding. When the CP rank is much less than the tensor dimension size, our sketched CP-ALS algorithm is more efficient than the existing sketched CP-ALS algorithm [88]. Our analysis also shows the optimality of an existing algorithm for tensor train rounding [89].

Approximate tensor network contraction algorithms In Part III, we propose two approximate tensor network contraction algorithms.

In Chapter 7, we present a swap-based algorithm named Contracting Arbitrary Tensor Network with Global Ordering (CATN-GO) that can efficiently approximate the contraction of arbitrary tensor networks. Our algorithm builds on the approach outlined in [41], which approximates each intermediate tensor generated during the contraction as an MPS with a bounded rank. When contracting two tensors, the algorithm merges two MPSs, with swaps of adjacent dimensions in the MPS being the bottleneck for complexity.

For a tensor network defined on $G = (V, E)$, we prove that the minimum number of swaps required during contraction is lower bounded by the least number of edge crossings in any vertex linear ordering of the tensor network graph, denoted by $\min_\sigma \mathrm{cr}(G, \sigma)$. A vertex linear ordering $\sigma : V \to \{1, \dots, |V|\}$ assigns each vertex a unique number, and two edges with adjacent vertex orders $(i, j), (k, l)$ cross if $i < k < j < l$. Hence, we reduce the problem of finding the minimum number of swaps to the problem of finding a vertex linear ordering that minimizes the number of edge crossings. In addition, for a fixed vertex ordering σ^V, the number of swaps used in CATN-GO equals the lower bound, $\mathrm{cr}(G, \sigma^V)$, implying optimality for this metric. Furthermore, CATN-GO includes a dynamic programming algorithm to select the contraction tree under a given vertex ordering. This algorithm aims to minimize the overall computational cost, under the assumption that all MPSs have a uniform rank. The uniform rank assumption makes the problem equivalent to minimizing the total length of the MPSs generated during the contractions and has a time complexity of $O(|V|^3|E|)$. Experimental results demonstrate that when contracting tensor networks defined on 3D lattices using the Ising model, our algorithm is more efficient than the algorithm proposed in [41] in terms of speed, and achieves a 5.9X speed-up while maintaining the same accuracy.

In Chapter 8, we propose another approximate tensor network contraction method named Partitioned Contract. Like similar methods proposed in [41], [85], [86], our algorithm approximates each intermediate tensor as a binary tree tensor network. Compared to previous works, the proposed algorithm has the flexibility to incorporate a larger portion of the environment when performing low-rank approximations. 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. In addition, our proposed algorithm includes a cost-efficient density matrix algorithm [90], [91] for approximating a tensor network with a general graph structure into a tree structure. The computational cost of the density matrix algorithm is asymptotically upper-bounded by that of the standard algorithm that uses canonicalization (the process of orthogonalizing all tensors except one in the tenosr network). Experimental results indicate that the proposed algorithm outperforms both algorithms proposed in [41] and [86] when considering tensor networks defined on lattices using the Ising model. Specifically, our approach achieves a 9.2X speed-up while maintaining the same level of accuracy.

Application of tensor decompositions in quantum computing In Part IV, we focus on the application of tensor decomposition techniques in quantum computing. We employ these techniques to simulate quantum algorithms on classical computers and analyze the entanglement properties of specific quantum states.

In Chapter 9, we examine the possibility of simulating a few quantum algorithms by using low-rank CP decomposition to represent the input and all intermediate states of these algorithms. We show that for some of the algorithms, including the Grover's search, quantum Fourier transform, and quantum phase estimation, the low-rank structure of specific input states is preserved, and thus they can be efficiently simulated on a classical computer for specific inputs. However, the rank of the intermediate states in other quantum algorithms can increase rapidly, making efficient simulation more difficult. To some extent, such difficulty reflects the potential advantage or superiority of a quantum computer over a classical computer.

In Chapter 10, we establish bounds on the CP decomposition rank of graph quantum states [92], which have significant applications in quantum information theory due to their connection with measurement-based computing and error correction. Previous studies have uncovered relationships between the graph structure of these states and their multipartite entanglement. Using tensor theory, we provide improved upper bounds on the CP decomposition rank of quantum states that are defined on ring graphs with an odd number of vertices ($|R_{2n+1}\rangle$). Our findings indicate that the rank of $|R_{2n+1}\rangle$ is constrained by $\mathrm{rank}(|R_{2n+1}\rangle) \leq 3 \cdot 2^{n-1}$.

Part I

ACCELERATING ALTERNATING MINIMIZATION OF TENSOR DECOMPOSITIONS

Chapter 2: PAIRWISE PERTURBATION FOR TENSOR DECOMPOSITIONS

This Chapter includes a novel inexact solver, *pairwise perturbation* [93], [94], that asymptotically accelerates alternating least squares (ALS) iteration complexity for CP and Tucker decompositions.

Each iteration of ALS is a sweep over quadratic optimization subproblems for each individual factor matrix composing the decomposition. For both CP and Tucker decomposition, computational cost of each sweep is dominated by the tensor contractions needed to setup the quadratic optimization subproblem for every factor matrix. These contractions are redone at every ALS sweep since they involve the factor matrices, all of which change after each sweep. We propose to circumvent these contractions in the scenario when the factor matrices are changing only slightly at each sweep, which is expected when ALS approaches a local minima. Our method approximates the setup of each quadratic optimization subproblem by computing perturbative corrections to the right-hand side due to the change in each factor matrix since a previous ALS sweep. To do so, pairwise perturbative operators are computed that propagate the change to each factor matrix to the subproblem needed to update each other factor matrix. Computing these operators costs slightly more than a typical ALS sweep. These operators are then reused to *approximately* perform more ALS sweeps until the changes to the factor matrices are deemed large, at which point, regular ALS sweeps are performed. Once the updates performed in these regular sweeps are again small, the pairwise operators are recomputed. Each sweep computed approximately in this way costs asymptotically less than a regular ALS sweep.

Within CP-ALS, the computational bottleneck of each sweep involves an operation called *the matricized tensor-times Khatri-Rao product* (MTTKRP). Similarly, the costliest operation in the ALS-based Tucker decomposition (Tucker-ALS) method is called *the tensor times matrix-chain* (TTMc) product. For an order N tensor with modes of dimension s, approximated computation of ALS sweeps via pairwise perturbation reduces the cost of that sweep from $O\left(s^N R\right)$ to $O\left(s^2 R + sR^2\right)$ for a rank-R CP decomposition and from $O\left(s^N R\right)$ to $O\left(s^2 R^{N-1}\right)$ for a rank-R Tucker decomposition.

To quantify the accuracy of the pairwise perturbation algorithm, in Section 2.3, we provide an error analysis for both MTTKRP and TTMc operations. For both operations, we first view the ALS procedure in terms of pairwise updates, pushing updates to least-squares problems of all factor matrices as soon as any one of them is updated. This reformulation is algebraically equivalent to the original ALS procedure. If the relative change to each factor matrix since pairwise perturbation operators were constructed is bounded by $O(\epsilon)$, we can bound the absolute error of the way pairwise perturbation propagates updates in MTTKRP/TTMc calculations due to changes in any one of the other factor matrices. For order three tensors, this absolute error bound yields a relative error bound that depends on a matrix condition number. For the TTMc operation in Tucker decomposition, we derive a 2-norm relative error bound for the overall TTMc calculations (as opposed to updates thereof) of $O\left(\epsilon^2\right)$ that holds when the residual of the Tucker decomposition is somewhat less than the norm of the original tensor. We also derive a Frobenius norm error bound of $O\left(\epsilon^2(s/R)^{N/2}\right)$ for TTMc, which only assumes that *higher-order singular value decomposition* (HOSVD) [10], [49] is performed to initialize Tucker-ALS (which is typical). In addition, in Section 2.7.4, we show that for the CP decomposition, if the factor matrices have changed by $O(\epsilon)$ in norm, the relative error in pairwise perturbation for the overall MTTKRP calculation is bounded by a term that scales with $O\left(\epsilon^2\right)$ and a tensor condition number. However, we also demonstrate, by connecting to the Hurwitz problem [95], that for large tensors, this condition number is generally infinite.

In order to evaluate the performance benefit of pairwise perturbation, in Section 2.4, we compare per ALS sweep and full decomposition performance using a NumPy-based [96] sequential implementation. Our microbenchmark results compare the performance of one CP-ALS sweep with different input tensor sizes. We consider the initialization sweep, in which the pairwise perturbation operators are calculated, as well as the approximated sweep, in which the operators are not recalculated, of the pairwise perturbation algorithm. These results show that the approximated pairwise perturbation sweeps are up to 6.3X faster than one ALS sweep with the dimension tree algorithm [58], [59], [62], [67], [97]–[99] for an order three tensor with dimension size 960, and up to 33.0X faster than one ALS sweep for an order six tensor. We then study the performance and numerical behavior of pairwise perturbation for the decomposition of synthetic tensors and application datasets. Our experimental results show that pairwise perturbation achieves fitness as high as standard ALS, and achieves speed-ups of up to 3.1X for CP decomposition and up to 1.13X for Tucker decomposition with respect to state of the art ALS algorithms.

We also evaluate the performance of pairwise perturbation based on a distributed-memory parallel implementation on many nodes of an Intel KNL system (Stampede2) using Cyclops Tensor Framework [100] and ScaLAPACK [72] libraries. Our experimental results show that pairwise perturbation achieves fitness as high as standard ALS, and achieves speed-ups of up to 1.75X with respect to a standard ALS implementation on top of the Cyclops library on Stampede2.

2.1 BACKGROUND

This section first outlines the notation used throughout this paper, then outlines the basic alternating least square algorithms for both CP and Tucker decomposition.

2.1.1 Notation and Definitions

Our analysis makes use of tensor algebra in both element-wise equations and specialized notation for tensor operations [5]. For vectors, bold lowercase Roman letters are used, e.g., $\mathbf{x}$. For matrices, bold uppercase Roman letters are used, e.g., $\mathbf{X}$. For tensors, bold calligraphic fonts are used, e.g., $\boldsymbol{\mathcal{X}}$. An order N tensor corresponds to an N-dimensional array with dimensions $s_1 \times \cdots \times s_N$. Elements of vectors, matrices, and tensors are denotes in parentheses, e.g., $\mathbf{x}(i)$ for a vector $\mathbf{x}$, $\mathbf{X}(i,j)$ for a matrix $\mathbf{X}$, and $\boldsymbol{\mathcal{X}}(i,j,k,l)$ for an order 4 tensor $\boldsymbol{\mathcal{X}}$. Columns of a matrix $\mathbf{X}$ are denoted by $\mathbf{x}_i = \mathbf{X}(:,i)$. The mode-$n$ matrix product of an order N tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{s_1\times\cdots\times s_N}$ with a matrix $\mathbf{A} \in \mathbb{R}^{J\times s_n}$ is denoted by $\boldsymbol{\mathcal{X}} \times_n \mathbf{A}$, with the result having dimensions $s_1 \times \cdots \times s_{n-1} \times J \times s_{n+1} \times \cdots \times s_N$. The mode-$n$ vector product of $\boldsymbol{\mathcal{X}}$ with a vector $\mathbf{v} \in \mathbb{R}^{s_n}$ is denoted by $\boldsymbol{\mathcal{X}} \times_n \mathbf{v}^T$, with the result having dimensions $s_1 \times \cdots \times s_{n-1} \times s_{n+1} \times \cdots \times s_N$. Matricization is the process of unfolding a tensor into a matrix. Given a tensor $\boldsymbol{\mathcal{X}}$ the mode-n matricized version is denoted by $\mathbf{X}_{(n)} \in \mathbb{R}^{s_n\times K}$ where $K = \prod_{m=1,m\neq n}^{N} s_m$. We generalize this notation to define the unfoldings of a tensor $\boldsymbol{\mathcal{X}}$ with dimensions $s_1 \times \cdots \times s_N$ into an order $M+1$ tensor, $\boldsymbol{\mathcal{X}}_{(i_1,\ldots,i_M)} \in \mathbb{R}^{s_{i_1}\times\cdots\times s_{i_M}\times K}$, where $K = \prod_{i\in\{1,\ldots,N\}\setminus\{i_1,\ldots,i_M\}} s_i$, e.g., $\boldsymbol{\mathcal{X}}(j,k,l,m) = \boldsymbol{\mathcal{X}}_{(1,3)}\left(j,l,k+(m-1)s_2\right)$. We use parenthesized superscripts as labels for different tensors, e.g., $\boldsymbol{\mathcal{X}}^{(1)}$ and $\boldsymbol{\mathcal{X}}^{(2)}$ are generally unrelated tensors.

The Hadamard product of two matrices $\mathbf{U}, \mathbf{V} \in \mathbb{R}^{I\times J}$ resulting in matrix $\mathbf{W} \in \mathbb{R}^{I\times J}$ is denoted by $\mathbf{W} = \mathbf{U} * \mathbf{V}$, where $\mathbf{W}(i,j) = \mathbf{U}(i,j)\mathbf{V}(i,j)$. We use $\mathop{\Huge *}$ to denote a chain of Hadamard products, e.g. $\mathop{\Huge *}_{i=1}^{n} \mathbf{A}^{(i)} = \mathbf{A}^{(1)} * \cdots * \mathbf{A}^{(n)}$. The outer product of K vectors $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(K)}$ of corresponding sizes $s_1, \ldots, s_K$ is denoted by $\boldsymbol{\mathcal{X}} = \mathbf{u}^{(1)} \circ \cdots \circ \mathbf{u}^{(K)}$ where $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{s_1\times\cdots\times s_K}$ is an order K tensor. The Kronecker product of vectors $\mathbf{u} \in \mathbb{R}^I$ and $\mathbf{v} \in \mathbb{R}^J$ is denoted by $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ where $\mathbf{w} \in \mathbb{R}^{IJ}$. For matrices $\mathbf{A} \in \mathbb{R}^{I\times K}$ and $\mathbf{B} \in \mathbb{R}^{J\times K}$, their Khatri-Rao product results in a matrix of size $(IJ) \times K$ defined by $\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1, \ldots, \mathbf{a}_K \otimes \mathbf{b}_K]$. We use $\bigodot$ to denote a chain of Khatri-Rao products, e.g. $\bigodot_{i=1}^{n} \mathbf{A}^{(i)} = \mathbf{A}^{(1)} \odot \cdots \odot \mathbf{A}^{(n)}$.

2.1.2 CP Decomposition with ALS

The CP tensor decomposition [7], [13] is a higher-order generalization of the matrix singular value decomposition (SVD). The CP decomposition is denoted by

$$\mathcal{X} \approx [\![\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(N)}]\!], \quad \text{where} \quad \mathbf{A}^{(i)} = \left[\mathbf{a}_1^{(i)}, \cdots, \mathbf{a}_R^{(i)}\right], \tag{2.1}$$

and serves to approximate a tensor by a sum of R tensor products of vectors,

$$\mathcal{X} \approx \sum_{r=1}^{R} \mathbf{a}_r^{(1)} \circ \cdots \circ \mathbf{a}_r^{(N)}. \tag{2.2}$$

The CP-ALS method alternates among quadratic optimization problems for each of the factor matrices $\mathbf{A}^{(n)}$, resulting in linear least squares problems for each row,

$$\mathbf{A}_{\text{new}}^{(n)} \mathbf{P}^{(n)T} \cong \mathbf{X}_{(n)}, \tag{2.3}$$

where the matrix $\mathbf{P}^{(n)} \in \mathbb{R}^{I_n \times R}$, where $I_n = s_1 \times \cdots \times s_{n-1} \times s_{n+1} \times \cdots \times s_N$, is formed by Khatri-Rao products of the other factor matrices,

$$\mathbf{P}^{(n)} = \mathbf{A}^{(1)} \odot \cdots \odot \mathbf{A}^{(n-1)} \odot \mathbf{A}^{(n+1)} \odot \cdots \odot \mathbf{A}^{(N)}. \tag{2.4}$$

These linear least squares problems are often solved via the normal equations [5]. We also adopt this strategy here to devise the pairwise perturbation method. The normal equations for the nth factor matrix are

$$\mathbf{A}_{\text{new}}^{(n)} \mathbf{\Gamma}^{(n)} = \mathbf{X}_{(n)} \mathbf{P}^{(n)}, \tag{2.5}$$

where $\mathbf{\Gamma} \in \mathbb{R}^{R \times R}$ can be computed via

$$\mathbf{\Gamma}^{(n)} = \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(n-1)} * \mathbf{S}^{(n+1)} * \cdots * \mathbf{S}^{(N)}, \quad \text{with each} \quad \mathbf{S}^{(i)} = \mathbf{A}^{(i)T}\mathbf{A}^{(i)}. \tag{2.6}$$

These equations also give the nth component of the optimality conditions for the unconstrained minimization of the nonlinear objective function,

$$f\left(\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}\right) = \frac{1}{2}\left\|\mathcal{X} - \left[\!\left[\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(N)}\right]\!\right]\right\|_F^2, \tag{2.7}$$

for which the nth component of the gradient is

$$\frac{\partial f}{\partial \mathbf{A}^{(n)}} = \mathbf{G}^{(n)} = \mathbf{A}^{(n)}\mathbf{\Gamma}^{(n)} - \mathbf{X}_{(n)}\mathbf{P}^{(n)} = \left(\mathbf{A}^{(n)} - \mathbf{A}_{\text{new}}^{(n)}\right)\mathbf{\Gamma}^{(n)}. \tag{2.8}$$

Algorithm 2.1 presents the basic ALS method described above, keeping track of the Frobenius norm of the N components of the overall gradient to ascertain convergence.

Algorithm 2.1: CP-ALS: ALS procedure for CP decomposition

```
1: Input: Tensor $\mathcal{X} \in \mathbb{R}^{s_1 \times \cdots s_N}$, stopping criteria $\Delta$
2: Initialize $[\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$ as uniformly distributed random matrices within $[0, 1]$, initialize $\mathbf{G}^{(n)} \leftarrow \mathbf{A}^{(n)}$, $\mathbf{S}^{(n)} \leftarrow \mathbf{A}^{(n)T}\mathbf{A}^{(n)}$ for $n \in \{1, \ldots, N\}$
3: while $\sum_{i=1}^{N} ||\mathbf{G}^{(i)}||_F > \Delta \|\mathcal{X}\|_F$ do
4:     for $n \in \{1, \ldots, N\}$ do
5:         $\mathbf{\Gamma}^{(n)} \leftarrow \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(n-1)} * \mathbf{S}^{(n+1)} * \cdots * \mathbf{S}^{(N)}$
6:         Update $\mathbf{M}^{(n)}$ based on the dimension tree algorithm shown in Fig. 2.1
7:         $\mathbf{A}^{(n)}_{\text{new}} \leftarrow \mathbf{M}^{(n)}\mathbf{\Gamma}^{(n)\dagger}$
8:         $\mathbf{G}^{(n)} \leftarrow \left(\mathbf{A}^{(n)} - \mathbf{A}^{(n)}_{\text{new}}\right)\mathbf{\Gamma}^{(n)}$
9:         $\mathbf{A}^{(n)} \leftarrow \mathbf{A}^{(n)}_{\text{new}}$
10:        $\mathbf{S}^{(n)} \leftarrow \mathbf{A}^{(n)T}\mathbf{A}^{(n)}$
11:    end for
12: end while
13: return $[\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$
```

The *Matricized Tensor Times Khatri-Rao Product* or MTTKRP computation, $\mathbf{M}^{(n)} = \mathbf{X}_{(n)}\mathbf{P}^{(n)}$, is the main computational bottleneck of CP-ALS [101]. The computational cost of MTTKRP is $\Theta(s^N R)$ if $s_n = s$ for all $n \in \{1, \ldots, N\}$. With the dimension tree algorithm, which will be detailed in Section 2.1.4, the computational complexity for all the MTTKRP calculations in one ALS sweep is $4s^N R$ to leading order in s. The normal equations worsen the conditioning, but are advantageous for CP-ALS, since $\mathbf{\Gamma}^{(n)}$ can be computed and inverted in just $O(s^2 R + R^3)$ cost and the MTTKRP can be amortized by dimension trees. If QR is used instead of the normal equations, the product of $\mathbf{Q}$ with the right-hand sides would have the cost $2s^N R$ and would need to be done for each linear least squares problem, increasing the overall leading order cost by a factor of $N/2$.

2.1.3 Tucker Decomposition with ALS

In this section we review the ALS method for computing a low-rank Tucker decomposition of a tensor [10]. Tucker decomposition approximates a tensor by a core tensor contracted by matrices with orthonormal columns along each mode. The Tucker decomposition is given by

$$\mathcal{X} \approx [\![\mathcal{G}; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!] = \mathcal{G} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)}. \tag{2.9}$$

Algorithm 2.2: Tucker-ALS: ALS procedure for Tucker decomposition

```
 1: Input: Tensor $\mathcal{X} \in \mathbb{R}^{s_1 \times \cdots \times s_N}$, decomposition ranks $\{R_1, \dots, R_N\}$, stopping criteria $\Delta$
 2: Initialize $[[\mathcal{G}; \mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}]]$ using HOSVD, initialize $\mathcal{F} \leftarrow \mathcal{G}$
 3: while $||\mathcal{F}||_F > \Delta \|\mathcal{X}\|_F$ do
 4:     for $n \in \{1, \dots, N\}$ do
 5:         Update $\mathcal{Y}^{(n)}$ based on the dimension tree algorithm
 6:         $\mathbf{A}^{(n)} \leftarrow R_n$ leading left singular vectors of $\mathbf{Y}^{(n)}_{(n)}$
 7:     end for
 8:     $\mathcal{G}_{\text{new}} \leftarrow \mathcal{Y}^{(N)} \times_N \mathbf{A}^{(N)T}$
 9:     $\mathcal{F} \leftarrow \mathcal{G}_{\text{new}} - \mathcal{G}$
10:     $\mathcal{G} \leftarrow \mathcal{G}_{\text{new}}$
11: end while
12: return $\left[\left[\mathcal{G}; \mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}\right]\right]$
```

The corresponding element-wise expression is

$$\mathcal{X}(x_1, \dots, x_N) \approx \sum_{\{z_1, \dots, z_N\}} \mathcal{G}(z_1, \dots, z_N) \prod_{r \in \{1, \dots, N\}} \mathbf{A}^{(r)}(x_r, z_r) . \tag{2.10}$$

The core tensor $\mathcal{G}$ is of order N with dimensions (Tucker ranks) $R_1 \times \cdots \times R_N$ (throughout error and cost analysis we assume each $R_n = R$ for $n \in \{1, \dots, N\}$). The matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{s_n \times R_n}$ have orthonormal columns.

The *higher-order singular value decomposition* (HOSVD) [10], [49] computes the leading left singular vectors of each one-mode unfolding of $\mathcal{X}$, providing a good starting point for the Tucker-ALS algorithm. The classical HOSVD computes the truncated SVD of $\mathbf{X}_{(n)} \approx \mathbf{U}^{(n)}\mathbf{\Sigma}^{(n)}\mathbf{V}^{(n)T}$ and sets $\mathbf{A}^{(n)} = \mathbf{U}^{(n)}$ for $n \in \{1, \dots, N\}$. The interlaced HOSVD [50], [51] instead computes the truncated SVD of

$$\mathbf{Z}^{(n)}_{(n)} = \mathbf{U}^{(n)}\mathbf{\Sigma}^{(n)}\mathbf{V}^{(n)T} \quad \text{where} \quad \mathcal{Z}^{(1)} = \mathcal{X} \quad \text{and} \quad \mathbf{Z}^{(n+1)}_{(n)} = \mathbf{\Sigma}^{(n)}\mathbf{V}^{(n)T}. \tag{2.11}$$

The interlaced HOSVD is cheaper, since the size of each $\mathcal{Z}^{(n)}$ is $s^{N-n+1}R^{n-1}$.

The ALS method for Tucker decomposition [5], [29], [30], which is also called the *higher-order orthogonal iteration* (HOOI), then proceeds by fixing all except one factor matrix, and computing a low-rank matrix factorization to update that factor matrix and the core tensor. To update the nth factor matrix, Tucker-ALS factorizes

$$\mathcal{Y}^{(n)} = \mathcal{X} \times_1 \mathbf{A}^{(1)T} \cdots \times_{n-1} \mathbf{A}^{(n-1)T} \times_{n+1} \mathbf{A}^{(n+1)T} \cdots \times_N \mathbf{A}^{(N)T}, \tag{2.12}$$

which is called the *Tensor Times Matrix-chain* or TTMc, into a product of an matrix with orthonormal columns $\mathbf{A}^{(n)}$ and the core tensor $\boldsymbol{\mathcal{G}}$, so that $\mathbf{Y}^{(n)}_{(n)} \approx \mathbf{A}^{(n)}\mathbf{G}_{(n)}$. This factorization can be done by taking $\mathbf{A}^{(n)}$ to be the R_n leading left singular vectors of $\mathbf{Y}^{(n)}_{(n)}$. This Tucker-ALS procedure is given in Algorithm 2.2.

As in previous work [102], [103], our implementation computes these singular vectors by finding the left eigenvectors of the Gram matrix $\mathbf{W} = \mathbf{Y}^{(n)}_{(n)}\mathbf{Y}^{(n)T}_{(n)}$. Computing the Gram matrix sacrifices some numerical stability, but avoids a large SVD and provides consistency of the signs of the singular vectors across ALS sweeps.

2.1.4 The Dimension Tree Algorithm

For CP-ALS, the tensor contractions for MTTKRP can be amortized across the linear least squares problems necessary for a given ALS sweep (for loop iteration in Algorithm 2.1). Such amortization techniques are referred to as dimension tree algorithms and a variety of dimension trees have been studied to minimize costs [58], [59], [62], [67], [97]–[99]. As our analysis focuses on leading order cost in s, simple binary dimension trees are an optimal choice. These dimension trees for $N = 3, 4$ are illustrated in Fig. 2.1a and Fig. 2.1b. We define the partially contracted MTTKRP intermediates $\boldsymbol{\mathcal{M}}^{(i_1,i_2,\ldots,i_m)}$ therein as follows,

$$\boldsymbol{\mathcal{M}}^{(i_1,i_2,\ldots,i_m)} = \boldsymbol{\mathcal{X}}_{(i_1,i_2,\ldots,i_m)} \bigodot_{j\in\{1,\ldots,N\}\setminus\{i_1,i_2,\ldots,i_m\}} \mathbf{A}^{(j)}. \tag{2.13}$$

Elementwise,

$$\begin{aligned}&\boldsymbol{\mathcal{M}}^{(i_1,i_2,\ldots,i_m)}(x_{i_1}, x_{i_2}, \ldots, x_{i_m}, k)\\ &= \sum_{\{x_1,\ldots,x_N\}\setminus\{x_{i_1},x_{i_2},\ldots,x_{i_m}\}} \boldsymbol{\mathcal{X}}(x_1,\ldots,x_N) \prod_{r\in\{1,\ldots,N\}\setminus\{i_1,i_2,\ldots,i_m\}} \mathbf{A}^{(r)}(x_r, k),\end{aligned} \tag{2.14}$$

where $\boldsymbol{\mathcal{M}}^{(1,\ldots,N)}$ is the input tensor $\boldsymbol{\mathcal{X}}$. The first level contractions (contractions between the input tensor and one factor matrix) can be done via matrix multiplications between the reshaped input tensor and the factor matrix. These contractions have a cost of $O\left(s^N R\right)$ and are generally the most time-consuming part of ALS. Other contractions (transforming one intermediate into another intermediate) can be done via batched matrix-vector products, and the complexity of an ith level contraction is $O\left(s^{N+1-i} R\right)$. Because two first level contractions are necessary for the construction of tree dimension tree, as is illustrated in Fig. 2.1a and Fig. 2.1b, to calculate all the $\mathbf{M}^{(n)}$ in one ALS sweep, to leading order in s, the computational complexity is $4s^N R$.

For Tucker-ALS, The *Tensor Times Matrix-chain* or TTMc that computes each $\boldsymbol{\mathcal{Y}}^{(n)}$ is the main computational bottleneck of Tucker-ALS [104] and can also be amortized by the dimension tree. The intermediates for Tucker dimension tree are the partially contracted TTMc, $\boldsymbol{\mathcal{Y}}^{(i_1,i_2,\ldots,i_m)}$, defined as follows,

$$\boldsymbol{\mathcal{Y}}^{(i_1,i_2,\ldots,i_m)} = \boldsymbol{\mathcal{X}} \bigtimes_{j\in\{1,\ldots,N\}\setminus\{i_1,i_2,\ldots,i_m\}} \mathbf{A}^{(j)T}, \tag{2.15}$$

where $\boldsymbol{\mathcal{X}}$ is contracted with all the matrices $\mathbf{A}^{(j)}$ except $\mathbf{A}^{(i_1)},\ldots,\mathbf{A}^{(i_m)}$. Each contraction can be done via matrix multiplications, and the complexity of an ith level contraction is $O\left(s^{N+1-i}R^i\right)$. Similar to CP-ALS, to calculate all the $\boldsymbol{\mathcal{Y}}^{(n)}$ in one ALS sweep, to leading order in s, the computational complexity is $4s^N R$.

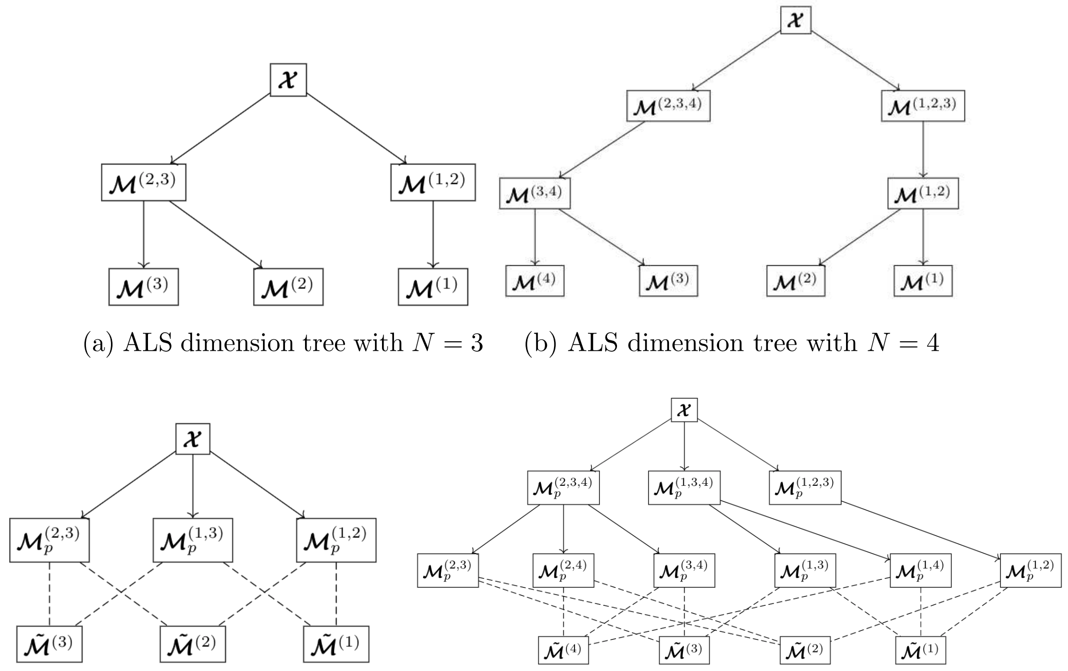
2.2 PAIRWISE PERTURBATION ALGORITHMS

We now introduce a pairwise perturbation (PP) algorithm to accelerate the ALS procedure when the iterative optimization steps are approaching a local minimum. We first derive the approximation for order three tensors, then generalize the algorithm to order N tensors. The key idea of the pairwise perturbation method is to compute *pairwise perturbation operators*, which correlate a pair of factor matrices. These tensors are then used to repeatedly update the quadratic subproblems for each tensor. As we will show, these updates are provably accurate if the factor matrices do not change significantly since their state at the time of formation of the pairwise perturbation operators.

2.2.1 Pairwise Perturbation for Order Three Tensors

CP-ALS The pairwise perturbation procedure for CP-ALS approximates the MTTKRP outputs. Consider an order three equi-dimensional tensor with size in each mode s and CP rank R, the first mode MTTKRP can be expressed as $\mathbf{M}^{(1)} = \mathbf{X}_{(1)}\left(\mathbf{A}^{(2)} \odot \mathbf{A}^{(3)}\right)$. Let $\mathbf{A}_p^{(n)}$ denote the $\mathbf{A}^{(n)}$ calculated with regular ALS at some number of sweeps prior to the current one. Then $\mathbf{A}^{(n)}$ at the current sweep can be expressed as

$$\mathbf{A}^{(n)} = \mathbf{A}_p^{(n)} + d\mathbf{A}^{(n)}, \tag{2.16}$$



(a) ALS dimension tree with $N = 3$ (b) ALS dimension tree with $N = 4$

(c) PP dimension tree with $N = 3$ (d) PP dimension tree with $N = 4$

Figure 2.1: Dimension trees for ALS and pairwise perturbation. In (c)(d), the solid arrows denote the data dependencies in building pairwise perturbation operators, and is calculated in the PP initialization step. The dashed lines denote the data dependencies in the PP approximated step calculations.

and $\mathbf{M}^{(1)}$ can be expressed as $\mathbf{M}^{(1)} =$

$$\underbrace{\mathbf{X}_{(1)}\left(\mathbf{A}_p^{(2)} \odot \mathbf{A}_p^{(3)}\right) + \mathbf{X}_{(1)}\left(\mathbf{A}_p^{(2)} \odot d\mathbf{A}^{(3)}\right) + \mathbf{X}_{(1)}\left(d\mathbf{A}^{(2)} \odot \mathbf{A}_p^{(3)}\right)}_{\mathbf{U}^{(1)}} + \mathbf{X}_{(1)}\left(d\mathbf{A}^{(2)} \odot d\mathbf{A}^{(3)}\right). \tag{2.17}$$

The pairwise perturbation procedure for CP-ALS approximates $\mathbf{M}^{(1)}$ with $\tilde{\mathbf{M}}^{(1)} = \mathbf{U}^{(1)} + \mathbf{V}^{(1)}$, where $\mathbf{U}^{(1)}$ is the first three terms in (2.17) and $\mathbf{V}^{(1)}$ approximates the final term through approximating the input tensor $\mathcal{X}$ by its approximate CP decomposition,

$$\begin{aligned}\mathbf{X}_{(1)}\left(d\mathbf{A}^{(2)} \odot d\mathbf{A}^{(3)}\right) \approx \mathbf{V}^{(1)} &= \left([\![\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \mathbf{A}^{(3)}]\!]\right)_{(1)}\left(d\mathbf{A}^{(2)} \odot d\mathbf{A}^{(3)}\right) \\ &= \mathbf{A}^{(1)}\Big(\left(\mathbf{A}^{(2)T} d\mathbf{A}^{(2)}\right) * \left(\mathbf{A}^{(3)T} d\mathbf{A}^{(3)}\right)\Big),\end{aligned} \tag{2.18}$$

which can be calculated with the cost of $O\left(sR^2\right)$. The remaining error term is

$$\left(\mathcal{X} - [\![\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \mathbf{A}^{(3)}]\!]\right)_{(1)}\left(d\mathbf{A}^{(2)} \odot d\mathbf{A}^{(3)}\right). \tag{2.19}$$

Therefore, the norm of the error scales as $O\left(C\epsilon^2\right)$ if each $||d\mathbf{A}^{(i)}||_2 \leq \epsilon$ and the decomposition residual norm is bounded by C.

The approximated MTTKRP, $\tilde{\mathbf{M}}^{(1)}$, can be rewritten as a function of $\boldsymbol{\mathcal{M}}_p^{(i_1,i_2,\ldots,i_m)}$, which is defined in the same way as $\boldsymbol{\mathcal{M}}^{(i_1,i_2,\ldots,i_m)}$ in (2.13) except that $\boldsymbol{\mathcal{X}}$ is contracted with $\mathbf{A}_p^{(j)}$ for $j \in \{1,\ldots,N\} \setminus \{i_1,i_2,\ldots,i_m\}$, thus $\mathbf{M}_p^{(1)} = \boldsymbol{\mathcal{X}}_{(1)}(\mathbf{A}_p^{(2)} \odot \mathbf{A}_p^{(3)})$, $\mathbf{M}_p^{(1,2)} = \boldsymbol{\mathcal{X}}_{(1,2)}\mathbf{A}_p^{(3)}$, $\mathbf{M}_p^{(1,3)} = \boldsymbol{\mathcal{X}}_{(1,3)}\mathbf{A}_p^{(2)}$. For each $x \in \{1,\ldots,s\}$ and $k \in \{1,\ldots,R\}$,

$$\begin{aligned}\tilde{\mathbf{M}}^{(1)}(x,k) = \mathbf{M}_p^{(1)}(x,k) &+ \sum_{y=1}^{s} \boldsymbol{\mathcal{M}}_p^{(1,2)}(x,y,k)\, d\mathbf{A}^{(2)}(y,k) \\ &+ \sum_{y=1}^{s} \boldsymbol{\mathcal{M}}_p^{(1,3)}(x,y,k)\, d\mathbf{A}^{(3)}(y,k) + \mathbf{V}^{(1)}(x,k).\end{aligned} \tag{2.20}$$

PP has two steps: the initialization step, where the terms $\mathbf{M}_p^{(1)}$ and pairwise perturbation operators $\boldsymbol{\mathcal{M}}_p^{(1,2)}$, $\boldsymbol{\mathcal{M}}_p^{(1,3)}$ are calculated, and the approximated step, where these terms are used in the equation above to calculate $\tilde{\mathbf{M}}^{(1)}$. Using the dimension tree structure shown in Fig. 2.1c, the initialization step for all the three modes can be done with the leading order cost of $6s^3R$, 1.5X the cost of the ALS dimension tree. Each approximated step for all the modes can be done with the leading order cost of $3\left(4s^2R + 6sR^2\right)$ overall.

Tucker-ALS We derive a similar pairwise perturbation algorithm for order three Tucker-ALS. The first mode of TTMc can be expressed as $\boldsymbol{\mathcal{Y}}^{(1)} = \boldsymbol{\mathcal{Y}} \times_2 \mathbf{A}^{(2)T} \times_3 \mathbf{A}^{(3)T}$. PP approximates $\boldsymbol{\mathcal{Y}}^{(1)}$ with

$$\tilde{\boldsymbol{\mathcal{Y}}}^{(1)} = \boldsymbol{\mathcal{X}} \times_2 \mathbf{A}_p^{(2)T} \times_3 \mathbf{A}_p^{(3)T} + \boldsymbol{\mathcal{X}} \times_2 \mathbf{A}_p^{(2)T} \times_3 d\mathbf{A}^{(3)T} + \boldsymbol{\mathcal{X}} \times_2 d\mathbf{A}^{(2)T} \times_3 \mathbf{A}_p^{(3)T}, \tag{2.21}$$

and the error term is $\boldsymbol{\mathcal{X}} \times_2 d\mathbf{A}^{(2)T} \times_3 d\mathbf{A}^{(3)T}$. The expression above can be rewritten as a function of $\boldsymbol{\mathcal{Y}}_p^{(i_1,i_2,\ldots,i_m)}$, which is defined in the same way as $\boldsymbol{\mathcal{Y}}^{(i_1,i_2,\ldots,i_m)}$ except that $\boldsymbol{\mathcal{X}}$ is contracted with $\mathbf{A}_p^{(j)}$ for $\boldsymbol{\mathcal{Y}}_p^{(i_1,i_2,\ldots,i_m)}$,

$$\tilde{\boldsymbol{\mathcal{Y}}}^{(1)} = \boldsymbol{\mathcal{Y}}_p^{(1)} + \boldsymbol{\mathcal{Y}}_p^{(1,2)} \times_2 d\mathbf{A}^{(2)T} + \boldsymbol{\mathcal{Y}}_p^{(1,3)} \times_3 d\mathbf{A}^{(3)T}. \tag{2.22}$$

Using the dimension tree structure, the initialization step for all the three modes can be done with the leading order cost of $6s^3R$, 1.5X the cost of the ALS dimension tree. Each approximated step for all the modes can be done with the leading order cost of $12s^2R^2$ overall.

2.2.2 General Pairwise Perturbation Algorithm

We now generalize PP to order N tensors.

CP-ALS The MTTKRP in nth mode, $\mathbf{M}^{(n)}$, can be expressed as

$$\mathbf{M}^{(n)} = \mathbf{X}_{(n)} \bigodot_{i=1,i\neq n}^{N} \left(\mathbf{A}_p^{(i)} + d\mathbf{A}^{(i)}\right). \tag{2.23}$$

$\mathbf{M}^{(n)}$ can be expressed as a function of $\mathcal{M}_p^{(i_1,i_2,\ldots,i_m)}$ as follows,

$$\begin{aligned}\mathbf{M}^{(n)}(y,k) =& \mathbf{M}_p^{(n)}(y,k) + \sum_{i=1,i\neq n}^{N}\sum_{x=1}^{s_i} \mathcal{M}_p^{(i,n)}(x,y,k)d\mathbf{A}^{(i)}(x,k) + \\ & \sum_{i=1,i\neq n}^{N}\sum_{j=i+1,j\neq n}^{N}\sum_{x=1}^{s_i}\sum_{z=1}^{s_j} \mathcal{M}_p^{(i,j,n)}(x,z,y,k)d\mathbf{A}^{(i)}(x,k)d\mathbf{A}^{(j)}(z,k) + \cdots .\end{aligned} \tag{2.24}$$

From the above expression we observe that, except the first two terms, all terms include the contraction between tensor $\mathcal{M}_p^{(i_1,i_2,\ldots,i_m)}$ and at least two matrices $d\mathbf{A}^{(i)}$, so that their norm scales quadratically with the norm of the perturbative updates $d\mathbf{A}^{(i)}$. Therefore, their norm scales as $O\left(\epsilon^2\right)$ if $||d\mathbf{A}^{(i)}||_2 \leq \epsilon$. The pairwise perturbation algorithm obtains an effective approximation by keeping the first two terms (these terms are illustrated in Fig. 2.1d for an order four tensor), and approximating the input tensor using its approximate CP decomposition in the third term to lower the error to a greater extent. For each $y \in \{1,\ldots,s_n\}$ and $k \in \{1,\ldots,R\}$,

$$\begin{aligned}\tilde{\mathbf{M}}^{(n)}(y,k) = \mathbf{M}_p^{(n)}(y,k) + \sum_{i=1,i\neq n}^{N}\sum_{x=1}^{s_i} \mathcal{M}_p^{(i,n)}(x,y,k)d\mathbf{A}^{(i)}(x,k) + \sum_{i,j=1,i,j\neq n,i\neq j}^{N} \mathbf{V}^{(n,i,j)}(y,k), \\ \text{where} \quad \mathbf{M}_p^{(n)} = \mathbf{X}_{(n)} \bigodot_{i=1,i\neq n}^{N} \mathbf{A}_p^{(i)}, \quad \mathcal{M}_p^{(i,n)} = \mathcal{X}_{(i,n)} \bigodot_{j\in\{1,\ldots,N\}\setminus\{i,n\}}^{N} \mathbf{A}_p^{(j)}, \\ \text{and} \quad \mathbf{V}^{(n,i,j)} = \mathbf{A}^{(n)}\left(\left(\mathbf{A}^{(i)T}d\mathbf{A}^{(i)}\right) * \left(\mathbf{A}^{(j)T}d\mathbf{A}^{(j)}\right) * \mathop{\scalebox{2}{$\ast$}}_{k=1,k\neq i,j,n}^{N} \left(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}\right)\right).\end{aligned} \tag{2.25}$$

We evaluate the benefit of including the $\mathbf{V}^{(n,i,j)}$ correction in Section 2.4.1. Given $\mathcal{M}_p^{(i,n)}$ and $\mathbf{M}_p^{(n)}$, calculation of $\tilde{\mathbf{M}}^{(n)}$ for $n \in \{1,\ldots,N\}$ requires $2N^2\left(s^2R + sR^2\right)$ operations overall. Further, we show in Section 2.3.1 that the column-wise relative approximation error of $\tilde{\mathbf{M}}^{(n)}$

with respect to $\mathbf{M}^{(n)}$ is small if each $||d\mathbf{a}_k^{(n)}||_2/||\mathbf{a}_k^{(n)}||_2$ for $n \in \{1,\dots,N\}, k \in \{1,\dots,R\}$ is sufficiently small. Algorithm 2.3 presents the PP-CP-ALS method described above.

Algorithm 2.3: PP-CP-ALS: Pairwise perturbation procedure for CP-ALS

```
1: Input: tensor $\mathcal{X} \in \mathbb{R}^{s_1\times\cdots\times s_N}$, stopping criteria $\Delta$, PP tolerance $\epsilon < 1$
2: Initialize $[[\mathbf{A}^{(1)},\dots,\mathbf{A}^{(N)}]]$ as uniformly distributed random matrices within $[0,1]$, initialize $\mathbf{G}^{(n)}, d\mathbf{A}^{(n)} \leftarrow \mathbf{A}^{(n)}$, $\mathbf{S}^{(n)} \leftarrow \mathbf{A}^{(n)T}\mathbf{A}^{(n)}$ for $i \in \{1,\dots,N\}$
3: while $\sum_{i=1}^N ||\mathbf{G}^{(i)}||_F > \Delta\|\mathcal{X}\|_F$ do
4:     if $\forall\, i \in \{1,\dots,N\}, ||d\mathbf{A}^{(i)}||_F < \epsilon||\mathbf{A}^{(i)}||_F$ then
5:         Compute $\mathcal{M}_p^{(i,n)}, \mathbf{M}_p^{(n)}$ for $i,n \in \{1,\dots,N\}$ via dimension tree in Section 2.2.2
6:         for $n \in \{1,\dots,N\}$ do
7:             $\mathbf{A}_p^{(n)} \leftarrow \mathbf{A}^{(n)}$, $d\mathbf{A}^{(n)} \leftarrow \mathbf{O}$
8:         end for
9:         while $\sum_{i=1}^N ||\mathbf{G}^{(i)}||_F > \Delta\|\mathcal{X}\|_F$ and $\forall\, i \in \{1,\dots,N\}, ||d\mathbf{A}^{(i)}||_F < \epsilon||\mathbf{A}^{(i)}||_F$ do
10:            for $n \in \{1,\dots,N\}$ do
11:                $\mathbf{\Gamma}^{(n)} \leftarrow \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(n-1)} * \mathbf{S}^{(n+1)} * \cdots * \mathbf{S}^{(N)}$
12:                Update $\tilde{\mathbf{M}}^{(n)}$ based on (2.25)
13:                $\mathbf{A}_{\text{new}}^{(n)} \leftarrow \tilde{\mathbf{M}}^{(n)}\mathbf{\Gamma}^{(n)\dagger}$
14:                $\mathbf{G}^{(n)} \leftarrow \left(\mathbf{A}^{(n)} - \mathbf{A}_{\text{new}}^{(n)}\right)\mathbf{\Gamma}^{(n)}$
15:                $\mathbf{A}^{(n)} \leftarrow \mathbf{A}_{\text{new}}^{(n)}$
16:                $\mathbf{S}^{(n)} \leftarrow \mathbf{A}^{(n)T}\mathbf{A}^{(n)}$
17:                $d\mathbf{A}^{(n)} = \mathbf{A}_{\text{new}}^{(n)} - \mathbf{A}_p^{(n)}$
18:            end for
19:        end while
20:    end if
21:    Perform regular ALS sweep as in Algorithm 2.1, taking $d\mathbf{A}^{(n)} \leftarrow \mathbf{A}_{\text{new}}^{(n)} - \mathbf{A}^{(n)}$ for each $n \in \{1,\dots,N\}$
22: end while
23: return $[[\mathbf{A}^{(1)},\dots,\mathbf{A}^{(N)}]]$
```

Tucker-ALS We derive a similar pairwise perturbation algorithm for Tucker-ALS. Similar to the expression for $\mathbf{M}^{(n)}$ in CP-ALS, $\mathcal{Y}^{(n)}$ can be expressed as

$$\mathcal{Y}^{(n)} = \mathcal{X} \bigtimes_{i=1, i\neq n}^{N} \left(\mathbf{A}_p^{(i)T} + d\mathbf{A}^{(i)T}\right). \tag{2.26}$$

The expression above can be rewritten as a function of $\mathcal{Y}_p^{(i_1,i_2,\dots,i_m)}$,

$$\mathcal{Y}^{(n)}=\mathcal{Y}_p^{(n)}+\sum_{i=1,i\neq n}^{N}\mathcal{Y}_p^{(i,n)}\times_i d\mathbf{A}^{(i)T}+\sum_{i=1,i\neq n}^{N}\sum_{j=i+1,j\neq n}^{N}\mathcal{Y}_p^{(i,j,n)}\times_i d\mathbf{A}^{(i)T}\times_j d\mathbf{A}^{(j)T}+\cdots. \tag{2.27}$$

The pairwise perturbation algorithm again takes only the first order terms in $d\mathbf{A}^{(i)}$, computing

$$\begin{aligned}&\tilde{\mathcal{Y}}^{(n)}=\mathcal{Y}_p^{(n)}+\sum_{i=1,i\neq n}^{N}\mathcal{Y}_p^{(i,n)}\times_i d\mathbf{A}^{(i)T},\\ &\text{where}\quad \mathcal{Y}_p^{(n)}=\mathcal{X}\bigtimes_{l=1,l\neq n}^{N}\mathbf{A}_p^{(l)T}\quad\text{and}\quad \mathcal{Y}_p^{(i,n)}=\mathcal{X}\bigtimes_{j\in\{1,\dots,N\}\setminus\{i,n\}}\mathbf{A}_p^{(j)T}.\end{aligned}\tag{2.28}$$

Given $\mathcal{Y}_p^{(i,n)}$ and $\mathcal{Y}_p^{(n)}$, $\tilde{\mathcal{Y}}^{(n)}$ for $n\in\{1,\dots,N\}$ can be calculated with $2N^2s^2R^{N-1}$ cost overall. In Section 2.3.2, we show that the relative Frobenius norm approximation error of $\tilde{\mathcal{Y}}^{(n)}$ with respect to $\mathcal{Y}^{(n)}$ is small, so long as each $||d\mathbf{A}^{(n)}||_F/||\mathbf{A}^{(n)}||_F$ is sufficiently small. Algorithm 2.4 presents the PP-Tucker-ALS method described above.

Dimension Trees for Pairwise Perturbation Operators Computation of the pairwise perturbation operators $\mathcal{M}_p^{(i,n)}$ and of $\mathcal{M}_p^{(n)}$ can benefit from amortization of common tensor contraction (Khatri-Rao product or multilinear multiplication) subexpressions. In the context of ALS, this technique is known as dimension trees and has been successfully employed to accelerate TTMc and MTTKRP. The same trees can be used for both CP and Tucker, although the tensor intermediates and contraction operations are different (Khatri-Rao products for CP and multilinear multiplication for Tucker). We describe the trees for CP decomposition, computing each $\mathcal{M}_p^{(i,n)}$ and $\mathcal{M}_p^{(n)}$. Fig. 2.1c and Fig. 2.1d describes the dimension tree for $N=3,4$. Our tree constructions assume that the tensors are equidimensional, if this is not the case, the largest dimensions should be contracted first.

The main goal of the dimension tree is to perform a minimal number of contractions to obtain each $\mathcal{M}_p^{(i,n)}$. Each matrix $\mathcal{M}_p^{(n)}$ can be simply obtained by a contraction with $\mathcal{M}_p^{(i,n)}$ for any $i\neq n$. Each level of the tree for $l=1,\dots,N-1$ should contain intermediate tensors containing $N-l+1$ uncontracted modes belonging to the original tensor (the root is the original tensor $\mathcal{X}=\mathcal{M}^{(1,\dots,N)}$). For any pair of the original tensor modes, each level should contain an intermediate for which these modes are uncontracted. Since the leaves at level $l=N-1$ have two uncontracted modes, they will include each $\mathcal{M}_p^{(i,n)}$ for $i<n$ and have $\binom{N}{2}$ tensors overall. At level l it then suffices to compute $\binom{l+1}{2}$ tensors $\mathcal{M}^{(i,j,l+2,l+3,\dots,N)}, \forall i,j\in\{1,\dots,l+1\}, i<j$. Each $\mathcal{M}^{(i,j,l+2,l+3,\dots,N)}$ can be computed by contraction of $\mathcal{M}^{(s,t,v,l+2,l+3,\dots,N)}$ and $\mathbf{A}^{(w)}$ where

Algorithm 2.4: PP-Tucker-ALS: Pairwise perturbation procedure for Tucker-ALS

```
1: Input: tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{s_1\times\cdots\times s_N}$, decomposition ranks $\{R_1,\dots,R_N\}$, stopping criteria $\Delta$,
   PP tolerance $\epsilon$
2: Initialize $[[\boldsymbol{\mathcal{G}};\mathbf{A}^{(1)},\dots,\mathbf{A}^{(N)}]]$ using HOSVD, initialize $d\mathbf{A}^{(n)} \leftarrow \mathbf{A}^{(n)}$ for $i\in\{1,\dots,N\}$,
   initialize $\boldsymbol{\mathcal{F}} \leftarrow \boldsymbol{\mathcal{G}}$
3: while $||\boldsymbol{\mathcal{F}}||_F > \Delta\|\boldsymbol{\mathcal{X}}\|_F$ do
4:     if $\forall\, i\in\{1,\dots,N\}, ||d\mathbf{A}^{(i)}||_F < \epsilon||\mathbf{A}^{(i)}||_F$ then
5:         Compute $\boldsymbol{\mathcal{Y}}_p^{(i,n)}, \boldsymbol{\mathcal{Y}}_p^{(n)}$ for $i,n\in\{1,\dots,N\}$ via dimension tree in Section 2.2.2
6:         for $n\in\{1,\dots,N\}$ do
7:             $\mathbf{A}_p^{(n)} \leftarrow \mathbf{A}^{(n)}$, $d\mathbf{A}^{(n)} \leftarrow \mathbf{O}$
8:         end for
9:         while $||\boldsymbol{\mathcal{F}}||_F > \Delta\|\boldsymbol{\mathcal{X}}\|_F$ and $||\boldsymbol{\mathcal{F}}||_F < \epsilon\|\boldsymbol{\mathcal{X}}\|_F$ do
10:            for $n\in\{1,\dots,N\}$ do
11:                $\boldsymbol{\mathcal{Y}}^{(n)} \leftarrow \boldsymbol{\mathcal{Y}}_p^{(n)} + \sum_{i=1,i\neq n}^{N} \boldsymbol{\mathcal{Y}}_p^{(i,n)} \times_i d\mathbf{A}^{(i)}$
12:                $\mathbf{A}^{(n)} \leftarrow R_n$ leading left singular vectors of $\mathbf{Y}_{(n)}^{(n)}$
13:                $d\mathbf{A}^{(n)} \leftarrow \mathbf{A}^{(n)} - \mathbf{A}_p^{(n)}$
14:            end for
15:            $\boldsymbol{\mathcal{G}}_{\text{new}} \leftarrow \boldsymbol{\mathcal{Y}}^{(N)} \times_N \mathbf{A}^{(N)T}$
16:            $\boldsymbol{\mathcal{F}} \leftarrow \boldsymbol{\mathcal{G}}_{\text{new}} - \boldsymbol{\mathcal{G}}$
17:            $\boldsymbol{\mathcal{G}} \leftarrow \boldsymbol{\mathcal{G}}_{\text{new}}$
18:        end while
19:    end if
20:    Perform regular ALS sweep as in Algorithm 2.2, taking $d\mathbf{A}^{(n)} \leftarrow \mathbf{A}_{\text{new}}^{(n)} - \mathbf{A}^{(n)}$ for each
   $n\in\{1,\dots,N\}$
21:    $\boldsymbol{\mathcal{G}}_{\text{new}} \leftarrow \boldsymbol{\mathcal{Y}}^{(N)} \times_N \mathbf{A}^{(N)T}$
22:    $\boldsymbol{\mathcal{F}} \leftarrow \boldsymbol{\mathcal{G}}_{\text{new}} - \boldsymbol{\mathcal{G}}$
23:    $\boldsymbol{\mathcal{G}} \leftarrow \boldsymbol{\mathcal{G}}_{\text{new}}$
24: end while
25: return $[[\boldsymbol{\mathcal{G}};\mathbf{A}^{(1)},\dots,\mathbf{A}^{(N)}]]$
```

$\{s,t,v\} = \{i,j,w\}$ with $w = \max_{w\in\{l-1,l,l+1\}\backslash\{i,j\}}(w)$ and $s<t<v$.

	DT ALS	PP initialization step	PP approximated step
CP	$4s^NR$	$6s^NR$	$2N^2(s^2R+sR^2)$
Tucker	$4s^NR$	$6s^NR$	$2N^2s^2R^{N-1}$

Table 2.1: Cost comparison between pairwise perturbation algorithm and ALS dimension tree algorithm for CP and Tucker decompositions.

The construction of pairwise perturbation operators for CP decomposition costs

$$2R\sum_{l=2}^{N-1}\binom{l+1}{2}s^{N-l+2} = 6s^N R + 12s^{N-1}R + O\left(s^{N-2}R^2\right). \tag{2.29}$$

The cost to form pairwise perturbation operators for Tucker decomposition is

$$2\sum_{l=2}^{N-1}\binom{l+1}{2}s^{N-l+2}R^{l-1} = 6s^N R + 12s^{N-1}R^2 + O\left(s^{N-2}R^3\right). \tag{2.30}$$

We summarize the leading order computational costs for both algorithms in Table 2.1. The PP initialization step, which involves the PP operator construction and does one more first level contraction, is computationally 1.5X more expensive than the ALS algorithm.

As for the memory footprint, ALS with the best choice of dimension tree requires intermediate tensors of size $O\left(s^{\lceil N/2\rceil}R\right)$. As an example, for the order four case shown in Fig. 2.1b, the first and second level contractions are combined to save memory, so that $\boldsymbol{\mathcal{M}}^{(3,4)}$ and $\boldsymbol{\mathcal{M}}^{(1,2)}$ are stored, both of size $O\left(s^2R\right)$. The PP dimension tree described above and in Fig. 2.1d needs at least $O\left(s^{N-1}R\right)$ auxiliary memory to store the first level contraction results. The memory needed for PP can be reduced similar to ALS. For example, when calculating the PP operator $\boldsymbol{\mathcal{M}}_p^{(1,3)}$ for an order four tensor, we can bypass the first level contraction and save its memory via directly performing a contraction between the input tensor and the Khatri-Rao product output $\mathbf{A}^{(1)}\odot\mathbf{A}^{(3)}$. Combining the first $l \leq N-2$ levels of contractions requires $O\left(s^{N-l}R + N^2s^2R\right)$ auxiliary memory, but incurs a cost of $O\left(l^2s^{N-1}R\right)$.

2.3 ERROR ANALYSIS

In this section, we formally bound the approximation error of the pairwise perturbation algorithm relative to ALS. We show that quadratic optimization problems computed by pairwise perturbation differ only slightly from ALS so long as the factor matrices have not changed significantly since the construction of the pairwise perturbation operators.

2.3.1 CP-ALS

To bound the error of pairwise perturbation, we view the ALS procedure for CP decomposition in terms of pairwise updates (Algorithm 2.5), pushing updates to least-squares problems of all tensors as soon as any one of them is updated. This reformulation is alge-

Algorithm 2.5: CP-ALS: Reinterpreted ALS procedure for CP decomposition

```
1: Input: Tensor $\mathcal{X} \in \mathbb{R}^{s_1\times\cdots\times s_N}$, stopping criteria $\Delta$
2: Initialize $[\![\mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)}]\!]$ as uniformly distributed random matrices within $[0,1]$, initialize $\mathbf{G}^{(n)}, \delta\mathbf{A}^{(n)} \leftarrow \mathbf{A}^{(n)}$, $\mathbf{S}^{(n)} \leftarrow \mathbf{A}^{(n)T}\mathbf{A}^{(n)}$ for $i \in \{1,\ldots,N\}$
3: for $n \in \{1,\ldots,N\}$ do
4:     Update $\mathbf{M}^{(n)}$ based on the dimension tree algorithm shown in Fig. 2.1
5: end for
6: while $\sum_{i=1}^{N} ||\mathbf{G}^{(i)}||_F > \Delta\|\mathcal{X}\|_F$ do
7:     for $n \in \{1,\ldots,N\}$ do
8:         $\mathbf{\Gamma}^{(n)} \leftarrow \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(n-1)} * \mathbf{S}^{(n+1)} * \cdots * \mathbf{S}^{(N)}$
9:         $\mathbf{A}^{(n)}_{\text{new}} \leftarrow \mathbf{M}^{(n)}\mathbf{\Gamma}^{(n)\dagger}$
10:        $\delta\mathbf{A}^{(n)} = \mathbf{A}^{(n)}_{\text{new}} - \mathbf{A}^{(n)}$
11:        $\mathbf{G}^{(n)} \leftarrow -\delta\mathbf{A}^{(n)}\mathbf{\Gamma}^{(n)}$
12:        $\mathbf{A}^{(n)} \leftarrow \mathbf{A}^{(n)}_{\text{new}}$
13:        $\mathbf{S}^{(n)} \leftarrow \mathbf{A}^{(n)T}\mathbf{A}^{(n)}$
14:        for $m \in \{1,\ldots,N\}, m \neq n$ do
15:            Update $\mathbf{M}^{(m)}$ as $\mathbf{M}^{(m)}(x,k) = \mathbf{M}^{(m)}(x,k) + \sum_{y=1}^{s_n}\mathcal{M}^{(m,n)}(x,y,k)\delta\mathbf{A}^{(n)}(y,k)$
16:        end for
17:    end for
18: end while
19: return $[\![\mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)}]\!]$
```

braically equivalent to Algorithm 2.1, but makes oracle-like use of $\mathcal{M}^{(m,n)}$ (shown in (2.13)), recomputing which would increase the computational cost. We can bound the error of the way pairwise perturbation propagates updates to any right-hand side $\mathbf{M}^{(m)}$ due to changes in any one of the other factor matrices $\delta\mathbf{A}^{(n)}$. We define the update $\mathbf{H}^{(m,n)}$ in terms of its columns,

$$\mathbf{h}_k^{(m,n)}(x) = \sum_{y=1}^{s_n} \mathcal{M}^{(m,n)}(x,y,k)\delta\mathbf{A}^{(n)}(y,k), \quad \text{where} \quad \delta\mathbf{A}^{(n)} = \mathbf{A}^{(n)}_{\text{new}} - \mathbf{A}^{(n)}. \tag{2.31}$$

Note that $\delta\mathbf{A}^{(n)}$ denotes the update of nth factor between two neighboring sweeps, which should be distinguished from $d\mathbf{A}^{(n)}$, denoting the perturbation of nth factor in PP. Based on the definition, the update of each $\mathbf{M}^{(m)}$ after an ALS sweep is the summation of $\mathbf{H}^{(m,n)}$ expressed as $\delta\mathbf{M}^{(m)} = \sum_{n=1,n\neq m}^{N} \mathbf{H}^{(m,n)}$.

For simplicity, we first perform an error analysis for the case where the second order correction terms $\mathbf{V}^{(n,i,j)}$ are not included in PP. In Theorem 2.1, we prove that when the column-wise norm of $d\mathbf{A}^{(n)} = \mathbf{A}^{(n)} - \mathbf{A}_p^{(n)}$ relative to the norm of $\mathbf{A}^{(n)}$ for $n \in \{1,\ldots,N\}$ is small, the absolute error of column-wise results for $\mathbf{H}^{(m,n)}$ calculated from pairwise perturbation with respect to that calculated from exact ALS is also small. Corollary 2.1 provides a simple relative error bound for third-order tensors. Overall, these bounds demonstrate that pairwise perturbation should generally compute updates with small relative error with respect to the magnitude of the perturbation of the factor matrices since the setup of the pairwise operators. However, this relative error can be amplified during other steps of ALS, which are ill-conditioned, i.e., can suffer from catastrophic cancellation (the same would hold for round-off error).

We then perform an error analysis for the case where the second order correction terms $\mathbf{V}^{(n,i,j)}$ are included in PP in Theorem 2.2. We show that the second order corrections can tighten the leading order error by a factor related to the CP decomposition accuracy.

Theorem 2.1. *For $k \in \{1,\dots,R\}$, if $||d\mathbf{a}_k^{(l)}||_2/||\mathbf{a}_k^{(l)}||_2 \le \epsilon < 1$ for all $l \in \{1,\dots,N\}$, the pairwise perturbation algorithm without second order corrections computes the update $\tilde{\mathbf{H}}^{(1,N)}$ with columnwise error,*

$$||\tilde{\mathbf{h}}_k^{(1,N)} - \mathbf{h}_k^{(1,N)}||_2 = O(N\epsilon)||\hat{\mathfrak{T}}||_2 \prod_{j=2}^{N-1} ||\mathbf{a}_k^{(j)}||_2, \tag{2.32}$$

where $\mathbf{H}^{(1,N)}$ is the update to the matrix $\mathbf{M}^{(1)}$ due to the change $\delta\mathbf{A}^{(N)}$ performed by a regular ALS sweep, and $\hat{\mathfrak{T}} = \mathfrak{X} \times_N \delta\mathbf{a}_k^{(N)T}$. Analogous bounds hold for $\mathbf{H}^{(m,n)}$ for any $m,n \in \{1,\dots,N\}$, $m \neq n$.

Proof. The ALS update and approximated update are

$$\mathbf{h}_k^{(1,N)} = \hat{\mathfrak{T}} \bigtimes_{i\in\{2,\dots,N-1\}} \mathbf{a}_k^{(i)T} \quad \text{and} \quad \tilde{\mathbf{h}}_k^{(1,N)} = \hat{\mathfrak{T}} \bigtimes_{i\in\{2,\dots,N-1\}} \left(\mathbf{a}_k^{(i)T} - d\mathbf{a}_k^{(i)T}\right). \tag{2.33}$$

We can expand the error as

$$\tilde{\mathbf{h}}_k^{(1,N)} - \mathbf{h}_k^{(1,N)} = \sum_{S\subset\{2,\dots,N-1\}, S\neq\emptyset} \hat{\mathfrak{T}} \bigtimes_{i\in\{2,\dots,N-1\}} \mathbf{v}_k^{(i)T}, \text{ where } \mathbf{v}_k^{(i)} = \begin{cases} -d\mathbf{a}_k^{(i)} & : i \in S \\ \mathbf{a}_k^{(i)} & : i \notin S \end{cases}. \tag{2.34}$$

Consequently, we can upper-bound the error due to terms with $|S| = d$ by

$$\binom{N-2}{d}\epsilon^d ||\hat{\mathfrak{T}}||_2 \prod_{j=2}^{N-1} ||\mathbf{a}_k^{(j)}||_2 = O(N\epsilon)^d ||\hat{\mathfrak{T}}||_2 \prod_{j=2}^{N-1} ||\mathbf{a}_k^{(j)}||_2. \tag{2.35}$$

Therefore, the error bound when $|S| = d$ scales as $O(N\epsilon)^d$, and the leading order error is $O(N\epsilon)$. Q.E.D.

Note that this error bound involves $\hat{\mathcal{T}}$, which is small in norm due to being constructed from contraction with $\delta\mathbf{a}_k^{(N)}$. Thus, the error norm generally scales as $O(\epsilon^2)$ relative to the norm of the original tensor $\mathcal{X}$, since $O(N\epsilon)||\hat{\mathcal{T}}||_2\prod_{j=2}^{N-1}||\mathbf{a}_k^{(j)}||_2 = O(N\epsilon^2)||\mathcal{X}||_2\prod_{j=2}^{N-1}||\mathbf{a}_k^{(j)}||_2$.

Corollary 2.1. *For $N = 3$, using the bounds from the proof of Theorem 2.1, under the same assumptions, we obtain the absolute error bound,*

$$||\tilde{\mathbf{h}}_k^{(1,3)} - \mathbf{h}_k^{(1,3)}||_2 \leq ||\hat{\mathbf{T}}||_2||\mathbf{a}_k^{(2)}||_2\epsilon, \tag{2.36}$$

where $\hat{\mathbf{T}} = \mathcal{X} \times_3 \delta\mathbf{a}_k^{(3)T}$. Further, since $\mathbf{h}_k^{(1,3)} = \hat{\mathbf{T}}\mathbf{a}_k^{(2)}$, the relative error is bounded by

$$\frac{||\tilde{\mathbf{h}}_k^{(1,3)} - \mathbf{h}_k^{(1,3)}||_2}{||\mathbf{h}_k^{(1,3)}||_2} \leq \kappa(\hat{\mathbf{T}})\epsilon. \tag{2.37}$$

From Theorem 2.1, we can conclude that the relative error in computing any column update $\mathbf{h}_k^{(i,j)}$ is $O(\epsilon)$ when $\epsilon \ll 1$ and the correct update is sufficiently large, e.g., for $i = 1$ and $j = N$, $||\mathbf{h}_k^{(1,N)}||_2 = \Omega\Big(||\hat{\mathcal{T}}||_2\prod_{i=2}^{N-1}||\mathbf{a}_k^{(l)}||_2\Big)$. When this is the case, we can also bound the error of the update to the columns of the right-hand sides $\delta\mathbf{M}^{(n)}$ formed in ALS, so long as the sum of the updates $\mathbf{H}^{(n,m)}$ for $m \neq n$ is not too small in norm relative to each update matrix.

We now perform analysis for the case where the second order corrections $\mathbf{V}^{(n,i,j)}$ are included in PP.

Theorem 2.2. *For $k \in \{1, \ldots, R\}$, if $||d\mathbf{a}_k^{(l)}||_2/||\mathbf{a}_k^{(l)}||_2 \leq \epsilon < 1$ for all $l \in \{1, \ldots, N\}$, the pairwise perturbation algorithm with second order correction terms computes the update term $\tilde{\mathbf{H}}^{(1,N)}$ with columnwise error,*

$$||\tilde{\mathbf{h}}_k^{(1,N)} - \mathbf{h}_k^{(1,N)}||_2 = O(N\epsilon)||\hat{\mathcal{P}} - \hat{\mathcal{T}}||_2\prod_{j=2}^{N-1}||\mathbf{a}_k^{(j)}||_2 + O\left((N\epsilon)^2\right)||\hat{\mathcal{T}}||_2\prod_{j=2}^{N-1}||\mathbf{a}_k^{(j)}||_2, \tag{2.38}$$

where $\hat{\mathcal{P}} = \mathcal{Z} \times_N \delta\mathbf{a}_k^{(N)T}$, and $\mathcal{Z} = [\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$ denotes the approximate CP decomposition of $\mathcal{X}$. $\mathbf{H}^{(1,N)}$ is the update to the matrix $\mathbf{M}^{(1)}$ due to the change $\delta\mathbf{A}^{(N)}$ performed by a regular ALS sweep, and $\hat{\mathcal{T}} = \mathcal{X} \times_N \delta\mathbf{a}_k^{(N)T}$. Analogous bounds hold for $\mathbf{H}^{(m,n)}$ for any $m, n \in \{1, \ldots, N\}$, $m \neq n$.

Proof. The ALS approximated update is

$$\tilde{\mathbf{h}}_k^{(1,N)} = \hat{\mathcal{T}} \bigtimes_{i\in\{2,\ldots,N-1\}} \left(\mathbf{a}_k^{(i)T} - d\mathbf{a}_k^{(i)T}\right) + \sum_{i\in\{2,\ldots,N-1\}} \hat{\mathcal{P}} \times_i d\mathbf{a}_k^{(i)T} \bigtimes_{j\in\{2,\ldots,N-1\},j\neq i} \mathbf{a}_k^{(j)T}. \tag{2.39}$$

We can expand the error as

$$\begin{aligned}\tilde{\mathbf{h}}_k^{(1,N)} - \mathbf{h}_k^{(1,N)} = & \sum_{S\subset\{2,\ldots,N-1\},|S|\geq 2} \hat{\boldsymbol{\mathcal{T}}} \bigtimes_{i\in\{2,\ldots,N-1\}} \mathbf{v}_k^{(i)T} \\ & + \sum_{i\in\{2,\ldots,N-1\}} \left(\hat{\boldsymbol{\mathcal{P}}} - \hat{\boldsymbol{\mathcal{T}}}\right) \times_i d\mathbf{a}_k^{(i)T} \bigtimes_{j\in\{2,\ldots,N-1\},j\neq i} \mathbf{a}_k^{(j)T},\end{aligned} \tag{2.40}$$

where $\mathbf{v}_k^{(i)} = -d\mathbf{a}_k^{(i)}$ if $i \in S$ and $\mathbf{v}_k^{(i)} = \mathbf{a}_k^{(i)}$ otherwise. By the same analysis as in Theorem 2.1, the error due to each term with $|S| = d$, $d \geq 2$ can be bounded as $O(N\epsilon)^d ||\hat{\boldsymbol{\mathcal{T}}}||_2 \prod_{j=2}^{N-1} ||\mathbf{a}_k^{(j)}||_2$. We can then upper-bound the error due to the second term by

$$O(N\epsilon)||\hat{\boldsymbol{\mathcal{P}}} - \hat{\boldsymbol{\mathcal{T}}}||_2 \prod_{j=2}^{N-1} ||\mathbf{a}_k^{(j)}||_2, \tag{2.41}$$

thus completing the proof. Q.E.D.

From Theorem 2.2, we can conclude that when the approximate CP decomposition is close to $\boldsymbol{\mathcal{X}}$, the term expressed in (2.41) will have small magnitude, making the absolute error second order accurate in terms of ϵ.

In Section 2.7.4, we also obtain relative error bounds on MTTKRPs (the right-hand sides in the linear least squares subproblems). However, this error bound is relative to the condition number of $\boldsymbol{\mathcal{X}}$ (defined in Section 2.7.1), which is infinite for sufficiently large tensors.

2.3.2 Tucker-ALS

For Tucker decomposition, the pairwise perturbation approximation satisfies better bounds than for CP decomposition, due to the orthogonality of the factor matrices. We can not only obtain the similar bound as Theorem 2.1, but also obtain stronger results in tensor spectral norm (defined in (2.42)) assuming that the residual of the Tucker decomposition is bounded (it suffices that the decomposition achieves one digit of accuracy in residual), and stronger results in Frobenius norm assuming that the ratio of rank to dimension is not too large.

The spectral norm of any tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{s_1\times\cdots s_N}$ is

$$||\boldsymbol{\mathcal{T}}||_2 \coloneqq \max_{\substack{\forall i\in\{2,\ldots,N\},\mathbf{x}^{(i)}\in\mathbb{R}^{s_i} \\ ||\mathbf{x}^{(2)}||_2=\cdots=||\mathbf{x}^{(N)}||_2=1}} \left\|\boldsymbol{\mathcal{T}} \bigtimes_{i\in\{2,\ldots,N\}} \mathbf{x}^{(i)T}\right\|_2, \tag{2.42}$$

where $\boldsymbol{\mathcal{T}}$ is contracted with $\mathbf{x}^{(i)}$ along its ith mode. The spectral tensor norm corresponds to the magnitude of the largest tensor singular value [105]. Computing the spectral norm is NP-hard [28], but can usually be done in practice by specialized variants of ALS [106]. The spectral norm is invariant under reordering of modes of $\mathcal{T}$. Lemma 2.1 shows submultiplicativity of this norm for the multilinear multiplication.

Lemma 2.1. *Given any tensor* $\mathcal{T} \in \mathbb{R}^{s_1 \times \cdots \times s_N}$ *and matrix* $\mathbf{M} \in \mathbb{R}^{s_N \times R}$*, if* $\mathcal{V} = \mathcal{T} \times_N \mathbf{M}^T$ *then* $||\mathcal{V}||_2 \leq ||\mathcal{T}||_2 ||\mathbf{M}||_2$.

Proof. There exist unit vectors $\mathbf{x}^{(2)}, \dots, \mathbf{x}^{(N)}$ such that

$$||\mathcal{V}||_2 = \left\| \mathcal{V} \bigtimes_{i \in \{2,\dots,N\}} \mathbf{x}^{(i)T} \right\|_2 = \left\| \mathcal{T} \bigtimes_{i \in \{2,\dots,N-1\}} \mathbf{x}^{(i)T} \times_N \left(\mathbf{M}\mathbf{x}^{(N)}\right)^T \right\|_2 . \tag{2.43}$$

Let $\mathbf{z} = \mathbf{M}\mathbf{x}^{(N)}$, so $||\mathbf{z}||_2 \leq ||\mathbf{M}||_2$. If $||\mathbf{z}||_2 = 0$, then $||\mathcal{V}||_2 = 0$, the inequality holds. Otherwise, since

$$\left\| \mathcal{T} \bigtimes_{i \in \{2,\dots,N-1\}} \mathbf{x}^{(i)T} \times_N \mathbf{z}^T \right\|_2 \leq \left\| \mathcal{T} \bigtimes_{i \in \{2,\dots,N-1\}} \mathbf{x}^{(i)T} \times_N \mathbf{z}^T \right\|_2 \frac{||\mathbf{M}||_2}{||\mathbf{z}||_2} \leq ||\mathcal{T}||_2 ||\mathbf{M}||_2, \tag{2.44}$$

the inequality still holds. Q.E.D.

Using Lemma 2.1, we prove in Lemma 2.2 that after contracting a tensor with a matrix with orthonormal columns, whose row length is higher or equal to the column length, the contracted tensor norm is the same as the original tensor norm.

Lemma 2.2. *Given tensor* $\mathcal{G} \in \mathbb{R}^{r_1 \times \cdots \times r_N}$*, the mode-$n$ product for any* $n \in \{1, \dots, N\}$*, with a matrix with orthonormal columns* $\mathbf{M} \in \mathbb{R}^{s \times r_n}$*,* $r_n \leq s$*, satisfies* $||\mathcal{G}||_2 = ||\mathcal{G} \times_n \mathbf{M}||_2$.

Proof. Based on the submultiplicative property of the tensor norm (Lemma 2.1),

$$||\mathcal{G}||_2 = ||\mathcal{G} \times_n (\mathbf{M}^T\mathbf{M})||_2 = ||\mathcal{G} \times_n \mathbf{M} \times_n \mathbf{M}^T||_2 \leq ||\mathcal{G} \times_n \mathbf{M}||_2 ||\mathbf{M}^T||_2 = ||\mathcal{G} \times_n \mathbf{M}||_2, \tag{2.45}$$

and simultaneously, $||\mathcal{G} \times_n \mathbf{M}||_2 \leq ||\mathcal{G}||_2 ||\mathbf{M}||_2 = ||\mathcal{G}||_2$. Q.E.D.

Below we demonstrate that

- similar to Algorithm 2.5 and Theorem 2.1, when we view the ALS procedure for Tucker decomposition of equidimensional tensors in terms of pairwise updates, we can bound the error of updates to any right-hand side $\mathcal{Y}^{(m)}$ due to changes in any one of the other factor matrices $\delta\mathbf{A}^{(n)}$. We define the update $\mathcal{J}^{(m,n)}$ as

$$\mathcal{J}^{(m,n)} = \mathcal{Y}^{(m,n)} \times_n \delta\mathbf{A}^{(n)T}, \quad \text{where} \quad \delta\mathbf{A}^{(n)} = \mathbf{A}^{(n)}_{\text{new}} - \mathbf{A}^{(n)}. \tag{2.46}$$

The columnwise absolute error bound for MTTKRP holds for $\boldsymbol{\mathcal{J}}^{(m,n)}$ when the columnwise 2-norm relative perturbations of the input matrices are bounded by $O(\epsilon)$ (Theorem 2.3),

- the relative error of $\boldsymbol{\mathcal{Y}}^{(m)}$ for $m \in \{1, \dots, N\}$ satisfies the bound of $O\left(\epsilon^2\right)$, so long as the residual of Tucker decomposition is small (Theorem 2.4),
- the relative error of $\boldsymbol{\mathcal{Y}}^{(m)}$ for $m \in \{1, \dots, N\}$ is bounded in Frobenius norm by $O\left(\epsilon^2\right)$ for a fixed problem size assuming that HOSVD is performed to initialize Tucker-ALS (Theorem 2.5).

Theorem 2.3. *For an order N tensor $\boldsymbol{\mathcal{X}}$ with dimension sizes s, if $||d\mathbf{a}_k^{(n)}||_2 / ||\mathbf{a}_k^{(n)}||_2 \leq \epsilon < 1$ for all $n \in \{1, \dots, N\}, k \in \{1, \dots, R\}$, the pairwise perturbation algorithm computes update $\boldsymbol{\mathcal{J}}^{(1,N)}$ with error,*

$$\left\|\tilde{\mathbf{j}}^{(1,N)}_{i_2,\dots,i_N} - \mathbf{j}^{(1,N)}_{i_2,\dots,i_N}\right\|_2 = O(N\epsilon)||\hat{\boldsymbol{\mathcal{T}}}||_2 \prod_{j=2}^{N-1} ||\mathbf{a}_k^{(j)}||_2, \tag{2.47}$$

where $\hat{\boldsymbol{\mathcal{T}}} = \boldsymbol{\mathcal{X}} \times_N \delta\mathbf{a}_{i_N}^{(N)T}$ and $\mathbf{j}^{(1,N)}_{i_2,\dots,i_N}(x) = \boldsymbol{\mathcal{J}}^{(1,N)}(x, i_2, \dots, i_N)$.

Proof. The proof is similar to that of Theorem 2.1. The ALS update and approximated update after a change $\delta\mathbf{A}^{(N)}$ are

$$\mathbf{j}^{(1,N)}_{i_2,\dots,i_N} = \hat{\boldsymbol{\mathcal{T}}} \bigtimes_{j=2}^{N-1} \mathbf{a}_{i_j}^{(j)T} \quad \text{and} \quad \tilde{\mathbf{j}}^{(1,N)}_{i_2,\dots,i_N} = \hat{\boldsymbol{\mathcal{T}}} \bigtimes_{j=2}^{N-1} \left(\mathbf{a}_{i_j}^{(j)T} - d\mathbf{a}_{i_j}^{(j)T}\right). \tag{2.48}$$

The error bound proceeds by analogy to the proof of Theorem 2.1. Q.E.D.

Using Lemma 2.2, we prove in Theorem 2.4 that when the relative error of the matrices $\mathbf{A}^{(n)}$ for $n \in \{1, \dots, N\}$ is small and the residual of the Tucker decomposition is loosely bounded, the relative error bound for the $\boldsymbol{\mathcal{Y}}^{(n)}$ is independent of the tensor condition number defined in Section 2.7.

Theorem 2.4. *Given tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{s_1 \times \cdots \times s_N}$, if $||d\mathbf{A}^{(n)}||_2 \leq \epsilon \ll 1$ for $n \in \{1, \dots, N\}$ and*

$$||\boldsymbol{\mathcal{X}} - [\![\boldsymbol{\mathcal{G}}; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(N)}]\!]||_2 \leq \frac{1}{3}||\boldsymbol{\mathcal{X}}||_2, \tag{2.49}$$

$\tilde{\boldsymbol{\mathcal{Y}}}^{(n)}$ is constructed with error,

$$\frac{||\tilde{\boldsymbol{\mathcal{Y}}}^{(n)} - \boldsymbol{\mathcal{Y}}^{(n)}||_2}{||\boldsymbol{\mathcal{Y}}^{(n)}||_2} = O\left(\epsilon^2\right). \tag{2.50}$$

Proof.

$$\begin{aligned}\frac{||\tilde{\mathcal{Y}}^{(n)}-\mathcal{Y}^{(n)}||_2}{||\mathcal{Y}^{(n)}||_2} &\le \binom{N}{2}\max_{i,j}\frac{||\mathcal{Y}_p^{(i,j,n)}\times_i d\mathbf{A}^{(i)T}\times_j d\mathbf{A}^{(j)T}||_2}{||\mathcal{Y}^{(n)}||_2} \\ &\le \binom{N}{2}\max_{i,j}\frac{||\mathcal{Y}_p^{(i,j,n)}||_2||d\mathbf{A}^{(i)}||_2||d\mathbf{A}^{(j)}||_2}{||\mathcal{Y}^{(n)}||_2}.\end{aligned} \tag{2.51}$$

Let $\tilde{\mathcal{X}} = [\![\mathcal{G};\mathbf{A}^{(1)},\mathbf{A}^{(2)},\ldots,\mathbf{A}^{(N)}]\!]$, $\mathcal{R}=\mathcal{X}-\tilde{\mathcal{X}}$. Define the tensors $\mathcal{Z}^{(i,j,n)}$ by contraction of $\mathcal{R}$ with all except three factor matrices,

$$\mathcal{Z}^{(i,j,n)} = \mathcal{R}\bigtimes_{r\in\{1,\ldots,N\}\backslash\{i,j,n\}}\mathbf{A}^{(r)T}. \tag{2.52}$$

For $||\mathcal{X}-\tilde{\mathcal{X}}||_2 = ||\mathcal{R}||_2 \le \frac{1}{3}||\mathcal{X}||_2$, we have $\frac{2}{3}||\mathcal{X}||_2 \le ||\tilde{\mathcal{X}}||_2 \le \frac{4}{3}||\mathcal{X}||_2$. Based on Lemma 2.2,

$$\begin{aligned}||\mathcal{Y}^{(n)}||_2 &= ||\mathcal{G}\times_n\mathbf{A}^{(n)}+\mathcal{Z}^{(i,j,n)}\times_i\mathbf{A}^{(i)T}\times_j\mathbf{A}^{(j)T}||_2 \\ &\ge ||\mathcal{G}||_2-||\mathcal{Z}^{(i,j,n)}||_2||\mathbf{A}^{(i)T}||_2||\mathbf{A}^{(j)T}||_2 \ge ||\mathcal{G}||_2-||\mathcal{R}||_2 \ge \frac{1}{3}||\mathcal{X}||_2.\end{aligned} \tag{2.53}$$

Additionally,

$$||\mathcal{Y}^{(i,j,n)}||_2 = ||\mathcal{G}\times_i\mathbf{A}^{(i)}\times_j\mathbf{A}^{(j)}\times_n\mathbf{A}^{(n)}+\mathcal{Z}^{(i,j,n)}||_2 \le ||\mathcal{G}||_2+||\mathcal{R}||_2 \le \frac{5}{3}||\mathcal{X}||_2. \tag{2.54}$$

Therefore,

$$\frac{||\tilde{\mathcal{Y}}^{(n)}-\mathcal{Y}^{(n)}||_2}{||\mathcal{Y}^{(n)}||_2} \le \binom{N}{2}\max_{i,j}\frac{||\mathcal{Y}_p^{(i,j,n)}||_2||d\mathbf{A}^{(i)}||_2||d\mathbf{A}^{(j)}||_2}{||\mathcal{Y}^{(n)}||_2} \le \binom{N}{2}\frac{\frac{5}{3}||\mathcal{X}||_2\epsilon^2}{\frac{1}{3}||\mathcal{X}||_2} = O\left(\epsilon^2\right). \tag{2.55}$$

Q.E.D.

We now derive a Frobenius norm error bound that is independent of residual norm and tensor condition number, and is based the ratio of the tensor dimensions and the Tucker rank. We arrive at this result (Theorem 2.5) by obtaining a lower bound on the residual achieved by the HOSVD (Lemma 2.3 and Lemma 2.4).

Lemma 2.3. *Given tensor $\mathcal{X}\in\mathbb{R}^{s_1\times\cdots\times s_N}$ and matrix $\mathbf{A}\in\mathbb{R}^{R\times s_n}$, where we have $R<\max\left\{s_n,\prod_{i=1,i\neq n}^N s_i\right\}$ and $\mathbf{A}$ consists of R leading left singular vectors of $\mathbf{X}_{(n)}$. Let $\mathcal{Z}=\mathcal{X}\times_n\mathbf{A}$, $||\mathcal{X}||_F\ge||\mathcal{Z}||_F\ge\sqrt{\frac{R}{s_n}}||\mathcal{X}||_F$.*

Proof. The singular values of $\mathbf{A}\mathbf{X}_{(n)}$ are the first R singular values of $\mathbf{X}_{(n)}$. Since the square of the Frobenius norm of a matrix is the sum of the squares of the singular values, $||\boldsymbol{\mathcal{Z}}||_F^2 = ||\mathbf{A}\mathbf{X}_{(n)}||_F^2 \geq (R/s_n)\, ||\mathbf{X}_{(n)}||_F^2 = (R/s_n)\, ||\boldsymbol{\mathcal{X}}||_F^2$ and $||\boldsymbol{\mathcal{Z}}||_F \leq ||\boldsymbol{\mathcal{X}}||_F$. Q.E.D.

Lemma 2.4. *For any equidimensional order N tensor $\boldsymbol{\mathcal{X}}$ with size s, we have $||\boldsymbol{\mathcal{Y}}^{(n)}||_F \geq \left(\frac{R}{s}\right)^{N/2} ||\boldsymbol{\mathcal{X}}||_F$ if Tucker-ALS starts from an interlaced HOSVD.*

Proof. In Tucker-ALS, $||\boldsymbol{\mathcal{G}}||_F$ is strictly increasing after each Tucker iteration, where $\boldsymbol{\mathcal{G}}$ is $\boldsymbol{\mathcal{X}}$'s HOSVD core tensor. Since the interlaced SVD computes each $\mathbf{A}^{(n)}$ from the truncated SVD of the product of $\boldsymbol{\mathcal{X}}$ and the first $n-1$ factor matrices, we can apply Lemma 2.3 N times,

$$\begin{aligned} ||\boldsymbol{\mathcal{X}} \times_1 \mathbf{A}^{(1)\mathbb{T}} \cdots \times_{N-1} \mathbf{A}^{(N-1)\mathbb{T}}||_F \geq ||\boldsymbol{\mathcal{G}}||_F &\geq \sqrt{\frac{R}{s}} ||\boldsymbol{\mathcal{X}} \times_1 \mathbf{A}^{(1)\mathbb{T}} \cdots \times_{N-1} \mathbf{A}^{(N-1)\mathbb{T}}||_F, \\ &\vdots \\ ||\boldsymbol{\mathcal{X}}||_F \geq ||\boldsymbol{\mathcal{G}}||_F &\geq (R/s)^{N/2} ||\boldsymbol{\mathcal{X}}||_F. \end{aligned} \tag{2.56}$$

Q.E.D.

Theorem 2.5. *Given any equidimensional order N tensor $\boldsymbol{\mathcal{X}}$ with size s, if $||d\mathbf{A}^{(n)}||_F \leq \epsilon$ for $n \in [1, N]$, $\tilde{\boldsymbol{\mathcal{Y}}}^{(n)}$ is constructed with error,*

$$\frac{||\tilde{\boldsymbol{\mathcal{Y}}}^{(n)} - \boldsymbol{\mathcal{Y}}^{(n)}||_F}{||\boldsymbol{\mathcal{Y}}^{(n)}||_F} = O\Big(\epsilon^2 \Big(\frac{s}{R}\Big)^{N/2}\Big), \tag{2.57}$$

assuming that HOSVD is used to initialize Tucker-ALS and the residual associated with factor matrices $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(n)}$ is no higher than that attained by HOSVD.

Proof.

$$\frac{||\tilde{\boldsymbol{\mathcal{Y}}}^{(n)} - \boldsymbol{\mathcal{Y}}^{(n)}||_F}{||\boldsymbol{\mathcal{Y}}^{(n)}||_F} \leq \binom{N}{2} \max_{i,j} \frac{||\boldsymbol{\mathcal{Y}}_p^{(i,j,n)} \times_i d\mathbf{A}^{(i)T} \times_j d\mathbf{A}^{(j)T}||_F}{||\boldsymbol{\mathcal{Y}}^{(n)}||_F}. \tag{2.58}$$

From Lemma 2.4, we have

$$\frac{||\boldsymbol{\mathcal{Y}}_p^{(i,j,n)} \times_i d\mathbf{A}^{(i)T} \times_j d\mathbf{A}^{(j)T}||_F}{||\boldsymbol{\mathcal{Y}}^{(n)}||_F} \leq \frac{||\boldsymbol{\mathcal{X}}||_F ||d\mathbf{A}^{(i)}||_F ||d\mathbf{A}^{(j)}||_F}{(\frac{R}{s})^{N/2} ||\boldsymbol{\mathcal{X}}||_F}. \tag{2.59}$$

Consequently, we can bound the relative error by

$$\frac{||\tilde{\boldsymbol{\mathcal{Y}}}^{(n)} - \boldsymbol{\mathcal{Y}}^{(n)}||_F}{||\boldsymbol{\mathcal{Y}}^{(n)}||_F} \leq \binom{N}{2} (s/R)^{N/2} \max_{i,j} ||d\mathbf{A}^{(i)}||_F ||d\mathbf{A}^{(j)}||_F = O\Big(\epsilon^2 \Big(\frac{s}{R}\Big)^{N/2}\Big). \tag{2.60}$$

Q.E.D.

2.4 EXPERIMENTS

We evaluate the performance of the pairwise perturbation algorithms on both synthetic tensors and application datasets. The synthetic experiments enable us to test tensors with known factors and to measure how effectively the algorithm works across many problem instances. We also consider publicly available tensor datasets as well as tensors of interest for quantum chemistry calculations and demonstrate the effectiveness of our algorithms on practical problems. We focus on the experiments on CP decomposition, because for many cases in Tucker decomposition, HOOI converges in small number of iterations with the initialization of HOSVD.

We use the metrics *relative residual* and *fitness* to evaluate the convergence of the decomposition. Let $\tilde{\mathcal{X}}$ denote the tensor reconstructed by the factor matrices and the core tensor, the relative residual and fitness are defined as follows,

$$r = \frac{\|\mathcal{X} - \tilde{\mathcal{X}}\|_F}{\|\mathcal{X}\|_F}, \qquad f = 1 - r. \tag{2.61}$$

We compare the performance of our own implementations of regular ALS with dimension trees to the pairwise perturbation algorithm. Both algorithms are implemented in Python with NumPy for sequential calculation and with Cyclops Tensor Framework (v1.5.5) [100], which is a distributed-memory library for matrix/tensor contractions that uses MPI for interprocessor communication and OpenMP for threading. We also make use of a wrapper Cyclops provides for ScaLAPACK [72] routines to solve symmetric positive definite linear systems of equations and compute the SVD[2].

The experimental results are collected on the Stampede2 supercomputer located at the University of Texas at Austin. We leverage the Knight's Landing (KNL) nodes exclusively, each of which consists of 68 cores, 96 GB of DDR RAM, and 16 GB of MCDRAM. These nodes are connected via a 100 Gb/sec fat-tree Omni-Path interconnect. For both NumPy and Cyclops implementations, we use Intel compilers and the MKL library for threaded BLAS routines, including batched BLAS routines, which are efficient for Khatri-Rao products arising in MTTKRP in CP decomposition, and employ the HPTT library [107] for high-performance tensor transposition. All storage and computation assumes the tensors are dense.

[2]All of our code is available at https://github.com/LinjianMa/tensor_decompositions.

2.4.1 Sequential Experimental Results

We collect the sequential results on one KNL node on Stampede2, leveraging 64 threads for MKL and HPTT routines.

We compare the per-sweep time of the ALS dimension tree to the pairwise perturbation initialization and approximated sweep in Fig. 2.2. Each initialization sweep constructs the PP operators and updates all the factor matrices, while an approximated sweep computes approximate updates to all the factor matrices using the PP operators constructed in the last initialization sweep. We also provide the reference per-sweep time of the ALS implementation from MATLAB Tensor Toolbox [108]. As can be seen, both ALS sweep times on top of NumPy and Cyclops are comparable to the Tensor Toolbox. For both decompositions and all the configurations, the time of an PP initialization sweep is 1.5-2.0X the time of a dimension tree based ALS sweep, while the approximated steps can have up to 6.3X speed-up for an order three tensor and 33.0X speed-up for an order six tensor for CP, and up to 10.6X speed-up for an order 6 tensor for Tucker. In addition, larger speed-up can be achieved with the increase of dimension size s and the tensor order N, which is consistent with Table 2.1.

We use five different tensors to test the sequential performance of pairwise perturbation. Sequential performance results are collected using NumPy, as NumPy has better sequential performance than Cyclops, as shown in Fig. 2.2a and Fig. 2.2b. For all the experiments, the pairwise perturbation tolerance is set as 0.1 for CP decomposition, and set as 0.3 for Tucker decomposition.

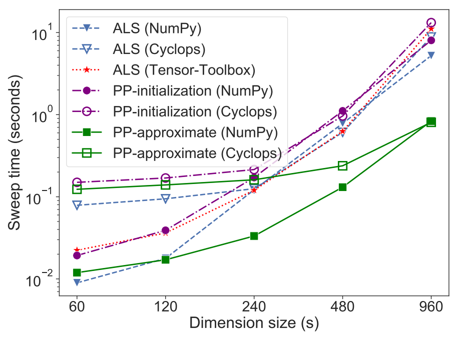
1. **Tensors with random collinearity** [80]. We create tensors based on known randomly-generated factor matrices $\mathbf{A}^{(n)}$. The factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{s\times R}$ are randomly generated so that the columns have collinearity defined based on a scalar C (selected randomly for the tensor from a given interval $[a,b)$), so that

$$\frac{\left\langle \mathbf{a}_i^{(n)}, \mathbf{a}_j^{(n)} \right\rangle}{||\mathbf{a}_i^{(n)}||_2 ||\mathbf{a}_j^{(n)}||_2} = C, \quad \forall i,j \in \{1,\ldots,R\}, i \neq j. \tag{2.62}$$

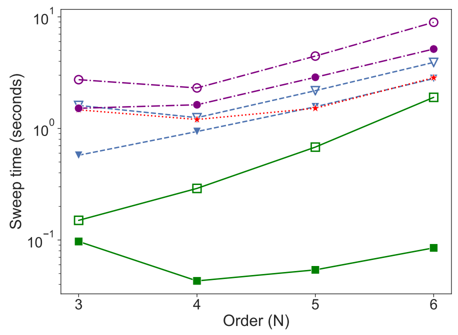
   Higher collinearity corresponds to greater overlap between columns within each factor matrix, which makes the convergence of CP-ALS slower [109].

2. **Tensors made by random matrices**. We create tensors based on known uniformly distributed randomly-generated factor matrices $\mathbf{A}^{(n)} \in [0,1]^{s\times R}$,

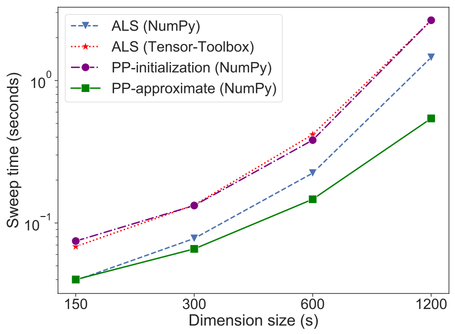
$$\boldsymbol{\mathcal{X}} = [[\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(N)}]] . \tag{2.63}$$



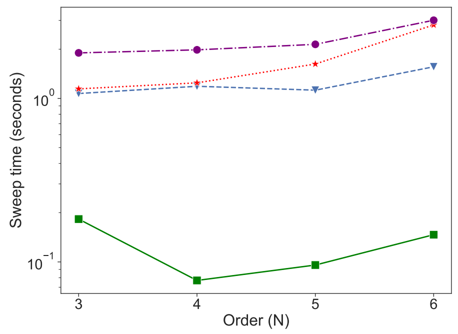
(a) CPD, $N = 3$, $R = s$



(b) CPD, $s = 960^{3/N}$, $R = 50$



(c) Tucker, $N = 3$, $R = 0.05s$



(d) Tucker, $s = 1200^{3/N}$, $R = 5$

Figure 2.2: Sequential ALS sweep time comparison for both CP and Tucker decompositions. Results are taken as the mean time across 5 sweeps. The line label of (b) is the same as (a), of (d) is the same as (c). In **(a)(c)**, we vary the dimension size and the decomposition rank, and fix the input tensor order. In **(b)(d)**, we vary the input tensor order, and fix the input tensor size and the decomposition rank.

In the experiments, we set R to be the same as the decomposition rank.

3. **Quantum chemistry tensor.** We also test on the density fitting intermediate tensor arising in quantum chemistry, which is the Cholesky factor of the two-electron integral tensor [25], [26]. For an order 4 two-electron integral tensor $\boldsymbol{\mathcal{T}}$, its Cholesky factor is an order 3 tensor $\boldsymbol{\mathcal{D}}$, with their relations shown as follows:

$$\boldsymbol{\mathcal{T}}(a, b, c, d) = \sum_{s=1}^{P} \boldsymbol{\mathcal{D}}(a, b, s)\boldsymbol{\mathcal{D}}(c, d, s), \tag{2.64}$$

where P is the third mode dimension size of $\boldsymbol{\mathcal{D}}$. CP decomposition can be performed on $\boldsymbol{\mathcal{D}}$ to provide the compressed form of the density fitting intermediate and can be used to speed up post Hartree-Fork calculations [26]. We generate the density fitting tensor via the PySCF library [110], which represents the compressed restricted Hartree-Fock wave function of an 8 water molecule chain system with a STO3G basis set. The generated tensor has size $904 \times 56 \times 56$. We set the CP rank to be 400.

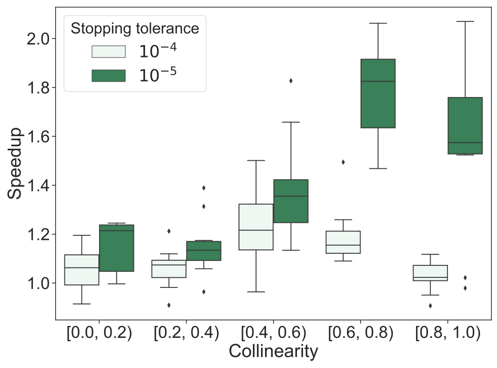
4. **COIL dataset**. COIL-100 is an image-recognition data set that contains images of objects in different poses [111] and has been used previously as a tensor decomposition benchmark [77], [80]. There are 100 different object classes, each of which is imaged from 72 different angles. Each image has 128×128 pixels in three color channels. Transferring the data into tensor format, we have a $128 \times 128 \times 3 \times 7200$ tensor. We fix the CP decomposition rank to be 15 and the Tucker decomposition rank to be $10 \times 10 \times 3 \times 50$.

5. **Time-Lapse hyperspectral radiance images**. We consider the 3D hyperspectral imaging dataset called "Souto wood pile" [112]. The dataset is usually used on the benchmark of nonnegative tensor decomposition [62], [64]. The hyperspectral data consists of a tensor with dimensions $1024 \times 1344 \times 33 \times 9$. We fix the CP decomposition rank to be 50 and the Tucker decomposition rank to be $100 \times 100 \times 3 \times 3$.

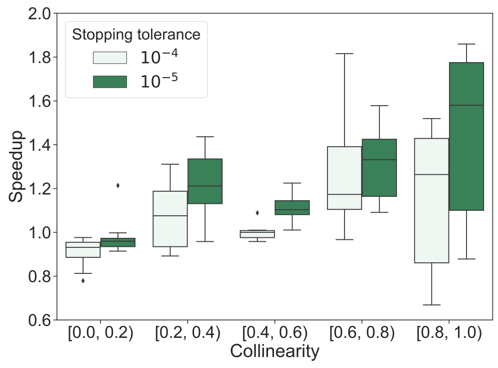
The order three tensors are tested to justify the relative error bound shown in Section 2.3.1. The performance of PP on higher order CP decompositions is also considered. The input tensors are explicitly given for all cases we considered. Note that for cases arising in scientific computing where the input tensors are given in the CP decomposition format, the efficient CP-to-Tucker-to-CP decomposition technique based on reduced HOSVD (RHOSVD) introduced in reference [113] can be used. We focus on the high rank CP decomposition, because for the cases with rank $R < s$, Tucker decomposition or HOSVD can be used to effectively compress the input tensor from dimensions of size s to R, and then CP decomposition can be performed [114], [115].

We test the synthetic tensors for CP decomposition. These tensors are all generated based on known factor matrices whose column sizes are equal to the decomposition rank, so these tensors have exact decompositions. For Tensor 1, we test on both order three tensors with both dimension sizes s and decomposition rank R equal to 400 and order four tensors with $s = R = 120$, and test the performance of pairwise perturbation on tensors with different collinearity for the exact input factor matrices. For Tensor 2, we test on order three tensors with $s = R$, and test the performance of pairwise perturbation with different dimension size and corresponding rank.

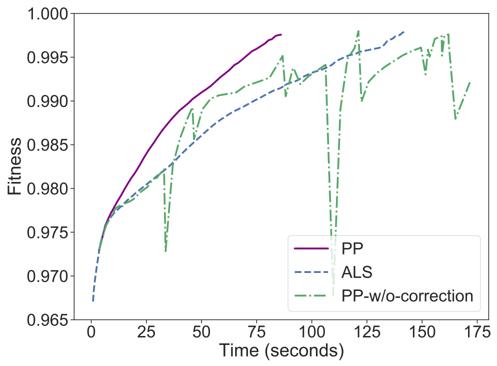
We display the speed-ups of pairwise perturbation compared to the dimension tree algorithm for synthetic tensors in Fig. 2.3. Fig. 2.3a and Fig. 2.3b show the speed-up



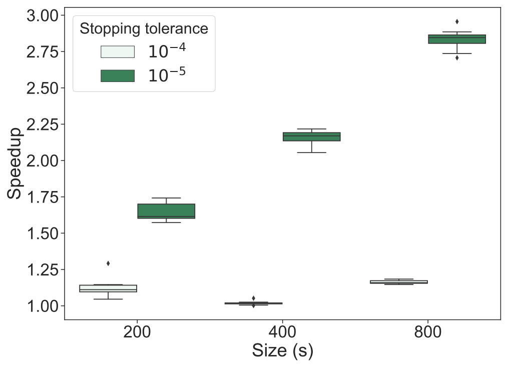
(a) $N = 3$, $s = R = 400$



(b) $N = 4$, $s = R = 120$



(c) $N = 3$, $s = R = 400$, collinearity$\in [0.6, 0.8)$



(d) Random tensors, $N = 3$, $s = R$

Figure 2.3: **(a)(b)** Box plot of the relation between PP speed-up and input collinearity ranges for tensors with specific collinearity. **(c)** Fitness-time relation for the decomposition of one tensor with specific collinearity. **(d)** Box plot of the relation between PP speed-up and size in each mode for order 3 random tensors. For all the box plots, each box is based on 10 experiments with different random seeds. Each box shows the 25th-75th quartiles, the median is indicated by a horizontal line inside the box, and outliers are displayed as dots.

distribution with different exact factor matrices collinearity. We stop the algorithm when the stopping tolerance (defined as the fitness difference between two neighboring sweeps) is reached. It can be seen that for both order three and order four tensors, PP achieves up to 2.0X speed-up, and high speed-up is achieved with tighter stopping tolerance. We find that the stricter stopping tolerance of 10^{-5} is valuable, as generally it permits about one more digit of accuracy to be achieved in fitness compared to a tolerance of 10^{-4}. In addition, experiments with a 10^{-4} stopping tolerance sometimes stop at transient swamps [116] with high decomposition residual, where ALS makes small progress for a period but the residual norm decreases more rapidly afterwards. In addition, PP tends to have higher speed-ups with relatively high collinearity. This is because tensors with high collinearity will converge in more sweeps, and more PP approximated sweeps are activated as can be seen in Table 2.2. PP starts working early for almost all the experiments, as can be observed in Fig. 2.3c, where PP starts to have speed-up when the fitness is around 0.975 and the experiment time is less than 20 seconds, and in Table 2.2, where almost all the PP initialization steps start within 20 sweeps. In addition, the fitness increases monotonically in Fig. 2.3c, indicating that PP controls the approximation error well.

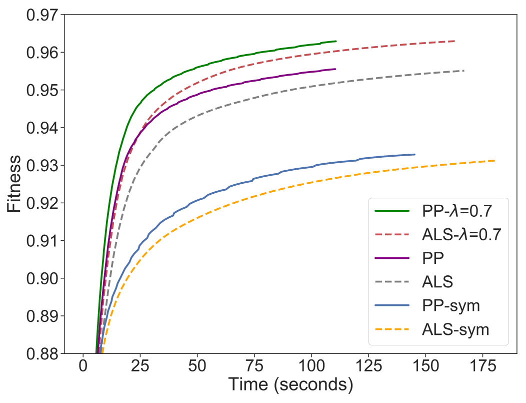
Fig. 2.3c also illustrates the importance of the second-order correction term, $\mathbf{V}^{(n,i,j)}$, in (2.25). We set the PP tolerance to be 0.02 for the PP experiment without corrections, which results in more conservative use of PP approximate steps than with the 0.1 tolerance we use for PP with the second-order correction. As can be seen, without the correction, PP suffers from more instability and no speed-up is achieved for this experiment. Therefore, for all other experiments, the correction terms are included as part of PP.

Fig. 2.3d shows the speed-up distribution with different dimension size for order three tensors made by random factor matrices. It can be seen from the figure than PP achieves up to 3.0X speed-up, and PP has larger speed-ups on larger tensors, consistent with the cost analysis.

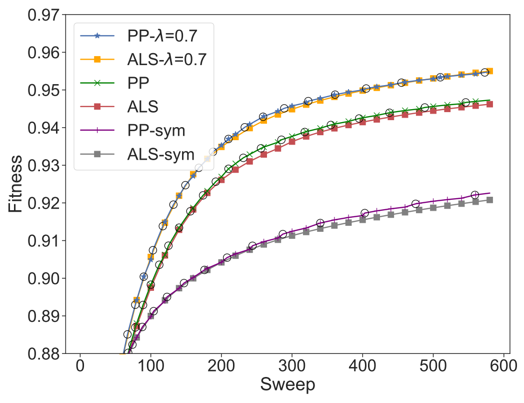
Configuration	Num-ALS	Num-PP-init	Num-PP-approx	PP-init-sweep	PP-init-fit	Final-fit
N=3, col$\in[0.0, 0.2)$	19.9	2.5	11.4	12.7	0.8203	0.9330
N=3, col$\in[0.2, 0.4)$	49.1	18.4	35.3	7.7	0.7937	0.9991
N=3, col$\in[0.4, 0.6)$	60.8	52.9	149.1	8.8	0.9345	0.9999
N=3, col$\in[0.6, 0.8)$	54.8	50.1	252.1	5.7	0.9751	0.9962
N=3, col$\in[0.8, 1.0)$	12.8	9.4	51.1	4.3	0.9940	0.9966
N=4, col$\in[0.0, 0.2)$	20.1	3.3	2.4	13.7	0.6802	0.8235
N=4, col$\in[0.2, 0.4)$	15.4	1.9	5.6	14.0	0.9525	0.9945
N=4, col$\in[0.4, 0.6)$	34.0	7.5	13.5	22.6	0.9477	0.9935
N=4, col$\in[0.6, 0.8)$	46.1	29.3	73.3	9.1	0.9365	0.9990
N=4, col$\in[0.8, 1.0)$	47.5	26.4	62.4	6.2	0.9831	0.9963

Table 2.2: Detailed statistics of the results shown in Fig. 2.3. From left to right: the tensor configuration (col stands for collinearity), number of exact ALS sweeps within the PP algorithm, number of PP initialization sweeps, number of PP approximated sweeps, index of sweep when PP is first initialized (approximation begins), the fitness when PP is first initialized, and the final fitness of the experiment. All the data are the average statistics from ten experiments.

We also test the performance of pairwise perturbation on CP decomposition of the quantum chemistry tensor, as is shown in Fig. 2.4, with detailed statistics shown in Table 2.3. In addition to the original ALS algorithm, we consider two other ALS variants for this problem: the ALS algorithm with different update step size, and the ALS algorithm with a symmetry constraint [25]. The algorithm with different update step size updates the factor



(a) Fitness-time relation



(b) Fitness-sweep relation

Figure 2.4: Comparison of PP and the dimension tree algorithm for CP decomposition on the quantum chemistry tensor with different variants. PP-sym/ALS-sym denotes the decomposition with symmetry constraint. PP-λ=0.7/ALS-λ=0.7 denotes the decomposition with step size chosen to be 0.7. (b) shows detailed fitness-sweep relation for part of the sweeps. In (b), squares on the dimension tree lines represent the results per 20 sweeps (including all PP initialization, PP approximated and ALS sweeps), and the black circles on pairwise perturbation lines represent the time when pairwise perturbation re-initializes.

Tensor	Num-ALS	Num-PP-init	Num-PP-approx	Time-ALS	Time-PP-init	Time-PP-approx
Chemistry (Fig. 2.4)	44	40	1416	0.1116	0.1655	0.0703
Coil (Fig. 2.5a)	31	22	147	2.357	3.660	0.0648
TimeLapse (Fig. 2.5b)	23	16	161	0.4087	0.9236	0.0562
Chemistry (Fig. 2.7)	88	54	1358	5.338	9.608	2.254

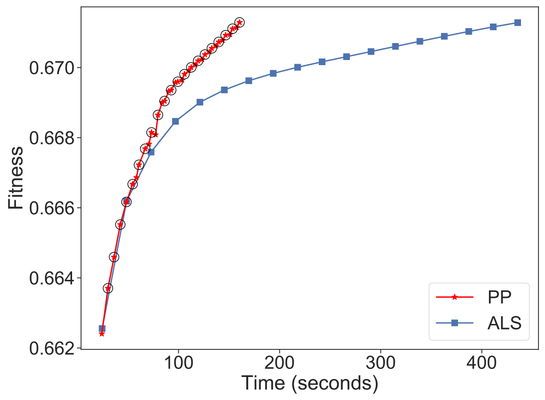
Table 2.3: Detailed statistics of different experiments. From left to right: the tensor type, number of ALS sweeps until PP experiments are finished, number of PP initialization sweeps, number of PP approximated sweeps, the average time of each ALS sweep, the average time of each PP initialization sweep, and average time of each PP approximated sweep.

matrices $\mathbf{A}^{(n)}$ based on

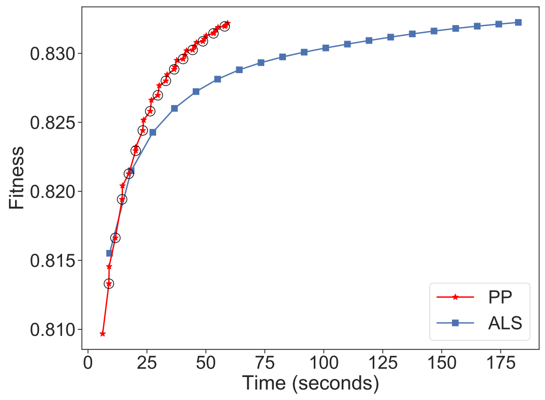
$$\mathbf{A}^{(n)}_{new} = (1-\lambda)\mathbf{A}^{(n)} + \lambda\mathbf{M}^{(n)}\mathbf{\Gamma}^{(n)\dagger}, \tag{2.65}$$

where λ is the update step size. A good choice of λ can help achieving better convergence. The symmetry constrained algorithm considers the input tensor is symmetric in the two equidimensional modes and restricts the two factor matrices for these two modes to be the same: $\boldsymbol{\mathcal{X}} = [\![\mathbf{A}, \mathbf{B}, \mathbf{B}]\!]$. We update $\mathbf{A}$ the same as the original ALS step, and update $\mathbf{B}$ with the update step size $\lambda = 0.8$ to avoid divergence.

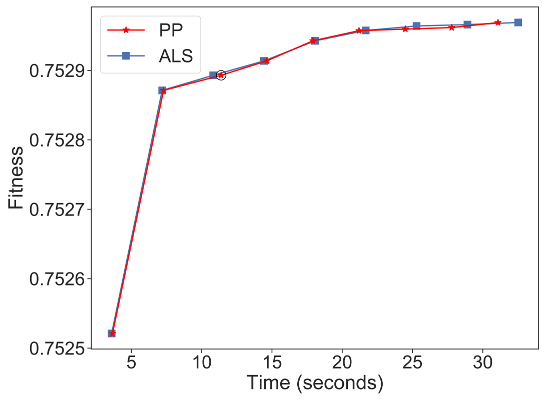
As is shown in Fig. 2.4a, for all the variants of ALS algorithms, PP performs better than the dimension tree algorithm, achieving 1.25-1.52X speed-up. All the experiments are stopped after 1500 sweeps. It can also be observed in Fig. 2.4b that PP usually restarts once approximately every 40 sweeps, and for each sweep, the fitness of both ALS and PP are almost the same, indicating that PP controls the approximation error well.



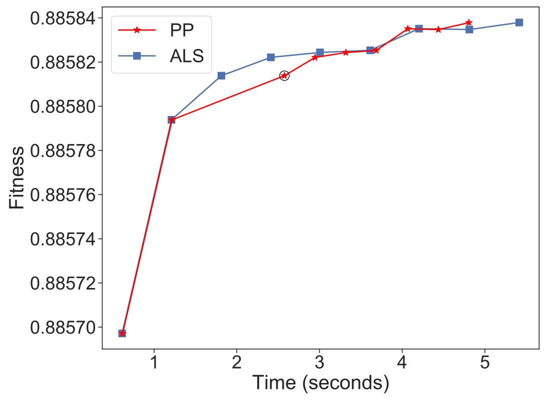
(a) CP decomposition of Coil Dataset



(b) CP decomposition of Time-Lapse Dataset



(c) Tucker decomposition of Coil Dataset



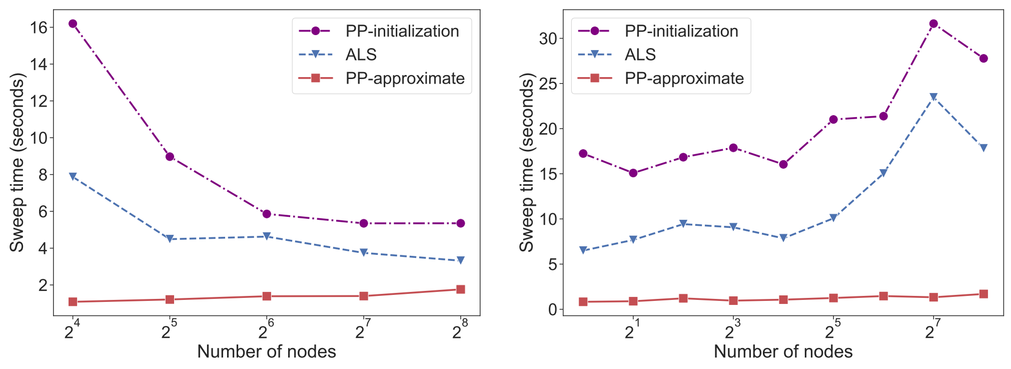
(d) Tucker decomposition of Time-Lapse Dataset

Figure 2.5: Experimental results on image datasets between pairwise perturbation and ALS for CP and Tucker decompositions. Each dot on the ALS/PP lines represents the results per 10 sweeps for CP and per sweep for Tucker decomposition (including all PP initialization, PP approximated and ALS sweeps), and the black circles on pairwise perturbation lines represent the time when pairwise perturbation restarts.

We test the performance of pairwise perturbation on real image datasets with NumPy in Fig. 2.5, with detailed statistics shown in Table 2.3. We display the fitness and execution time for CP decomposition of the two image datasets in Fig. 2.5a and Fig. 2.5b. We observe that pairwise perturbation achieves a lower execution time for them. The speed-up for the Coil Dataset is 2.72X and for the Time-Lapse Dataset is 3.1X.

Pairwise perturbation is also used to speedup HOOI procedure in Tucker decomposition. However, as noted in other work [117], we observed that ALS sweeps do not significantly lower the residual beyond what is achieved by the first sweep (HOSVD). We display the fitness and the execution time for Tucker decomposition of the two real datasets in Fig. 2.5c and Fig. 2.5d. The speed-up for the Coil Dataset is 1.05X and for the Time-Lapse Dataset is 1.13X. The reason for no obvious speed-up for the Coil Dataset is that the tensor is not equidimensional (one dimension is 7200, while others are all smaller or equal to 128). Therefore, when updating the factor matrix with a dimension of 7200, the number of operations necessary to construct the SVD input for PP are similar to that for the dimension tree Tucker algorithm. For the Time-Lapse Dataset, the tensor dimensions are more evenly distributed (two dimensions are greater than 1000), and we observe a greater speed-up. We conclude that the proposed Tucker PP algorithm performs better when used on the tensors whose dimensions are approximately equal.

2.4.2 Parallel Performance



(a) Strong scaling of CP decomposition (b) Weak scaling of CP decomposition

Figure 2.6: Benchmark results for ALS sweeps with Cyclops, taken as the mean time across 5 sweeps.

We perform a parallel scaling analysis to compare the simulation time for one ALS sweep of the dimension tree algorithm to the initialization and the approximated step of the pairwise perturbation algorithm with Cyclops in Fig. 2.6. Parallelism is used to accelerate the tensor contractions via calling Cyclops kernels as well as the linear system solve via calling ScaLAPACK kernels. The Cyclops library reduces each tensor contraction to a matrix multiplication. For the PP initialization step, this approach either keeps the input tensor in place, performs local multiplications, and afterwards performs a reduction on the output tensor when the rank R is small, or performs a general 3D parallel matrix multiplication when R is high. For the PP approximated step, this approach parallelizes small-sized batched matrix-vector products and result in over-parallelization. We direct readers to the reference

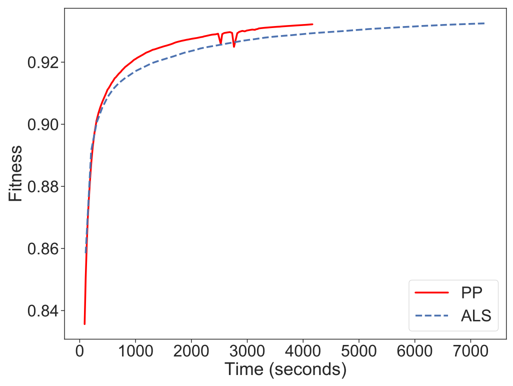
[94] for a detailed communication cost analysis and a more communication efficient algorithm for parallel pairwise perturbation.

We use 8 processes per node and 8 threads per process for the benchmark experiments. The pairwise perturbation initialization step includes the construction of the pairwise perturbation operators, and is therefore much slower than the approximated steps. For strong scaling, we consider order $N = 6$ tensors with dimension $s = 50$ and rank $R = 6$ CP and Tucker decompositions. For weak scaling, on p processors, we consider order $N = 6$ tensors with dimension $s = \lfloor 32p^{1/6} \rfloor$ and rank $R = \lfloor 4p^{1/6} \rfloor$.

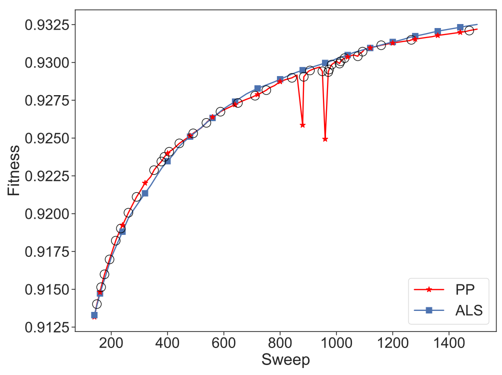
For weak scaling, Fig. 2.6 shows that with the increase of number of nodes, the step time for all three steps increases. The approximated step time of pairwise perturbation is always much faster (7.8 and 10.5 times faster on 1 node and 256 nodes, respectively, compared to the dimension tree based ALS step time) than the two other steps, showing the good scalability of pairwise perturbation. For strong scaling, the figure shows that the approximated step time of pairwise perturbation increases with the number of nodes, while the two other step times decrease. The PP approximated step is much cheaper computationally and becomes dominated by communication with increasing node counts, thereby slowing down in step time. For the two other steps, the matrix calculation time will be decreased a lot with the increase of node number, thereby the step time is decreased. Overall, we observe that the potential performance benefit of pairwise perturbation is greater for weak scaling.

2.4.3 Parallel Experimental Results

We test the parallel performance of pairwise perturbation with Cyclops on a quantum chemistry tensor. Similar to Section 2.4.1, we generate the order three density fitting tensor representing the compressed restricted Hartree-Fock wave function of an 40 water molecule chain system with a STO3G basis set. The generated tensor has size $4520 \times 280 \times 280$. We set the CP rank to be 1800. We show the parallel performance with Cyclops for the quantum chemistry tensor in Fig. 2.7, with detailed statistics shown in Table 2.3. We perform experiments on 4 KNL nodes, leveraging 64 processors on each node. For the PP experiment, after first level contractions of the PP dimension tree, we redistribute the resulting tensor across all the processes so that it is partitioned in the rank mode, which makes the PP approximated steps faster. It can be seen that PP performs better than the dimension tree algorithm, achieving 1.75X speed-up to reach a fitness of 0.933. It can also be observed in Fig. 2.7b that for most of the sweeps, the fitness of both the dimension tree algorithm and PP are almost the same, indicating that PP controls the approximation error well.



(a) Fitness-time relation



(b) Fitness-sweep relation

Figure 2.7: Comparison of PP and the dimension tree algorithm for CP decomposition on the quantum chemistry tensor with Cyclops. (b) shows detailed fitness-sweep relation for part of the sweeps. In (b), squares on the dimension tree lines represent the results per 20 sweeps (including all PP initialization, PP approximated and ALS sweeps), and the black circles on pairwise perturbation lines represent the time when pairwise perturbation re-initializes.

2.5 DISCUSSIONS

One disadvantage of the standard CP-ALS algorithm is that it could be slow or has no convergence when a solution with high resolution is required, which is also called the 'swamp' phenomenon [118]. Consequently, researchers have been looking at different alternatives to CP-ALS, including various regularization techniques [119], [120] and line search [109], [121]. These alternatives usually have higher convergence rate compared to the standard CP-ALS. We can combine some of these algorithms with pairwise perturbation to design algorithms with both faster convergence rate and cheaper per-sweep cost. For example, we show in Section 2.8 that pairwise perturbation can be combined with the enhanced line search (ELS-ALS) algorithm [109], which is an effective line search algorithm on top of the standard ALS for CP decomposition, to accelerate the scheme.

2.6 CONCLUSION

We have provided the pairwise perturbation algorithm for both CP and Tucker decompositions for dense tensors. The advantage of this algorithm is that it uses perturbative corrections rather than recomputing the tensor contractions to set up the quadratic optimization subproblems. Our error analysis demonstrates that it is accurate when the factor matrices change little. Specifically, our implementation of pairwise perturbation shows speed-ups for CP-ALS of up to 3.1X across all synthetic and application data with respect to the best known method for exact CP-ALS with the NumPy-based sequential implementation.

We leave analysis and benchmarking of pairwise perturbation with sparse tensors for future work. Since contraction between the input tensor and the first factor matrix requires fewer operations, there is less likely to be a benefit in using pairwise perturbation. Additionally, it is likely of interest to investigate more efficient adaptations of pairwise perturbation for non-equidimensional tensors and to experiment with alternative schemes for switching between regular ALS and pairwise perturbation.

2.7 ERROR BOUNDS BASED ON A TENSOR CONDITION NUMBER

We provide relative error bounds for the pairwise perturbation algorithm for both CP-ALS and Tucker-ALS for tensors that are 'well-conditioned', in a sense that is defined in this section. However, results related to the Hurwitz problem regarding multiplicative relations of quadratic forms [95], imply that equidimensional order three tensors can have a bounded condition number only if their dimension is $s \in \{1, 2, 4, 8\}$. We provide families of tensors with unit condition number with such dimensions. The results shed further light on the stability of MTTKRP as well as ALS, and yield nontrivial bounds for small tensors. For factorization of large tensors, the bounds proven in this section are not meaningful, since their condition number is necessarily infinite for at least one ordering of modes.

2.7.1 Tensor Condition Number

We consider a notion of tensor condition number that corresponds to a global bound on the conditioning of the multilinear vector-valued function, $\mathbf{g}_{\boldsymbol{\mathcal{T}}} : \otimes_{i=2}^{N} \mathbb{R}^{s_i} \to \mathbb{R}^{s_1}$ associated with the product of the tensor with vectors along all except the first mode,

$$\mathbf{g}_{\boldsymbol{\mathcal{T}}}\left(\mathbf{x}^{(2)}, \dots, \mathbf{x}^{(N)}\right) = \boldsymbol{\mathcal{T}} \bigtimes_{i\in\{2,\dots,N\}} \mathbf{x}^{(i)T}, \tag{2.66}$$

where $\boldsymbol{\mathcal{T}}$ is contracted with $\mathbf{x}^{(i)}$ along its ith mode. The norm and condition number are given by extrema of the norm amplification of $\mathbf{g}_{\boldsymbol{\mathcal{T}}}$, which are described by the amplification function $\mathbf{f}_{\boldsymbol{\mathcal{T}}} : \otimes_{i=2}^{N} \mathbb{R}^{s_i} \to \mathbb{R}$,

$$\mathbf{f}_{\boldsymbol{\mathcal{T}}}\left(\mathbf{x}^{(2)}, \dots, \mathbf{x}^{(N)}\right) = \frac{||\mathbf{g}_{\boldsymbol{\mathcal{T}}}(\mathbf{x}^{(2)}, \dots, \mathbf{x}^{(N)})||_2}{||\mathbf{x}^{(2)}||_2 \cdots ||\mathbf{x}^{(N)}||_2}. \tag{2.67}$$

The spectral norm of the tensor corresponds to its supremum,

$$||\mathcal{T}||_2 = \sup\{\mathbf{f}_{\mathcal{T}}\}. \tag{2.68}$$

The tensor condition number can be defined as

$$\kappa(\mathcal{T}) = \sup\{\mathbf{f}_{\mathcal{T}}\}/\inf\{\mathbf{f}_{\mathcal{T}}\}, \tag{2.69}$$

which enables quantification of the worst-case relative amplification of error with respect to input for the product of a tensor with vectors along all except the first mode. In particular, $\kappa(\mathcal{T})$ provides an upper bound on the relative norm of the perturbation of $\mathbf{g}_{\mathcal{T}}$ with respect to the relative norm of any perturbation to any input vector.

For a matrix $\mathbf{M} \in \mathbb{R}^{s_1 \times s_2}$, if $s_1 > s_2$ the above notion of condition number gives $\kappa(\mathbf{M}) = \sigma_{\max}(\mathbf{M})/\sigma_{\min}(\mathbf{M})$ where $\sigma_{\min}(\mathbf{M})$ is the smallest singular value of $\mathbf{M}$ in the reduced SVD, while if $s_1 < s_2$, then $\kappa(\mathbf{M}) = \infty$. When tensor dimensions are unequal, the condition number is infinite if the first dimension is not the largest, so for some i, $s_i > s_1$. Aside from this condition, the ordering of modes of $\mathcal{T}$ does not affect the condition number, since for any $m > 1$, the supremum/infimum of $\mathbf{f}_{\mathcal{T}}$ over the domain of unit vectors are for some choice of $\mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m-1)}, \mathbf{x}^{(m+1)}, \ldots, \mathbf{x}^{(N)}$ the maximum/minimum singular values of

$$\mathbf{K} = \mathcal{T} \bigtimes_{i \in \{2,\ldots,m-1,m+1,\ldots,N\}} \mathbf{x}^{(i)T}. \tag{2.70}$$

2.7.2 Well-Conditioned Tensors

We provide two examples of order three tensors that have unit condition number. Other perfectly conditioned tensors can be obtained by multiplying the above tensors by an orthogonal matrix along any mode (we prove below that such transformations preserve condition number). The first example has $s_i = 2$, and yields a Givens rotation when contracted with a vector along the last mode. It is composed of two slices:

$$\begin{bmatrix} 1 & \\ & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} & 1 \\ -1 & \end{bmatrix}. \tag{2.71}$$

The second example has $s_i = 4$ and is composed of four slices:

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & -1 \end{bmatrix}, \begin{bmatrix} & 1 & & \\ -1 & & & \\ & & & 1 \\ & & 1 & \end{bmatrix}, \begin{bmatrix} & & & 1 \\ & & 1 & \\ & -1 & & \\ 1 & & & \end{bmatrix}, \begin{bmatrix} & & -1 & \\ & & & 1 \\ 1 & & & \\ & 1 & & \end{bmatrix}. \tag{2.72}$$

Finally, for $s_i = 8$, we provide an example by giving matrices $\mathbf{M}$ and $\mathbf{N}$, so that the tensor has nonzeros $\mathfrak{T}(i, j, \mathbf{M}(i, j)) = \mathbf{N}(i, j)$ for each entry in $\mathbf{M}$,

$$\mathbf{M} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 1 & 4 & 3 & 6 & 5 & 8 & 7 \\ 3 & 4 & 1 & 2 & 7 & 8 & 5 & 6 \\ 4 & 3 & 2 & 1 & 8 & 7 & 6 & 5 \\ 5 & 6 & 7 & 8 & 1 & 2 & 3 & 4 \\ 6 & 5 & 8 & 7 & 2 & 1 & 4 & 3 \\ 7 & 8 & 5 & 6 & 3 & 4 & 1 & 2 \\ 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}, \mathbf{N} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 \\ -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 \\ -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \end{bmatrix}. \tag{2.73}$$

The fact that the latter two tensors have unit condition number can be verified by symbolic algebraic manipulation or numerical tests.

These tensors provide solutions to special cases of the Hurwitz problem [95], which seeks bilinear forms $z_1, \ldots, z_n$ in variables $x_1, \ldots, x_l$ and $y_1, \ldots, y_m$ such that

$$\left(x_1^2 + \cdots + x_l^2\right)\left(y_1^2 + \cdots + y_m^2\right) = z_1^2 + \cdots + z_n^2.$$

Consequently, if for $\mathfrak{T}$ and any vectors $\mathbf{x}$, $\mathbf{y}$,

$$\frac{||\mathfrak{T} \times_2 \mathbf{x}^T \times_3 \mathbf{y}^T||_2}{||\mathbf{x}||_2 ||\mathbf{y}||_2} = 1 \quad \Rightarrow \quad ||\mathfrak{T} \times_2 \mathbf{x}^T \times_3 \mathbf{y}^T||_2^2 = ||\mathbf{x}||_2^2 ||\mathbf{y}||_2^2, \tag{2.74}$$

so we can define bilinear forms,

$$z_i = \sum_j \sum_k \mathfrak{T}(i, j, k) x_j y_k, \tag{2.75}$$

that provide a solution to the Hurwitz problem. Such equidimensional tensors with unit condition number exist for dimension $s \in \{1, 2, 4, 8\}$ [122], corresponding to the Hurwitz problem with $l = m = n = s$. However, solutions to the Hurwitz problem with $l = m = n$

cannot exist for any other dimension. Furthermore, tight bounds exist on the dimension s_3 for a tensor of dimensions $s \times s \times s_3$ to have bounded condition number ($\inf\{\mathbf{f}_{\mathcal{T}}\} > 0$). This problem is equivalent to finding s_3 matrices of dimension $s \times s$, such that any nonzero linear combination thereof is invertible. Factorizing $s = 2^{4a+b}c$, where $b \in \{0,1,2,3\}$ and c is odd, $s_3 \leq 8a + 2^b$ [123], [124].

2.7.3 Properties of the Tensor Condition Number

In our analysis, we make use of the following submultiplicativity property of the tensor condition number with respect to multilinear multiplication (the property also generalizes to pairs of arbitrary order tensors contracted over one mode).

Lemma 2.5. *For any $\mathcal{T}\in\mathbb{R}^{s_1\times\cdots\times s_N}$ and matrix $\mathbf{M}$, if $\mathcal{V}=\mathcal{T}\times_N \mathbf{M}^T$ then $\kappa(\mathcal{V})\leq\kappa(\mathcal{T})\kappa(\mathbf{M})$.*

Proof. Assume $\kappa(\mathcal{V}) > \kappa(\mathcal{T})\kappa(\mathbf{M})$, then there exist unit vectors $\mathbf{x}^{(2)},\dots,\mathbf{x}^{(N)}$ and unit vectors $\mathbf{y}^{(2)},\dots,\mathbf{y}^{(N)}$ such that

$$\kappa(\mathcal{T})\kappa(\mathbf{M}) < \kappa(\mathcal{V}) = \frac{||\mathcal{V} \bigtimes_{i\in\{2,\dots,N\}} \mathbf{x}^{(i)T}||_2}{||\mathcal{V} \bigtimes_{i\in\{2,\dots,N\}} \mathbf{y}^{(i)T}||_2} = \frac{||\mathcal{T} \bigtimes_{i\in\{2,\dots,N-1\}} \mathbf{x}^{(i)T} \times_N \left(\mathbf{M}\mathbf{x}^{(N)}\right)^T||_2}{||\mathcal{T} \bigtimes_{i\in\{2,\dots,N-1\}} \mathbf{y}^{(i)T} \times_N \left(\mathbf{M}\mathbf{y}^{(N)}\right)^T||_2}. \tag{2.76}$$

Let $\mathbf{u} = \mathbf{M}\mathbf{x}^{(N)}$ and $\mathbf{v} = \mathbf{M}\mathbf{y}^{(N)}$, so $||\mathbf{u}||_2/||\mathbf{v}||_2 \leq \kappa(\mathbf{M})$, yielding a contradiction,

$$\kappa(\mathcal{V}) \leq \frac{||\mathcal{T} \bigtimes_{i\in\{2,\dots,N-1\}} \mathbf{x}^{(i)T} \times_N (\mathbf{u}/||\mathbf{u}||_2)^T||_2}{||\mathcal{T} \bigtimes_{i\in\{2,\dots,N-1\}} \mathbf{y}^{(i)T} \times_N (\mathbf{v}/||\mathbf{v}||_2)^T||_2}\kappa(\mathbf{M}) \leq \kappa(\mathcal{T})\kappa(\mathbf{M}). \tag{2.77}$$

Q.E.D.

Applying Lemma 2.5 with a vector, i.e. when $\mathbf{M} \in \mathbb{R}^{s_N\times 1}$ and so has condition number $\kappa(\mathbf{M}) = 1$, implies $\kappa\left(\mathcal{T} \times_N \mathbf{M}^T\right) \leq \kappa(\mathcal{T})$. By an analogous argument to the proof of Lemma 2.5, we can also conclude that the norm and infimum of such a product of $\mathcal{T}$ with unit vectors are bounded by those of $\mathcal{T}$, giving the following corollary.

Corollary 2.2. *For any $\mathcal{T} \in \mathbb{R}^{s_1\times\cdots\times s_N}$, vector $\mathbf{u} \in \mathbb{R}^{s_n}$, and any $n \in \{1,\dots,N\}$ such that $\exists m \in \{1,\dots,N\}$ with $s_m \geq s_n$ and $m \neq n$, if $\mathcal{V} = \mathcal{T} \times_n \mathbf{u}^T$, then $||\mathcal{V}||_2 \leq ||\mathbf{u}||_2||\mathcal{T}||_2$, $\inf\{\mathbf{f}_{\mathcal{V}}\} \geq ||\mathbf{u}||_2 \inf\{\mathbf{f}_{\mathcal{T}}\}$, and $\kappa(\mathcal{V}) \leq \kappa(\mathcal{T})$.*

For an orthogonal matrix $\mathbf{M}$, Lemma 2.5 can be applied in both directions, namely for $\mathcal{V} = \mathcal{T} \times_N \mathbf{M}^T$ and $\mathcal{T} = \mathcal{V} \times_N \mathbf{M}$, so we observe that $\kappa(\mathcal{V}) = \kappa(\mathcal{T})$. Using this fact, we demonstrate in the following theorem that any tensor $\mathcal{T}$ can be transformed by orthogonal matrices along each mode, so that one of its fibers has norm $||\mathcal{T}||_2/\kappa(\mathcal{T})$.

Theorem 2.6. *For any* $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{s_1\times\cdots\times s_N}$*, there exist orthogonal matrices* $\mathbf{Q}^{(2)}\ldots\mathbf{Q}^{(N)}$*, with* $\mathbf{Q}^{(i)} \in \mathbb{R}^{s_i\times s_i}$*, such that* $\boldsymbol{\mathcal{V}} = \boldsymbol{\mathcal{T}} \times_2 \mathbf{Q}^{(2)} \cdots \times_N \mathbf{Q}^{(N)}$ *satisfies* $\kappa(\boldsymbol{\mathcal{V}}) = \kappa(\boldsymbol{\mathcal{T}})$*,* $||\boldsymbol{\mathcal{V}}||_2 = ||\boldsymbol{\mathcal{T}}||_2$*, and the first fiber of* $\boldsymbol{\mathcal{V}}$*, i.e. the vector* $\mathbf{v}$ *with* $\mathbf{v}(i) = \boldsymbol{\mathcal{V}}(i, 0, \ldots, 0)$*, satisfies* $||\mathbf{v}||_2 = ||\boldsymbol{\mathcal{T}}||_2/\kappa(\boldsymbol{\mathcal{T}})$*.*

Proof. Given a tensor $\boldsymbol{\mathcal{T}}$ with infinite condition number, there must exist $N-1$ unit vectors $\mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}$, such that $||\boldsymbol{\mathcal{T}} \bigtimes_{i\in\{2,\ldots,N\}} \mathbf{x}^{(i)T}||_2 = ||\boldsymbol{\mathcal{T}}||_2/\kappa(\boldsymbol{\mathcal{T}})$. We define $N-1$ orthogonal matrices $\mathbf{Q}^{(2)}, \ldots, \mathbf{Q}^{(N)}$ such that $\mathbf{Q}^{(i)T}\mathbf{x}^{(i)} = \mathbf{e}_i$. We can then contract $\boldsymbol{\mathcal{T}}$ with these matrices along the last $N-1$ modes, resulting in $\boldsymbol{\mathcal{V}}$, with the same condition number as $\boldsymbol{\mathcal{T}}$ (by Lemma 2.5) and the same norm (by a similar argument). Then, we have that the first fiber of $\boldsymbol{\mathcal{V}}$ is

$$\mathbf{v} = \boldsymbol{\mathcal{V}} \bigtimes_{i\in\{2,\ldots,N\}} \mathbf{e}_i^T = \boldsymbol{\mathcal{T}} \bigtimes_{i\in\{2,\ldots,N\}} \mathbf{x}^{(i)T}, \tag{2.78}$$

and consequently $||\mathbf{v}||_2 = ||\boldsymbol{\mathcal{T}}||_2/\kappa(\boldsymbol{\mathcal{T}})$. Q.E.D.

By Theorem 2.6, the condition number of a tensor is infinity if and only if it can be transformed by products with orthogonal matrices along the last $N-1$ modes into a tensor with a zero fiber. Further, any tensor $\boldsymbol{\mathcal{T}}$ may be perturbed to have infinite condition number by adding to it some $\delta\boldsymbol{\mathcal{T}}$ with relative norm $||\delta\boldsymbol{\mathcal{T}}||_2/||\boldsymbol{\mathcal{T}}||_2 = 1/\kappa(\boldsymbol{\mathcal{T}})$.

2.7.4 PP-CP-ALS Error Bound using Tensor Condition Number

For CP decomposition, we obtain condition-number-dependent column-wise error bounds on $\mathbf{M}^{(n)}$ (the right-hand sides in the linear least squares subproblems), based on the magnitude of the relative perturbation to $\mathbf{A}^{(n)}$ since the formation of the pairwise perturbation operators.

Theorem 2.7. *If* $\frac{||d\mathbf{a}_k^{(n)}||_2}{||\mathbf{a}_k^{(n)}||_2} \le \epsilon \ll 1$ *for* $n \in \{1,\ldots,N\}, k \in \{1,\ldots,R\}$ *and* $s_m \le s_n$ *for any* $m \in \{1,\ldots,N\}$*, the pairwise perturbation algorithm without second order corrections computes* $\tilde{\mathbf{M}}^{(n)}$ *with column-wise error,*

$$\frac{||\tilde{\mathbf{m}}_k^{(n)} - \mathbf{m}_k^{(n)}||_2}{||\mathbf{m}_k^{(n)}||_2} = O\left(\epsilon^2\kappa(\boldsymbol{\mathcal{X}})\right), \tag{2.79}$$

where $\mathbf{M}^{(n)}$ *is the matrix given by a regular ALS sweep.*

Proof. We bound the error due to second order perturbations in $d\mathbf{A}^{(1)}, \ldots, d\mathbf{A}^{(n)}$, by similar analysis, higher-order perturbations would lead to errors smaller by a factor of $O(\text{poly}(N)\epsilon)$ and are consequently negligible if $\epsilon \ll 1$. Consider the order four tensors $\boldsymbol{\mathcal{M}}^{(i,j,n)}$ (shown in (2.13)) based on the current factor matrices $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$ and the pairwise perturbation operators $\mathcal{M}_p^{(i,j,n)}$ based on past factor matrices $\mathbf{A}_p^{(1)}, \ldots, \mathbf{A}_p^{(N)}$. The contribution of second order terms to the error is

$$\tilde{\mathbf{m}}_k^{(n)}(x) - \mathbf{m}_k^{(n)}(x) \approx \sum_{\substack{i,j \in \{1,\ldots,n-1,n+1,\ldots,N\} \\ i \neq j}} \sum_{y=1}^{s} \sum_{z=1}^{s} \mathcal{M}_p^{(i,j,n)}(x,y,z,k) d\mathbf{a}_k^{(i)}(y) d\mathbf{a}_k^{(j)}(z). \tag{2.80}$$

This absolute error has magnitude,

$$||\tilde{\mathbf{m}}_k^{(n)} - \mathbf{m}_k^{(n)}||_2 \leq \binom{N}{2} \max_{i,j} ||\mathcal{M}_p^{(i,j,n)}(:,:,:,k)||_2 ||d\mathbf{a}_k^{(i)}||_2 ||d\mathbf{a}_k^{(j)}||_2. \tag{2.81}$$

Using the fact that for any i, j we can express $\mathbf{m}_k^{(n)}$ as

$$\mathbf{m}_k^{(n)}(x) = \sum_{y=1}^{s} \sum_{z=1}^{s} \mathcal{M}^{(i,j,n)}(x,y,z,k) \mathbf{a}_k^{(i)}(y) \mathbf{a}_k^{(j)}(z), \tag{2.82}$$

we can lower bound the magnitude of the answer with respect to any $\mathcal{M}^{(i,j,n)}$,

$$||\mathbf{m}_k^{(n)}||_2 \geq \sup\{||\mathbf{f}_{\mathcal{M}^{(i,j,n)}(:,:,:,k)}||_2\} ||\mathbf{a}_k^{(i)}||_2 ||\mathbf{a}_k^{(j)}||_2. \tag{2.83}$$

Combining the upper bound on the absolute error with the lower bound on norm,

$$\frac{||\tilde{\mathbf{m}}_k^{(n)} - \mathbf{m}_k^{(n)}||_2}{||\mathbf{m}_k^{(n)}||_2} \leq \binom{N}{2} \max_{i,j} \frac{||\mathcal{M}_p^{(i,j,n)}(:,:,:,k)||_2 ||d\mathbf{a}_k^{(i)}||_2 ||d\mathbf{a}_k^{(j)}||_2}{\sup\{||\mathbf{f}_{\mathcal{M}^{(i,j,n)}(:,:,:,k)}||_2\} ||\mathbf{a}_k^{(i)}||_2 ||\mathbf{a}_k^{(j)}||_2}. \tag{2.84}$$

Lemma 2.5 implies that for any i, j, k,

$$||\mathcal{M}_p^{(i,j,n)}(:,:,:,k)||_2 \leq ||\mathcal{X}||_2 \prod_{l \in \{1,\ldots,N\} \setminus \{i,j,n\}} ||\mathbf{A}_p^{(l)}(:,k)||_2 \tag{2.85}$$

and that

$$\sup\{||\mathbf{f}_{\mathcal{M}^{(i,j,n)}(:,:,:,k)}||_2\} \geq \sup\{||\mathbf{f}_{\mathcal{X}}||_2\} \prod_{l \in \{1,\ldots,N\} \setminus \{i,j,n\}} ||\mathbf{A}^{(l)}(:,k)||_2. \tag{2.86}$$

Since, $||\mathbf{A}_p^{(l)}(:,k)||_2 \leq (1+\epsilon)||\mathbf{A}^{(l)}(:,k)||_2$, we obtain the bound,

$$\frac{||\tilde{\mathbf{m}}_k^{(n)} - \mathbf{m}_k^{(n)}||_2}{||\mathbf{m}_k^{(n)}||_2} \leq \binom{N}{2} \kappa(\mathcal{X})(1+\epsilon)^{N-3}\epsilon^2 \approx \binom{N}{2} \kappa(\mathcal{X})\epsilon^2. \tag{2.87}$$

Q.E.D.

This error bound is relative to the condition number of $\boldsymbol{\mathcal{X}}$, which means the bound is sensitive to the input tensor and that the error may be unbounded if $\boldsymbol{\mathcal{X}}$ has an exact CP decomposition of rank at most $\min_i s_i$.

2.7.5 PP-Tucker-ALS Error Bound using Tensor Condition Number

For Tucker decomposition, we again obtain bounds based on the perturbation to $\mathbf{A}^{(n)}$, this time for $\boldsymbol{\mathcal{Y}}^{(n)}$ (the tensors on whose matricizations a truncated SVD is performed). Using Lemma 2.5, we prove in Theorem 2.8 that when the tensor has same length in each mode and the relative error of the matrices $\mathbf{A}^{(n)}$ for $n \in \{1, \dots, N\}$ is small, the relative error for the $\tilde{\boldsymbol{\mathcal{Y}}}^{(n)}$ is also small.

Theorem 2.8. *Given an order N equidimensional tensor $\boldsymbol{\mathcal{X}}$ with size s, if $||d\mathbf{A}^{(n)}||_2 \leq \epsilon \ll 1$ for $n \in \{1, \dots, N\}$, the pairwise perturbation algorithm computes $\tilde{\boldsymbol{\mathcal{Y}}}^{(n)}$ with error,*

$$\frac{||\tilde{\boldsymbol{\mathcal{Y}}}^{(n)} - \boldsymbol{\mathcal{Y}}^{(n)}||_2}{||\boldsymbol{\mathcal{Y}}^{(n)}||_2} = O\left(\epsilon^2 \kappa(\boldsymbol{\mathcal{X}})\right). \tag{2.88}$$

Proof. As in Theorem 2.7, we bound the error due to second-order terms,

$$\frac{||\tilde{\boldsymbol{\mathcal{Y}}}^{(n)} - \boldsymbol{\mathcal{Y}}^{(n)}||_2}{||\boldsymbol{\mathcal{Y}}^{(n)}||_2} = \binom{N}{2} \max_{i,j} \frac{||\boldsymbol{\mathcal{Y}}_p^{(i,j,n)} \times_i d\mathbf{A}^{(i)T} \times_j d\mathbf{A}^{(j)T}||_2}{||\boldsymbol{\mathcal{Y}}^{(i,j,n)} \times_i \mathbf{A}^{(i)T} \times_j \mathbf{A}^{(j)T}||_2}. \tag{2.89}$$

From Lemma 2.5, we have

$$\frac{||\boldsymbol{\mathcal{Y}}_p^{(i,j,n)} \times_i d\mathbf{A}^{(i)T} \times_j d\mathbf{A}^{(j)T}||_2}{||\boldsymbol{\mathcal{Y}}^{(i,j,n)} \times_i \mathbf{A}^{(i)T} \times_j \mathbf{A}^{(j)T}||_2} \leq \frac{||\boldsymbol{\mathcal{Y}}_p^{(i,j,n)}||_2 ||d\mathbf{A}^{(i)}||_2 ||d\mathbf{A}^{(j)}||_2}{\sup\{||\mathbf{f}_{\boldsymbol{\mathcal{Y}}^{(i,j,n)}}||_2\} ||\mathbf{A}^{(i)}||_2 ||\mathbf{A}^{(j)}||_2}. \tag{2.90}$$

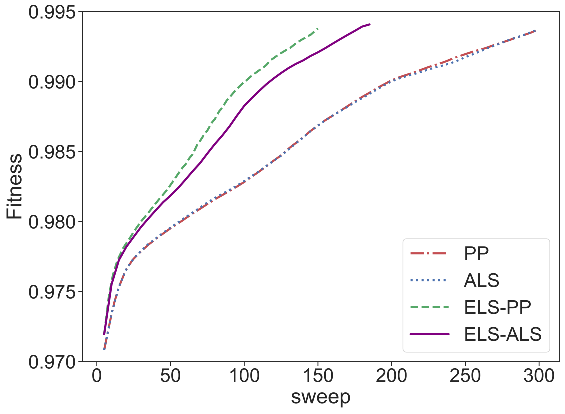
Since $\mathbf{A}^{(i)}$ and $\mathbf{A}^{(j)}$ are both matrices with orthonormal columns,

$$\frac{||\tilde{\boldsymbol{\mathcal{Y}}}^{(n)} - \boldsymbol{\mathcal{Y}}^{(n)}||_2}{||\boldsymbol{\mathcal{Y}}^{(n)}||_2} \leq \binom{N}{2} \max_{i,j} \frac{||\boldsymbol{\mathcal{Y}}_p^{(i,j,n)}||_2 ||d\mathbf{A}^{(i)}||_2 ||d\mathbf{A}^{(j)}||_2}{\sup\{||\mathbf{f}_{\boldsymbol{\mathcal{Y}}^{(i,j,n)}}||_2\}} = O\left(\epsilon^2 \kappa(\boldsymbol{\mathcal{X}})\right). \tag{2.91}$$

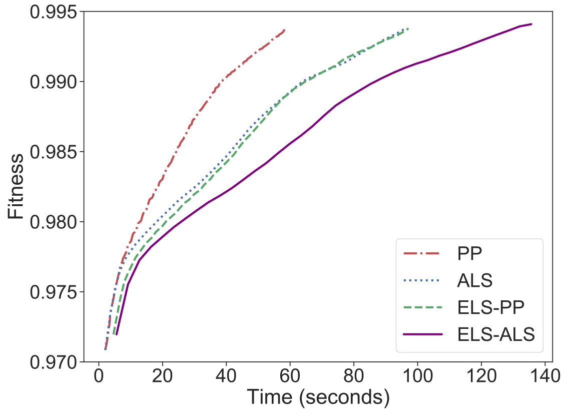
Q.E.D.

2.8 COMBINING PAIRWISE PERTURBATION WITH ENHANCED LINE SEARCH

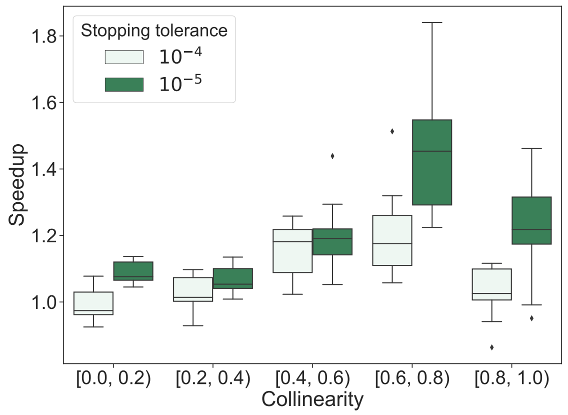
In this section, we compare the enhanced line search (ELS-ALS) [109] algorithm with an algorithm that combines pairwise perturbation with ELS (ELS-PP) for CP decomposition.



(a) $N = 3$, $s = R = 400$, collinearity$\in [0.6, 0.8)$



(b) $N = 3$, $s = R = 400$, collinearity$\in [0.6, 0.8)$



(c) $N = 3$, $s = R = 400$

Figure 2.8: **(a)** Fitness-sweep relation for the decomposition of one tensor with specific collinearity. **(b)** Fitness-time relation for the decomposition of one tensor with specific collinearity. **(c)** Box plot showing the speed-up of ELS-PP compared to ELS-ALS. Each box shows the 25th-75th quartiles, the median is indicated by a horizontal line inside the box, and outliers are displayed as dots.

The ELS-ALS algorithm is presented in Algorithm 2.6. For an order N input tensor $\boldsymbol{\mathcal{X}}$, the step size α_{ELS} is chosen based on

$$\alpha_{\text{ELS}} = \operatorname*{argmin}_{\alpha} \left\| \mathbf{X}_{(1)} - \left[\mathbf{A}^{(1)} + \alpha\left(\mathbf{A}^{(1)}_{\text{new}} - \mathbf{A}^{(1)}\right)\right] \left[\bigodot_{i=2}^{N} \mathbf{A}^{(i)} + \alpha\left(\mathbf{A}^{(i)}_{\text{new}} - \mathbf{A}^{(i)}\right)\right]^T \right\|_F^2, \quad (2.92)$$

Algorithm 2.6: ELS-ALS: enhanced line search for CP decomposition

```
1: **Input:** Tensor $\mathcal{X} \in \mathbb{R}^{s_1 \times \cdots s_N}$
2: Initialize $\left[\left[\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}\right]\right]$ as uniformly distributed random matrices within $[0,1]$
3: **while** not converge **do**
4:     Update $\mathbf{A}^{(n)}_{\text{new}}$ based on Line 7 in Algorithm 2.1 for $n \in \{1, \ldots, N\}$
5:     Get the ELS step size $\alpha_{\text{ELS}}$ based on (2.92)
6:     $\mathbf{A}^{(n)} \leftarrow \mathbf{A}^{(n)} + \alpha_{\text{ELS}} \left(\mathbf{A}^{(n)}_{\text{new}} - \mathbf{A}^{(n)}\right)$ for $n \in \{1, \ldots, N\}$
7: **end while**
8: **return** $\left[\left[\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}\right]\right]$
```

Algorithm 2.7: PP-CP-ALS: Pairwise perturbation procedure for CP-ALS

```
1: **Input:** tensor $\mathcal{X} \in \mathbb{R}^{s_1 \times \cdots \times s_N}$, PP tolerance $\epsilon < 1$
2: Initialize $\left[\left[\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}\right]\right]$ as uniformly distributed random matrices within $[0,1]$, initialize $d\mathbf{A}^{(n)} \leftarrow \mathbf{A}^{(n)}$
3: **while** not converge **do**
4:     **if** $\forall\, i \in \{1, \ldots, N\}, ||d\mathbf{A}^{(i)}||_F < \epsilon ||\mathbf{A}^{(i)}||_F$ **then**
5:         $\mathbf{A}^{(n)}_p \leftarrow \mathbf{A}^{(n)}$, $d\mathbf{A}^{(n)} \leftarrow \mathbf{O}$ for $n \in \{1, \ldots, N\}$
6:         **while** not converge and $\forall\, i \in \{1, \ldots, N\}, ||d\mathbf{A}^{(i)}||_F < \epsilon ||\mathbf{A}^{(i)}||_F$ **do**
7:             Update $\mathbf{A}^{(n)}_{\text{new}}$ based on Line 13 in Algorithm 2.3 for $n \in \{1, \ldots, N\}$
8:             $d\mathbf{A}^{(n)} = \mathbf{A}^{(n)}_{\text{new}} - \mathbf{A}^{(n)}_p$ for $n \in \{1, \ldots, N\}$
9:             Get the ELS step size $\alpha$ based on (2.94)
10:            $\mathbf{A}^{(n)} \leftarrow \mathbf{A}^{(n)}_p + \alpha_{\text{ELS}} \left(d\mathbf{A}^{(n)}\right)$ for $n \in \{1, \ldots, N\}$
11:        **end while**
12:    **end if**
13:    Perform an ELS-ALS sweep as in Algorithm 2.6, taking $d\mathbf{A}^{(n)} \leftarrow \alpha_{\text{ELS}} \left(\mathbf{A}^{(n)}_{\text{new}} - \mathbf{A}^{(n)}\right)$ for each $n \in \{1, \ldots, N\}$
14: **end while**
15: **return** $[[\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]]$
```

which minimizes an order $2N$ polynomial. When $N = 3$, (2.92) can be simplified to

$$\alpha_{\text{ELS}} = \underset{\alpha}{\operatorname{argmin}} \left\| \mathbf{X}_{(1)} - \left[\mathbf{A}^{(1)} + \alpha \cdot \delta\mathbf{A}^{(1)}\right] \left\{ \left[\mathbf{A}^{(2)} + \alpha \cdot \delta\mathbf{A}^{(2)}\right] \odot \left[\mathbf{A}^{(3)} + \alpha \cdot \delta\mathbf{A}^{(3)}\right] \right\}^T \right\|_F^2, \tag{2.93}$$

where $\delta\mathbf{A}^{(1)} = \mathbf{A}^{(1)}_{\text{new}} - \mathbf{A}^{(1)}$, $\delta\mathbf{A}^{(2)} = \mathbf{A}^{(2)}_{\text{new}} - \mathbf{A}^{(2)}$, and $\delta\mathbf{A}^{(3)} = \mathbf{A}^{(3)}_{\text{new}} - \mathbf{A}^{(3)}$. We direct readers to the reference [109] for an a detailed computational cost analysis of (2.92).

The ALS-PP algorithm is presented in Algorithm 2.7. When the algorithm is in the pairwise perturbation phase (satisfying the if condition in Line 4), the step size is chosen based on

$$\alpha_{\mathrm{ELS}} = \operatorname*{argmin}_{\alpha} \left\| \mathbf{X}_{(1)} - \left[\mathbf{A}_p^{(1)} + \alpha\left(\mathbf{A}_{\mathrm{new}}^{(1)} - \mathbf{A}_p^{(1)}\right)\right] \left[\bigodot_{i=2}^{N} \mathbf{A}_p^{(i)} + \alpha\left(\mathbf{A}_{\mathrm{new}}^{(i)} - \mathbf{A}_p^{(i)}\right)\right]^T \right\|_F^2 . \quad (2.94)$$

Note that the search directions are $\left(\mathbf{A}_{\mathrm{new}}^{(i)} - \mathbf{A}_p^{(i)}\right)$ for $i \in \{1, \ldots, N\}$ and w.r.t. $\mathbf{A}_p^{(i)}$ rather than $\mathbf{A}^{(i)}$, which is different from (2.92). Forming the explicit polynomial expression in (2.94) can leverage pre-computed pairwise perturbation operators and is cheaper.

We display the speed-ups of ELS-PP compared to ELS-ALS for synthetic tensors in Fig. 2.8. Fig. 2.8c shows the speed-up distribution with different exact factor matrices collinearity. We stop the algorithm when the stopping tolerance (defined as the fitness difference between two neighboring sweeps) is reached. It can be seen that ELS-PP achieves up to 1.8X speed-up, and high speed-up is achieved with tighter stopping tolerance.

Fig. 2.8a shows that both ELS-PP and ELS-ALS have faster convergence rate compared to PP and ALS algorithms. In addition, ELS-PP converges a bit faster than ELS-ALS. Fig. 2.8b shows that ELS-PP takes less time than ELS-ALS to reach the same final accuracy. Note that both ELS-PP and ELS-ALS take longer time compared to the standard ALS. This is consistent with the findings in the reference [109], where forming the line search polynomial could take a relatively long time.

Chapter 3: DISTRIBUTED PARALLEL CP DECOMPOSITION ALGORITHMS

In this Chapter, we efficiently parallelize two algorithms that accelerate the matricized tensor-times Khatri-Rao product (MTTKRP) calculations in CP-ALS for *dense* tensors [94]. These two algorithms are computationally more efficient than the state-of-the-art dimension tree-based CP-ALS algorithm. Additionally, our parallelization strategy is efficient in communication across processors.

First, we propose the *multi-sweep dimension tree* (MSDT) algorithm, which requires the tensor-times-matrix (TTM) between an order-N input tensor with dimension size s and the first-contracted input matrix once every $\frac{N-1}{N}$ sweeps and reduce the leading per-sweep computational cost of a rank-R CP-ALS to $2\frac{N}{N-1}s^N R$. This algorithm can produce exactly the same results as the standard dimension tree, i.e., it has no accuracy loss. Our algorithm leverages a parallelization strategy similar to previous work [62], [73] that performs the dimension tree calculations locally. Benchmark results of MSDT show a speed-up of 1.25X compared to the state-of-the-art dimension tree running on 1024 processors.

Second, we propose a communication-efficient pairwise perturbation (PP) algorithm. The pairwise perturbation algorithm is detailed in Chapter 2. The approach reduces communication cost by decomposing MTTKRP into small local MTTKRPs and constructing first-order PP operators in the PP initialization step as well as MTTKRP approximations in the PP approximated step for each local MTTKRP. Our benchmark results show that the PP approximated step achieves a speed-up of 1.94X compared to the state-of-the-art dimension tree running on 1024 processors.

3.1 BACKGROUND

3.1.1 Notations and Definitions

We use both element-wise and specialized tensor algebra notation [5]. Vectors are denoted with bold lowercase Roman letters (e.g., $\mathbf{v}$), matrices are denoted with bold uppercase Roman letters (e.g., $\mathbf{M}$), and tensors are denoted with bold calligraphic fonts (e.g., $\boldsymbol{\mathcal{T}}$). An order N tensor corresponds to an N-dimensional array. Elements of vectors, matrices, and tensors are denoted in parentheses, e.g., $\mathbf{v}(i)$ for a vector $\mathbf{v}$, $\mathbf{M}(i, j)$ for a matrix $\mathbf{M}$, and $\boldsymbol{\mathcal{T}}(i, j, k, l)$ for an order 4 tensor $\boldsymbol{\mathcal{T}}$. The ith column of $\mathbf{M}$ is denoted by $\mathbf{M}(:, i)$. Parenthesized superscripts are used to label different vectors, matrices and tensors (e.g. $\boldsymbol{\mathcal{T}}^{(1)}$ and $\boldsymbol{\mathcal{T}}^{(2)}$ are unrelated tensors).

Algorithm 3.1: CP-ALS: ALS for CP decomposition

```
1:  Input: Tensor $\mathcal{T} \in \mathbb{R}^{s_1 \times \cdots s_N}$, stopping criteria $\Delta$, rank $R$
2:  Initialize $[\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$ with $\mathbf{A}^{(i)} \in \mathbb{R}^{s_i \times R}$ with uniformly distributed random elements
    in $[0,1]$, $\mathbf{S}^{(i)} \leftarrow \mathbf{A}^{(i)T}\mathbf{A}^{(i)}$ for $i \in \{1, \ldots, N\}$
3:  $r \leftarrow 1, r_{\text{old}} \leftarrow 0$                              ▷ Initialize the relative residual
4:  while $|r - r_{\text{old}}| > \Delta$ do
5:      for $i \in \{1, \ldots, N\}$ do
6:          $\mathbf{\Gamma}^{(i)} \leftarrow \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(i-1)} * \mathbf{S}^{(i+1)} * \cdots * \mathbf{S}^{(N)}$
7:          Update $\mathbf{M}^{(i)}$ via dimension tree in Section 3.1.3
8:          $\mathbf{A}^{(i)} \leftarrow \mathbf{M}^{(i)}\mathbf{\Gamma}^{(i)\dagger}$,   $\mathbf{S}^{(i)} \leftarrow \mathbf{A}^{(i)T}\mathbf{A}^{(i)}$
9:      end for
10:     $r_{\text{old}} = r$
11:     Update $r$ based on (3.5)
12: end while
13: return $[\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$
```

The pseudo-inverse of matrix $\mathbf{A}$ is denoted with $\mathbf{A}^\dagger$. The Hadamard product of two matrices is denoted with $*$. The outer product of two or more vectors is denoted with $\circ$. The Kronecker product of two vectors/matrices is denoted with $\otimes$. For matrices $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{J \times K}$, their Khatri-Rao product resulting in a matrix of size $(IJ) \times K$ defined by $\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1, \ldots, \mathbf{a}_K \otimes \mathbf{b}_K]$. The mode-$n$ TTM of an order N tensor $\mathcal{T} \in \mathbb{R}^{s_1 \times \cdots \times s_N}$ with a matrix $\mathbf{A} \in \mathbb{R}^{J \times s_n}$ is denoted by $\mathcal{T} \times_n \mathbf{A}$, whose output size is $s_1 \times \cdots \times s_{n-1} \times J \times s_{n+1} \times \cdots \times s_N$. Matricization is the process of unfolding a tensor into a matrix. The mode-n matricized version of $\mathcal{T}$ is denoted by $\mathbf{T}_{(n)} \in \mathbb{R}^{s_n \times K}$ where $K = \prod_{m=1, m \neq n}^{N} s_m$. We generalize this notation to define the unfoldings of a tensor $\mathcal{T}$ with dimensions $s_1 \times \cdots \times s_N$ into an order $M+1$ tensor, $\mathcal{T}_{(i_1, \ldots, i_M)} \in \mathbb{R}^{s_{i_1} \times \cdots \times s_{i_M} \times K}$, where $K = \prod_{j \in \{1, \ldots, N\} \setminus \{i_1, \ldots, i_M\}} s_j$. For instance, if $\mathcal{T}$ is an order 4 tensor, $\mathcal{T}(j,k,l,m) = \mathcal{T}_{(1,3)}(j, l, k + (m-1)s_2)$.

We use calligraphic fonts (e.g., $\mathcal{P}$) to denote the tensor representing a logical multidimensional processor grid. Similar to the representation of tensor elements, the index of a specific processor in the grid is denoted in parentheses, e.g., $\mathcal{P}(i,j)$ denotes one processor indexed by i, j in the 2-dimensional grid. For a processor grid $\mathcal{P}$ with size $I_1 \times \cdots \times I_N$ and a tensor $\mathcal{T} \in \mathbb{R}^{s_1 \times \cdots \times s_N}$, let $x = (x_1, \ldots, x_N)$ denote the processor index, we use $\mathcal{T}_{\mathcal{P}(x)}$ to denote the local tensor residing on a processor indexed by x. The size of the local tensor will be $\lceil \frac{s_1}{I_1} \rceil \times \cdots \times \lceil \frac{s_N}{I_N} \rceil$. When $\frac{s_i}{I_i}$ is not an integer, padding is added to the tensor.

3.1.2 CP Decomposition with ALS

The goal of the CP tensor decomposition is to minimize the following objective function,

$$f(\mathbf{A}^{(1)},\dots,\mathbf{A}^{(N)}) \coloneqq \frac{1}{2}\|\boldsymbol{\mathcal{T}} - [\![\mathbf{A}^{(1)},\cdots,\mathbf{A}^{(N)}]\!]\|_F^2, \tag{3.1}$$

where $[\![\mathbf{A}^{(1)},\cdots,\mathbf{A}^{(N)}]\!] := \sum_{r=1}^{R} \mathbf{A}^{(1)}(:,r)\circ\cdots\circ\mathbf{A}^{(N)}(:,r)$. CP-ALS alternates among subproblems for each of the factor matrices $\mathbf{A}^{(n)}$. Each subproblem is quadratic, with the optimality condition $\frac{\partial f}{\partial \mathbf{A}^{(n)}} = 0$ resulting in the update expression $\mathbf{A}^{(n)}_{\text{new}}\mathbf{\Gamma}^{(n)} = \mathbf{T}_{(n)}\mathbf{P}^{(n)}$, where the matrix $\mathbf{P}^{(n)} \in \mathbb{R}^{I_n\times R}$, with $I_n = \prod_{i=1,i\neq n}^{N} s_i$, is formed by Khatri-Rao products of the other factor matrices,

$$\mathbf{P}^{(n)} = \mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)}, \tag{3.2}$$

and $\mathbf{\Gamma} \in \mathbb{R}^{R\times R}$ is computed via a chain of Hadamard products,

$$\mathbf{\Gamma}^{(n)} = \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(n-1)} * \mathbf{S}^{(n+1)} * \cdots * \mathbf{S}^{(N)}, \tag{3.3}$$

with each $\mathbf{S}^{(i)} = \mathbf{A}^{(i)T}\mathbf{A}^{(i)}$. The MTTKRP computation $\mathbf{M}^{(n)} = \mathbf{T}_{(n)}\mathbf{P}^{(n)}$ is the main computational bottleneck of CP-ALS. The computational cost of MTTKRP is $\mathcal{O}(s^N R)$ if $s_n = s$ for all $n \in \{1,\dots,N\}$. The naive implementation of CP-ALS for a dense tensor calculates N MTTKRPs for each ALS sweep, leading to the cost of $\mathcal{O}(Ns^N R)$. With the dimension tree algorithm, the computational complexity for all the MTTKRP calculations in one ALS sweep is $4s^N R$ flops to leading order in s. The dimension tree algorithm will be detailed in Section 3.1.3. Algorithm 3.1 presents the CP-ALS procedure described above, performing updates until the relative decomposition residual of the neighboring sweeps is sufficiently small. Let $\tilde{\boldsymbol{\mathcal{T}}}$ denote the tensor reconstructed by the factor matrices, the relative residual norm is defined as

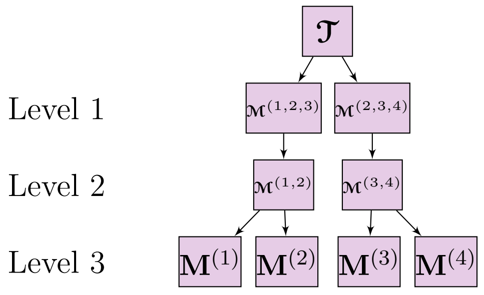
$$r = \frac{\|\boldsymbol{\mathcal{T}} - \tilde{\boldsymbol{\mathcal{T}}}\|_F}{\|\boldsymbol{\mathcal{T}}\|_F}. \tag{3.4}$$

As shown in [62], [64], [68], r can be calculated efficiently via

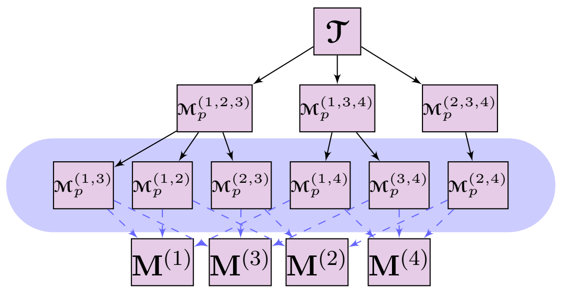
$$r = \frac{\sqrt{\|\boldsymbol{\mathcal{T}}\|_F + \langle \mathbf{\Gamma}^{(N)}, \mathbf{A}^{(N)T}\mathbf{A}^{(N)}\rangle - 2\langle \mathbf{M}^{(N)}, \mathbf{A}^{(N)}\rangle}}{\|\boldsymbol{\mathcal{T}}\|_F}, \tag{3.5}$$

assuming that terms $\|\boldsymbol{\mathcal{T}}\|_F, \mathbf{\Gamma}^{(N)}, \mathbf{M}^{(N)}$ are all amortized before the residual calculations.

3.1.3 The Dimension Tree Algorithm



(a) ALS dimension tree with $N = 4$



(b) PP dimension tree with $N = 4$

Figure 3.1: Dimension trees for ALS and pairwise perturbation. The blue region in (b) denotes the PP operators.

In each CP-ALS sweep, the TTM and TTV operations for MTTKRP calculations can be amortized and reused. Such amortization strategies are referred to as dimension tree algorithms. A dimension tree data structure partitions the mode indices of an order N tensor hierarchically and constructs the intermediate tensors accordingly [58], [59], [62]. The root of the tree corresponds to the input tensor and the leaves consist of all the N factor matrices. It is assumed in the literature that each ALS sweep uses the same tree.

It has been shown in [58] that for CP decomposition, an optimal dimension tree must be binary. Therefore, our analysis will focus on binary trees. We illustrate one dimension tree for $N = 4$ in Fig. 3.1a. The first level contractions (contractions between the input tensor and one factor matrix) are done via TTM. For an equidimensional tensor with size s and rank R, these contractions have a computational cost of $\mathcal{O}(s^N R)$ and are generally the most time-consuming part of ALS. Other contractions (transforming one intermediate into another intermediate) are done via batched TTV (also called multi-TTV/mTTV), and the computational cost of an ith level contraction, where $0 < i < N$, is $\mathcal{O}(s^{N+1-i}R)$. Because two first level contractions are necessary for the construction of the dimension tree, as is illustrated in Fig. 3.1a, to calculate all the MTTKRP results in one ALS sweep, to leading order in s, the computational complexity is $4s^N R$ flops.

Algorithm 3.2: PP-CP-ALS: Pairwise perturbation for CP-ALS

```
1: Input: Tensor $\mathcal{T} \in \mathbb{R}^{s_1\times\cdots\times s_N}$, stopping criteria $\Delta$, PP tolerance $\epsilon < 1$
2: Initialize $[\![\mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)}]\!]$ with $\mathbf{A}^{(i)} \in \mathbb{R}^{s_i\times R}$ with uniformly distributed random elements
   in $[0,1]$, $d\mathbf{A}^{(i)} \leftarrow \mathbf{A}^{(i)}$, $\mathbf{S}^{(i)} \leftarrow \mathbf{A}^{(i)T}\mathbf{A}^{(i)}$ for $i \in \{1,\ldots,N\}$
3: $r \leftarrow 1, r_{\text{old}} \leftarrow 0$                                   ▷ Initialize the relative residual
4: while $|r - r_{\text{old}}| > \Delta$ do
5:     if $\forall\, i \in \{1,\ldots,N\}, ||d\mathbf{A}^{(i)}||_F < \epsilon||\mathbf{A}^{(i)}||_F$ then
6:         for $i \in \{1,\ldots,N\}$ do
7:             $\mathbf{A}_p^{(i)} \leftarrow \mathbf{A}^{(i)}$, $d\mathbf{A}^{(i)} \leftarrow \mathbf{O}$
8:         end for
9:         Compute $\mathcal{M}_p^{(i,n)}, \mathbf{M}_p^{(n)}$ for $i, n \in \{1,\ldots,N\}$ via dimension tree
10:        while $\forall\, i \in \{1,\ldots,N\}, ||d\mathbf{A}^{(i)}||_F < \epsilon||\mathbf{A}^{(i)}||_F$ do
11:            for $j \in \{1,\ldots,N\}$ do
12:                $\mathbf{\Gamma}^{(j)} \leftarrow \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(j-1)} * \mathbf{S}^{(j+1)} * \cdots * \mathbf{S}^{(N)}$
13:                Update $\tilde{\mathbf{M}}^{(j)}$ based on (3.7)
14:                $\mathbf{A}^{(j)} \leftarrow \tilde{\mathbf{M}}^{(j)}\mathbf{\Gamma}^{(j)\dagger}$
15:                $d\mathbf{A}^{(j)} = \mathbf{A}^{(j)} - \mathbf{A}_p^{(j)}$, $\mathbf{S}^{(j)} \leftarrow \mathbf{A}^{(j)T}\mathbf{A}^{(j)}$
16:            end for
17:        end while
18:    end if
19:    Perform a regular ALS sweep as in Algorithm 3.1 (line 5-9), taking $d\mathbf{A}^{(i)}$ as the
       update of $\mathbf{A}^{(i)}$ in one sweep for each $i \in \{1,\ldots,N\}$
20:    $r_{\text{old}} = r$
21:    Update $r$ based on (3.5)
22: end while
23: return $[\![\mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)}]\!]$
```

3.1.4 The Pairwise Perturbation Algorithm

Before the introduction of PP, we define the partially contracted MTTKRP intermediates $\mathcal{M}^{(i_1,i_2,\ldots,i_m)}$ as follows,

$$\mathcal{M}^{(i_1,i_2,\ldots,i_m)} = \mathcal{T}_{(i_1,i_2,\ldots,i_m)} \bigodot_{j\in\{1,\ldots,N\}\setminus\{i_1,i_2,\ldots,i_m\}} \mathbf{A}^{(j)}, \tag{3.6}$$

where $\mathcal{M}^{(1,\ldots,N)}$ is the input tensor $\mathcal{T}$. Let $\mathbf{A}_p^{(n)}$ denote the $\mathbf{A}^{(n)}$ calculated with regular ALS at some number of sweeps prior to the current one, we also define $\mathcal{M}_p^{(i_1,i_2,\ldots,i_m)}$ in the same way as $\mathcal{M}^{(i_1,i_2,\ldots,i_m)}$ in (3.6), except that $\mathcal{T}$ is contracted with $\mathbf{A}_p^{(j)}$ for $j \in \{1,\ldots,N\}\setminus\{i_1,i_2,\ldots,i_m\}$.

Pairwise perturbation (PP) uses perturbative corrections to the subproblems rather than recomputing the tensor contractions and contains two steps. The initialization step calculates the PP operators $\mathcal{M}_p^{(i,j)}$ for $\forall i,j \in \{1,\ldots,N\}, i < j$. Similar to CP-ALS, computation of these operators can also benefit from dimension trees. Fig. 3.1b describes the PP dimension tree for $N = 4$. In the PP dimension tree, $\binom{l+1}{2}$ tensors $\mathcal{M}_p^{(i,j,j+1,\ldots,j+N-l-1)}, \forall i, j \in \{1, \ldots, l+1\}, i < j$ are calculated at the lth level. The construction of PP operators with the dimension tree costs $4s^N R$ flops to leading order (despite the fact that three first level intermediates are needed, only two of them need to be recalculated, and the remaining one, e.g., $\mathcal{M}_p^{(2,3,4)}$ in Fig. 3.1b, can be amortized from the last DT sweep), which is computationally as expensive as a sweep of the ALS dimension tree algorithm.

The approximated step uses the PP operators to approximate the MTTKRP $\mathbf{M}^{(n)}$ with $\tilde{\mathbf{M}}^{(n)}$ as follows,

$$\tilde{\mathbf{M}}^{(n)} = \mathbf{M}_p^{(n)} + \sum_{i=1, i\neq n}^{N} \mathbf{U}^{(n,i)} + \mathbf{V}^{(n)}, \tag{3.7}$$

$$\text{where} \quad \mathbf{U}^{(n,i)}(x,k) = \sum_{y=1}^{s_i} \mathcal{M}_p^{(n,i)}(x,y,k) d\mathbf{A}^{(i)}(y,k), \tag{3.8}$$

$$\mathbf{V}^{(n)} = \mathbf{A}^{(n)} \left(\sum_{i,j=1\neq n, i<j}^{N} d\mathbf{S}^{(i)} * d\mathbf{S}^{(j)} * \mathop{\Large\ast}_{k=1, k\neq i,j,n}^{N} \mathbf{S}^{(k)} \right), \tag{3.9}$$

and $d\mathbf{S}^{(i)} = \mathbf{A}^{(i)T} d\mathbf{A}^{(i)}$. Terms $\mathbf{U}^{(n,i)}$ are the first-order corrections computed via the PP operators, and the term $\mathbf{V}^{(n)}$ is the second-order correction to lower the error to a greater extent. Given $\mathcal{M}_p^{(n,i)}$ and $\mathbf{M}_p^{(n)}$, calculation of $\tilde{\mathbf{M}}^{(n)}$ for $n \in \{1, \ldots, N\}$ requires $2N(Ns^2R + NR^2 + sR^2)$ flops overall. Algorithm 3.2 presents the PP-CP-ALS method described above. The PP initialization step is shown on line 9 and the PP approximated step is shown on line 13. We direct readers to Section 3.1 in reference [93] for an illustration of the algorithm on order 3 tensors.

3.1.5 Cost Model

We use the BSP $\alpha - \beta - \gamma$ model for parallel cost analysis [125], [126]. In addition to considering the communication cost of sending data among processors (horizontal communication cost), we use another parameter, ν, to measure the cost of transferring data between slow memory and cache (vertical communication cost) [127]. α represents the cost of sending/receiving a single message (latency cost), β represents the cost of moving a single word among processors, ν represents the cost of moving a single word between the main memory and cache, and γ represents the cost to perform one floating point operation. We assume $\alpha \gg \beta \gg \gamma$ and $\nu \leq \gamma \cdot \sqrt{H}$, where H is the cache size. Below, we summarize the communication collective routines used in the parallel algorithms, including All-Gather, Reduce-Scatter and All-Reduce, with costs assuming on a fully-connected network,

- All-Gather(v, Procs) collects a vector v that is distributed across P processors (Procs) and stores the concatenation of all the data of total size n redundantly on all processors. Its cost is $\log P \cdot \alpha + n\delta(P) \cdot \beta$, where $\delta(P) = 1$ if $P > 1$ and $\delta(P) = 0$ otherwise.

- Reduce-Scatter(v, Procs) sums a vector v that is distributed across Procs and partitions the result across Procs. Its cost is $\log P \cdot \alpha + n\delta(P) \cdot \beta$.

- All-Reduce(v, Procs) sums a vector v that is distributed across Procs and stores the result redundantly on all processors. Its cost is $2 \log P \cdot \alpha + 2n\delta(P) \cdot \beta$.

3.1.6 Parallel CP-ALS

Algorithm 3.3: Par-CP-ALS: Parallel CP-ALS

```
1: Input: Processor grid $\mathcal{P}$ with dimension $I_1 \times \cdots \times I_N$, where $I = \prod_{i=1}^{N} I_i$, tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{s_1 \times \cdots s_N}$ distributed over $\mathcal{P}$
2: $\mathcal{Q} \leftarrow \mathcal{P}$ reshaped to a 2-d array with dimension $I \times 1$
3: $\triangleright$ $x$ denotes one processor in the grid whose index is $(x_1, \ldots, x_N)$ in $\mathcal{P}$, and the index is $(x, 1)$ in $\mathcal{Q}$
4: for $i \in \{2, \ldots, N\}$ do
5:     Initialize $\mathbf{A}^{(i)}_{\mathcal{Q}(x)} \in \mathbb{R}^{\lceil \frac{s_i}{I} \rceil \times R}$
6:     $\mathbf{S}^{(i)}_{\mathcal{Q}(x)} \leftarrow \mathbf{A}^{(i)T}_{\mathcal{Q}(x)} \mathbf{A}^{(i)}_{\mathcal{Q}(x)}$
7:     $\mathbf{S}^{(i)}_{\mathcal{Q}(x)} \leftarrow$ All-Reduce($\mathbf{S}_{\mathcal{Q}(x)}$, All-Procs)
8:     $\mathbf{A}^{(i)}_{\mathcal{P}(x)} \leftarrow$ All-Gather($\mathbf{A}^{(i)}_{\mathcal{Q}(x)}$, Proc-Slice($\mathcal{P}_{(i)}(x_i, :)$)
9: end for
10: while Stopping criteria not reached do
11:     for $i \in \{1, \ldots, N\}$ do
12:         $\mathbf{\Gamma}^{(i)}_{\mathcal{Q}(x)} \leftarrow \mathbf{S}^{(1)}_{\mathcal{Q}(x)} * \cdots * \mathbf{S}^{(i-1)}_{\mathcal{Q}(x)} * \mathbf{S}^{(i+1)}_{\mathcal{Q}(x)} * \cdots * \mathbf{S}^{(N)}_{\mathcal{Q}(x)}$
13:         $\mathbf{M}^{(i)}_{\mathcal{P}(x)} \leftarrow$ Local-MTTKRP($\boldsymbol{\mathcal{T}}_{\mathcal{P}(x)}, \{\mathbf{A}^{(1)}_{\mathcal{P}(x)}, \ldots, \mathbf{A}^{(N)}_{\mathcal{P}(x)}\}, i$) via dimension tree in Section 3.1.3
14:         $\mathbf{M}^{(i)}_{\mathcal{Q}(x)} \leftarrow$ Reduce-Scatter($\mathbf{M}^{(i)}_{\mathcal{P}(x)}$, Proc-Slice($\mathcal{P}_{(i)}(x_i, :)$))
15:         $\mathbf{A}^{(1)}_{\mathcal{Q}(x)} \leftarrow \mathbf{M}^{(i)}_{\mathcal{Q}(x)} \mathbf{\Gamma}^{(i)\dagger}_{\mathcal{Q}(x)}$,    $\mathbf{S}^{(i)}_{\mathcal{Q}(x)} \leftarrow \mathbf{A}^{(i)T}_{\mathcal{Q}(x)} \mathbf{A}^{(i)}_{\mathcal{Q}(x)}$
16:         $\mathbf{S}^{(i)}_{\mathcal{Q}(x)} \leftarrow$ All-Reduce($\mathbf{S}_{\mathcal{Q}(x)}$, All-Procs)
17:         $\mathbf{A}^{(i)}_{\mathcal{P}(x)} \leftarrow$ All-Gather($\mathbf{A}^{(i)}_{\mathcal{Q}(x)}$, Proc-Slice($\mathcal{P}_{(i)}(x_i, :)$)
18:     end for
19: end while
20: return $[\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$.
```

Our parallel CP-ALS on dense tensors are based on Algorithm 3.3, which is introduced in [62], [73]. The input $\boldsymbol{\mathcal{T}}$ with order N is uniformly distributed across an order N processor grid $\mathcal{P}$, and all the factor matrices are initially distributed such that each processor owns a subset of the rows. The main idea of the algorithm is to decompose the MTTKRP into local MTTKRP calculations to reduce the expensive communication cost, e.g., if $\mathbf{A}^{(1)} = \text{MTTKRP}(\boldsymbol{\mathcal{T}}, \mathbf{A}^{(2)}, \mathbf{A}^{(3)})$ then

$$\mathbf{A}^{(1)}_{\mathcal{P}(i,j,k)} = \sum_{j,k} \text{Local-MTTKRP}\left(\boldsymbol{\mathcal{T}}_{\mathcal{P}(i,j,k)}, \mathbf{A}^{(2)}_{\mathcal{P}(i,j,k)}, \mathbf{A}^{(3)}_{\mathcal{P}(i,j,k)}\right), \tag{3.10}$$

where i, j, k denote the location of a processor in a 3D grid. Before the MTTKRP calculation, all the factor matrices are redistributed (lines 8,17). For the ith mode factor matrix $\mathbf{A}^{(i)}$, all processors having the same ith index in the processor grid $\mathcal{P}$ redundantly own the same $\lceil \frac{s_i}{I_i} \rceil$ rows of $\mathbf{A}^{(i)}$. This setup prepares all the local factor matrices $\mathbf{A}^{(1)}_{\mathcal{P}(x)}, \ldots, \mathbf{A}^{(N)}_{\mathcal{P}(x)}$ necessary for a local MTTKRP on each processor indexed x. The local MTTKRP routine (line 13) independently performs MTTKRP calculations on each processor, and this step requires no communication. The local MTTKRP can be efficiently calculated with the dimension tree techniques. After the local MTTKRP, Reduce-Scatter (line 14) is performed to sum-up the local contributions and the MTTKRP outputs $\mathbf{M}^{(i)}$ are distributed such that each processor owns a subset of the rows. Moreover, all the Gram matrices $\mathbf{S}^{(i)}$, once updated, are then distributed redundantly on all the processors via All-Reduce (lines 7,16), allowing for the linear systems to be solved locally.

For each processor, the leading order computational cost is $\frac{4s^N R}{P} \cdot \gamma$, assuming the dimension size is s and the state-of-the-art dimension tree algorithm is used. Three collective routines (lines 14,16,17) are called for each factor matrix update. When $I_1 = \cdots = I_N = P^{\frac{1}{N}}$, the horizontal communication cost for each ALS sweep is $\mathcal{O}(N \log(P) \cdot \alpha + N(sR/P^{\frac{1}{N}} + R^2) \cdot \beta)$.

Unlike Algorithm 3.3, our implementation calculates all Hadamard products and linear system solves in a parallel way, leveraging a distributed-memory matrix library for solving symmetric positive definite linear systems. Distributing the work in the solve reduces the computational and bandwidth costs, while raising the latency cost. Our performance evaluation in Section 3.4.2 also includes the PLANC implementation of CP-ALS [73], which makes use of a sequential linear system solve. As we will show, the cost of solving linear systems is often small for both approaches, since the MTTKRP calculations are the major bottleneck.

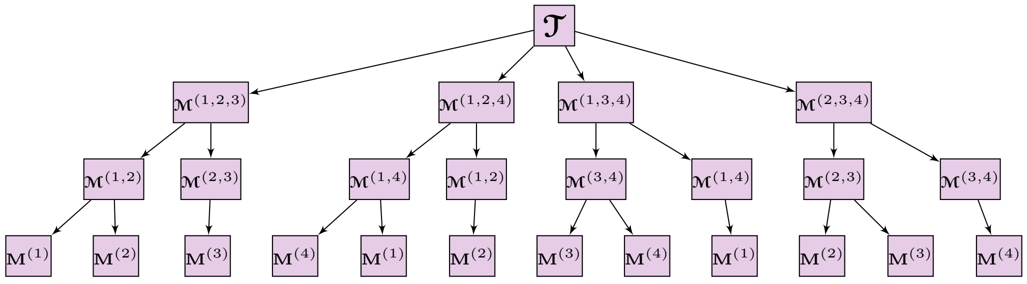


Figure 3.2: Multi-sweep dimension tree with $N = 4$.

The standard dimension tree for CP-ALS constructs the tensor contraction paths based on a fixed amortization scheme for different ALS sweeps. It requires two first level TTM calculations, one for the MTTKRP of right-half modes and the other for the left-half modes, as is shown in Fig. 3.1a. However, cost of CP-ALS can be reduced further by amortizing first-level TTM contractions across sweeps. Given an order N tensor $\mathcal{T}$, the first level TTM $\mathcal{T} \times_i \mathbf{A}^{(i)}$ can be used for the MTTKRP of all the modes except i. For example, with order $N = 4$, the contraction between $\mathcal{T}$ and $\mathbf{A}^{(4)}$ can be used for the construction of $\mathbf{M}^{(1)}, \mathbf{M}^{(2)}, \mathbf{M}^{(3)}$. After this, the TTM with $\mathbf{A}^{(3)}$ can be used to compute not only the remaining term for this sweep, $\mathbf{M}^{(4)}$, but also $\mathbf{M}^{(1)}$ and $\mathbf{M}^{(2)}$ for the next sweep. Given that each first-level TTM can be used to compute $N-1$ terms $\mathbf{M}^{(i)}$, we should be able to compute $N-1$ sweeps using N such TTMs, as opposed to the $2(N-1)$ needed by a typical dimension tree.

Our multi-sweep dimension tree (MSDT) algorithm achieves this goal, and we illustrate the tree in Fig. 3.2. Each MSDT tree is responsible for the MTTKRP calculations of $N-1$ sweeps. It includes N subtrees, and the root of ith subtree is the first level contraction $\mathcal{T} \times_{N-i+1} \mathbf{A}^{(N-i+1)}$, which is used for the MTTKRP of all the $N-1$ modes except $N-i+1$, with the calculation order being $(N-i+1)+1 \mod N \prec \cdots \prec (N-i+1)+N-1 \mod N$. Each subtree can be constructed with the traditional binary dimension tree. MSDT achieves the computational cost of $2\frac{N}{N-1}s^N R + \mathcal{O}(s^{N-1}R)$ flops for each sweep. Its leading order cost is only $\frac{N}{2(N-1)}$ times that of the state-of-the-art dimension tree, thus speeding up the CP-ALS algorithm. Note that the MSDT algorithm can also be used to accelerate the Higher Order Orthogonal Iteration (HOOI) algorithm for Tucker decompositions [5] in a similar way.

Algorithm 3.4: Par-PP-CP-ALS-subroutine

```
1: Assume: Local matrices $\mathbf{A}^{(i)}_{\mathcal{P}(x)}, d\mathbf{A}^{(i)}_{\mathcal{P}(x)}$ $\forall i \in \{1,\dots,N\}$ prepared for local-MTTKRPs
2: Call Local-PP-init($\boldsymbol{\mathcal{T}}_{\mathcal{P}(x)}, \{\mathbf{A}^{(1)}_{\mathcal{P}(x)},\dots,\mathbf{A}^{(N)}_{\mathcal{P}(x)}\}$) to update all $\boldsymbol{\mathcal{M}}^{(i,n)}_{p,\mathcal{P}(x)}, \mathbf{M}^{(n)}_{p,\mathcal{P}(x)}$
3: while $\forall\ i \in \{1,\dots,N\}, ||d\mathbf{A}^{(i)}||_F < \epsilon||\mathbf{A}^{(i)}||_F$ do
4:     for $n \in \{1,\dots,N\}$ do
5:         for $i \in \{1,\dots,N\}, i \neq n$ do
6:             Update $\mathbf{U}^{(n,i)}_{\mathcal{P}(x)}$ with $\mathbf{U}^{(n,i)}_{\mathcal{P}(x)}(a,k) \leftarrow \sum_b \boldsymbol{\mathcal{M}}^{(n,i)}_{p,\mathcal{P}(x)}(a,b,k) d\mathbf{A}^{(i)}_{\mathcal{P}(x)}(b,k)$
7:         end for
8:         $\tilde{\mathbf{M}}^{(n)}_{\mathcal{P}(x)} \leftarrow \mathbf{M}^{(n)}_{p,\mathcal{P}(x)} + \sum_{i=1,i\neq n}^{N} \mathbf{U}^{(n,i)}_{\mathcal{P}(x)}$
9:         Update global $\tilde{\mathbf{M}}^{(n)}$ based on $\tilde{\mathbf{M}}^{(n)}_{\mathcal{P}(x)}$
10:        Calculate $\mathbf{V}^{(n)}$ based on (3.9)
11:        $\tilde{\mathbf{M}}^{(n)} \leftarrow \tilde{\mathbf{M}}^{(n)} + \mathbf{V}^{(n)}$
12:        $\boldsymbol{\Gamma}^{(n)} \leftarrow \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(n-1)} * \mathbf{S}^{(n+1)} * \cdots * \mathbf{S}^{(N)}$
13:        $\mathbf{A}^{(n)} \leftarrow \tilde{\mathbf{M}}^{(n)} \boldsymbol{\Gamma}^{(n)\dagger}$
14:        $d\mathbf{A}^{(n)} = \mathbf{A}^{(n)} - \mathbf{A}^{(n)}_p$, $\mathbf{S}^{(n)} \leftarrow \mathbf{A}^{(n)T}\mathbf{A}^{(n)}$
15:        Prepare local factors $\mathbf{A}^{(n)}_{\mathcal{P}(x)}, d\mathbf{A}^{(n)}_{\mathcal{P}(x)}$ from $\mathbf{A}^{(n)}, d\mathbf{A}^{(n)}$
16:    end for
17: end while
```

	Seq computational cost	Local computational cost	Auxiliary memory	Horizontal communication cost	Vertical communication cost
DT	$4s^NR\cdot\gamma$	$4s^NR/P\cdot\gamma$	$(s^N/P)^{1/2}R$	$\mathcal{O}(N\log(P)\cdot\alpha + NsR/P^{\frac{1}{N}}\cdot\beta)$	$\mathcal{O}((s^N/P + (s^N/P)^{1/2}R)\cdot\nu)$
MSDT	$\frac{2N}{N-1}s^NR\cdot\gamma$	$\frac{2N}{N-1}s^NR/P\cdot\gamma$	$(s^N/P)^{\frac{N-1}{N}}R$	$\mathcal{O}(N\log(P)\cdot\alpha + NsR/P^{\frac{1}{N}}\cdot\beta)$	$\mathcal{O}((s^N/P + (s^N/P)^{\frac{N-1}{N}}R)\cdot\nu)$
PP-init	$4s^NR\cdot\gamma$	$4s^NR/P\cdot\gamma$	$(s^N/P)^{\frac{N-1}{N}}R$	/	$\mathcal{O}((s^N/P + (s^N/P)^{\frac{N-1}{N}}R)\cdot\nu)$
PP-init-ref	$4s^NR\cdot\gamma$	$4s^NR/P\cdot\gamma$	$s^{N-1}R/P$	$\mathcal{O}(N\log(P)\cdot\alpha + N(s^NR/P)^{2/3}\cdot\beta)$ or $\mathcal{O}(N\log(P)\cdot\alpha + N(s^N/p)^{\frac{N-1}{N}}R\cdot\beta)$	$\mathcal{O}((s^N/P + (s^N/P)^{\frac{N-1}{N}}R)\cdot\nu)$
PP-approx	$2N^2(s^2R+R^2)\cdot\gamma$	$2N^2(s^2R/P^{\frac{2}{N}} + R^2/P)\cdot\gamma$	$N^2s^2R/P^{\frac{2}{N}} + NR^2/P$	$\mathcal{O}(N\log(P)\cdot\alpha + NsR/P^{\frac{1}{N}}\cdot\beta)$	$\mathcal{O}(N^2(s^2R/P^{\frac{2}{N}} + R^2/P)\cdot\nu)$
PP-approx-ref	$2N^2(s^2R+R^2)\cdot\gamma$	$2N^2(s^2R/P + R^2/P)\cdot\gamma$	$N^2s^2R/P + NR^2/P$	$\mathcal{O}(N^2\log(P)\cdot\alpha + N^2sR/P\cdot\beta)$	$\mathcal{O}(N^2(s^2R/P + R^2/P)\cdot\nu)$

Table 3.1: Comparison of the leading order sequential computational cost, leading order local computational cost, asymptotic communication cost and the leading order auxiliary memory necessary on each processor for the MTTKRP calculation. PP-init-ref, PP-approx-ref denote the PP initialization and approximated kernels implemented in the reference [93].

3.3 PARALLEL ALGORITHMS

The parallel CP-ALS algorithm introduced in Section 3.1.6 can be easily combined with the MSDT algorithm, where only the DT routine in the local-MTTKRP calculations need to be replaced by the MSDT routine. The computational cost will be cheaper, and the horizontal communication cost will be the same for both algorithms.

We detail the parallel PP algorithm in Algorithm 3.4. In the algorithm, we show the parallel version of the PP initialization step and the approximated step (lines 9-18 in Algorithm 3.2). In Algorithm 3.4, the core idea is to perform all the contractions in the

PP initialization steps (line 2), and first-order corrections in the PP approximated steps (line 6) locally, similar to the local-MTTKRP routine introduced in Section 3.1.6. After the calculations of the local first-order MTTKRP corrections, we use the Reduce-Scatter collective routine to update the global $\tilde{\mathbf{M}}^{(n)}$ (line 9). The second-order correction (line 10) only involves Hadamard products and a small matrix multiplication, which we calculate in parallel.

We show the comparison of both the sequential and the parallel computational cost, the horizontal communication cost, the vertical communication cost, and the auxiliary memory needed among DT, MSDT and PP in Table 3.1. The leading order computational cost of MSDT is a factor of $\frac{2(N-1)}{N}$ smaller than the cost of DT. The PP initialization step has the same leading order cost as DT, and the PP approximated step reduces the local computational cost to $\mathcal{O}(N^2(s^2R/P^{\frac{2}{N}} + R^2/P))$.

As to the auxiliary memory usage, both MSDT and PP require more memory compared to DT. Both MSDT and PP can use less memory through combining several upper level contractions. For example, when calculating the PP operator $\mathcal{M}_p^{(1,3)}$ for an order four tensor, we can bypass the first level contraction and save its memory via directly performing a contraction between the input tensor and the Khatri-Rao product output $\mathbf{A}^{(1)} \odot \mathbf{A}^{(3)}$. If the upper $l \leq N-2$ levels of contractions are combined, both PP and MSDT would require $(s^N/P)^{\frac{N-l}{N}}R$ local auxiliary memory. However, the local computational cost of the PP initialization step would increase to $(l+2)(l+1)s^N R/P$, and the local computational cost of MSDT would increase to $\frac{2N}{N-l}s^N R/P$.

DT, MSDT, and PP all have the same asymptotic horizontal communication cost. In addition, our new PP implementation is more efficient in horizontal communication compared to the reference implementation in [93] for both PP initialization and approximated steps. The new implementation does not involve any horizontal communication cost in the PP initialization step, and requires $\Theta(N)$ less for the PP approximated step. The reference implementation reduces each contraction to a matrix multiplication. For the PP initialization step, this approach will either keep the input tensor in place, perform local multiplications and afterwards perform a reduction on the output tensor when R is small, or perform a general 3D parallel matrix multiplication when R is high. For the PP approximated step, this approach will parallelize small-sized mTTVs and result in over-parallelization. Both MSDT and the PP initialization step have higher vertical communication costs compared to DT, on account of the larger intermediates formed in the dimension trees. For high order cases, one can combine the upper l levels of contractions to achieve a trade-off between the computational cost and the vertical communication cost. The vertical communication cost of the PP approximated step is greater than the local computational cost, implying that the PP approximated step is likely to be bottlenecked by vertical communication. Our performance evaluation in Section 3.4.2 confirms this.

Note that tensor transposes are not necessary for the contractions in DT but are necessary in the PP initialization step when the order is greater than 3 (for the PP operator calculation). They are also necessary in MSDT to contract the input tensor with a middle mode factor matrix. For example, when performing the TTM of an order three tensor and a matrix, e.g., $\boldsymbol{\mathcal{T}} \times_2 \mathbf{A}^{(2)T}$, the transpose of the second and third modes of $\boldsymbol{\mathcal{T}}$ needs to be performed first, then the TTM can be cast as a matrix-multiplication and calculated via calling BLAS routines. Performing tensor transposes will incur a larger leading order constant in the vertical communication cost. The transposes in the PP initialization step will affect the PP performance, as we will show in Section 3.4. The cost can be amortized in MSDT if transposes of the input tensor are stored. For both order 3 and order 4 tensors, one copy of the transposed input tensor is necessary. Our implementations use this method to avoid transposes in MSDT.

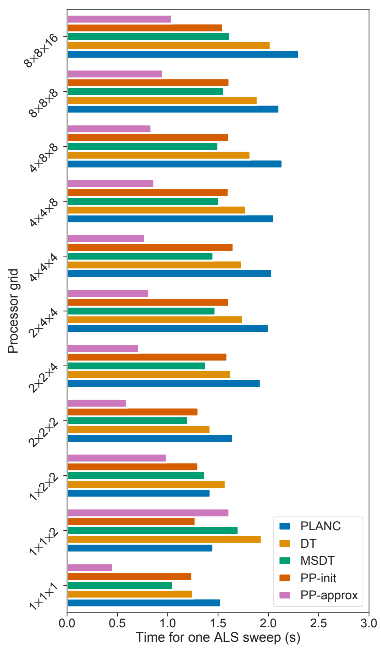
3.4 EXPERIMENTAL RESULTS

Processor grid	$2\times4\times4$ (3D)	$4\times4\times4$ (3D)	$4\times4\times8$ (3D)	$4\times8\times8$ (3D)	$2\times2\times2\times4$ (4D)	$2\times2\times4\times4$ (4D)	$2\times4\times4\times4$ (4D)	$4\times4\times4\times4$ (4D)
PP-init	1.6105	1.6535	1.6045	1.6060	0.7627	0.7713	0.85	0.8715
PP-init-ref	12.9300	12.1920	11.9075	11.3710	19.8695	19.6200	16.018	13.3695
PP-approx	0.4579	0.4509	0.4410	0.4433	0.0541	0.0526	0.0553	0.0533
PP-approx-ref	6.7128	5.7927	5.3655	4.5682	0.3540	0.2916	0.2757	0.2887

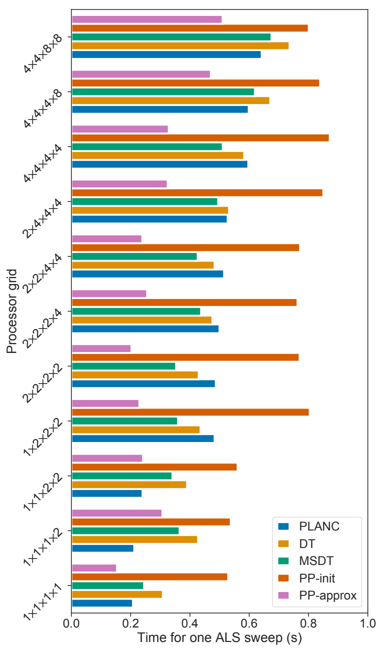
Table 3.2: Comparison of the per-sweep MTTKRP calculation time between our PP initialization step (PP-init) and the PP approximation step (PP-approx) kernels to the ones (PP-init-ref, PP-approx-ref) implemented in the reference [93]. The tensor size and CP ranks under each processor grid configuration is the same as in Fig. 3.3a and Fig. 3.3b.

3.4.1 Implementations, Platforms and Tensors

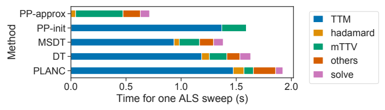
We implement parallel DT, MSDT, and PP algorithms with Cyclops Tensor Framework (v1.5.5) [100], which is a distributed-memory library for matrix/tensor contractions that uses MPI for inter-processor communication and OpenMP for threading. On each processor, we use Intel compilers and the MKL library for threaded BLAS routines, including batched BLAS routines, which are efficient for mTTV arising in MTTKRP, and employ the HPTT library [107] for high-performance tensor transpositions. We also use a wrapper provided by Cyclops for ScaLAPACK [72] routines to solve symmetric positive definite linear systems of equations. All storage and computations assume the tensors are dense. Our PP algorithm calls



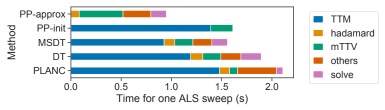
(a) $N = 3, s_{\text{local}} = 400, R = 400$



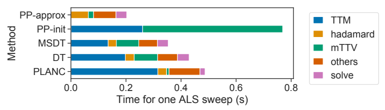
(b) $N = 4, s_{\text{local}} = 75, R = 200$



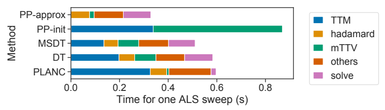
(c) $N = 3$, grid $= 2 \times 4 \times 4$



(d) $N = 3$, grid $= 8 \times 8 \times 8$



(e) $N = 4$, grid $= 2 \times 2 \times 2 \times 2$



(f) $N = 4$, grid $= 4 \times 4 \times 4 \times 4$

Figure 3.3: Benchmark results for order 3 and 4 tensors. The reported time is the mean time across 5 sweeps. **(a)(b)** Weak scaling of synthetic tensors. For each plot, the rank R is fixed, and the local tensor size on each processor is fixed with dimension size s_{local}. **(c)(d)(e)(f)** Time breakdown under specific processor grid configurations. Each per-sweep time is broken into 5 categories: TTM, mTTV, solve (linear systems solves), hadamard (the Hadamard products, which appear in (3.3) and (3.9)), and others. The tensor sizes and CP ranks of (c)(d) are the same as in (a), and of (e)(f) are the same as in (b).

the MSDT subroutine for the regular ALS sweeps (line 19 in Algorithm 3.2) to improve the performance. All of our code is available at `https://github.com/LinjianMa/parallel-pp`.

The experimental results are collected on the Stampede2 supercomputer located at the University of Texas at Austin. Experiments are performed on the Knight's Landing (KNL) nodes, each of which consists of 68 cores, 68 threads, 96 GB of DDR RAM, and 16 GB of MCDRAM. These nodes are connected via a 100 Gb/sec fat-tree Omni-Path interconnect.

We compare the performance of different algorithms on both synthetic tensors and application datasets. The application datasets we considered include publicly available tensor datasets as well as tensors of interest for quantum chemistry calculations. These tensors enable us to demonstrate the effectiveness of our algorithms on practical problems. We use the following four tensors to test the performance. For all the experiments, we use 16 MPI processes on each KNL node, and each process uses 4 threads.

1. **Tensors with given collinearity** [80]. We create tensors based on randomly-generated factor matrices $\mathbf{A}^{(n)}$, where $n \in \{1, \dots, N\}$. Each factor matrix $\mathbf{A}^{(n)} \in \mathbb{R}^{s \times R}$ is randomly generated so that the columns have collinearity defined based on a scalar C (selected randomly from a given interval $[a, b)$),

$$\frac{\langle \mathbf{a}_i^{(n)}, \mathbf{a}_j^{(n)} \rangle}{||\mathbf{a}_i^{(n)}||_2 ||\mathbf{a}_j^{(n)}||_2} = C, \quad \forall i, j \in \{1, \dots, R\}, i \neq j. \tag{3.11}$$

The generated tensor has dimension size s in each mode, and its CP rank is bounded by R. Higher collinearity corresponds to greater overlap between columns within each factor matrix, which makes the convergence of the rank-R CP-ALS procedure slower [109]. We experiment on tensors with dimensions $1600 \times 1600 \times 1600$. We run the experiments on 64 processors, and the processor grid has dimensions $4 \times 4 \times 4$.

2. **Quantum chemistry tensor.** We also test on the density fitting intermediate tensor arising in quantum chemistry, which is the Cholesky factor of the two-electron integral tensor [24], [25]. For the order 4 two-electron integral tensor $\mathcal{T}$, its Cholesky factor is an order 3 tensor $\mathcal{D}$, with their element-wise relation shown as $\mathcal{T}(a, b, c, d) = \sum_{e=1}^{E} \mathcal{D}(a, b, e)\mathcal{D}(c, d, e)$, where E is the third mode dimension size of $\mathcal{D}$. CP decomposition can be performed on $\mathcal{D}$ to compress the intermediate and can accelerate the post Hartree-Fork calculations [26]. We generate the density fitting tensor via the PySCF library [110], which represents the compressed restricted Hartree-Fock wave function of an 40 water molecule chain system with a STO-3G basis set. The generated tensor has size $4520 \times 280 \times 280$. We run the experiments on 32 processors, and the processor grid has dimensions $8 \times 2 \times 2$.

3. **COIL dataset**. COIL-100 is an image-recognition dataset that contains images of objects in different poses [111]. It has been used previously as a tensor decomposition benchmark [77], [80], [93]. Transferring the data into tensor format results in a tensor of size $128 \times 128 \times 3 \times 7200$. We run the experiments on 16 processors, and the processor grid has dimensions $2 \times 2 \times 1 \times 4$.

4. **Time-Lapse hyperspectral radiance images**. We consider the 3D hyperspectral imaging dataset called “Souto wood pile” [112]. The dataset is usually used to benchmark nonnegative tensor decomposition [62], [64]. The hyperspectral data consists of a tensor with dimensions $1024 \times 1344 \times 33 \times 9$. We run the experiments on 16 processors, and the processor grid has dimensions $4 \times 4 \times 1 \times 1$.

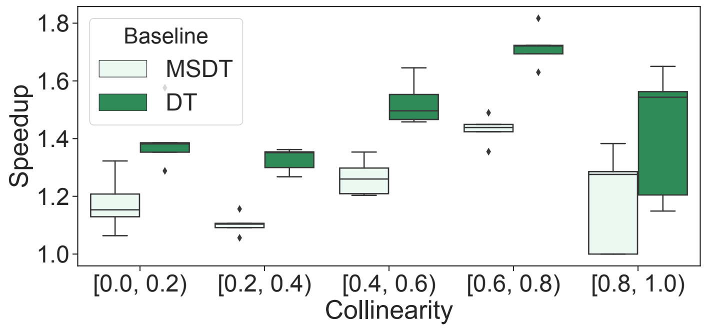


Figure 3.4: Relation between PP speed-ups and input collinearity for order 3 tensors. The dimension size $s = 1600$, the rank $R = 400$, and the experiments run on the $4 \times 4 \times 4$ processor grid. The PP tolerance is set as 0.2. Each box is based on 5 experiments with different random seeds and shows the 25th-75th quartiles. The median is indicated by a horizontal line inside the box and outliers are displayed as dots.

Configuration	Num-DT	Num-PP-init	Num-PP-approx
Col$\in [0.0, 0.2)$	21.2	1.4	14.4
Col$\in [0.2, 0.4)$	50.8	11.6	39.6
Col$\in [0.4, 0.6)$	65.6	29.0	169.6
Col$\in [0.6, 0.8)$	34.8	17.4	180.2
Col$\in [0.8, 1.0)$	10.2	3.4	22.8

Table 3.3: Detailed statistics of the results shown in Fig. 3.4. From left to right: the tensor configuration (Col stands for collinearity), number of standard dimension tree sweeps, number of PP initialization sweeps and number of PP approximated sweeps. All the data are the average statistics from 5 experiments.

3.4.2 Benchmarks

We perform a parallel scaling analysis to compare the per-ALS sweep simulation time for DT, MSDT, the PP initialization step and the PP approximated step, and present the results in Fig. 3.3. We also show the simulation time using PLANC [73] for reference, which contains state-of-the-art parallel DT implementations. The benchmarks are performed on both order 3 and order 4 tensors.

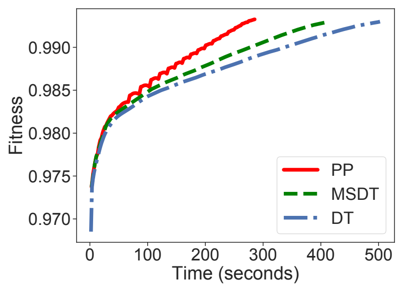
We show the order 3 weak scaling results in Fig. 3.3a. As can be seen, the performance of our DT implementation is comparable, and slightly better than PLANC on most of the processor grid configurations. MSDT performs consistently better than DT. For the largest processor grid configuration, MSDT is 1.25X better than DT. In addition, the time spent on each PP initialization step is consistently less than each DT sweep, and the PP approximated step achieves a speed-up of 1.94X compared to DT for the largest processor grid, showing good scalability for our parallel PP algorithm. We show the detailed time breakdown under the grid configurations $2 \times 4 \times 4$, $8 \times 8 \times 8$ in Fig. 3.3c and Fig. 3.3d. TTM is the major bottleneck for all the kernels except the PP approximated step, which is bottlenecked by the mTTV kernel. Note that as discussed in Section 3.3, the mTTV kernel is vertical communication (memory bandwidth) bound, which inhibits the realization of the large speed-up in flops.

We show the order 4 weak scaling results in Fig. 3.3b. Since the PP initialization step for the order 4 tensors involves several tensor transposes, we use 8 MPI processes per KNL node and 8 threads per process for the benchmark, so that the transposes can be accelerated with a relatively large number of threads for each process. The performance of our DT implementation is comparable to PLANC for most of the processor grid configurations. When the number of processors used is large, our DT implementation is slightly slower than PLANC. This can be explained by the slow global linear system solves shown in Fig. 3.3f. Since the CP rank is relatively small, using too many processors results in over-parallelization. MSDT performs consistently better than DT, and achieves a speed-up of 1.10X under the largest processor grid configuration. The PP initialization step is slower than the DT sweep, which is due to several tensor transposes in the mTTV kernels. It also accounts for the slow mTTV kernels shown in Fig. 3.3e and Fig. 3.3f. The PP approximated step achieves a speed-up of 1.44X compared to DT under the largest processor grid. Overall, the speed-up of PP is less on order 4 compared to order 3 tensors.

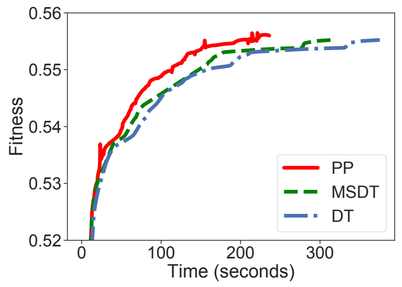
We also show the per-sweep MTTKRP calculation time comparison between our parallel PP implementation and that in [93] in Table 3.2. As can be seen, the current implementation is more efficient for both kernels under all the processor grid configurations, consistent with the analysis in Section 3.3. The local dimension tree calculations greatly decrease the horizontal communication cost.

3.4.3 Performance Comparison

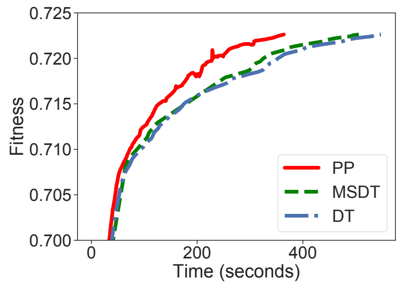
We test algorithms on both the synthetic tensors (Tensor 1) and real datasets (Tensor 2,3,4). We use the metrics relative residual, r (defined in (3.4)), and fitness, $f = 1 - r$, to evaluate the convergence progress of the decomposition. Fig. 3.4 shows the speed-up distribution with different exact factor matrices collinearity. We stop the algorithm when the stopping tolerance (defined as the fitness difference between two neighboring sweeps) reaches 10^{-5}, or the maximum number of sweeps (300) is reached. The speedup is calculated based on the ratio of the ALS time to the PP time to reach the same accuracy. For these synthetic tensors, PP achieves up to 1.8X speed-up compared to DT. In addition, PP tends to have higher speed-ups when the collinearity is within $[0.4, 0.8)$. This is because tensors within



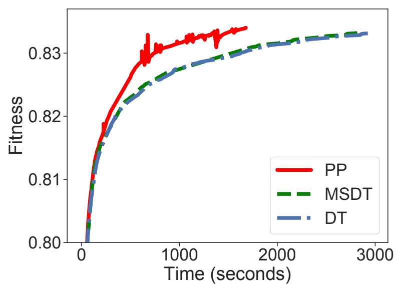
(a) Col$\in [0.6, 0.8)$, $R = 400$



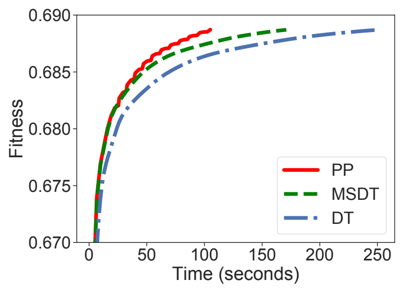
(b) Chemistry, $R = 300$



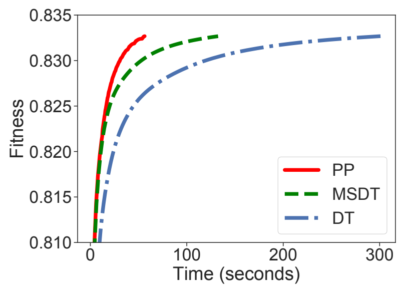
(c) Chemistry, $R = 600$



(d) Chemistry, $R = 1000$



(e) Coil dataset, $R = 20$



(f) Time-lapse dataset, $R = 50$

Figure 3.5: Comparison of PP, MSDT, DT on different tensors.

Tensor	N-DT	N-PP-init	N-PP-approx	T-ALS	T-PP-init	T-PP-approx
Fig. 3.5b	96	39	477	0.5064	0.2335	0.3443
Fig. 3.5c	76	29	468	0.8177	0.3801	0.5839
Fig. 3.5d	216	63	1129	1.4307	0.795	1.1095
Fig. 3.5e	34	11	115	2.3907	4.2253	0.1785
Fig. 3.5f	30	10	155	0.6703	0.3631	0.1554

Table 3.4: Detailed statistics of the results shown in Fig. 3.5. From left to right: the tensor type, number of standard dimension tree sweeps, number of PP initialization sweeps, number of PP approximated sweeps, the average time of each dimension tree sweep, the average time of each PP initialization sweep, and the average time of each PP approximated sweep.

above collinearity range will converge in more sweeps, and more PP approximated sweeps are activated, as can be seen in Table 3.3. When the collinearity is in the range of $[0.0, 0.4)$ and $[0.8, 1.0)$, the experiments stop after a small number of sweeps, which results in less benefit of PP. In addition, we show the fitness-time relation for one experiment with the collinearity$\in [0.6, 0.8)$ in Fig. 3.5a. The fitness increases monotonically, indicating that the approximation error in PP is not problematic. The results show that substantial performance gains on large synthetic tensors can be achieved for PP under parallel execution. This conclusion is a supplement to the results in [93], which show that PP can speed up CP decomposition on small (each tensor has size $400 \times 400 \times 400$) synthetic tensors.

We also compare the performance on application tensors. The PP tolerance is set as 0.1 for these tensors. Fig. 3.5b, Fig. 3.5c, and Fig. 3.5d show the results on the quantum chemistry tensor with different CP ranks. The detailed statistics are shown in Table 3.4. As is shown in the figures, for all the variants of the CP ranks, PP performs better than DT, achieving 1.52-1.78X speed-ups. Fig. 3.5e and Fig. 3.5f show the results on the image datasets. The detailed statistics are shown in Table 3.4. As is shown in the figures, PP achieves 2.4X speed-up on the Coil dataset and 5.4X speed-up on the Time-lapse dataset.

3.5 CONCLUSION

We propose two parallel algorithms, multi-sweep dimension tree (MSDT) and communication efficient pairwise perturbation (PP), to accelerate MTTKRP calculations in CP-ALS for dense tensors. These algorithms are both computationally more efficient than the standard dimension tree algorithm, and are efficient in horizontal communication. MSDT reduces the leading order computational cost by a factor of $2(N-1)/N$ relative to the standard dimension tree algorithm. Our parallel PP algorithm reduces the communication cost to a greater extent compared to the implementations in reference [93]. Our experimental results show that substantial performance improvements are achieved for both algorithms relative to prior approaches. However, our theoretical analysis and results reveal that speed-ups obtained via MSDT and PP are inhibited by the lower arithmetic intensity of these two more work-efficient algorithms.

Chapter 4: A SYSTEM FOR AUTOMATIC DIFFERENTIATION OF TENSOR NETWORKS

In this Chapter, we introduce AutoHOOT, an automatic differentiation computer system that targets tensor network applications. In particular, AutoHOOT incorporates tensor algebra-specific transformations, and includes algorithms to automatically generate dimension tree implementations for alternating minimization.

Derivatives, mostly in the form of gradients, are ubiquitous in the optimization algorithms for tensor-related problems. For neural networks, they are used to calculate the gradients of the loss function w.r.t. the model parameters. For tensor decomposition and tensor networks, first-order and higher-order derivatives are necessary to construct the operators used in the alternating optimization. Gradients of computational chemistry methods are used for optimization of the electronic geometry to identify stable states and state transitions [128]. Automatic differentiation (AD) frameworks, including popular Python tools such as PyTorch [129], JAX [130], and TensorFlow [131], can generate derivatives in all of these contexts. However, in tensor decomposition, tensor networks, and quantum chemistry, gradient calculations are most often done via manually written codes, as careful numerical and performance considerations are required in these more complex settings.

AD transforms a software or mathematical expression of a function into code for computation of its derivatives with respect to the desired parameters. Although mathematically correct, the output programs for the derivatives may be sub-optimal in computational cost, use of efficient kernels such as the BLAS, memory footprint, and numerical stability. Components of different frameworks address these problems jointly or independently. For example, transformations of the computational graph and operator fusion are used to improve computational efficiency and parallelizability [129], [131], [132]. Gradient checkpointing and garbage collection are used to address memory bottlenecks [129], [131]. For large scale tensor computations, computational and memory demands leave little leeway for error in these aspects.

Common commercial AD frameworks such as PyTorch [129], JAX [130], and TensorFlow [131] are focused on first-order numerical optimization methods on deep learning models. In the context of tensor decompositions, tensor network optimization, and differentiation of tensor methods, three major additional challenges arise.

1. These domains predominantly employ alternating second-order optimization methods, as they provide monotonic convergence and rapid progress at almost the same per-iteration cost as first-order methods. These methods employ implicit representations of the Jacobian and Hessian to solve linear systems. Existing AD frameworks have

limited logical constructs for second-order derivative information, and consequently generate code that can be sub-optimal in cost by orders of magnitude.

2. Most tensor operations involved in the deep learning applications are related to small tensors, while in tensor network and tensor decomposition applications, there are many tensor contractions over high order (multidimensional) tensors with a large number of elements. Therefore, tensor network applications require better optimization algorithms to select optimized contraction order and eliminate redundant calculations.

3. Deep learning computational graphs usually have large depth with many nonlinear operations, making the freedom to optimize tensor operations limited. On the other hand, in tensor decomposition and tensor network applications, the computational graphs are usually wide and have small depth, so there is more freedom to optimize the computation.

Although many frameworks, such as Tensorly [133], TensorNetwork [134] and Quimb [135], provide interfaces to optimize the tensor decomposition / networks algorithms with AD frameworks such as TensorFlow and PyTorch, the optimization algorithms are the general first-order methods and its variants. These frameworks explicitly implement popular second-order methods for these problems, such as Alternating Least Squares (ALS) for tensor decompositions and Density Matrix Renormalization Group (DMRG) for 1D tensor networks, rather than using AD. The ability to generate efficient expressions of these methods automatically via AD, would accelerate the development of new variants and their deployment on shared-memory, GPU, and distributed-memory architectures.

We propose a new AD framework for tensor computations, Automatic High-Order Optimization for Tensors (AutoHOOT). The library is publicly available at `https://github.com/LinjianMa/AutoHOOT`. AutoHOOT encapsulates the following novel ideas and capabilities:

- a new AD module that generates more efficient representations for higher-order derivative constructs such as Jacobians and Hessians, which are needed for tensor computation applications,

- a new computational graph optimization module that extends beyond the traditional optimization techniques for compilers with tensor-algebra specific transformations, such as distributivity of matrix inversion over the Kronecker product,

- portability via high-level support for different tensor contraction backends: NumPy for multi-core CPU, TensorFlow for GPUs, and Cyclops [100] for distributed memory systems,

- substantial improvements in sequential and parallel performance for tensor network and tensor decomposition optimizations over other AD libraries and competitive or improved performance w.r.t. manually-optimized implementations.

4.1 BACKGROUND

4.1.1 Notation and Definitions

For vectors, bold lowercase Roman letters are used, e.g., $\mathbf{x}$. For matrices, bold uppercase Roman letters are used, e.g., $\mathbf{X}$. For tensors, bold calligraphic uppercase Roman letters are used, e.g., $\boldsymbol{\mathcal{X}}$. An order N tensor corresponds to an N-dimensional array with dimensions $s_1 \times \cdots \times s_N$. Elements of vectors, matrices, and tensors are denoted in parentheses, e.g., $\mathbf{x}(i)$ denotes the ith entry of a vector $\mathbf{x}$, $\mathbf{X}(i,j)$ denotes the (i,j)th element of a matrix $\mathbf{X}$, and $\boldsymbol{\mathcal{X}}(i,j,k,l)$ denotes the (i,j,k,l)th element of an order 4 tensor $\boldsymbol{\mathcal{X}}$. Subscripts are used to label different vectors, matrices, tensors and functions (e.g. $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$, f_1 and f_2).

Matricization is the process of unfolding a tensor into a matrix. Given a tensor $\boldsymbol{\mathcal{X}}$ the mode-n matricized version is denoted by $\mathbf{X}_{(n)} \in \mathbb{R}^{s_n \times K}$ where $K = \prod_{m=1,m\neq n}^{N} s_m$. We generalize this matricization definition, so that $\mathbf{X}_{(i:j)}$ means that the dimensions from the ith index to the jth index are unfolded to the column dimension of the matrix, and all the other dimensions are unfolded to the row dimension of the matrix.

For a scalar output function $y = f(\mathbf{a}_1, \ldots, \mathbf{a}_N)$, We use the $\mathbf{g}^{[f]}_{[\mathbf{a}_i]}$ and $\mathbf{H}^{[f]}_{[\mathbf{a}_i]}$ to denote the gradient vector and Hessian matrix of f w.r.t the input vectors $\mathbf{a}_i$. When the inputs are tensors, the gradient and the Hessian will also be a tensor and denote $\boldsymbol{\mathcal{G}}^{[f]}_{[\boldsymbol{\mathcal{A}}_i]}$ and $\boldsymbol{\mathcal{H}}^{[f]}_{[\boldsymbol{\mathcal{A}}_i]}$. For a function with non-scalar output $\mathbf{y} = f(\mathbf{a}_1, \ldots, \mathbf{a}_N)$, we use $\mathbf{J}^{[f]}_{[\mathbf{a}_i]}$ to denote the Jacobian matrix of the function f w.r.t one of the input vectors $\mathbf{a}_i$. The shape of the Jacobian matrix will be $\mathbb{R}^{|\mathbf{y}| \times |\mathbf{a}_i|}$. If $\boldsymbol{\mathcal{Y}}$ is an output tensor with size $\mathbb{R}^{s_1 \times \ldots \times s_M}$, and $\boldsymbol{\mathcal{A}}_i$ is an input tensor with size $\mathbb{R}^{r_1 \times \ldots \times r_K}$, then the Jacobian will be a tensor denoted as $\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{A}}_i]}$ with dimensions $\mathbb{R}^{s_1 \times \ldots \times s_M \times r_1 \times \ldots \times r_K}$.

We also define generalized Vector Jacobian Product (VJP), Jacobian Vector Product (JVP) and Hessian Vector Product (HVP). When both Jacobian and Hessian are matrices, these are matrix-vector multiplication operations. When Jacobian and Hessian are both tensors defined above, these are tensor contractions, whose results are the same as unfolding the tensors into matrices and performing the matrix-vector product.

4.1.2 Numerical Optimization Algorithms for Tensor Computations

We consider two tensor numerical problems: the nonlinear least squares fitting and the eigenvalue problem. For both problems, we denote $\boldsymbol{\mathcal{X}}$ as the input tensor which can be an explicit tensor or implicit tensor network (e.g., Matrix Product Operator [136]), f as a tensor network function and $\boldsymbol{\mathcal{A}}_1, \dots, \boldsymbol{\mathcal{A}}_N$ as the optimization variables. Then the objective for the nonlinear least squares problem is defined as

$$\min_{\boldsymbol{\mathcal{A}}_1,\dots,\boldsymbol{\mathcal{A}}_N} \frac{1}{2}\|\boldsymbol{\mathcal{X}} - f(\boldsymbol{\mathcal{A}}_1, \dots, \boldsymbol{\mathcal{A}}_N)\|^2, \tag{4.1}$$

which finds a generalized low rank approximation of the input tensor $\boldsymbol{\mathcal{X}}$. The objective for the eigenvalue problem is defined as

$$\min_{\boldsymbol{\mathcal{A}}_1,\dots,\boldsymbol{\mathcal{A}}_N} \frac{\mathbf{v}_{(1:N)}^T \mathbf{X}_{(1:N)} \mathbf{v}_{(1:N)}}{\|\boldsymbol{\mathcal{V}}\|_F^2}, \tag{4.2}$$

where $\boldsymbol{\mathcal{V}} = f(\boldsymbol{\mathcal{A}}_1, \dots, \boldsymbol{\mathcal{A}}_N)$ and the output of f serves as a generalized low rank approximation of the eigenvector of a Hermitian matrix that is a matricization of $\boldsymbol{\mathcal{X}}$.

Three categories of algorithms are generally used to optimize the problems: second-order methods, including Newton's method and its variants, alternating minimization, which updates each input / site at one time, and first-order methods such as gradient descent and its variants.

Newton's method and its variants. Newton's method and its variants, such as Gauss-Newton (GN) method, are popular methods to solve non-linear least squares problems for a quadratic objective function defined in (4.1). Let $\mathbf{a}$ denote the concatenation of all the vectorized sites $\text{vec}(\boldsymbol{\mathcal{A}}_i)$ and $\hat{f}(\mathbf{a}) = \text{vec}(f(\boldsymbol{\mathcal{A}}_1, \dots, \boldsymbol{\mathcal{A}}_N))$, so that $r(\mathbf{a}) := \text{vec}(\boldsymbol{\mathcal{X}}) - \hat{f}(\mathbf{a})$ denotes the vectorized residual. Further, let $r_i(\mathbf{a})$ denote the ith element of the output of function r. The gradient and the Hessian matrix of

$$\phi(\boldsymbol{\mathcal{A}}_1, \dots, \boldsymbol{\mathcal{A}}_N) := \frac{1}{2}\|\boldsymbol{\mathcal{X}} - f(\boldsymbol{\mathcal{A}}_1, \dots, \boldsymbol{\mathcal{A}}_N)\|^2, \tag{4.3}$$

can be expressed as

$$\nabla\phi(\mathbf{a}) = \mathbf{J}_{[\mathbf{a}]}^{[r]T} r(\mathbf{a}), \quad \text{and} \quad \mathbf{H}_{[\mathbf{a}]}^{[\phi]} = \mathbf{J}_{[\mathbf{a}]}^{[r]T}\mathbf{J}_{[\mathbf{a}]}^{[r]} + \sum_i r_i(\mathbf{a})\mathbf{H}_{[\mathbf{a}]}^{[r_i]}. \tag{4.4}$$

The Newton iteration performs the update based on

$$\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} - (\mathbf{H}_{[\mathbf{a}^{(k)}]}^{[\phi]})^{-1}\mathbf{J}_{[\mathbf{a}^{(k)}]}^{[r]T} r(\mathbf{a}^{(k)}), \tag{4.5}$$

while the Gauss-Newton method leverages the fact that $\mathbf{H}_{[\mathbf{a}]}^{[r_i]}$ is negligible as its norm is small when the residual is small, therefore the update can be performed as

$$\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} - (\mathbf{J}_{[\mathbf{a}^{(k)}]}^{[r]T} \mathbf{J}_{[\mathbf{a}^{(k)}]}^{[r]})^{-1} \mathbf{J}_{[\mathbf{a}^{(k)}]}^{[r]T} r(\mathbf{a}^{(k)}), \tag{4.6}$$

where $\mathbf{a}^{(k)}$ represents the $\mathbf{a}$ at kth iteration. The Gauss-Newton updates can be regarded as normal equations for the linear least squares problem. Both Newton and Gauss-Newton methods can be solved via the conjugate gradient method with matrix-vector products performed with an implicit form of the Jacobian / Hessian to avoid costly matrix inversion [137], [138].

Alternating minimization. For tensor numerical optimization, in many cases both the input and output dimensions are large, and it's computationally expensive to form the explicit Hessian / Jacobian matrix w.r.t. all the variables and perform the second-order method directly. On the other hand, when optimizing a subset of variables, forming the Hessian or Jacobian with respect to those variables is affordable and effective. Most often, alternating minimization procedures update one tensor operand at a time. For (4.1), such subproblem can be formulated as

$$\min_{\boldsymbol{\mathcal{A}}_i} \frac{1}{2} \|\boldsymbol{\mathcal{X}} - f(\boldsymbol{\mathcal{A}}_1, \dots, \boldsymbol{\mathcal{A}}_N)\|^2. \tag{4.7}$$

Each $\boldsymbol{\mathcal{A}}_i$ for $i \in \{1, \dots, N\}$ is updated once via its subproblem during an optimization sweep. For tensor decompositions and tensor networks, each subproblem is often quadratic, allowing for the minima to be found directly, often at a similar cost to updating $\boldsymbol{\mathcal{A}}_i$ with a first-order method. Alternating minimization also generally provides monotonic convergence.

In each sweep, many terms necessary to form the subproblems have many equivalent intermediates, and choosing the proper contraction paths to form and also amortize them can greatly save the cost. This scheme, called the *dimension tree* algorithm, is critical to the algorithm performance, and has been used in both tensor decompositions [59], [93], [99] and DMRG to save the cost.

First-order methods. The efficacy of the first-order methods on tensor computations is dependent on the applications. The first-order methods are shown to be advantageous on achieving high fitting accuracies on some tensor decomposition problems [139], while they also perform worse than alternating minimization in achieving high accuracy for large scale tensor completion problems [140]. The per-iteration cost of first-order methods is often comparable to that of both second-order methods and the alternating minimization method, due to the structure of tensor networks f in (4.1) and (4.2).

Traditional AD frameworks can generate efficient kernels for first-order methods, while their performance on the kernels in higher-order methods is suboptimal. In this paper, we focus on the performance optimization over both second-order method and alternating minimization methods, to accelerate future development of efficient high-order methods for various applications. However, we believe our graph optimization techniques also have the potential to produce efficient formulations for first-order methods, where the objective involves contractions of high-order tensors, which arise in quantum chemistry methods [141].

4.1.3 Previous Work

Optimization for tensor computations requires three essential building blocks, automatic differentiation, optimization of the generated set of tensor operations, and a computational backend for individual tensor operations. Existing software for tensor computations, including Tensorly [133], TensorNetwork [134] and Quimb [135] permit the use of multiple backends for individual tensor operations, and provide some constructs to make use of AD. However, when using AD, these libraries employ general AD backends such as JAX or TensorFlow in a black-box fashion.

Automatic differentiation is generally provided via one of two ways, operator overloading [129], [130], [142]–[144] or source code transformation (SCT) [131], [145], [146]. Operator overloading requires the user to write functions in terms of the provided library constructs and constructs the derivatives at run-time, while SCT uses precompilation to generate code for derivative computation. Operator overloading provides a similar mental programming model as normal computer programs [142], yielding code that is easier to interpret and debug than SCT. On the flip side, SCT has more potential to optimize the computational graph with global graph information. Consequently, SCT is generally the method of choice for AD libraries that aim to achieve high performance (e.g., [131]).

Our work on graph optimization builds on substantial efforts for optimization of computational graphs of tensor operations. Tensor contraction can be optimized via parallelization [100], [147]–[149], efficient transposition [107], blocking [150]–[153], exploiting symmetry [100], [141], [154], and sparsity [149], [155]–[158]. For complicated tensor graphs, specialized compilers like XLA [159] and TVM [160] rewrite the computational graph to optimize program execution and memory allocation on dedicated hardware. For machine independent optimization, Grappler in TensorFlow [131] and TASO [132] use rule based symbolic substitution to simplify the execution flow. Classical compiler optimization also includes relevant techniques such as common subexpression elimination [161] are widely used as well [131], [162]. Previous work, such as Opt_einsum [163] has yielded approaches for

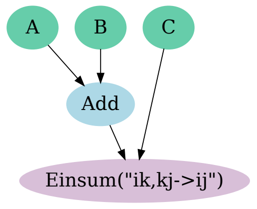


Figure 4.1: An example of a computational graph. We use green nodes to denote input variables, purple nodes to denote output nodes, and blue nodes to denote intermediate or constant nodes.

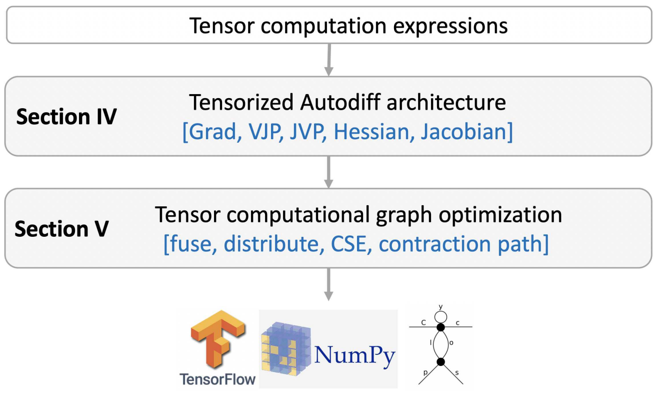

Tensor computation expressions
Section IV
Tensorized Autodiff architecture
[Grad, VJP, JVP, Hessian, Jacobian]
Section V
Tensor computational graph optimization
[fuse, distribute, CSE, contraction path]
TensorFlow
NumPy

Figure 4.2: System overview of AutoHOOT. The arrows show the computation flow.

automatically determining efficient contraction orderings and selecting the best intermediates [141], [164]–[166]. The approaches generally rely on heuristic or exhaustive search to select a contraction path, as finding the optimal contraction order is NP-hard [167].

4.2 OVERALL ARCHITECTURE

The computations in AutoHOOT are described by *computational graphs*, which are directed graphs revealing the data dependency between different operations. Each *node* can be a source, intermediate or sink. Source / Sink nodes are inputs / outputs of the graph. Sink and intermediate nodes can be any mathematical computation, while input nodes are fed by the user or constants. An *edge* connecting two nodes represents the data dependency between them. An example of a computational graph is shown in Fig. 4.1, where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are source nodes, the Einsum node is the sink, and the graph computes $(\mathbf{A} + \mathbf{B})\mathbf{C}$. We typically refer a node with its type, e.g., an Einsum node, which represents the tensor computations based on the Einstein summation convention. An *Einsum graph* is defined as a graph of nodes where all the nodes except the sources are Einsum nodes. An *Einsum tree* is defined as a tree of nodes where all the nodes except the sources are Einsum nodes.

AutoHOOT has two major components: an automatic differentiation architecture for tensor computations and a tensor computational graph optimizer. Fig. 4.2 shows the system overview. For an input computation expression, the AD module will generate its tensorized differentiation expressions. Both the input expressions and the differentiation expressions will be optimized through the graph optimization module. With the optimized expressions, users have the choice to directly run the optimized expressions using the framework backends, including NumPy, TensorFlow and Cyclops, or to generate the Python source code through the source generation module.

In Fig. 4.3 we show an example of performing the CP decomposition based on alternating least squares using the framework. Rather than constructing each subproblem and building the dimension tree-based algorithm manually, we only need to construct the updates of Newton's method for each subproblem, and the *optimize* function will reorganize the computational graph to minimize execution time automatically.

In the AD module, we implement the reverse mode AD for first-order derivatives (Jacobian, VJP and JVP), as well as for higher-order derivatives, including Hessian and HVP. Both Jacobian and Hessian are formulated with a new algorithm, such that their calculations are not dependent on the JVP and HVP routines, which is more amenable to parallel execution as well as graph optimizations. We describe this approach in detail in Section 4.3.

The graph optimizer provides optimizations for tensor computational graphs. We adopt many machine independent optimization algorithms for common tensor computational graphs, such as selection of optimal contraction path and common sub-expression elimination. For second-order methods, the graph optimizer rewrites the structured inverse, such as the inverse of a Kronecker product, so that the inverses are operated on smaller tensors. For alternating methods, we developed a path selection algorithm with constraints to construct the dimension trees. We describe this algorithm in detail in Section 4.4.

4.3 COMPUTATIONAL GRAPHS FOR HIGH-ORDER DERIVATIVES

We implement the reverse-mode AD based on the source code transformation (SCT) method, explicitly transforming the primal computation expression prior to execution to the adjoint expression. It allows us to flexibly perform the computational graph optimization after the adjoint expression production.

Our AD module supports the operations which calculate the Jacobian / Hessian expressions implicitly (VJP, JVP and HVP), and also explicit Jacobian and Hessian calculations. The implicit calculations are widely used in many other frameworks, because it is computationally cheaper. For example, for a Hessian matrix with size $n \times n$, explicitly forming the matrix

```
# construct input expressions
A, B, C, input_tensor, loss = cpd_graph(size, rank)

def update_site(site):
    hes = ad.hessian(loss, [site])
    grad, = ad.gradients(loss, [site])
    new_site = ad.tensordot(
        ad.tensorinv(hes[0][0]), grad)
    # return the optimized computational graph
    return optimize(new_site)
new_A = update_site(A)
new_B = update_site(B)
new_C = update_site(C)

# This executor is shared among all updates.
executor = ad.Executor([loss, new_A, new_B, new_C])
# ALS iterations
for i in range(num_iter):
    A_val = executor.run(feed_dict={
        input_tensor: input_tensor_val,
        A: A_val, B: B_val, C: C_val
    }, out=[new_A])
    B_val = executor.run(feed_dict={
        input_tensor: input_tensor_val,
        A: A_val, B: B_val, C: C_val
    }, out=[new_B])
    C_val = executor.run(feed_dict={
        input_tensor: input_tensor_val,
        A: A_val, B: B_val, C: C_val
    }, out=[new_C])
    loss_val = executor.run(feed_dict={
        input_tensor: input_tensor_val,
        A: A_val, B: B_val, C: C_val
    }, out=[loss])
```

Figure 4.3: An example of performing CP decomposition based on alternating least squares using AutoHOOT.

costs $O(n^2)$, while the HVP calculation will only cost $O(n)$ leveraging the back-propagation gradient functions. For the explicit Jacobian and Hessian calculations, we introduce a new back-propagation algorithm that can produce a computational graph is more amenable to parallelization and downstream optimizations. The algorithm is detailed in Section 4.3.2.

4.3.1 VJP, JVP, and HVP

Our implementation of VJP is similar to many other frameworks [129]–[131], and is based on the reverse-mode AD. For functions involving matrix / vector operations whose inputs and outputs are both vectors,

$$\mathbf{x}_{i+1} = f_i(\mathbf{x}_i), i \in [1, \ldots, N], \tag{4.8}$$

consider a computational graph consisting of a chain of these functions,

$$\mathbf{y} = f(\mathbf{x}_1) = f_N \cdots f_1(\mathbf{x}_1), \tag{4.9}$$

the VJP adjoint of $\mathbf{x}_i$, $\mathbf{v}^T \mathbf{J}^{[f]}_{[\mathbf{x}_i]}$, is calculated based on the VJP adjoint of $\mathbf{x}_{i+1}$,

$$\text{VJP}(\mathbf{v}, f, \mathbf{x}_i) = \mathbf{v}^T \mathbf{J}^{[f]}_{[\mathbf{x}_i]} = (\mathbf{v}^T \mathbf{J}^{[f]}_{[\mathbf{x}_{i+1}]}) \mathbf{J}^{[f_i]}_{[\mathbf{x}_i]} = \text{VJP}(\mathbf{v}, f, \mathbf{x}_{i+1}) \mathbf{J}^{[f_i]}_{[\mathbf{x}_i]}. \tag{4.10}$$

Therefore, the VJP of all the inputs / intermediates $\mathbf{x}_i$, $i \in [1, \ldots, N]$ will be calculated with one backward propagation. It is also computationally efficient, because only matrix-vector product is necessary for each calculation.

Note that for the cases where sub function inputs and outputs contain matrices or tensors, VJP with reverse-mode AD is still valid and efficient, since we can think of each matrix or tensor as a reshaped vector. For the case where the output is a scalar, the gradient expression is implemented based on the VJP, if we fix the vector as a unit length vector with element being one.

Our JVP implementation is based on the VJP function[3]. Although it's more computationally efficient to implement JVP based on forward mode AD [168], we choose to implement it based on our reverse mode AD module, and optimize the computational graph afterwards to achieve computationally efficient expressions. The JVP implementation is based on calling the VJP function twice. First, we construct a function g, whose expression is as follows,

$$g(\mathbf{u}) = \text{VJP}(\mathbf{u}, f, \mathbf{x})^T = (\mathbf{u}^T \mathbf{J}^{[f]}_{[\mathbf{x}]})^T. \tag{4.11}$$

Afterwards, we perform another VJP operation on the function g with related to its input $\mathbf{u}$, and can get the JVP expression,

$$\text{VJP}(\mathbf{v}, g, \mathbf{u})^T = (\mathbf{v}^T \mathbf{J}^{[g]}_{[\mathbf{u}]})^T = (\mathbf{v}^T \mathbf{J}^{[f]T}_{[\mathbf{x}]})^T = \mathbf{J}^{[f]}_{[\mathbf{x}]} \mathbf{v} = \text{JVP}(\mathbf{v}, f, \mathbf{x}). \tag{4.12}$$

We also implement the HVP function based on the gradient function. We only consider the case when the function output is a scalar, because it is the general case where Hessian

[3]The JVP implementation is based on the technique introduced at https://j-towns.github.io/2017/06/12/A-new-trick.html.

matrices are used. The HVP is formulated based on two gradient calculations, because HVP is equivalent to the gradient of the gradient-vector inner product. The expression is shown as follows,

$$\text{HVP}(\mathbf{v}, f, \mathbf{x}) = \mathbf{H}^{[f]}_{[\mathbf{x}]}\mathbf{v} = \frac{\partial \mathbf{g}^{[f]}_{[\mathbf{x}]}}{\partial \mathbf{x}}\mathbf{v} = \frac{\partial \mathbf{g}^{[f]}_{[\mathbf{x}]}}{\partial \mathbf{x}}\mathbf{v} + \mathbf{g}^{[f]T}_{[\mathbf{x}]}\frac{\partial \mathbf{v}}{\partial \mathbf{x}} = \frac{\partial(\mathbf{g}^{[f]T}_{[\mathbf{x}]}\mathbf{v})}{\partial \mathbf{x}} = \text{grad}(\text{grad}(f, \mathbf{x})^T\mathbf{v}, \mathbf{x}). \tag{4.13}$$

4.3.2 Explicit Jacobian and Hessian

To the best of our knowledge, all of the popular AD frameworks calculate explicit Jacobian and Hessian based on the VJP and HVP routines [129]–[131]. Taking the Jacobian calculation of

$$f(\mathbf{x}) = \mathbf{A}_1\mathbf{A}_2\mathbf{x} \tag{4.14}$$

as an example: when both $\mathbf{x}$ and $f(\mathbf{x})$ are of size n, current methods will compute the ith row of the Jacobian via VJP $\mathbf{e}_i^T\mathbf{J}^{[f]}_{[\mathbf{x}]}$ for $i \in \{1, \dots, n\}$, where $\mathbf{e}_i$ is the ith elementary vector. There are two major disadvantages to this approach:

- It changes the BLAS-3 level matrix-matrix multiplications to multiple BLAS-2 level matrix-vector multiplications, and less flop intensity can be achieved. Although many frameworks provide the routine to compute all the matrix-vector multiplications in parallel, the parallelism is still sub-optimal and less efficient than the matrix multiplications, because the flop-to-byte ratio is $O(1)$ versus $O(n)$.

- The computational graph produced is difficult to optimize. Although having high dimensions, many Jacobians / Hessians in tensor computation operations are highly structured and the computational cost can be greatly reduced if being well optimized. However, calculating them based on matrix-vector products adds one more matrix-vector product operation, which usually break the structure and increase the cost. For example, if $\mathbf{A}_1 = \mathbf{B} \otimes \mathbf{C}$ and $\mathbf{A}_2 = \mathbf{D} \otimes \mathbf{E}$ and $\mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}$ have sizes $n \times n$, performing matrix-vector product for the Jacobian and each elementary vector costs $O(n^4)$ and the overall Jacobian calculation cost is $O(n^6)$. However, if we calculate the Jacobian directly, we can use the mixed-product property of the Kronecker product to optimize the expression,

$$(\mathbf{B} \otimes \mathbf{C})(\mathbf{D} \otimes \mathbf{E}) = (\mathbf{BD}) \otimes (\mathbf{CE}), \tag{4.15}$$

reducing the overall cost to $O(n^4)$.

To alleviate these disadvantages, we produce both Jacobian and Hessian expressions in a way that's independent of VJP and HVP routines.

For the Jacobian expression, our implementations are also based on the chain rule to perform back propagation, using

$$\text{Jacobian}(f, \mathbf{x}_i) {=} \mathbf{J}^{[f]}_{[\mathbf{x}_i]} {=} \mathbf{J}^{[f]}_{[\mathbf{x}_{i+1}]} \mathbf{J}^{[f_i]}_{[\mathbf{x}_i]} {=} \text{Jacobian}(f, \mathbf{x}_{i+1}) \mathbf{J}^{[f_i]}_{[\mathbf{x}_i]}. \tag{4.16}$$

Therefore, the Jacobian of one target node is the matrix-matrix product between the Jacobian of its output node and the Jacobian of the local function. Note that when both $\mathbf{x}_i$ and the Jacobian have the tensor format, the above equation still holds, except that the matrix-matrix product is expressed in the form of tensor contractions (Einsums).

For linear operations, such as addition, subtraction, scalar-tensor multiplication and Einsum, we formulate the Jacobian expressions as an Einsum. To achieve that, we introduce the *Identity node*, which is a node that applies an identity matrix, to express the constraints in Jacobian tensors. For example, for the addition operations of two order N tensors,

$$f(\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}) = \boldsymbol{\mathcal{A}} + \boldsymbol{\mathcal{B}}, \tag{4.17}$$

its Jacobian is a tensor of order $2N$, where $\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{A}}]}(x_1, \dots, x_{2N}) = 1$ if and only if $x_i = x_{i+N}$ for $i \in \{1, \dots, N\}$, and other elements are 0. This constraint can be easily specified with identity nodes. For the order 3 addition, the Jacobian of $\boldsymbol{\mathcal{A}}$ can be expressed as

$$\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{A}}]}(i, j, k, l, m, n) = \mathbf{I}(i, l)\mathbf{I}(j, m)\mathbf{I}(k, n). \tag{4.18}$$

Similarly, we can use the method to express the Jacobians for all the other linear operations. For example, for an Einsum expression below, its Jacobians are written as

$$f(\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}})(i, j, k) = \sum_l \boldsymbol{\mathcal{A}}(i, k, l)\boldsymbol{\mathcal{B}}(j, k, l), \tag{4.19}$$

$$\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{A}}]}(i, j, k, m, n, o) = \mathbf{I}(i, m)\mathbf{I}(k, n)\boldsymbol{\mathcal{B}}(j, n, o), \tag{4.20}$$

$$\boldsymbol{\mathcal{J}}^{[f]}_{[\boldsymbol{\mathcal{B}}]}(i, j, k, m, n, o) = \mathbf{I}(j, m)\mathbf{I}(k, n)\boldsymbol{\mathcal{A}}(i, n, o). \tag{4.21}$$

Although we have introduced several identity nodes, they can be easily pruned so that only necessary identity nodes are left, which will be introduced in Section 4.4. The Hessian routines are based on the Jacobian routines: we perform Jacobian calculations twice to get the Hessian

Algorithm 4.1: Graph optimization

```
1: Input: Input Graph G
2: G = FuseAllEinsum(Distribution(G))                    ▷ Provide longer Einsums
3: G = SymbolicExecution(G)           ▷ Decompose Inverse / Prune identity / SymPy
4: G = OptContractPath(G)                          ▷ Find efficient contraction order
5: OG = CSE(G)                                 ▷ Common Subexpression Elimination
6: Return: Optimized Graph OG
```

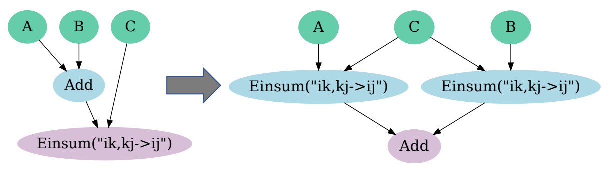
expressions. The advantage of this Jacobian / Hessian generation method is three-fold: first, we can leverage BLAS-3 level operations to perform most of the tensor contractions and can achieve higher performance. Second, the expressions are much easier to optimize, as will be introduced below. Third, the source code for Jacobian / Hessian expressions can be easily acquired, which is beneficial for both debugging and research purposes.

4.4 GRAPH OPTIMIZATIONS

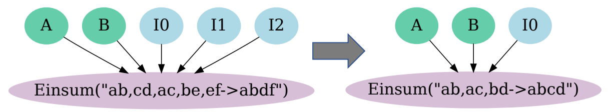
We built a compiler to optimize tensor computational graphs. The compiler is specifically designed for tensor expressions with multilinear operations, including tensor contractions (Einsum) and linear algebra operations (addition, multiplication, summation, inversion and so on). Our goal is to reduce the computational cost by transforming the graph to an equivalent form. Given the fact that retrieving the optimal execution graph is NP-hard, we devise several application-driven heuristic strategies:

- *Generation of longer Einsum nodes:* To achieve this, we implement two kernels, Einsum distribution and Einsum fusion.
- *Symbolic rule execution:* We implement the structured inverse node decomposition and redundant node pruning kernels. In addition, we use SymPy [169] to simplify elementary algebraic operations.
- *Contraction order selection:* We select the contraction path on fully simplified expressions.
- *Constrained contraction path construction:* To accelerate alternating minimization, we provide a kernel to reuse intermediates between optimization subproblems.

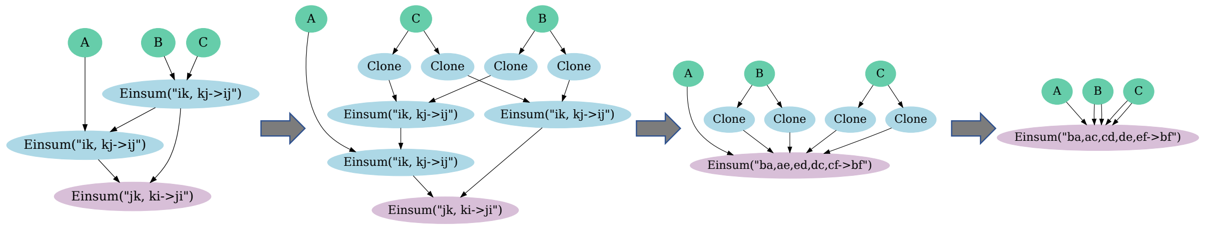
Traditional compiler techniques, such as common sub-expressions elimination, are applied after the strategies above. The overall algorithm is described in Algorithm 4.1.



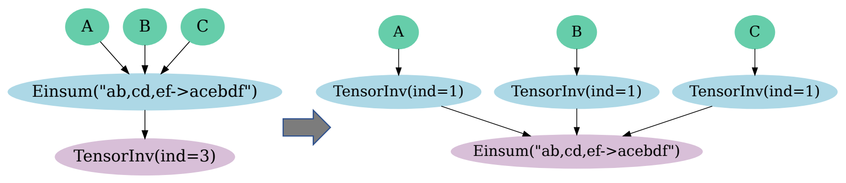
(a) Einsum distribution.



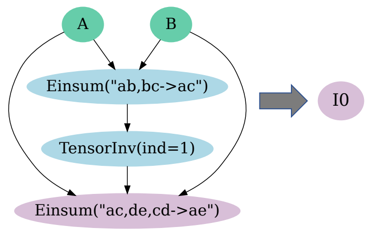
(b) Identity node pruning. The nodes whose name starts from "I" are identity nodes.



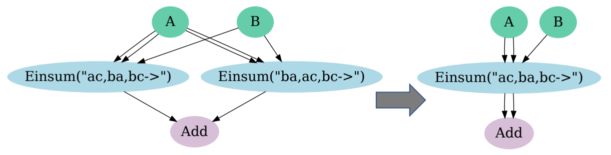
(c) Einsum fusion. It transforms an Einsum graph into one single Einsum node.



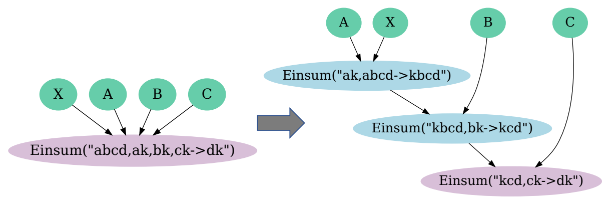
(d) Optimization of tensor inversion



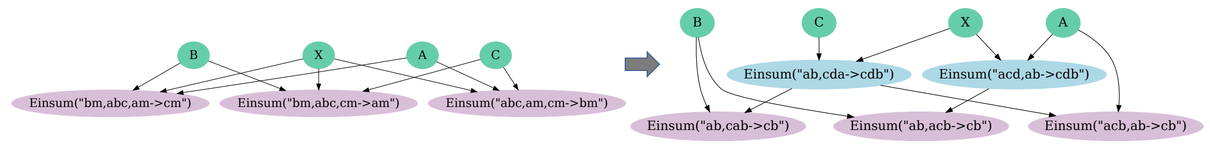
(e) Inverse node pruning



(f) Common subexpression elimination



(g) Optimal contraction path



(h) Dimension tree generation

Figure 4.4: Visualization of different graph optimization kernels.

4.4.1 Longer Einsum Nodes Generation

We aim to transform the computational graph into Einsum nodes with as many inputs as possible. This optimization will empower the contraction path selection with a global view and ease the discovery of optimizable patterns for downstream algorithms. To achieve this, we introduce two transformation kernels.

Einsum distribution. Einsum distribution recursively leverages distributivity of tensor

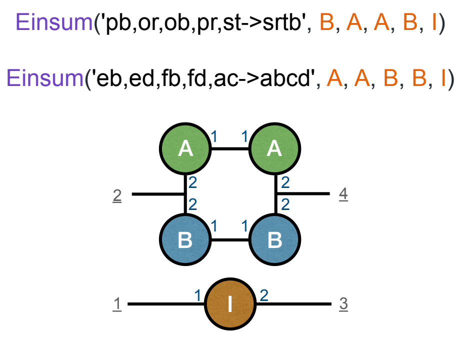

Einsum('pb,or,ob,pr,st->srtb', B, A, A, B, I)
Einsum('eb,ed,fb,fd,ac->abcd', A, A, B, B, I)
A
A
B
B
I
1
2
3
4

Figure 4.5: Tensor diagram of two Einsum expressions with the same tensor computations. The numbers around the input tensor denote the dimension numbers that are contracted by specific edges. The underlined numbers denote the dimension number of the output tensor. Two Einsum expressions with the same tensor diagram express the same tensor computations.

contraction over tensor addition (or another distributive operation) to generate larger Einsum graphs. Larger Einsum graphs are the prerequisite for further graph depth reduction. This optimization moves the nodes performing the distributive operation (`dist_op`) closer to the graph sinks based on the programmatic rule in Fig. 4.6. Fig. 4.4a illustrates the idea of an application of the algorithm. while the pseudo-code can be found in Section 4.7.3.

```
Einsum(dist_op(g1, g2), g3) =
dist_op(Einsum(g1, g3), Einsum(g2, g3))
```

Figure 4.6: Illustration of einsum distribution.

Einsum fusion. Einsum fusion transforms an Einsum graph into several distinct Einsum nodes with the same set of source vertices (inputs) leveraging associativity of tensor contractions. It is a prerequisite for downstream graph optimization steps, such as contraction path selection and identity node pruning. An example can be seen in Fig. 4.4c.

Einsum fusion has three steps: linearization of the graph, fusion of the generated Einsum Tree, and removal of the redundant clone nodes. The linearization step changes the input Einsum graph into an Einsum tree. When a source node is used in multiple Einsums, we create a clone of it for each Einsum. If an Einsum node has more than one output, we copy the subgraph defining its computation, including itself, and repeat until all nodes have a single output, yielding a forest (set of disconnected trees). The fusion step fuses each generated Einsum tree. It leverages a union-find data structure, which puts two dimensions from two Einsum nodes into one set if they have the same subscript in one Einsum expression. After that, each disjoint set is assigned an unique character for the generation of the subscript of the new Einsum node. Finally, the clone node removal step removes the redundant clone nodes and returns an Einsum node. We illustrate both the pseudo-code sketch of the algorithm and the union-find data structure in Section 4.7.3.

4.4.2 Symbolic Execution

We employ several linear algebra constructs that can simplify the computational graph and reduce the computational cost.

Structured Tensor inverse decomposition. An inverse of an Einsum graph may be the bottleneck of the computational graph because of the cubic order complexity. Fortunately, structured information may guide the optimization, e.g, the inverse of a Kronecker Product can be decomposed into the Kronecker product of inverses through $(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}$. We develop an algorithm to detect and break large tensor inverses into products of smaller tensor inverses so that the computation is cheaper. To keep it simple, the algorithm limits its applicability to specific forms of the tensors, and further details are described in Section 4.7.2. An illustrative example is shown is Fig. 4.4d.

Redundant node pruning. We prune the redundant nodes, including the Identity nodes and the inverse nodes, to simplify the expressions. Identity nodes are essential building blocks for the explicit Jacobian and the Hessian expressions, as is shown in Section 4.3.2. During the AD, redundant Identity nodes are introduced to aid the construction of the graph. Hence, we implement an algorithm to eliminate the unnecessary identity nodes afterwards for better efficiency. Identity nodes are removed unless they express necessary constraints in the output tensor structure, such as the tensor symmetry shown in the right graph of Fig. 4.4b. In addition, we prune the unnecessary inverse nodes, as is shown in Fig. 4.4e. When there exists an matrix multiplication between an Einsum Node and its corresponding inverse node, we directly return an identity node.

Elementary algebraic simplification. For elementary operations, such as addition, subtraction and multiplication, we use the SymPy library [169] to optimize them. SymPy can help us easily simplify the expressions. For the example shown in Fig. 4.7, it helps reducing the expression to one term.

```
sympy_simplify(
    (A-(((A*0.5)-(T*0.5))+((A*0.5)-(T*0.5))))
) = T
```

Figure 4.7: Illustration of algebraic simplification.

4.4.3 Optimized Contraction Path Selection

We identify the optimal contraction path for the Einsum expression after all the above transformations. For one Einsum node with multiple inputs, we provide an function to decompose it into an Einsum graph with the optimized contraction path, as is shown in Fig. 4.4g. Our strategy is designed for the common tensor contractions with the following two assumptions:

- For simplicity, we only discuss the case where tensors are dense, and for a long Einsum expression with multiple inputs, it will first be split into multiple small Einsum expressions, each has only two inputs, and then dense tensor contractions will be executed.

- The chosen contraction path is hardware oblivious. We assume the contraction time for each operation is proportional to the flop counts. Other factors, such as the communication cost among different processes under the parallel execution settings, are not considered.

These assumptions allow us to implement the algorithm based on an interface provided by Opt_Einsum [163]. Note that whether we can find the optimal contraction path is based on the optimization algorithm, but we generally found that a greedy search algorithm is able to provide an optimal path for most of the Einsum expressions in tensor computation applications.

In addition, the assumptions above are not limitations of our overall approach. AutoHOOT is also capable of extracting the contraction path based on other libraries, such as Cyclops [140], where hardware and tensor sparsity are considered in the algorithm.

4.4.4 Constrained Contraction Path Construction

We provide a constrained contraction path selection routine, such that the contraction path is optimized under the constraint that partial inputs' contraction order is fixed. This routine is critical for the dimension tree construction used in the alternating minimization algorithms. Consider (4.7), with the update sequence in each sweep starting from $\boldsymbol{\mathcal{A}}_1$ and ending at $\boldsymbol{\mathcal{A}}_N$, for the Einsum node used to update $\boldsymbol{\mathcal{A}}_i$, where $i \in \{1, \dots, N\}$, we generate the contraction path such that it is optimized under the constraint that the contraction order for all the target sites is $\boldsymbol{\mathcal{A}}_N \prec \cdots \prec \boldsymbol{\mathcal{A}}_{i+1} \prec \boldsymbol{\mathcal{A}}_1 \prec \cdots \prec \boldsymbol{\mathcal{A}}_{i-1}$. This order ensures that the tensor that is updated just previously, $\boldsymbol{\mathcal{A}}_{i-1}$, affects only the last part of the contraction path, enabling the reuse of the calculations prior to it in the path as much as possible.

CPD Kernel	Size (s)	Backend	Backend AD	AD	AD + OPT1	AD + OPT1,2	AD + OPT1,2,3	Overall speed-up
GN Jacobian	25	JAX	0.1449s	0.0632s	0.0126s	0.0126s	0.0126s	11X
		TensorFlow	1.5201s	0.1037s	0.0029s	0.0029s	0.0029s	524X
GN HVP	40	JAX	0.0107s	0.0011s	0.0012s	0.0011s	0.0011s	9X
		TensorFlow	0.0040s	0.0027s	0.0048s	0.0048s	0.0048s	0.8X
	640	JAX	0.3742s	0.776s	0.0056s	0.0054s	0.0051s	73X
		TensorFlow	0.9669s	0.9746s	0.4470s	0.3422s	0.2795s	3X
ALS Hessian	40	JAX	0.0713s	OOM	0.0017s	0.0017s	0.0017s	41X
		TensorFlow	0.3643s	OOM	0.0021s	0.0021s	0.0014s	260X
	160	JAX	OOM	OOM	1.0682s	1.0682s	0.8141s	/
		TensorFlow	OOM	OOM	3.0164s	3.0164s	1.5405s	/
ALS Hessian inv	40	JAX	0.1623s	OOM	0.0908s	0.0090s	0.0090s	18X
		TensorFlow	0.4237s	OOM	0.0278s	0.0028s	0.0028s	151X
	160	JAX	OOM	OOM	13.13s	1.5160s	1.5110s	/
		TensorFlow	OOM	OOM	OOM	0.5786s	0.5585s	/

Table 4.1: Detailed performance gain from each graph optimization technique on different CPD kernels. The rank is set the same as the input tensor dimension/size along each mode (s). Results are collected on an NVIDIA Titan X GPU. We denote each technique as: Einsum fusion + distribution: OPT1, Symbolic optimization: OPT2, CSE: OPT3.

The constrained path selection algorithm is illustrated in Algorithm 4.2, and is implemented on top of the unconstrained one and uses the greedy search heuristic. We find that this heuristic works well for all the dimension tree selection in the tensor computation applications tested in Section 4.5. An example is shown in Fig. 4.4h, which illustrate the dimension construction for the Matricized Tensor Times Khatri-Rao Product (MTTKRP) calculations of an order 3 CP decomposition. The pseudo-code is illustrated in Section 4.7.3.

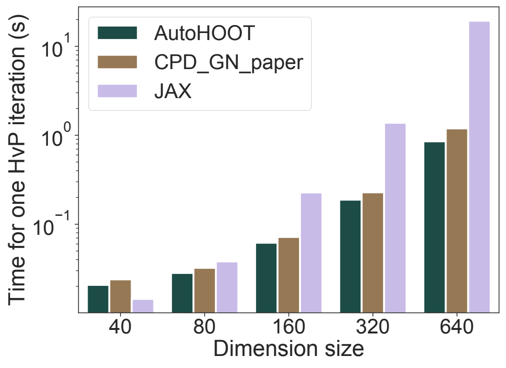
4.4.5 Common Subexpression Elimination (CSE)

CSE is used to remove the duplicated Einsum expressions generated from the path selection above. We show one example in Fig. 4.4f, where CSE helps saving one Einsum calculation. However, CSE is nontrivial for Einsum nodes because different Einsum subscripts may represent the same computation. We show an example in Fig. 4.5 where two Einsum nodes represent the same calculation despite different input ordering and subscripts. Hence, we transfer an Einsum expression into a tensor diagram graph, and compare the graph structures between two expressions.

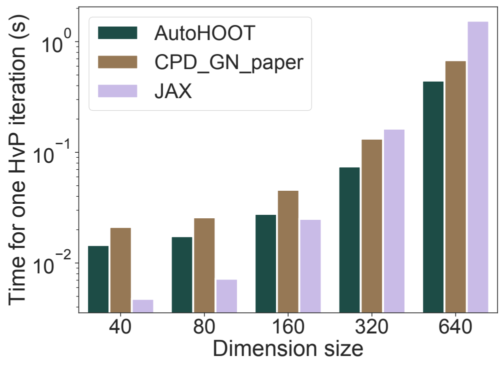
Moreover, two nodes in an Einsum graph may be transpositions of each other. After detecting such conditions, we replace one of the nodes with its transpose node and update its outputs' expressions therein. This optimization greatly reduces the computation cost when transposes of large tensors appear in the graph.

Algorithm 4.2: Opt_contraction_path_w_constraint

```
1:  Input: Einsum Node N, Contraction order list L
2:  $n$ = length(L)
3:  T = N    ▷ Initialize tree with single Einsum node
4:  for i $\in \{1, \dots, n\}$ do
5:      split_T = SplitEinsum(T, L[i+1:n])    ▷ Split T into an Einsum node that contracts all input nodes apart from L[i+1:n] and the subgraph induced by the remaining nodes, returning the former
6:      opt_contract_subtree = Optcontractionpath(split_T)    ▷ Unconstrained optimized contraction path
7:      opt_contract_subtree = ancestor(opt_contract_subtree, L[i])    ▷ Get the tree whose sink is the nearest ancestor of L[i]
8:      T = substitute(T, opt_contract_subtree) ▷ Return the equivalent graph of T whose inputs contain opt_contract_subtree
9:  end for
10: Return: Einsum Tree T
```



(a) KNL CPU



(b) TESLA K80 GPU

Figure 4.8: Performance comparison among AutoHOOT, JAX and the existing implementation for the HVP kernel in the Gauss-Newton algorithm for the CP decomposition. The implementation of CPD_GN_paper comes from reference [138]. The tensor order is set as $N = 3$, and the CP rank is set equal to the dimension size. Each bar is the average result of 10 iterations.

4.5 BENCHMARKS

We evaluate the performance of AutoHOOT on both the Gauss-Newton method and the alternating minimization method discussed in Section 4.1.2. The performance of the critical Gauss-Newton kernel, the Hessian-Vector Product, is evaluated on the CP decomposition application, where Gauss-Newton with conjugate gradient update is commonly used to achieve high accuracy [138], [170]. The performance of alternating minimization kernels

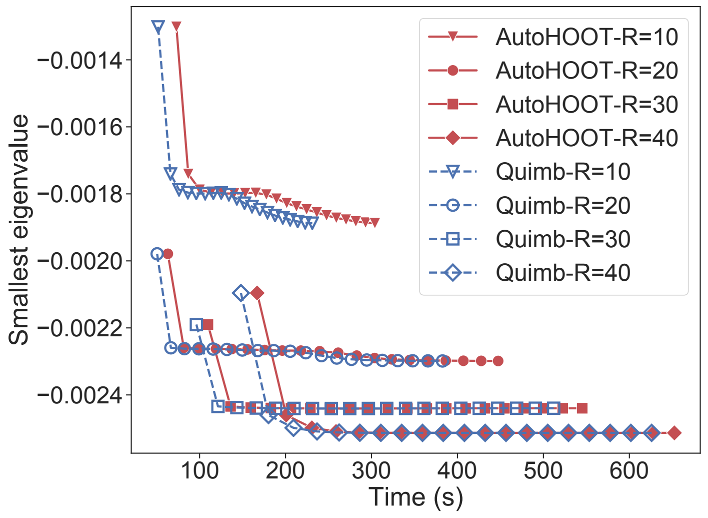
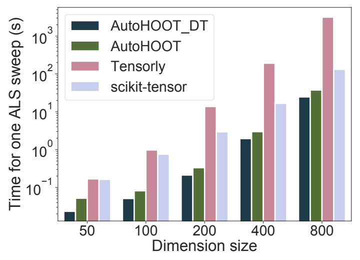


Figure 4.9: Comparison between AutoHOOT and Quimb on the full DMRG running curve. The input MPO is random and symmetric, has 6 sites, and its physical leg size equals 10 and MPO rank size equals 20. We compare the performance under different largest MPS rank constraints.

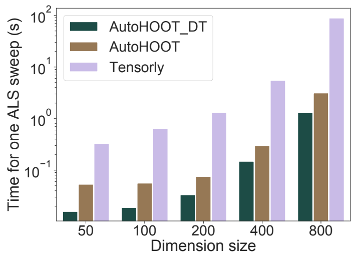
generated by AutoHOOT is evaluated on both CP and Tucker decompositions, as well as the DMRG algorithm in tensor network applications used to calculate the smallest eigenvalue and eigenvector for a matrix product state.

The experiments are run on both CPUs and GPUs. On CPUs, we test the performance on both one process with the NumPy backend, and on the distributed parallel system with the Cyclops backend. The results are collected on the Stampede2 supercomputer located at the University of Texas at Austin. We leverage the Knight's Landing (KNL) nodes, each of which consists of 68 cores, 96 GB of DDR RAM, and 16 GB of MCDRAM. These nodes are connected via a 100 Gb/sec fat-tree Omni-Path interconnect. We use Intel compilers and the MKL library for threaded BLAS routines for both sequential and parallel experiments. We use 16 processes per node and 16 threads per process for the Cyclops benchmark experiments. We also collected results with both TensorFlow and JAX backends on both single NVIDIA TESLA K80 GPU and single NVIDIA Titan X GPU.

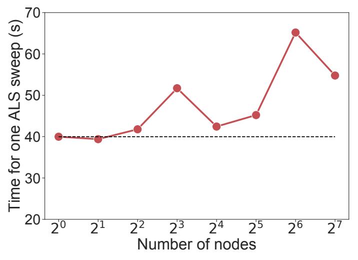
We first compare the detailed performance gain from each graph optimization technique proposed in Section 4.4. The experiments are performed on the Jacobians and HVPs kernels in the Gauss-Newton (GN) methods, as well as Hessians and Hessian inverses used in the ALS algorithm for CP decompositions and are shown in Table 4.1. As can be seen in the table, Einsum fusion and distribution are critical for almost all the calculations, and Symbolic optimization is critical for tensor/matrix inverse. In addition, CSE provides incremental



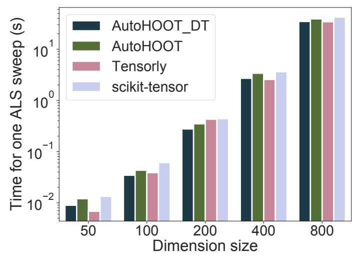
(a) NumPy, CPD, $R = s$



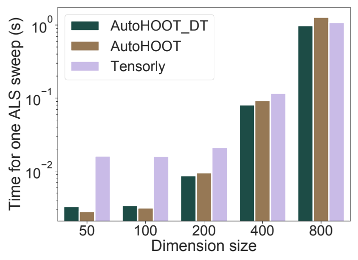
(b) TensorFlow, CPD, $R = s$



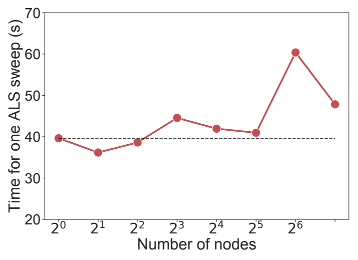
(c) Cyclops, $R = 400, s = \lfloor 1320n^{\frac{1}{3}} \rfloor$



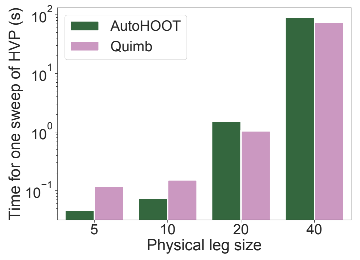
(d) NumPy, Tucker, $R = 0.5s$



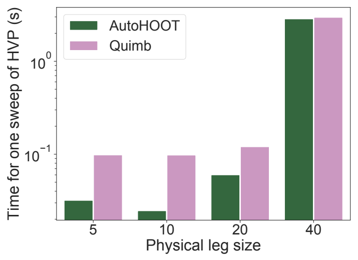
(e) TensorFlow, Tucker, $R = 0.5s$



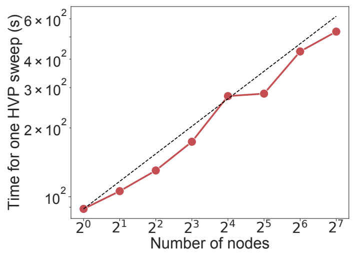
(f) Cyclops, $R = 400, s = \lfloor 1240n^{\frac{1}{3}} \rfloor$



(g) NumPy, DMRG, $s = R$



(h) TensorFlow, DMRG, $s = R$



(i) Cyclops, $R = s = \lfloor 50n^{\frac{1}{5}} \rfloor$

Figure 4.10: AutoHOOT performance for kernels in the alternating minimization. TensorFlow results are collected on an NVIDIA TESLA K80 GPU. **(a)-(c)**: Results for the CP decomposition. The tensor order is set as $N = 3$ for all the experiments. **(d)-(f)**: Results for the Tucker decomposition. The tensor order is set as $N = 3$ for all the experiments. **(g)-(i)**: Results for the DMRG experiment. The number of sites is set as $N = 10$ for the experiments with NumPy and TensorFlow, and set as $N = 6$ for the experiments with Cyclops. For the Cyclops benchmark, the dotted line denotes the perfect scaling curve. Each bar/dot is the average result of 10 iterations.

performance gain.

The performance of the HVP kernels in the Gauss-Newton algorithm for the CP decomposition is shown in Fig. 4.8. As can be seen, AutoHOOT has at least 2X speed-up on the GPU and at least 7X speed-up on the CPU compared to JAX when the dimension size $s \geq 320$. Note that JAX performs better for small HVP kernels, because the experiments with AutoHOOT are performed on TensorFlow, where JAX has faster small contractions. It can be seen that the speed-up increases with the increase of the dimension size, indicating the advantage of AutoHOOT for large scale tensor computations. In addition, the AutoHOOT performance is comparable compared to the manually designed algorithms in the reference [138], indicating that the kernels generated by AutoHOOT reaches the state-of-art performance boundary.

The performance of the alternating minimization kernels for both tensor decompositions and the DMRG algorithm are shown in Fig. 4.10. For the tensor decompositions, we compare the performance of AutoHOOT output expressions, both with and without dimension tree optimizations, to the popular tensor decomposition libraries Tensorly [133], both with NumPy and TensorFlow backend, and scikit-tensor[4] with NumPy backend. For the DMRG algorithm, we compare the performance to Quimb [135], which is an efficient library for tensor networks.

The benchmark results for the CP decomposition with both NumPy and TensorFlow can be seen in Fig. 4.10a and Fig. 4.10b. We compare the performance with different CP ranks (R) and dimension size (s). As can be seen, the expressions generated with the dimension tree algorithm outperform all the other implementations. Note that Tensorly's performance is not as expected for the CP decomposition, because it slices the factor matrices over the rank mode and sums over all the MTTKRP results of the input tensor and the sliced factor matrices, which is not favorable. The weak scaling benchmark is also performed on the distributed parallel system with Cyclops, shown in Fig. 4.10c, where we consider weak scaling with fixed input size and work per processor. The expressions generated from AutoHOOT scale well, obtaining 73% parallel scaling efficiency on 128 nodes (2048 cores).

The benchmark results for the Tucker decomposition with both NumPy and TensorFlow can be seen in Fig. 4.10d and Fig. 4.10e. We compare the performance with different Tucker ranks (R) and dimension size (s). Note that we are only comparing the performance of the kernel generated through AutoHOOT to the Tensor Times Matrix-chain (TTMc) implementation in other libraries, which doesn't contain the low rank factorization step of splitting the factor matrix from the core tensor. The expressions generated with the dimension tree algorithm is comparable to all the other implementations. The weak scaling benchmark is shown in Fig. 4.10f. Similar to the CP decomposition, the expressions generated from AutoHOOT scale with high efficiency.

The performance results for DMRG can be seen in Fig. 4.10g, Fig. 4.10h, and Fig. 4.10i. We benchmark over sweep of the HVP kernels with different MPO and MPS rank size (R) and physical dimension size (s), where the Hessian denotes the local Hessian of the DMRG

[4]`https://github.com/mnick/scikit-tensor`

loss function w.r.t. each local site. In DMRG, the HVP calculations are important kernels for the sparse eigensolver. Multiple HVP calculations are necessary for each site to get the local smallest eigenvalue, making it the computation bottleneck. The expressions generated with the dimension tree algorithm achieve comparable performance to the implementations in Quimb. In addition, the expressions generated from AutoHOOT scale nearly perfectly with Cyclops up to at least 128 nodes[5].

We also compare the performance between AutoHOOT and Quimb on the full DMRG experiments. Like Quimb, we use the sparse eigensolver in SciPy [171], and set the solver parameters the same as Quimb. The results are shown in Fig. 4.9. We test the four cases where the maximum MPS rank ranges from 10 to 40, and the results show that both libraries have the similar performance, while AutoHOOT has a small fixed overhead.

Note that we did not report the ALS results of other AD libraries, because their performance is far worse than both AutoHOOT and other tensor computation libraries. For both CP and Tucker decompositions, existing AD libraries cannot efficiently decompose the structured inverse operations, leading to a big overhead from inverting large tensors. For the DMRG experiment, existing libraries fail to choose an optimized contraction path, and produce large intermediates which require too much memory.

4.6 CONCLUSION

AutoHOOT is the first automatic differentiation framework targeting high-order optimization for tensor computations. AutoHOOT contains a new explicit Jacobian / Hessian expression generation kernel whose outputs keep the input tensors' granularity and are easy to optimize. It also contains a new computational graph optimization module that combines both the traditional optimization techniques for compilers and techniques based on specific tensor algebra. The optimization module generates expressions as good as manually written codes in other frameworks for the numerical algorithms of tensor computations. AutoHOOT is compatible with other numerical computation libraries, and users can execute the generated expressions on CPU with NumPy, GPU with TensorFlow, and distributed parallel systems with Cyclops Tensor Framework. Experimental results show that AutoHOOT has competitive performance on both tensor decomposition and tensor network applications compared to both existing AD software and other tensor computation libraries with manually written kernels, both on CPU and GPU architectures.

[5]In these experiments, we constrain the physical leg size to be equal to the rank, e.g. $s = R$, so the computational cost is $O(R^7)$ and the memory footprint is $O(R^5)$.

4.7 ADDITIONAL BACKGROUND AND RESULTS

4.7.1 Background of Tensor Computation Applications

In this Section we provide background on CP decomposition, Tucker decomposition, and Density Matrix Renormalization Group (DMRG).

CP decomposition. The CP tensor decomposition [7], [13] serves to approximate a tensor by a sum of R tensor products of vectors. For an order N input tensor $\boldsymbol{\mathcal{X}}$ with size $s_1 \times \cdots \times s_N$, CP decomposition compresses it into N factor matrices $\mathbf{A}_1, \ldots, \mathbf{A}_N$, size of each is $s_i \times R$ for $i \in \{1, \ldots, N\}$. The optimization for the CP decomposition is a least squares problem, where element-wise expression for the output of the tensor network f in (4.1) denotes

$$f(\mathbf{A}_1, \ldots, \mathbf{A}_N)(x_1, \ldots, x_N) := \sum_{k=1}^{R} \prod_{i \in \{1,\ldots,N\}} \mathbf{A}_i(x_i, k). \tag{4.22}$$

Both the Gauss-Newton method and the alternating minimization method, which is also called alternating least squares (ALS) are effective and popular for the CP decomposition.

The CP-ALS method alternates among quadratic optimization problems for each of the factor matrices $\mathbf{A}_n$, resulting in linear least squares problems for each row, are often solved via the normal equations [5],

$$\mathbf{A}_n \mathbf{\Gamma}_n \leftarrow \mathbf{X}_{(n)} \mathbf{P}_n, \tag{4.23}$$

where the matrix $\mathbf{P}_n \in \mathbb{R}^{I_n \times R}$, where $I_n = \prod_{i=1, i\neq n}^{N} s_i$, is formed by Khatri-Rao products of the other factor matrices,

$$\mathbf{P}_n = \mathbf{A}_N \odot \cdots \odot \mathbf{A}_{n+1} \odot \mathbf{A}_{n-1} \odot \cdots \odot \mathbf{A}_1, \tag{4.24}$$

and $\mathbf{\Gamma}_n \in \mathbb{R}^{R \times R}$ can be computed via

$$\mathbf{\Gamma}_n = \mathbf{S}_1 * \cdots * \mathbf{S}_{n-1} * \mathbf{S}_{n+1} * \cdots * \mathbf{S}_N, \tag{4.25}$$

with each $\mathbf{S}_i = \mathbf{A}_i^T \mathbf{A}_i$. The *Matricized Tensor Times Khatri-Rao Product* or MTTKRP computation $\mathbf{M}_n = \mathbf{X}_{(n)} \mathbf{P}_n$ is the main computational bottleneck of CP-ALS [101]. Within MTTKRP, the bottleneck is the contraction between the input tensor and the first-contracted matrix. For a rank-R CP decomposition, this computation has the leading cost of $2s^N R$ if $s_n = s$ for all $n \in \{1, \ldots, N\}$. The dimension-tree algorithm for ALS [59], [99] uses a fixed amortization scheme to update MTTKRP in each ALS sweep. This scheme only needs to perform two first contraction calculations for each ALS sweep, decreasing the leading order cost of a sweep from $2Ns^NR$ to $4s^NR$.

Another alternative is to solve the least squares problem via the Gauss-Newton method. Although directly inverting the Hessian matrix for the problem is expensive (costs $O(N^3s^3R^3)$ if each dimension has size s), the matrices involved in the linear system are sparse and have much implicit structure. The cost of direct Hessian inversion can be reduced to $O(NR^6)$ [137] and of using implicit conjugate gradient method is $O(N^2sR^2)$ for each iteration [170]. Additionally, it has been shown that higher decomposition accuracy can be reached with Gauss-Newton rather than ALS [138]. We refer readers to references for details of the Gauss-Newton implementations for CP decomposition [138], [170].

Tucker decomposition. Tucker decomposition [10] approximates a tensor by a core tensor contracted by orthogonal matrices along each mode. For an order N input tensor $\boldsymbol{\mathcal{X}}$ with size $s_1 \times \cdots \times s_N$, Tucker decomposition compresses it into N matrices with orthogonal columns $\mathbf{A}_1, \cdots, \mathbf{A}_N$, size of each is $s_i \times R_i$ for $i \in \{1, \dots, N\}$, and a core tensor $\boldsymbol{\mathcal{G}}$ with size $R_1 \times \cdots \times R_N$. Similar to CP decomposition, the optimization for the Tucker decomposition is a least squares problem, where element-wise expression for the output of the tensor network f in (4.1) denotes

$$f(\boldsymbol{\mathcal{G}}, \mathbf{A}_1, \dots, \mathbf{A}_N)(x_1, \dots, x_N) = \sum_{\{z_1,\dots,z_N\}} \boldsymbol{\mathcal{G}}(z_1, \dots, z_N) \prod_{r \in \{1,\dots,N\}} \mathbf{A}_r(x_r, z_r). \tag{4.26}$$

The ALS method for Tucker decomposition [5], [29], which is also called the *higher-order orthogonal iteration* (HOOI), proceeds by fixing all except one factor matrix, and computing a low-rank matrix factorization on the *Tensor Times Matrix-chain* (TTMc) $\boldsymbol{\mathcal{Y}}_n$ for $n \in \{1, \dots, N\}$, to update that factor matrix and the core tensor. $\boldsymbol{\mathcal{Y}}_n$ is expressed as

$$\boldsymbol{\mathcal{Y}}_n(z_1, \dots, z_{n-1}, x_n, z_{n+1}, \dots, z_N) = \sum_{\{x_1,\dots,x_{n-1},x_{n+1},\dots x_N\}} \boldsymbol{\mathcal{X}}(x_1, \dots, x_N) \prod_{r \in \{1,\dots,N\}, r \neq n} \mathbf{A}_r(x_r, z_r). \tag{4.27}$$

Then $\boldsymbol{\mathcal{Y}}_n$ is factored into a product of an orthogonal matrix $\mathbf{A}_n$ and the core tensor $\boldsymbol{\mathcal{G}}$, so that $\mathbf{Y}_{n,(n)} \approx \mathbf{A}_n\mathbf{G}_{(n)}$. This factorization can be done by taking $\mathbf{A}_n$ to be the R_n leading left singular vectors of $\mathbf{Y}_{n,(n)}$. TTMc is the computational bottleneck of Tucker-ALS. With the use of dimensions trees same as CP-ALS, the computational complexity for a sweep of TTMc has the leading order $O(4s^NR)$.

Density Matrix Renormalization Group (DMRG). DMRG calculates the smallest eigenvalue of a Matrix Product Operator (MPO) with the corresponding eigenvector represented by a Matrix Product State (MPS). MPS, which is also called tensor train, represents a high dimensional tensor into a linear tensor network. For an order N input tensor $\boldsymbol{\mathcal{V}}$ with size $s_1 \times \cdots \times s_N$, the MPS decomposition is expressed as

$$\boldsymbol{\mathcal{V}}(x_1, \ldots, x_N) = \sum_{\alpha_0, \ldots, \alpha_N} \prod_{i=1}^{N} \boldsymbol{\mathcal{A}}_i(\alpha_{i-1}, x_i, \alpha_i), \tag{4.28}$$

where $\boldsymbol{\mathcal{A}}_i \in \mathbb{R}^{R_{i-1} \times s_i \times R_i}$ and $R_0 = R_N = 1$. The MPO has the similar linear structure, each site is a 4-D tensor,

$$\boldsymbol{\mathcal{W}}(x_1, \ldots, x_N, y_1, \ldots, y_N) = \sum_{\alpha_0, \ldots, \alpha_N} \prod_{i=1}^{N} \boldsymbol{\mathcal{A}}_i(\alpha_{i-1}, x_i, y_i, \alpha_i), \tag{4.29}$$

and for each $i \in \{1, \ldots, N\}$, the ith and $i + N$th mode of $\boldsymbol{\mathcal{W}}$ have the same size. The objective of DMRG is expressed as

$$\min_{\boldsymbol{\mathcal{V}}} \psi(\boldsymbol{\mathcal{V}}) := \frac{\mathbf{v}_{(1:N)}^T \mathbf{W}_{(1:N)} \mathbf{v}_{(1:N)}}{\|\mathbf{v}_{(1:N)}\|^2}, \tag{4.30}$$

where we are optimizing $\boldsymbol{\mathcal{V}}$ under the constraint that it has the MPS structure. DMRG optimizes this objective via alternating minimization, where in each local step the minimum of the objective w.r.t. one or two neighboring sites is achieved, and performs sweeps of the local steps until the results converged. We refer readers to the tensornetwork website for algorithm details[6].

4.7.2 Proofs for Structured Inverse Node Decomposition

In our program, for an implicit tensor constructed through several input tensors and an Einsum expression, our optimization algorithm finds the form of its decomposed tensors that obey the rules in Theorem 4.1, thus helping the inverse.

At first, we define the terms *decomposable tensor*, *tensor inverse* and *identity tensor* as follows:

Definition 4.1. *A tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{s_1 \times \cdots \times s_N}$ is decomposable if it can be written as the outer product of 2 smaller tensors. It can be written as*

$$\boldsymbol{\mathcal{T}}(x_1, \ldots, x_N) = \boldsymbol{\mathcal{A}}(y_1, \ldots, y_M) \boldsymbol{\mathcal{B}}(z_1, \ldots, z_K), \tag{4.31}$$

[6]https://tensornetwork.org/mps/algorithms/dmrg/

where $\{y_1, \dots, y_M\} \cup \{z_1, \dots, z_K\} = \{x_1, \dots, x_N\}$ *and* $\{y_1, \dots, y_M\} \cap \{z_1, \dots, z_K\} = \emptyset$.

Definition 4.2. *For an even order tensor* $\boldsymbol{\mathfrak{T}} \in \mathbb{R}^{s_1 \times \cdots \times s_{2N}}$, *let* $R_1 = \prod_{i=1}^{N} s_i, R_2 = \prod_{i=N+1}^{2N} s_i$, *if* $R_1 = R_2$, *its tensor inverse* $\boldsymbol{\mathfrak{T}}^{-1}$ *is defined as the inverse of the matricized tensor* $\mathbf{T}$, *where* $\mathbf{T} \in \mathbb{R}^{R_1 \times R_2}$.

Definition 4.3. *We use* $\boldsymbol{\mathfrak{I}}_T$ *to denote a tensor has the same shape as* $\boldsymbol{\mathfrak{T}}$, *and the matricized* $\boldsymbol{\mathfrak{I}}_T$ *is an identity matrix.*

Using the definitions above, we will show that if a tensor meets the requirement described in Theorem 4.1, then we can transfer the tensor inverse into the inverse of its decomposed parts.

Theorem 4.1. *For an even order tensor* $\boldsymbol{\mathfrak{T}} \in \mathbb{R}^{s_1 \times \cdots \times s_{2N}}$, *if it can be decomposed into 2 tensors* $\boldsymbol{\mathcal{A}}$ *and* $\boldsymbol{\mathfrak{B}}$ *as:*

$$\boldsymbol{\mathfrak{T}}(x_1, \dots, x_{2N}) = \boldsymbol{\mathcal{A}}(y_1, \dots, y_{2M}) \boldsymbol{\mathfrak{B}}(z_1, \dots, z_{2K}), \tag{4.32}$$

and the indices satisfy the following requirements:

1. $\{y_1, \dots, y_M\} \cup \{z_1, \dots, z_K\} = \{x_1, \dots, x_N\}$, *and* $\{y_1, \dots, y_M\} \cap \{z_1, \dots, z_K\} = \emptyset$,
2. $y_{i+M} = x_{j+N}$ *if* $y_i = x_j$ *for* $\forall i \in \{1, \dots, M\}$, $z_{i+K} = x_{j+N}$ *if* $z_i = x_j$ *for* $\forall i \in \{1, \dots, K\}$,
3. $\boldsymbol{\mathcal{A}}$ *and* $\boldsymbol{\mathfrak{B}}$ *are both invertible,*

then we have $\boldsymbol{\mathfrak{T}}^{-1}(x_1, \dots, x_{2N}) = \boldsymbol{\mathcal{A}}^{-1}(y_1, \dots, y_{2M}) \boldsymbol{\mathfrak{B}}^{-1}(z_1, \dots, z_{2K})$.

Proof. Let tensor $\boldsymbol{\mathfrak{C}}$ be the outer product of tensor $\boldsymbol{\mathcal{A}}^{-1}$ and $\boldsymbol{\mathfrak{B}}^{-1}$ based on the following element-wise expression:

$$\boldsymbol{\mathfrak{C}}(x_1, \dots, x_{2N}) = \boldsymbol{\mathcal{A}}^{-1}(y_1, \dots, y_{2M}) \boldsymbol{\mathfrak{B}}^{-1}(z_1, \dots, z_{2K}), \tag{4.33}$$

we can rewrite the equation above with different index notations:

$$\begin{aligned} &\boldsymbol{\mathfrak{C}}(x_{N+1}, \dots, x_{2N}, a_1, \dots, a_N) \\ &= \boldsymbol{\mathcal{A}}^{-1}(y_{M+1}, \dots, y_{2M}, y_{2M+1}, \dots, y_{3M}) \boldsymbol{\mathfrak{B}}^{-1}(z_{K+1}, \dots, z_{2K}, z_{2K+1}, \dots, z_{3K}), \end{aligned} \tag{4.34}$$

where

$$\{y_{2M+1}, \dots, y_{3M}\} \cup \{z_{2K+1}, \dots, z_{3K}\} = \{a_1, \dots, a_N\}, \tag{4.35}$$

$$\{y_{2M+1}, \dots, y_{3M}\} \cap \{z_{2K+1}, \dots, z_{3K}\} = \emptyset, \tag{4.36}$$

and $y_{i+2M} = a_j$ if $y_i = x_j$ for $\forall i \in \{1, \dots, M\}$, $z_{i+2K} = a_j$ if $z_i = x_j$ for $\forall i \in \{1, \dots, K\}$.

We denote the matrix multiplication of the matricized $\mathfrak{T}$ and $\mathfrak{C}$ as $\mathfrak{Z}$, and their relations can be shown as

$$\begin{aligned}
\mathfrak{Z}(x_1, \dots, x_N, a_1, \dots, a_N) &= \mathfrak{T}(x_1, \dots, x_{2N})\mathfrak{C}(x_{N+1}, \dots, x_{2N}, a_1, \dots, a_N) \\
&= \boldsymbol{\mathcal{A}}(y_1, \dots, y_{2M})\boldsymbol{\mathcal{A}}^{-1}(y_{M+1}, \dots, y_{2M}, y_{2M+1}, \dots, y_{3M}) \\
&\quad \boldsymbol{\mathcal{B}}(z_1, \dots, z_{2K})\boldsymbol{\mathcal{B}}^{-1}(z_{K+1}, \dots, z_{2K}, z_{2K+1}, \dots, z_{3K}) \\
&= \mathfrak{I}_A(y_1, \dots, y_M, y_{2M+1}, \dots, y_{3M})\mathfrak{I}_B(z_1, \dots, z_K, z_{2K+1}, \dots, z_{3K}) \\
&= \mathfrak{I}_T(x_1, \dots, x_N, a_1, \dots, a_N).
\end{aligned} \tag{4.37}$$

Therefore, the theorem is proved. Q.E.D.

4.7.3 Detailed Optimization Algorithms

In this Section, we provide detailed explanations on the union-find data structure for Einsum. In addition, we provide detailed pseudo-codes for the distribution, Einsum fusion and dimension tree generation kernels discussed in Section 4.4.

Union-find for Einsum. The union-find (UF) representation for Einsum is a key ingredient of the optimization kernels. In the UF graph, each node represents one dimension of a specific tensor in the Einsum graph, and each edge represents a connection between two dimensions in an Einsum expression, where the connection is denoted by two dimensions sharing the same character in an Einsum string. Downstream tasks can leverage this canonical representation of the Einsum graph for analysis. The algorithm of graph building is illustrated in Algorithm 4.3. We use Einsum_subscript to denote the Einsum expression of an Einsum node, for example the string '$ij, jk \to ik$'.

4.7.4 Additional Benchmark Results

We present the additional benchmark results for the kernels in the alternating minimization problems in Fig. 4.11. For both CP and Tucker decompositions, we fix the tensor size in each mode and the decomposition rank, and compare the performance between AutoHOOT and other libraries with different input tensor order. As can be seen, the expressions generated with the dimension tree algorithm outperform all the other implementations. For DMRG, we fix the physical dimension sizes and the MPO/MPS ranks, and compare the performance

Algorithm 4.3: BuildUF

```
1: **Input**: Einsum Graph G
2: Initialize a union-find data structure UF
3: Initialize a map from Einsum character to tensor dimension DM
4: for all einsum nodes $N$ in G do
5:     for all characters C1 in N.einsum_subscript do
6:         for all characters C2 in N.einsum_subscript do
7:             if C1 == C2 then
8:                 UF.connect(DM[C1], DM[C2])
9:             end if
10:        end for
11:    end for
12: end for
13: Output: Union-find data structure UF
```

Algorithm 4.4: Distribution

```
1: Input: Graph G
2: DG = G
3: while True do
4:     for All DistributeOp nodes {Ops} in DG do
5:         if All Einsum nodes are topologically ahead of {Ops} then
6:             return DG
7:         end if
8:         for Op ∈ {Ops} do
9:             DG = Distribute(Op, DG)        ▷ Distribute does Einsum((a+b),c) →
   Einsum(a,c) + Einsum(b,c), where + is the Op
10:        end for
11:    end for
12: end while
13: Output: Distributed Graph DG
```

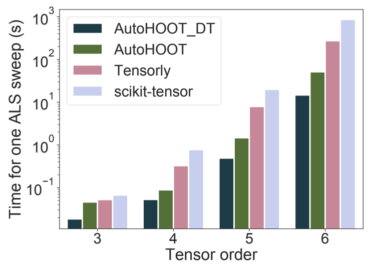
Algorithm 4.5: Einsum fusion

```
1: Input: Einsum Tree T
2: LT = Linearize(T)
3: UF = BuildUF(LT)
4: UF.Assign()                    ▷ Assign each disjoint subset an unique character.
5: Init FN (sink: T.root, source: T.leaves)
6: FN.genereateSubscript()        ▷ Generate FT.subscript based on input nodes' assigned
   characters.
7: FN = Declone(FN)
8: Return: Fused Einsum Node FN
```

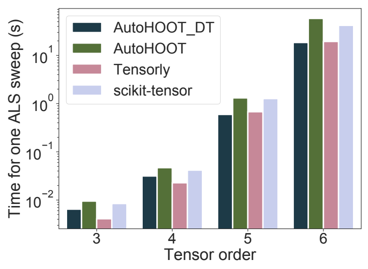
Algorithm 4.6: Dimension tree construction

```
1: Input: Einsum node List: NL, Input node list: IL
2: $n$ = length(NL)
3: UL = NL
4: for i $\in \{1, \dots, n\}$ do
5:     contract_order_list = [I[N], . . . , I[i+1], I[1], . . . , I[i-1]]
6:     contract_order_list = part of contract_order_list where all elements are in UL[i].inputs
7:     UL[i] = optconstraint(UL[i], contract_order_list)
8: end for
9: Return: Updated Einsum node List: UL
```

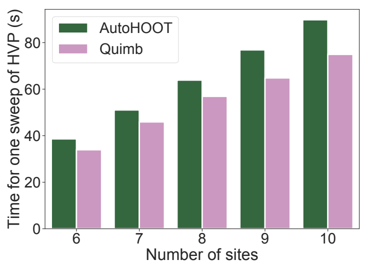
with different number of sites. As can be seen, AutoHOOT and Quimb have comparable performance.



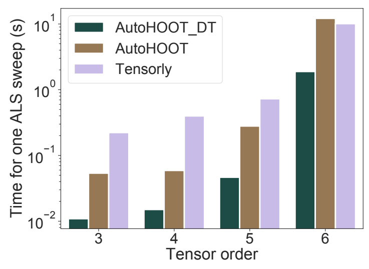
(a) CPD, $s = R = 30$



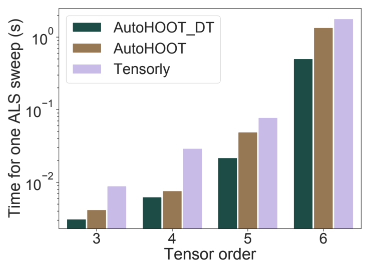
(b) Tucker, $s = 30, R = 10$



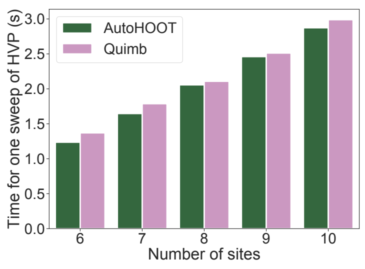
(c) DMRG, $s = R = 40$



(d) CPD, $s = R = 30$



(e) Tucker, $s = 30, R = 10$



(f) DMRG, $s = R = 40$

Figure 4.11: Additional AutoHOOT performance results for kernels in the alternating minimization. (a)(b)(c) Results are collected with NumPy backend and are executed on a single process on a CPU. (d)(e)(f) Results are collected with TensorFlow backend and are executed on an GPU.

Part II

SKETCHING FOR TENSOR DECOMPOSITIONS AND TENSOR NETWORKS

Chapter 5: SKETCHING FOR TENSOR DECOMPOSITIONS

For both CP and Tucker decompositions, each subproblem of alternating minimization is a (constrained) linear least squares problem with the left-hand-side matrix being tall and skinny with a specific tensor network structure. The left-hand-side is a Kronecker product of matrices for Tucker decomposition, and is a Khatri-Rao product of matrices for CP decomposition. In this Chapter, we propose a fast sketching algorithm to accelerate each linear least squares problem for Tucker decomposition, which is then further used to accelerate CP decomposition.

We propose a sketched high-order orthogonal iteration (HOOI) algorithm for low-rank Tucker decomposition of large and sparse tensors [172]. In this algorithm, sketching is directly applied on each *rank-constrained* linear least squares problem $\min_{\mathbf{X},\mathrm{rank}(\mathbf{X})\leq R} \|\mathbf{A}\mathbf{X}-\mathbf{B}\|_F$, with the left-hand-side matrix $\mathbf{A}$ composed of orthonormal columns and having a Kronecker product structure. To the best of our knowledge, the relative error analysis of sketching techniques for this problem has not been discussed in the literature. Existing works either only provide sketch size upper bounds for the relaxed problem [173], where rank constraint is relaxed with a nuclear norm constraint, or provide upper bounds for general constrained problems [60]. We provide tighter sketch size upper bounds to achieve $\mathcal{O}(\epsilon)$ relative error with two state-of-the-art sketching techniques, TensorSketch [174] (a special type of CountSketch [175] with the hash map being restricted to a specific format to allow fast multiplication of the sketching matrix with the chain of Kronecker products) and leverage score sampling [176].

Experimental results show that this new sketched HOOI algorithm, combined with a new initialization scheme based on the randomized range finder, yields decomposition accuracy comparable to the standard higher-order orthogonal iteration (HOOI) algorithm. The new algorithm achieves up to 22.0% relative decomposition residual improvement compared to the state-of-the-art sketched randomized algorithm for Tucker decomposition of various synthetic and real datasets. This sketched HOOI algorithm can be further used to accelerate CP decomposition, by using randomized Tucker compression followed by CP decomposition of the Tucker core tensor.

5.1 BACKGROUND

We introduce the notation used throughout the paper, and briefly review ALS algorithms for Tucker and CP decompositions, and TensorSketch as well as leverage score sampling in this section.

Notation Our analysis makes use of tensor algebra in both element-wise equations and specialized notation for tensor operations [5]. Vectors are denoted with bold lowercase Roman letters (e.g., $\mathbf{v}$), matrices are denoted with bold uppercase Roman letters (e.g., $\mathbf{M}$), and tensors are denoted with bold calligraphic font (e.g., $\boldsymbol{\mathcal{T}}$). An order N tensor corresponds to an N-dimensional array. Elements of vectors, matrices, and tensors are denoted in parentheses, e.g., $\mathbf{v}(i)$ for a vector $\mathbf{v}$, $\mathbf{M}(i,j)$ for a matrix $\mathbf{M}$, and $\boldsymbol{\mathcal{T}}(i,j,k,l)$ for an order 4 tensor $\boldsymbol{\mathcal{T}}$. The ith column of $\mathbf{M}$ is denoted by $\mathbf{M}(:,i)$, and the ith row is denoted by $\mathbf{M}(i,:)$. Parenthesized superscripts are used to label different vectors, matrices and tensors (e.g. $\boldsymbol{\mathcal{T}}^{(1)}$ and $\boldsymbol{\mathcal{T}}^{(2)}$ are unrelated tensors). Number of nonzeros of the tensor $\boldsymbol{\mathcal{T}}$ is denoted by $\text{nnz}(\boldsymbol{\mathcal{T}})$. The pseudo-inverse of matrix $\mathbf{A}$ is denoted with $\mathbf{A}^{\dagger}$. The Hadamard product of two matrices is denoted with $*$. The outer product of two or more vectors is denoted with $\circ$. The Kronecker product of two vectors/matrices is denoted with $\otimes$. For matrices $\mathbf{A} \in \mathbb{R}^{m\times k}$ and $\mathbf{B} \in \mathbb{R}^{n\times k}$, their Khatri-Rao product results in a matrix of size $(mn) \times k$ defined by $\mathbf{A} \odot \mathbf{B} = [\mathbf{A}(:,1) \otimes \mathbf{B}(:,1), \ldots, \mathbf{A}(:,k) \otimes \mathbf{B}(:,k)]$. The mode-$n$ tensor times matrix of an order N tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{s_1\times\cdots\times s_N}$ with a matrix $\mathbf{A} \in \mathbb{R}^{J\times s_n}$ is denoted by $\boldsymbol{\mathcal{T}} \times_n \mathbf{A}$, whose output size is $s_1 \times \cdots \times s_{n-1} \times J \times s_{n+1} \times \cdots \times s_N$. Matricization is the process of unfolding a tensor into a matrix. The mode-n matricized version of $\boldsymbol{\mathcal{T}}$ is denoted by $\mathbf{T}_{(n)} \in \mathbb{R}^{s_n \times K}$ where $K = \prod_{m=1,m\neq n}^{N} s_m$. We use $[N]$ to denote $\{1,\ldots,N\}$. $\widetilde{\mathcal{O}}$ denotes the asymptotic cost with logarithmic factors ignored.

Tucker decomposition with ALS Throughout the analysis we assume the input tensor has order N and size $s \times \cdots \times s$, and the Tucker ranks are $R \times \cdots \times R$. Tucker decomposition approximates a tensor by a core tensor contracted along each mode with matrices that have orthonormal columns. The goal of Tucker decomposition is to minimize the objective function, $f(\boldsymbol{\mathcal{C}}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}) := \frac{1}{2}\left\|\boldsymbol{\mathcal{T}} - \boldsymbol{\mathcal{C}} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)}\right\|_F^2$. The core tensor $\boldsymbol{\mathcal{C}}$ is of order N with dimensions $R \times \cdots \times R$. Each matrix $\mathbf{A}^{(n)} \in \mathbb{R}^{s\times R}$ for $n \in [N]$ has orthonormal columns. The ALS method for Tucker decomposition [5], [29], [30], called the *higher-order orthogonal iteration* (HOOI), proceeds by updating one of the factor matrices along with the core tensor at a time. The nth subproblem can be written as

$$\min_{\boldsymbol{\mathcal{C}},\mathbf{A}^{(n)}} \frac{1}{2}\left\|\mathbf{P}^{(n)}\mathbf{C}_{(n)}^{T}\mathbf{A}^{(n)T} - \mathbf{T}_{(n)}^{T}\right\|_F^2, \tag{5.1}$$

where $\mathbf{P}^{(n)} = \mathbf{A}^{(1)} \otimes \cdots \otimes \mathbf{A}^{(n-1)} \otimes \mathbf{A}^{(n+1)} \otimes \cdots \otimes \mathbf{A}^{(N)}$. This problem can be formulated as a rank-constrained linear least squares problem,

$$\min_{\mathbf{B}^{(n)}} \frac{1}{2}\left\|\mathbf{P}^{(n)}\mathbf{B}^{(n)} - \mathbf{T}_{(n)}^{T}\right\|_F^2, \quad \text{such that} \quad \text{rank}(\mathbf{B}^{(n)}) \leq R. \tag{5.2}$$

$\mathbf{A}^{(n)}$ corresponds to the right singular vectors of the optimal $\mathbf{B}^{(n)}$, while $\mathbf{C}^T_{(n)} = \mathbf{B}^{(n)}\mathbf{A}^{(n)}$. Since $\mathbf{P}^{(n)}$ contains orthonormal columns, the optimal $\mathbf{B}^{(n)}$ can be obtained by calculating the *Tensor Times Matrix-chain* (TTMc),

$$\boldsymbol{\mathcal{Y}}^{(n)} = \boldsymbol{\mathcal{T}} \times_1 \mathbf{A}^{(1)T} \cdots \times_{n-1} \mathbf{A}^{(n-1)T} \times_{n+1} \mathbf{A}^{(n+1)T} \cdots \times_N \mathbf{A}^{(N)T}, \tag{5.3}$$

and taking $\mathbf{B}^{(n)}$ to be the transpose of the mode-n matricized $\boldsymbol{\mathcal{Y}}^{(n)}$, $\mathbf{Y}^{(n)T}_{(n)}$. Calculating $\boldsymbol{\mathcal{Y}}^{(n)}$ costs $\mathcal{O}\left(s^N R\right)$ for dense tensors and $\mathcal{O}\left(\text{nnz}(\boldsymbol{\mathcal{T}})R^{N-1}\right)$ for sparse tensors. Before the HOOI procedure, the factor matrices are often initialized with the *higher-order singular value decomposition* (HOSVD) [10], [49]. HOSVD computes the truncated SVD of each $\mathbf{T}_{(n)} \approx \mathbf{U}^{(n)}\mathbf{\Sigma}^{(n)}\mathbf{V}^{(n)T}$, and sets $\mathbf{A}^{(n)} = \mathbf{U}^{(n)}$ for $n \in [N]$. If performing SVD via randomized SVD [177], updating $\mathbf{A}^{(n)}$ for each mode costs $\mathcal{O}\left(s^N R\right)$ for dense tensors, and costs $\mathcal{O}\left(s^{N-1}R^2 + \text{nnz}(\boldsymbol{\mathcal{T}})R\right)$ for sparse tensors.

CP decomposition with ALS CP tensor decomposition [7], [13] decomposes the input tensor into a sum of outer products of vectors. Throughout analysis we assume the input tensor has order N and size $s \times \cdots \times s$, and the CP rank is R. The goal of CP decomposition is to minimize the objective function, $f(\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}) := \frac{1}{2}\left\|\boldsymbol{\mathcal{T}} - \sum_{r=1}^{R} \mathbf{A}^{(1)}(:, r) \circ \cdots \circ \mathbf{A}^{(N)}(:, r)\right\|_F^2$, where $\mathbf{A}^{(i)} \in \mathbb{R}^{s\times R}$ for $i \in [N]$ are called factor matrices. CP-ALS is the mostly widely used algorithm to get the factor matrices. In each ALS sweep, we solve N subproblems, and the objective for the update of $\mathbf{A}^{(n)}$, with $n \in [N]$, is expressed as,

$$\mathbf{A}^{(n)} = \arg\min_{\mathbf{A}} \frac{1}{2}\left\|\mathbf{P}^{(n)}\mathbf{A}^T - \mathbf{X}^T_{(n)}\right\|_F^2, \tag{5.4}$$

where $\mathbf{P}^{(n)} = \mathbf{A}^{(1)} \odot \cdots \odot \mathbf{A}^{(n-1)} \odot \mathbf{A}^{(n+1)} \odot \cdots \odot \mathbf{A}^{(N)}$. Solving the linear least squares problem above has a cost of $\mathcal{O}\left(s^N R\right)$. For instance, when solving via normal equations the term $\mathbf{P}^{(n)T}\mathbf{X}^T_{(n)}$, which is called MTTKRP, needs to be calculated, and it costs $\mathcal{O}\left(s^N R\right)$ for dense tensors and $\mathcal{O}\left(\text{nnz}(\boldsymbol{\mathcal{T}})R\right)$ for sparse tensors.

A major disadvantage of CP-ALS is its slow convergence. There are many cases where CP-ALS takes a large number of sweeps to converge when high resolution is necessary [118]. When $R < s$, the procedure can be accelerated by performing Tucker-ALS first, which typically converges in fewer sweeps, and then computing a CP decomposition of the core tensor [77], [114], [178], which only has $\mathcal{O}\left(R^N\right)$ elements.

TensorSketch and leverage score sampling In this paper, we sketch the linear least squares problems using both TensorSketch and leverage score sampling. TensorSketch is a special type of CountSketch, where the hash map is restricted to a specific format to allow fast multiplication of the sketching matrix with the chain of Kronecker products. Leverage score sampling picks important rows based on leverage scores to form the sampled/sketched problem. Both algorithms can be efficiently applied to a chain of Kronecker products, and the detailed analysis is presented in Section 5.6.

In the paper, we test two forms of leverage score sampling, random sampling, where we perform importance random sampling based on leverage scores, and deterministic sampling [179], where we deterministically sample rows having the largest leverage scores. Both ideas are also used in [82] for randomized CP decomposition. Papailiopoulos et al. [180] show that if the leverage scores follow a moderately steep power-law decay, then deterministic sampling can be provably as efficient and even better than the random sampling. We compare both leverage score sampling techniques in Section 5.4.

Previous work Randomized algorithms have been applied to both Tucker and CP decompositions in several previous works. For Tucker decomposition, Ahmadi-Asl et al. [181] review a variety of random projection, sampling and sketching based randomized algorithms. Methods introduced in [75]–[78] accelerate the traditional HOSVD/HOOI via random projection, where factor matrices are updated based on performing SVD on the matricization of the randomly projected input tensor. For these methods, random projections are all performed based on Gaussian embedding matrices, and the core tensor is calculated via TTMc among the input tensor and all the factor matrices, which costs $\Omega(\text{nnz}(\boldsymbol{\mathcal{T}})R)$ and is computationally inefficient for large sparse tensors. Sun et al. [182] introduce randomized algorithms for Tucker decompositions for streaming data.

The most similar work to ours is Becker and Malik [79]. This work computes Tucker decomposition via a sketched ALS scheme where in each optimization subproblem, one of the factor matrices or the core tensor is updated. They also solve each sketched linear least squares subproblem via TensorSketch. Our new scheme provides more accurate results compared to this method. Another work that is closely relevant to us is [183]. This work introduces structure-preserving decomposition, which is similar to Tucker decomposition but the factor matrices are not necessary orthogonal, and the entries of the core tensor are explicitly taken from the original tensor. The authors design an algorithm based on rank-revealing QR [184], which is efficient for sparse tensors, to calculate the decomposition. However, their experimental results show that the relative error of the algorithm for sparse tensors is much worse than that of the traditional HOSVD [183].

Several works discuss algorithms for sparse Tucker decomposition. Oh et al. [185] propose PTucker, which provides algorithms for parallel sparse Tucker decomposition. Kaya and

Ucar [104] provide parallel algorithms for sparse Tucker decompositions. Li et al. [186] introduce SGD-Tucker, which uses stochastic gradient descent to perform Tucker decomposition of sparse tensors.

For CP decomposition, Battaglino et al. [80] and Jin et al. [81] introduce a randomized algorithm based on Kronecker fast Johnson-Lindenstrauss Transform (KFJLT) to accelerate CP-ALS. However, KFJLT is effective only for the decomposition of dense tensors. Aggour et al. [187] introduce adaptive sketching for CP decomposition. Song et al. [188] discuss the theoretical relative error of various tensor decompositions based on sketching. The work by Cheng et al. [189] and Larsen and Kolda [82] accelerate CP-ALS based on leverage score sampling. Cheng et al. [189] use leverage score sampling to accelerate MTTKRP calculations. Larsen and Kolda [82] propose an approximate leverage score sampling scheme for the Khatri-Rao product, and they show with $\mathcal{O}\left(R^{(N-1)}\log(1/\delta)/\epsilon^2\right)$ number of samples, each unconstrained linear least squares subproblem in CP-ALS can be solved with $\mathcal{O}\left(\epsilon\right)$-relative error with probability at least $1-\delta$. Zhou et al. [77] and Erichson et al. [190] accelerate CP decomposition via performing randomized Tucker decomposition of the input tensor first, and then performing CP decomposition of the smaller core tensor.

Several other works discuss techniques to parallelize and accelerate the computation of CP-ALS. Ma and Solomonik [93], [94] approximate MTTKRP within CP-ALS based on information from previous sweeps. For sparse tensors, parallelization strategies for MTTKRP have been developed both on shared memory systems [65], [66] and distributed memory systems [67]–[69]. Researchers have also been looking at different alternatives to accelerate the convergence of CP-ALS, including various regularization techniques [119], [120], line search [109], [121], [191], and gradient-based methods [137]–[139], [192], [193].

5.2 SKETCHED RANK-CONSTRAINED LINEAR LEAST SQUARES

Each subproblem of Tucker HOOI solves a linear least squares problem with the following properties,

1. the left-hand-side matrix is a chain of Kronecker products of factor matrices,
2. the left-hand-side matrix has orthonormal columns, since each factor matrix has orthonormal columns,
3. the rank of the output solution is constrained to be less than R, as is shown in (5.2).

To the best of our knowledge, the relative error analysis of sketching techniques for this problem have not been discussed in the literature. In the following two theorems, we will show the sketch sizes of TensorSketch and leverage score sampling that are sufficient for the relative residual norm error of the problems to be bounded by $\mathcal{O}(\epsilon)$ with at least $1-\delta$ probability. The detailed proofs are presented in Section 5.10.

Theorem 5.1 (TensorSketch for Rank-constrained Linear Least Squares)**.** *Consider matrices* $\mathbf{P} = \mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)} \otimes \cdots \otimes \mathbf{A}^{(N-1)}$*, where each* $\mathbf{A}^{(i)} \in \mathbb{R}^{s\times R}$ *has orthonormal columns,* $s > R$*, and* $\mathbf{B} \in \mathbb{R}^{s^{N-1}\times n}$*. Let* $\mathbf{S} \in \mathbb{R}^{m\times s^{N-1}}$ *be an order* $N-1$ *TensorSketch matrix. Let* $\widetilde{\mathbf{X}}_r$ *be the best rank-*R *approximation of the solution of the problem* $\min_{\mathbf{X}} ||\mathbf{SPX}-\mathbf{SB}||_F$*, and let* $\mathbf{X}_r = \arg\min_{\mathbf{X},rank(\mathbf{X})=R} \|\mathbf{PX}-\mathbf{B}\|_F$*. With*

$$m = \mathcal{O}\left((R^{(N-1)}\cdot 3^{N-1})\cdot(R^{(N-1)}+1/\epsilon^2)/\delta\right), \tag{5.5}$$

the approximation,

$$\left\|\mathbf{A}\widetilde{\mathbf{X}}_r-\mathbf{B}\right\|_F^2 \leq (1+\mathcal{O}(\epsilon))\left\|\mathbf{A}\mathbf{X}_r-\mathbf{B}\right\|_F^2, \tag{5.6}$$

holds with probability at least $1-\delta$*.*

Theorem 5.2 (Leverage Score Sampling for Rank-constrained Linear Least Squares)**.** *Given matrices* $\mathbf{P} = \mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)} \otimes \cdots \otimes \mathbf{A}^{(N-1)}$*, where each* $\mathbf{A}^{(i)} \in \mathbb{R}^{s\times R}$ *has orthonormal columns,* $s > R$*, and* $\mathbf{B} \in \mathbb{R}^{s^{N-1}\times n}$*. Let* $\mathbf{S} \in \mathbb{R}^{m\times s^{N-1}}$ *be a leverage score sampling matrix for* $\mathbf{P}$*. Let* $\widetilde{\mathbf{X}}_r$ *be the best rank-*R *approximation of the solution of the problem* $\min_{\mathbf{X}} ||\mathbf{SPX}-\mathbf{SB}||_F$*, and let* $\mathbf{X}_r = \arg\min_{\mathbf{X},rank(\mathbf{X})=R} \|\mathbf{PX}-\mathbf{B}\|_F$*. With* $m = \mathcal{O}\left(R^{(N-1)}/(\epsilon^2\delta)\right)$*, the approximation,*

$$\left\|\mathbf{A}\widetilde{\mathbf{X}}_r-\mathbf{B}\right\|_F^2 \leq (1+\mathcal{O}(\epsilon))\left\|\mathbf{A}\mathbf{X}_r-\mathbf{B}\right\|_F^2, \tag{5.7}$$

holds with probability at least $1-\delta$*.*

Therefore, for leverage score sampling, $\mathcal{O}\left(R^{(N-1)}/(\epsilon^2\delta)\right)$ number of samples are sufficient to get $(1+\mathcal{O}(\epsilon))$-accurate residual with probability at least $1-\delta$. The sketch size upper bound for TensorSketch is higher than that for leverage score sampling, suggesting that leverage score sampling is better. As can be seen in (5.5), when $R^{N-1} \leq 1/\epsilon^2$, the sketch size bound for TensorSketch is $\mathcal{O}\left(3^{N-1}\right)$ times that for leverage score sampling. When $R^{N-1} > 1/\epsilon^2$, the ratio is even higher. The accuracy comparison of the two methods is discussed further in Section 5.4.

While TensorSketch has a worse upper bound compared to leverage score sampling, it is more flexible since the sketching matrix is independent of the left-hand-side matrix. One can derive a sketch size bound that is sufficient to get $(1+\mathcal{O}(\epsilon))$-accurate residual norm for linear least squares with general (not necessarily rank-based) constraints based on existing proof techniques (detailed in Section 5.11). Although that bound is applicable for general constraints, it is looser than (5.5). For leverage score sampling, we do not provide a sample size bound for general constrained linear least squares.

Sketching method	Rank-constrained least squares	Unconstrained least squares
Leverage score sampling	$\mathcal{O}\left(R^{(N-1)}/(\epsilon^2\delta)\right)$ (Theorem 5.2)	$\mathcal{O}\left(R^{(N-1)}/(\epsilon\delta)\right)$ (Theorem 5.6) or $\mathcal{O}\left(R^{(N-1)}\log(1/\delta)/\epsilon^2\right)$ [82]
TensorSketch	$\mathcal{O}\left((3R)^{(N-1)}\cdot(R^{(N-1)}+1/\epsilon^2)/\delta\right)$ (Theorem 5.1)	$\mathcal{O}\left((3R)^{(N-1)}\cdot(R^{(N-1)}+1/\epsilon)/\delta\right)$ (Theorem 5.5)

Table 5.1: Comparison of sketch size upper bounds for rank-constrained linear least squares and unconstrained linear least squares. The upper bounds are sufficient for the relative residual norm error to be bounded by $\mathcal{O}(\epsilon)$ with at least $1-\delta$ probability.

We also compare the sketch size upper bounds for rank-constrained linear least squares and unconstrained linear least squares in Table 5.1. For both leverage score sampling and TensorSketch, the upper bounds for rank-constrained problems are at most $\mathcal{O}(1/\epsilon)$ times the upper bounds for unconstrained linear least squares problem. The error of sketched rank-constrained solution consists of two parts, the error of the sketched unconstrained linear least squares solution, and the error from low-rank approximation of the unconstrained solution. To make sure the second error term has a relative error bound of $\mathcal{O}(\epsilon)$, we restrict the first error term to be relatively bounded by $\mathcal{O}(\epsilon^2)$, incurring a larger sketch size upper bound.

5.3 MAIN ALGORITHM

Our main algorithm is presented in Algorithm 5.1. To improve the robustness of leverage score sampling, we use an initialization scheme that uses the randomized range finder (RRF) [177] to initialize the factor matrices (lines 3-5). In this scheme, the composition of CountSketch and Gaussian random matrix is used as the RRF embedding matrix, which only requires one pass over the non-zero elements of the input tensor. The detailed initialization algorithm and its cost analysis is detailed in Section 5.7.

We provide detailed cost analysis for Algorithm 5.1. Note that for leverage score sampling, lines 8 and 9 need to be recalculated for every sweep, since $\mathbf{S}^{(n)}$ is dependent on the factor matrices. On the other hand, the TensorSketch embedding is oblivious to the state of the factor matrices, so we can choose to use the same $\mathbf{S}^{(n)}$ for all the sweeps for each mode n to save cost. This strategy is also used in [79]. Detailed cost analysis for each part of

Algorithm 5.1: Sketch-Tucker-ALS: Sketched ALS procedure for Tucker decomposition

```
1: Input: Input tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{s_1\times\cdots\times s_N}$, Tucker ranks $\{R_1,\ldots,R_N\}$, maximum number of
   sweeps $I_{\max}$, sketching tolerance $\epsilon$
2: $\boldsymbol{\mathcal{C}} \leftarrow \boldsymbol{\mathcal{O}}$
3: for $n \in \{2,\ldots,N\}$ do
4:     $\mathbf{A}^{(n)} \leftarrow$ Init-RRF$(\mathbf{T}_{(n)}, R_n, \epsilon)$
5: end for
6: for $i \in \{1,\ldots,I_{\max}\}$ do
7:     for $n \in \{1,\ldots,N\}$ do
8:         Build the sketching matrix $\mathbf{S}^{(n)}$
9:         $\mathbf{Y} \leftarrow \mathbf{S}^{(n)}\mathbf{T}_{(n)}$
10:        $\mathbf{Z} \leftarrow \mathbf{S}^{(n)}(\mathbf{A}^{(1)} \otimes \cdots \otimes \mathbf{A}^{(n-1)} \otimes \mathbf{A}^{(n+1)} \otimes \cdots \otimes \mathbf{A}^{(N)})$
11:        $\mathbf{C}_{(n)}^T, \mathbf{A}^{(n)} \leftarrow$ RSVD-LRLS$(\mathbf{Z}, \mathbf{Y}, R)$
12:    end for
13: end for
14: Return: $\{\boldsymbol{\mathcal{C}}, \mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)}\}$
```

Algorithm for Tucker	LS subproblem cost	Sketch size (m)	Prep cost
ALS	$\Omega(\text{nnz}(\boldsymbol{\mathcal{T}})R)$	/	/
ALS+TensorSketch [79]	$\widetilde{O}\left(msR + mR^N\right)$	$\mathcal{O}\left((3R)^{N-1}/\delta \cdot (R^{N-1} + 1/\epsilon)\right)$	$\mathcal{O}\left(N\,\text{nnz}(\boldsymbol{\mathcal{T}})\right)$
ALS+TTMTS [79]	$\widetilde{O}\left(msR^{N-1}\right)$	Not shown	$\mathcal{O}\left(N\,\text{nnz}(\boldsymbol{\mathcal{T}})\right)$
<u>ALS + TensorSketch</u>	$\mathcal{O}\left(msR + mR^{2(N-1)}\right)$	$\mathcal{O}\left((3R)^{N-1}/\delta \cdot (R^{N-1} + 1/\epsilon^2)\right)$ (Theorem 5.1)	$\mathcal{O}\left(N\,\text{nnz}(\boldsymbol{\mathcal{T}})\right)$
<u>ALS+leverage scores</u>	$\mathcal{O}\left(msR + mR^{2(N-1)}\right)$	$\mathcal{O}\left(R^{N-1}/(\epsilon^2\delta)\right)$ (Theorem 5.2)	/

Table 5.2: Comparison of algorithm complexity between Tucker-ALS (HOOI), ALS with the TensorSketch/leverage score sampling, and the sketched Tucker-ALS algorithms introduced in [79]. The third column shows the sketch size sufficient for the sketched linear least squares to be $(1 + \mathcal{O}(\epsilon))$ accurate with probability at least $1 - \delta$. Underlined algorithms are our new contributions.

Algorithm 5.1 is listed below, where we assume $s_1 = \cdots = s_N = s$ and $R_1 = \cdots = R_N = R$.

- Line 3-5: the cost is $\mathcal{O}\left(N\,\text{nnz}(\boldsymbol{\mathcal{T}}) + NsR^3/\epsilon\right)$ by the analysis in Section 5.7.

- Line 8: if using leverage score sampling, the cost is $\mathcal{O}(sR)$ per subproblem (for computing the leverage scores of the previously updated $\mathbf{A}^{(i)}$). If using TensorSketch, the cost is $\mathcal{O}(Ns)$, which is only incurred for the first sweep.

- Line 9: if using leverage score sampling, the cost is $\mathcal{O}(ms)$ per subproblem; if using TensorSketch, the cost is $\mathcal{O}\left(N\,\text{nnz}(\boldsymbol{\mathcal{T}})\right)$, and is only incurred for the first sweep.

Algorithm 5.2: RSVD-LRLS: Low-rank approximation of least squares solution via randomized SVD

```
1: Input: Left-hand-side matrix $\mathbf{Z} \in \mathbb{R}^{m\times r}$, right-hand-side matrix $\mathbf{Y} \in \mathbb{R}^{m\times s}$, rank $R$
2: Initialize $\mathbf{S} \in \mathbb{R}^{s\times \mathcal{O}(R)}$ as a random Gaussian sketching matrix
3: $\mathbf{B} \leftarrow (\mathbf{Z}^T\mathbf{Z})^{-1}$
4: $\mathbf{C} \leftarrow \mathbf{B}\mathbf{Z}^T\mathbf{Y}\mathbf{S}$
5: $\mathbf{Q}, \mathbf{R} \leftarrow \texttt{qr}(\mathbf{C})$
6: $\mathbf{D} \leftarrow \mathbf{Q}^T\mathbf{B}\mathbf{Z}^T\mathbf{Y}$
7: $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V} \leftarrow \texttt{svd}(\mathbf{D})$
8: Return: $\mathbf{Q}\mathbf{U}(:,: R)\mathbf{\Sigma}(: R,: R), \mathbf{V}(:,: R)$
```

- Line 10: if using leverage score sampling, the cost is $\mathcal{O}\left(mR^{N-1}\right)$ per subproblem; if using TensorSketch, the cost is $\mathcal{O}\left(NsR + m\log(m)R^{N-1}\right)$ per subproblem, as analyzed in Section 5.6.

- Line 11: the cost is $\mathcal{O}\left(msR + mR^{2(N-1)}\right)$ per subproblem, under the condition that $m \geq R^{N-1}$ and using randomized SVD as detailed in Algorithm 5.2.

Therefore, the cost for each subproblem (lines 8-11) is $\mathcal{O}\left(msR + mR^{2(N-1)}\right)$, for both leverage score sampling and TensorSketch. For TensorSketch, another cost of $\mathcal{O}\left(N\,\text{nnz}(\boldsymbol{\mathcal{T}})\right)$ is incurred at the first sweep to sketch the right-hand-side matrix, which we refer to as preparation cost. Using the initialization scheme based on RRF to initialize the factor matrices would increase the cost of both sketching techniques by $\mathcal{O}\left(N\,\text{nnz}(\boldsymbol{\mathcal{T}}) + NsR^3/\epsilon\right)$.

We compare the cost of each linear least squares subproblem between our sketched ALS algorithms with both HOOI and the sketched ALS algorithms introduced in [79] in Table 5.2. For the ALS + TensorSketch algorithm in [79], $N+1$ subproblems are solved in each sweep, and in each subproblem either one factor matrix or the core tensor is updated based on the sketched *unconstrained* linear least squares solutions. For the ALS + TTMTS algorithm, TensorSketch is simply used to accelerate the TTMc operations, and it has been shown to be less accurate compared to the reference ALS + TensorSketch algorithm in [79].

For the solutions of sketched linear least squares problems to be unique, we need $m \geq R^{N-1}$ and hence $m = \Omega(R^{N-1})$. With this condition, the cost of each linear least squares subproblem of our sketched ALS algorithms is less than that for ALS + TTMTS, but is more expensive with related to R compared to the ALS + TensorSketch algorithm in [79], since our cost involves a term $mR^{2(N-1)}$. However, as will be discussed in Section 5.9.1, this term does not dominate in the low-rank decomposition regime. In addition, as shown in Section 5.4, our algorithms provide better accuracy as a result of updating more variables at a time. We also show the sketch size upper bound sufficient to get a $(1 + \mathcal{O}\left(\epsilon\right))$-accurate approximation in residual norm. As can be seen in the table, our sketching algorithm with leverage score sampling has the smallest sketch size, making it the best algorithm considering both the cost of each subproblem and the sketch size. In [79], the authors give an error bound for the approximate matrix multiplication in ALS + TTMTS, but the relative error of the overall linear least squares problem is not given. For the ALS + TensorSketch algorithm in [79], the sketch size upper bound in Table 5.2 comes from the upper bound for the unconstrained linear least squares problem.

Note that the analysis generalizes to the case with non-uniform input tensor dimensions and Tucker ranks. For the decomposition of an order N tensor with dimensions $s_1 \times \cdots \times s_N$ and the Tucker ranks $R_1 \times \cdots \times R_N$, the least squares subproblem cost for the ith mode for both ALS with TensorSketch and ALS with leverage score sampling generalize from $O(msR + mR^{2(N-1)})$ (shown in Table 5.2) to $O(ms_iR_i + m\prod_{j=1,j\neq i}^{N} R_j^2)$. For ALS with leverage score sampling, the sketch size bound changes to $O(\prod_{j=1,j\neq i}^{N} R_j/(\epsilon^2\delta))$. For ALS with TensorSketch, the sketch size bound changes to $O\Big(3^{N-1}\prod_{j=1,j\neq i}^{N} R_j \cdot (\prod_{j=1,j\neq i}^{N} R_j + 1/\epsilon^2)/\delta\Big)$.

Algorithm 5.1 can also be used to accelerate CP decomposition when $R \ll s$. Tucker compression is performed first, and then CP decomposition is applied to the core tensor. The detailed algorithm and the cost analysis is presented in Section 5.8.

5.4 EXPERIMENTS

In this section, we compare our randomized algorithms with reference algorithms for both Tucker and CP decompositions on several synthetic and real tensors. We evaluate accuracy based on the final fitness f for each algorithm, defined as $f = 1 - \frac{\|\mathcal{T}-\widetilde{\mathcal{T}}\|_F}{\|\mathcal{T}\|_F}$, where $\mathcal{T}$ is the input tensor and $\widetilde{\mathcal{T}}$ is the reconstructed low-rank tensor. For Tucker decomposition, we focus on the comparison of accuracy and robustness of attained fitness across various synthetic datasets for different algorithms. For CP decomposition, we focus on the comparison of accuracy and sweep count. Our experiments are carried out on an Intel Core i7 2.9 GHz Quad-Core machine using NumPy [194] routines in Python.

5.4.1 Experiments for Tucker Decomposition

We compare five different algorithms for Tucker decomposition. Two baselines from previous work are considered, standard HOOI and the original TensorSketch-based randomized Tucker-ALS algorithm, which optimizes only one factor in Tucker decomposition at a time [79]. We compare these to our new randomized algorithm (Algorithm 5.1) based on

TensorSketch, random leverage score sampling, and deterministic leverage score sampling. For each randomized algorithm, we test both random initialization for factor matrices as well as the initialization scheme based on RRF. For the baseline HOOI algorithm, we report the performance with both random and HOSVD initializations. We use the following four synthetic tensors and real datasets to evaluate these algorithms.

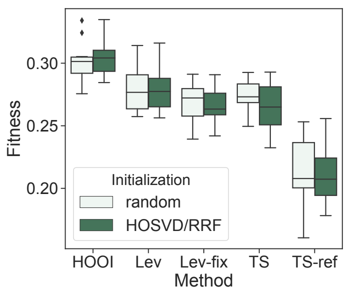
1. **Dense tensors with specific Tucker rank**. We create order 3 tensors based on randomly-generated factor matrices $\mathbf{B}^{(n)} \in \mathbb{R}^{s\times R_{\text{true}}}$ and a core tensor $\boldsymbol{\mathcal{C}}$,
$$\boldsymbol{\mathcal{T}} = \boldsymbol{\mathcal{C}} \times_1 \mathbf{B}^{(1)} \times_2 \mathbf{B}^{(2)} \times_3 \mathbf{B}^{(3)}. \tag{5.8}$$
Each element in the core tensor and the factor matrices are i.i.d. normally distributed random variables $\mathcal{N}(0,1)$. The ratio R_{true}/R, where R is the decomposition rank, is denoted as α.

2. **Dense tensors with strong low-rank signal**. We also test on dense tensors with strong low-rank signal, $\boldsymbol{\mathcal{T}}^{(b)} = \boldsymbol{\mathcal{T}} + \sum_{i=1}^{n} \lambda_i \mathbf{a}_i^{(1)} \circ \mathbf{a}_i^{(2)} \circ \mathbf{a}_i^{(3)}$. $\boldsymbol{\mathcal{T}}$ is generated based on (5.8), and each vector $\mathbf{a}_i^{(j)}$ has unit 2-norm. The magnitudes λ_i for $i \in [n]$ are constructed based on the power-law distribution, $\lambda_i = C\frac{\|\boldsymbol{\mathcal{T}}\|_F}{i^{1+\eta}}$. In our experiments, we set $n = 5, C = 3$ and $\eta = 0.5$. This tensor is used to model data whose leading low-rank components obey the power-law distribution, which is common in real datasets.

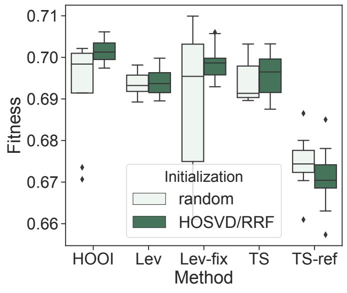
3. **Tensors with large coherence**. We also test on tensors with large coherence, $\boldsymbol{\mathcal{T}}^{(b)} = \boldsymbol{\mathcal{T}} + \boldsymbol{\mathcal{N}}$. $\boldsymbol{\mathcal{T}}$ is generated based on (5.8), and $\boldsymbol{\mathcal{N}}$ contains $n \ll s$ elements with random positions and same large magnitude. In our experiments, we set $n = 10$, and each nonzero element in $\boldsymbol{\mathcal{N}}$ has the i.i.d. normal distribution $\mathcal{N}(\|\boldsymbol{\mathcal{T}}\|_F/\sqrt{n}, 1)$, which means the expected norm ratio $\mathbb{E}[\|\boldsymbol{\mathcal{N}}\|_F/\|\boldsymbol{\mathcal{T}}\|_F] = 1$. This tensor has large coherence and is used to test the robustness of sketching techniques.

4. **Real image datasets**. We test on two image datasets, COIL-100 [111] and a Time-Lapse hyperspectral radiance images dataset called “Souto wood pile” [112], both have been used previously as a tensor decomposition benchmark [77], [80], [93]. Transferring the data into tensor format results in a tensor of size $128 \times 128 \times 7200$ for COIL-100, and $1024 \times 1344 \times 33$ for the Time-Lapse dataset.

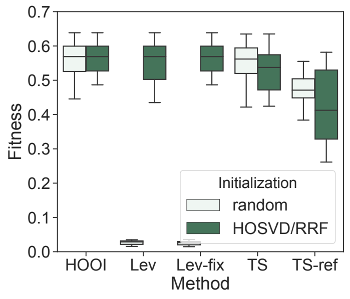
For all the experiments, we run 5 ALS sweeps unless otherwise specified, and calculate the fitness based on the output factor matrices as well as the core tensor. We observe that 5 sweeps are sufficient for both HOOI and randomized algorithms to converge. For each randomized algorithm, we set the sketch size to be KR^2. The constant factor K reveals the accuracy of each subproblem. For the randomized SVD routine in Algorithm 5.2, we set the dimension sizes of the random matrix $\mathbf{S}$ as $s \times (R+5)$, where the oversampling size is 5. We find that this yields accurate randomized SVD solutions. Let $\mathbf{C}_{(n)}^T, \mathbf{A}^{(n)}$ be the output of Line 11, Algorithm 5.1 via calling accurate SVD, and let $\hat{\mathbf{C}}_{(n)}^T, \hat{\mathbf{A}}^{(n)}$ be the output via calling randomized SVD. We observe that the error $||\mathbf{C}_{(n)}^T\mathbf{A}^{(n)} - \hat{\mathbf{C}}_{(n)}^T\hat{\mathbf{A}}^{(n)}||_F$ is always smaller than 10^{-10} for all experiments.



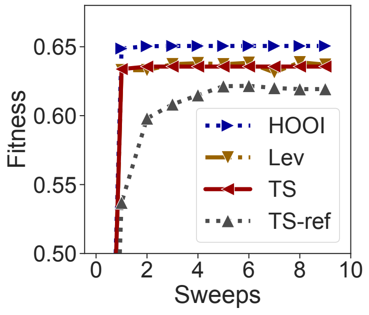
(a) Tensor 1 with $s = 200$



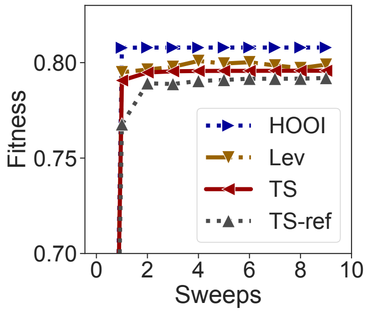
(b) Tensor 2 with $s = 200$



(c) Tensor 3 with $s = 1000$



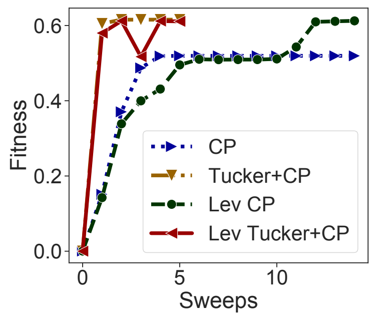
(d) COIL100 dataset



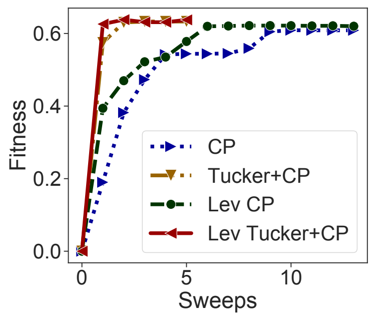
(e) Time-Lapse dataset

Figure 5.1: Experimental results for Tucker decomposition. For all experiments, the Tucker rank is $5 \times 5 \times 5$ and the sketch size parameter $K = 16$. For synthetic tensors, we set $\alpha = 1.6$. HOSVD/RRF means HOOI is initialized with HOSVD, and all other methods are initialized with RRF. Lev, Lev-fix, TS denote our new sketched Tucker-ALS scheme with leverage score random sampling, leverage score deterministic sampling, and TensorSketch, respectively. TS-ref denotes the reference ALS-TensorSketch algorithm [79]. **(a)(b)(c)** Box plots of the final fitness for each algorithm with different synthetic tensors. Each box is based on 10 experiments with different random seeds. Each box shows the 25th-75th quartiles, the median is indicated by a horizontal line inside the box, and outliers are displayed as dots. **(d)(e)** Detailed fitness-sweeps relation for real image datasets. HOOI is initialized with HOSVD, and all other methods are initialized with RRF.

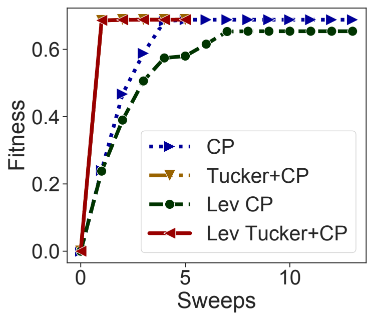
We show the experimental results in Fig. 5.1. As can be seen in the figure, our new randomized ALS scheme, with either leverage score sampling or TensorSketch, outperforms the reference randomized algorithm for all the synthetic and real tensors. The relative fitness



(a) $p = 0.5$



(b) $p = 0.1$



(c) $p = 0.02$

Figure 5.2: Detailed fitness-sweeps relation for CP decomposition of three tensors with different parameters. For all the experiments, we set $s = 2000, R = 10, \alpha = 1.2$ and $K = 16$. Markers represent the results per sweep. For Tucker + CP algorithms, the fitness shown for ith sweep is the output fitness after running i Tucker sweeps along with 25 CP-ALS sweeps on core tensors afterwards.

improvement ranges from 4.5% (Fig. 5.1b,5.3b) to 22.0% (Fig. 5.1a,5.3a) when $K = 16$ for synthetic tensors. With our new randomized scheme, the relative final fitness difference between HOOI and the randomized algorithms is less than 8.5% when $K = 16$, indicating the efficacy of our new scheme.

Fig. 5.1a,5.1b,5.1c include a comparison between random initialization and the initialization scheme based on RRF detailed in Algorithm 5.3. For Tensor 1, both initialization schemes have similar performance. For the deterministic leverage score sampling on Tensor 2 (Fig. 5.1b), using RRF-based initialization substantially decreases variability of attained accuracy. For leverage score sampling on Tensor 3 (Fig. 5.1c), we observe that the random initialization is not effective, resulting in approximately zero final fitness. This is because the random initializations are far from the accurate solutions, and some elements with large amplitudes are not sampled in all the ALS sweeps. With the RRF-based initialization, the output fitness of the algorithms based on leverage score sampling is close to HOOI. Therefore, our proposed initialization scheme is important for improving the robustness of leverage score sampling.

We present additional experiments on dense synthetic tensors in Section 5.9.1, where we show the computational cost comparison of different algorithms, the relation between the sketch size and the least squares subproblem accuracy, as well as the perturbation of factor matrices for each randomized algorithm relative to the baseline HOOI.

Although the analysis in Section 5.2 shows leverage score sampling has a better sketch size upper bound, the random leverage score sampling scheme performs similar to TensorSketch for the tested dense tensors. In Section 5.9.1 and Section 5.9.2, we provide additional experimental results on sparse tensors, and results with other sketch sizes. Results show that for multiple sparse tensors and several experiments with smaller sketch sizes, leverage score sampling performs better than TensorSketch.

5.4.2 Experiments for CP Decomposition

We show the efficacy of accelerating CP decomposition via performing Tucker compression first. We compare four different algorithms, the standard CP-ALS algorithm, the Tucker HOOI + CP-ALS algorithm, sketched CP-ALS, where the sketching matrix is applied to each linear least squares subproblem (5.4), as well as the sketched Tucker-ALS + CP-ALS algorithm, where Tucker compression is performed first, and then CP decomposition is applied to the core tensor. Random leverage score sampling is used for sketching, since it has been shown to be efficient for both Tucker (Section 5.4.1) and CP (reference [82]) decompositions. We use the synthetic tensor to evaluate these four algorithms, $\boldsymbol{\mathcal{T}} = \sum_{i=1}^{R_{\text{true}}} \mathbf{a}_i^{(1)} \circ \mathbf{a}_i^{(2)} \circ \mathbf{a}_i^{(3)}$, where each element in $\mathbf{a}_i^{(j)}$ is an i.i.d normally distributed random variable $\mathcal{N}(0,1)$ with probability p and is zero otherwise. This guarantees that the expected sparsity of $\boldsymbol{\mathcal{T}}$ is lower-bounded by $1 - R_{\text{true}}p^3$. The ratio R_{true}/R, where R is the decomposition rank, is denoted as α.

We show the detailed fitness-sweeps relation in Fig. 5.2. The detailed experimental set-up and additional results with different parameter α are presented in Section 5.9.3. We observe that for (sketched) CP-ALS, more than 10 sweeps are necessary for the algorithms to converge. On the contrary, less than 5 Tucker-ALS sweeps are needed for the sketched Tucker + CP scheme, making it more efficient. In summary, we observe CP decomposition can be accurately calculated with fewer passes over the tensor data based on the sketched Tucker + CP method.

5.5 CONCLUSIONS

In this work, we propose a fast and accurate sketching based ALS algorithm for Tucker decomposition, which consists of a sequence of sketched rank-constrained linear least squares subproblems. Theoretical sketch size upper bounds are provided to achieve $\mathcal{O}(\epsilon)$-relative residual norm error for each subproblem with two sketching techniques, TensorSketch and leverage score sampling. For both techniques, our bounds are at most $\mathcal{O}(1/\epsilon)$ times the sketch size upper bounds for the unconstrained linear least squares problem. In particular, the TensorSketch-based sketching algorithm only requires one pass over the input tensor and can be effective in the streaming setting [195].

We also propose an initialization scheme based on randomized range finder to improve the accuracy of leverage score sampling based randomized Tucker decomposition of tensors with high coherence. Experimental results show that this new ALS algorithm is more accurate than the existing sketching based randomized algorithm for Tucker decomposition. This Tucker decomposition algorithm also yields an efficient CP decomposition method, where randomized Tucker compression is performed first, and CP decomposition is applied to the Tucker core tensor afterwards. Experimental results show this algorithm not only converges faster, but also yields more accurate CP decompositions.

We leave high-performance implementation of our sketched ALS algorithm as well as testing its performance on large-scale real sparse datasets for future work. Additionally, although our theoretical analysis shows a much tighter sketch size upper bound for leverage score sampling compared to TensorSketch, their experimental performance under the same sketch size for multiple tensors are similar. Therefore, it will be of interest to investigate potential improvements to sketch size bounds for TensorSketch.

5.6 BACKGROUND ON SKETCHING

Throughout the paper we consider the linear least squares problem,

$$\min_{\mathbf{X}\in\mathcal{C}} \frac{1}{2} \|\mathbf{PX}-\mathbf{B}\|_F^2 , \tag{5.9}$$

where $\mathbf{P} = \mathbf{A}^{(1)} \otimes \cdots \otimes \mathbf{A}^{(N)} \in \mathbb{R}^{s^N \times R^N}$ is a chain of Kronecker products, $N \geq 2$, $\mathbf{P}$ is dense and $\mathbf{B}$ is sparse. In each subproblem of Tucker HOOI, the feasible region $\mathcal{C}$ contains matrices with the rank constraint, as is shown in (5.2). The associated sketched problem is

$$\min_{\mathbf{X}\in\mathcal{C}} \frac{1}{2} \|\mathbf{SPX}-\mathbf{SB}\|_F^2 , \tag{5.10}$$

where $\mathbf{S} \in \mathbb{R}^{m\times s^N}$ is the sketching matrix with $m \ll s^N$. We refer to m as the sketch size throughout the paper.

The Kronecker product structure of $\mathbf{P}$ prevents efficient application of widely-used sketching matrices, including Gaussian matrices and CountSketch matrices. For these sketching matrices, the computation of $\mathbf{SP}$ requires forming $\mathbf{P}$ explicitly, which has a cost of $\mathcal{O}\left(s^N R^N\right)$. We consider two sketching techniques, TensorSketch and leverage score sampling, that are efficient for the problem. With these two sketching techniques, $\mathbf{SP}$ can be calculated without explicitly forming $\mathbf{P}$, and $\mathbf{SB}$ can be calculated efficiently as well (with a cost of $\mathcal{O}\left(\text{nnz}(\mathbf{B})\right)$).

5.6.1 TensorSketch

TensorSketch is a special type of CountSketch, where the hash map is restricted to a specific format to allow fast multiplication of the sketching matrix with the chain of Kronecker products. We introduce the definition of CountSketch and TensorSketch below.

Definition 5.1 (CountSketch)**.** *The CountSketch matrix is defined as* $\mathbf{S} = \mathbf{\Omega}\mathbf{D} \in \mathbb{R}^{m\times n}$*, where*

- $h : [n] \to [m]$ *is a hash map such that* $\forall i \in [n]$ *and* $\forall j \in [m]$*,* $\Pr[h(i) = j] = 1/m$*,*
- $\mathbf{\Omega} \in \mathbb{R}^{m\times n}$ *is a matrix with* $\mathbf{\Omega}(j,i) = 1$ *if* $j = h(i)$ $\forall i \in [n]$ *and* $\mathbf{\Omega}(j,i) = 0$ *otherwise,*
- $\mathbf{D} \in \mathbb{R}^{n\times n}$ *is a diagonal matrix whose diagonal is a Rademacher vector (each entry is* $+1$ *or* -1 *with equal probability).*

Definition 5.2 (TensorSketch [174])**.** *The order* N *TensorSketch matrix* $\mathbf{S} = \mathbf{\Omega}\mathbf{D} \in \mathbb{R}^{m\times \prod_{i=1}^{N} s_i}$ *is defined based on two hash maps* H *and* S *defined below,*

$$H : [s_1]\times[s_2]\times\cdots\times[s_N] \to [m] : (i_1,\ldots,i_N) \mapsto \left(\sum_{n=1}^{N}(H_n(i_n)-1) \mod m\right) + 1, \tag{5.11}$$

$$S : [s_1]\times[s_2]\times\cdots\times[s_N] \to \{-1,1\} : (i_1,\ldots,i_N) \mapsto \prod_{n=1}^{N} S_n(i_n), \tag{5.12}$$

where each H_n *for* $n \in [N]$ *is a 3-wise independent hash map that maps* $[s_n] \to [m]$*, and each* S_n *is a 4-wise independent hash map that maps* $[s_n] \to \{-1,1\}$*. A hash map is k-wise independent if any designated k keys are independent random variables. Two matrices* $\mathbf{\Omega}$ *and* $\mathbf{D}$ *are defined based on* H *and* S*, respectively,*

- $\mathbf{\Omega} \in \mathbb{R}^{m\times \prod_{i=1}^{N} s_i}$ *is a matrix with* $\mathbf{\Omega}(j,i) = 1$ *if* $j = H(i)$ $\forall i \in \left[\prod_{i=1}^{N} s_i\right]$*, and* $\mathbf{\Omega}(j,i) = 0$ *otherwise,*
- $\mathbf{D} \in \mathbb{R}^{n\times n}$ *is a diagonal matrix with* $\mathbf{D}(i,i) = S(i)$*.*

Above we use the notation $S(i) = S(i_1,\ldots,i_N)$ *where* $i = i_1 + \sum_{k=2}^{N}\left(\prod_{\ell=1}^{k-1} s_l\right)(i_k - 1)$*, and similar for* H*.*

The restricted hash maps (5.11),(5.12) used in $\mathbf{S}$ make it efficient to multiply with a chain of Kronecker products. Define $\mathbf{S}^{(n)} := \mathbf{\Omega}^{(n)}\mathbf{D}^{(n)} \in \mathbb{R}^{m\times s_n}$, where $\mathbf{\Omega}^{(n)} \in \mathbb{R}^{m\times s_n}$ is defined based on H_n and $\mathbf{D}^{(n)} \in \mathbb{R}^{s_n \times s_n}$ defined based on S_n, and let $\mathbf{P} = \mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)} \otimes \cdots \otimes \mathbf{A}^{(N)}$ with $\mathbf{A}^{(n)} \in \mathbb{R}^{s_n \times R_n}$ for $n \in [N]$,

$$\mathbf{SP} = \mathrm{FFT}^{-1}\left(\left(\bigodot_{n=1}^{N} \left(\mathrm{FFT}\left(\mathbf{S}^{(n)}\mathbf{A}^{(n)}\right)\right)^T\right)^T\right). \tag{5.13}$$

Calculating each $\mathrm{FFT}\left(\mathbf{S}^{(n)}\mathbf{A}^{(n)}\right)$ costs $\mathcal{O}\left(s_n R_n + m \log m R_n\right)$, and performing the Kronecker product as well as the outer FFT costs $\mathcal{O}\left(m \log m \prod_{n=1}^{N} R_n\right)$. When each $s_n = s$ and $R_n = R$, the overall cost is $\mathcal{O}\left(NsR + m \log m R^N\right)$.

5.6.2 Leverage Score Sampling

Leverage score sampling is a useful tool to pick important rows to form the sampled/sketched linear least squares problem. Intuitively, let $\mathbf{Q}_P$ be an orthogonal basis for the column space of $\mathbf{P}$. Then large-norm rows of $\mathbf{Q}_P$ suggest large contribution to $\mathbf{Q}_P^T\mathbf{B}$, which is part of the linear least squares right-hand-side we can solve for.

Definition 5.3 (Leverage Scores [176], [196]). *Let $\mathbf{P} \in \mathbb{R}^{s\times R}$ with $s > R$, and let $\mathbf{Q} \in \mathbb{R}^{s \times R}$ be any orthogonal basis for the column space of $\mathbf{P}$. The* leverage scores *of the rows of $\mathbf{P}$ are given by*

$$\ell_i(\mathbf{P}) := (\mathbf{Q}\mathbf{Q}^T)(i,i) = \|\mathbf{Q}(i,:)\|_2^2 \quad \textit{for all} \quad i \in [s]. \tag{5.14}$$

Definition 5.4 (Importance Sampling based on Leverage Scores). *Let $\mathbf{P} \in \mathbb{R}^{s\times R}$ be a full-rank matrix and $s > R$. The leverage score sampling matrix of $\mathbf{P}$ is defined as $\mathbf{S} = \mathbf{D}\boldsymbol{\Omega}$, where $\boldsymbol{\Omega} \in \mathbb{R}^{m\times s}$, $m < s$ is the sampling matrix, and $\mathbf{D} \in \mathbb{R}^{m\times m}$ is the rescaling matrix. For each row $j \in [m]$ of $\boldsymbol{\Omega}$, one column index $i \in [s]$ is picked independently with replacement with probability $p_i = \ell_i(\mathbf{P})/R$, and we set $\boldsymbol{\Omega}(j,i) = 1, \mathbf{D}(j,j) = \frac{1}{\sqrt{mp_i}}$. Other elements of $\boldsymbol{\Omega}, \mathbf{D}$ are 0.*

To calculate the leverage scores of $\mathbf{P}$, one can get the matrix $\mathbf{Q}$ via QR decomposition, and the scores can be retrieved via calculating the norm of each row of $\mathbf{Q}$. However, performing QR decomposition of $\mathbf{P}$ is almost as costly as solving the linear least squares problem. The lemma below shows that leverage scores of $\mathbf{P}$ can be efficiently calculated from smaller QR decompositions of the Kronecker product factors composing $\mathbf{P}$.

Lemma 5.1 (Leverage Scores for Kronecker product [189]). *Let $\mathbf{P} = \mathbf{A}^{(1)} \otimes \cdots \otimes \mathbf{A}^{(N)} \in$*

$\mathbb{R}^{s^N \times R^N}$, *where* $\mathbf{A}^{(i)} \in \mathbb{R}^{s \times R}$ *and* $s > R$. *Leverage scores of* $\mathbf{P}$ *satisfy*

$$\ell_i(\mathbf{P}) = \prod_{k=1}^{N} \ell_{i_k}(\mathbf{A}^{(k)}), \quad \textit{where } i = 1 + \sum_{k=1}^{N} (i_k - 1) s^{k-1}. \tag{5.15}$$

Proof. To show (5.15), we only need to show the case when $N = 2$, since it can then be easily generalized to arbitrary N. Consider the reduced QR decomposition of $\mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)}$,

$$\mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)} = \mathbf{Q}^{(1)}\mathbf{R}^{(1)} \otimes \mathbf{Q}^{(2)}\mathbf{R}^{(2)} = (\mathbf{Q}^{(1)} \otimes \mathbf{Q}^{(2)})(\mathbf{R}^{(1)} \otimes \mathbf{R}^{(2)}) = \mathbf{Q}\mathbf{R}. \tag{5.16}$$

The reduced $\mathbf{Q}$ term for $\mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)}$ is $\mathbf{Q}^{(1)} \otimes \mathbf{Q}^{(2)}$. Therefore, the leverage score of the ith row in $\mathbf{Q}$, ℓ_i, can be expressed as,

$$\begin{aligned} \ell_i(\mathbf{P}) &= \|\mathbf{Q}(i,:)\|_2^2 = \left\|\mathbf{Q}^{(1)}(i_1,:) \otimes \mathbf{Q}^{(2)}(i_2,:)\right\|_2^2 \\ &= \left\|\mathbf{Q}^{(1)}(i_1,:)\right\|_2^2 \left\|\mathbf{Q}^{(2)}(i_2,:)\right\|_2^2 = \ell_{i_1}(\mathbf{A}^{(1)})\ell_{i_2}(\mathbf{A}^{(2)}). \end{aligned} \tag{5.17}$$

Q.E.D.

Let $p_i = \ell_i(\mathbf{P})/R^N$ denote the leverage score sampling probability for ith index, and $p_{i_k}^{(k)} = \ell_{i_k}(\mathbf{A}^{(k)})/R$ for $k \in [N]$ denote the leverage score sampling probability for i_kth index of $\mathbf{A}^{(k)}$. Based on Lemma 5.1, we have

$$p_i = p_{i_1}^{(1)} \cdots p_{i_N}^{(N)}. \tag{5.18}$$

Therefore, leverage score sampling can be efficiently performed by sampling the row of $\mathbf{P}$ associated with multi-index $(i_1, \ldots, i_N)$, where i_k is selected with probability $p_{i_k}^{(k)}$. To calculate the leverage scores of each $\mathbf{A}^{(k)}$, N QR decompositions are needed, which in total cost $\mathcal{O}(NsR^2)$. In addition, the cost of this sampling process would be $\mathcal{O}(Nm)$ if m samples are needed, making the overall cost $\mathcal{O}(NsR^2 + Nm)$. To calculate $\mathbf{SP}$, for each sampled multi-index $(i_1, \ldots, i_N)$, we need to perform the Kronecker product,

$$\mathbf{A}^{(1)}(i_1,:) \otimes \cdots \otimes \mathbf{A}^{(N)}(i_N,:), \tag{5.19}$$

which costs $\mathcal{O}\left(R^N\right)$. Therefore, including the cost of QR decompositions, the overall cost is $\mathcal{O}\left(NsR^2 + mR^N\right)$.

Rather than performing importance random sampling based on leverage scores, another way introduced in [179] to construct the sketching matrix is to deterministically sample rows having the largest leverage scores. This idea is also used in [82] for randomized CP decomposition. Papailiopoulos et al. [180] show that if the leverage scores follow a moderately steep power-law decay, then deterministic sampling can be provably as efficient and even better than random sampling. We compare both leverage score sampling techniques in Section 5.4. For the sampling complexity analysis in Section 5.2 and Section 5.3, we only consider the random sampling technique.

5.7 INITIALIZATION OF FACTOR MATRICES VIA THE RANDOMIZED RANGE FINDER

Algorithm 5.3: Init-RRF: Initialization based on randomized range finder

```
1: Input: Matrix $\mathbf{M} \in \mathbb{R}^{n\times m}$, rank $R$, tolerance $\epsilon$
2: Initialize $\mathbf{S} \in \mathbb{R}^{m\times k}$, with $k = \mathcal{O}(R/\epsilon)$, as a composite sketching matrix (see Definition 5.5)
3: $\mathbf{B} \leftarrow \mathbf{MS}$
4: $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V} \leftarrow \texttt{SVD}(\mathbf{B})$
5: Return: $\mathbf{U}(:, :R)$
```

The effectiveness of sketching with leverage score sampling for Tucker-ALS is dependent on finding a good initialization of the factor matrices. This sensitivity arises because in each subproblem (5.2), only part of the input tensor being sampled is taken into consideration, and some non-zero input tensor elements are unsampled in all ALS linear least squares subproblems if the initialization of the factor matrices are far from the accurate solutions. Initialization is not a big problem for CountSketch/TensorSketch, since all the non-zero elements in the input tensor appear in the sketched right-hand-side.

An unsatisfactory initialization can severely affect the accuracy of leverage score sampling if elements of the tensor have large variability in magnitudes, a property known as high coherence. The coherence [197] of a matrix $\mathbf{U} \in \mathbb{R}^{n\times r}$ with $n > r$ is defined as $\mu(\mathbf{U}) = \frac{n}{r}\max_{i<n} \|\mathbf{Q}_U^T \mathbf{e}_i\|$, where $\mathbf{Q}_U$ is an orthogonal basis for the column space of $\mathbf{U}$ and $\mathbf{e}_i$ for $i \in [n]$ is a standard basis. Large coherence means that the orthogonal basis $\mathbf{Q}_U$ has large row norm variability. A tensor $\boldsymbol{\mathfrak{T}}$ has high coherence if all of its matricizations $\mathbf{T}_{(i)}^T$ for $i \in [N]$ have high coherence.

We use an example to illustrate the problem of bad initializations for leverage score sampling on tensors with high coherence. Suppose we seek a rank R Tucker decomposition of $\boldsymbol{\mathfrak{T}} \in \mathbb{R}^{s\times s\times s}$ expressed as

$$\boldsymbol{\mathfrak{T}} = \boldsymbol{\mathfrak{C}} \times_1 \mathbf{A} \times_2 \mathbf{A} \times_3 \mathbf{A} + \boldsymbol{\mathfrak{D}}, \tag{5.20}$$

where $\boldsymbol{\mathfrak{C}} \in \mathbb{R}^{R\times R\times R}$ is a tensor with elements drawn from a normal distribution, $\boldsymbol{\mathfrak{D}} \in \mathbb{R}^{s\times s\times s}$

is a very sparse tensor (has high coherence), and $\mathbf{A} \in \mathbb{R}^{s\times R}$ is an orthogonal basis for the column space of a matrix with elements drawn from a normal distribution. Let all the factor matrices be initialized by $\mathbf{A}$. Consider $R \ll s$ and let the leverage score sample size $m = R$. Since $\boldsymbol{\mathcal{D}}$ is very sparse, there is a high probability that most of the non-zero elements in $\boldsymbol{\mathcal{D}}$ are not sampled in all the sketched subproblems, resulting in a decomposition error proportional to $\|\boldsymbol{\mathcal{D}}\|_F$.

This problem can be fixed by initializing factor matrices using the randomized range finder (RRF) algorithm. For each matricization $\mathbf{T}^{(i)} \in \mathbb{R}^{s\times s^{N-1}}$, where $i \in [N]$, we first find a good low-rank subspace $\mathbf{U} \in \mathbb{R}^{s\times m}$, where $m = \mathcal{O}(R/\epsilon)$, such that it is ϵ-close to the rank-R subspace defined by its leading left singular vectors,

$$\left\|\mathbf{T}^{(i)} - \mathbf{U}\mathbf{U}^T\mathbf{T}^{(i)}\right\|_F^2 \leq (1+\epsilon) \min_{\text{rank}(\mathbf{X})\leq R} \left\|\mathbf{T}^{(i)} - \mathbf{X}\right\|_F^2, \tag{5.21}$$

and then initialize $\mathbf{A}^{(i)}$ based on the first R columns of $\mathbf{U}$. To calculate $\mathbf{U}$, we use a composite sketching matrix $\mathbf{S}$ defined in Definition 5.5, such that $\mathbf{U}$ is calculated via performing SVD on the sketched matrix $\mathbf{T}^{(i)}\mathbf{S}$. Based on Theorem 5.3, (5.21) holds with high probability.

Definition 5.5 (Composite sketching matrix [198], [199])**.** *Let* $k_1 = \mathcal{O}(R/\epsilon)$ *and* $k_2 = \mathcal{O}(R^2 + R/\epsilon)$*. The composite sketching matrix* $\mathbf{S} \in \mathbb{R}^{s\times k_1}$ *is defined as* $\mathbf{S} = \mathbf{T}\mathbf{G}$*, where* $\mathbf{T} \in \mathbb{R}^{s\times k_2}$ *is a CountSketch matrix (defined in Definition 5.1), and* $\mathbf{G} \in \mathbb{R}^{k_2\times k_1}$ *contains elements selected randomly from a normal distribution with variance* $1/k_1$*.*

Theorem 5.3 (Good low-rank subspace [198])**.** *Let* $\mathbf{T}$ *be an* $m \times n$ *matrix,* $R < \text{rank}(\mathbf{T})$ *be a rank parameter, and* $\epsilon > 0$ *be an accuracy parameter. Let* $\mathbf{S} \in \mathbb{R}^{n\times k}$ *be a composite sketching matrix defined as in Definition 5.5. Let* $\mathbf{B} = \mathbf{T}\mathbf{S}$ *and let* $\mathbf{Q} \in \mathbb{R}^{m\times k}$ *be any orthogonal basis for the column space of* $\mathbf{B}$*. Then, with probability at least 0.99,*

$$\left\|\mathbf{T} - \mathbf{Q}\mathbf{Q}^T\mathbf{T}\right\|_F^2 \leq (1+\epsilon)\left\|\mathbf{T} - \widetilde{\mathbf{T}}\right\|_F^2, \tag{5.22}$$

where $\widetilde{\mathbf{T}}$ *is the best rank-R approximation of* $\mathbf{T}$*.*

The algorithm is shown in Algorithm 5.3. The multiplication $\mathbf{T}^{(i)}\mathbf{S}$ has a cost of $\mathcal{O}(\text{nnz}(\boldsymbol{\mathcal{T}}) + sR^3/\epsilon)$, and the SVD step has a cost of $\mathcal{O}(sR^2/\epsilon)$, making the cost of the initialization step $\mathcal{O}(\text{nnz}(\boldsymbol{\mathcal{T}}) + sR^3/\epsilon)$. Since we need at least go over all the non-zero elements of the input tensor for a good initialization guess, the cost is $\Omega(\text{nnz}(\boldsymbol{\mathcal{T}}) + sR)$. Consequently, Algorithm 5.3 is computationally efficient for small R.

Note that since $\mathbf{A}^{(i)}$ is only part of $\mathbf{U}$, the error $\left\|\mathbf{T}^{(i)} - \mathbf{A}^{(i)}\mathbf{A}^{(i)T}\mathbf{T}^{(i)}\right\|_F^2$ is generally higher than that shown in (5.21), so further ALS sweeps are necessary to further decrease the residual. Based on the experimental results shown in Section 5.4, this initialization greatly enhances the performance of leverage score sampling for tensors with high coherence.

5.8 ALGORITHM FOR CP DECOMPOSITION

Algorithm 5.4: CP-Sketch-Tucker: CP decomposition with sketched Tucker-ALS

```
1: Input: Tensor $\mathcal{T} \in \mathbb{R}^{s_1 \times \cdots s_N}$, rank $R$, maximum number of Tucker-ALS sweeps $I_{max}$,
   Tucker sketching tolerance $\epsilon$
2: $\left\{\mathcal{C}, \mathbf{B}^{(1)}, \ldots, \mathbf{B}^{(N)}\right\} \leftarrow$ Rand-Tucker-ALS$(\mathcal{T}, \{R, \ldots, R\}, I_{max}, \epsilon)$
3: $\left\{\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}\right\} \leftarrow$ CP-ALS$(\mathcal{C}, R)$
4: Return: $\{\mathbf{B}^{(1)}\mathbf{A}^{(1)}, \ldots, \mathbf{B}^{(N)}\mathbf{A}^{(N)}\}$
```

When $R \ll s$, sketched Tucker-ALS can also be used to accelerate CP decomposition. When an exact CP decomposition of the desired rank exists, it is attainable from a Tucker decomposition of the same or greater rank. In particular, given a CP decomposition of the desired rank for the core tensor from Tucker decomposition, it suffices to multiply respective factor matrices of the CP and Tucker decompositions to obtain a CP decomposition of the original tensor. For the exact case, Tucker decomposition can be computed exactly via the sequentially truncated HOSVD, and for approximation, the Tucker model is generally easier to fit than CP. Consequently, Tucker decomposition has been employed as a pre-processing step prior to running CP decomposition algorithms such as CP-ALS [77], [114], [178], [190].

We leverage the ability of Tucker decomposition to preserve low-rank CP structure to apply our fast randomized Tucker algorithms to low-rank CP decomposition. We show the algorithm in Algorithm 5.4. In practice, the randomized Tucker-ALS algorithm takes a small number of sweeps (less than 5) to converge, and then CP-ALS can be applied on the core tensor, which is computationally inexpensive.

The state-of-the-art approach for randomized CP-ALS [82] is to use leverage score sampling to solve each subproblem (5.4). The cost sufficient to get $(1+\mathcal{O}(\epsilon))$-accurate residual norm for each subproblem is $\mathcal{O}\left(sR^N \log(1/\delta)/\epsilon^2\right)$. With the same criteria, the cost for sketched Tucker-ALS with leverage score sampling is $\mathcal{O}\left(sR^N/(\epsilon^2\delta) + R^{3(N-1)}/(\epsilon^2\delta)\right)$. As we can see, when $R \ll s$, the cost of each Tucker decomposition subproblem is only slightly higher than that of CP decomposition, and the fast convergence of Tucker-ALS makes this Tucker + CP method more efficient than directly applying CP decomposition on the input tensor.

5.9 ADDITIONAL EXPERIMENTS

In this section, we provide additional experimental results for both Tucker and CP decompositions. In Section 5.9.1, we present results for Tucker decomposition of dense tensors. In Section 5.9.2, we present results for Tucker decomposition of sparse tensors. In Section 5.9.3, we provide additional results for CP decomposition.

5.9.1 Additional Results for Tucker Decomposition of Dense Synthetic Tensors

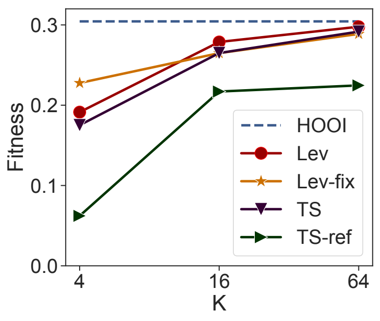
Size (s)	ALS	ALS+leverage scores	ALS+TensorSketch	ALS+TensorSketch [79]
2×10^2	5.06×10^8	1.58×10^8	1.77×10^8	2.10×10^8
2×10^3	4.82×10^{11}	5.15×10^8	5.25×10^8	3.84×10^8
2×10^4	4.80×10^{14}	4.08×10^9	4.00×10^9	2.12×10^9
2×10^5	4.80×10^{17}	3.97×10^{10}	3.88×10^{10}	2.05×10^{10}

Table 5.3: Comparison of per-sweep computational cost of different methods. The input tensors are assumed to be dense with size $s \times s \times s$, and the Tucker rank is $R = 10$. For sketching algorithms, we set the sketch size as $16R^2$.

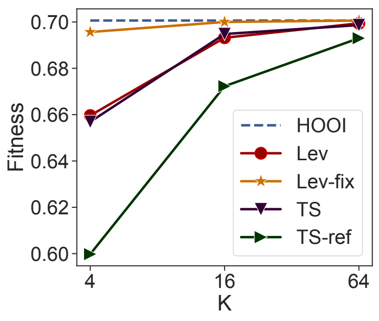
Cost comparison We compare the per-sweep computational cost (number of floating point operations (FLOPs)) between the standard HOOI, our ALS + leverage score sampling algorithm, our ALS + TensorSketch, and the reference ALS + TensorSketch algorithm [79]. As can be seen from Table 5.3, when the Tucker rank is small, the per-iteration cost of our algorithms are a bit higher than the algorithm in [79]. In addition, the cost ratio of our algorithm over the reference is bounded by 2. Although the per-iteration cost increases slightly, the output accuracy has a large improvement compared to the reference algorithm.

K	4	16	64
e	0.22	0.05	0.01

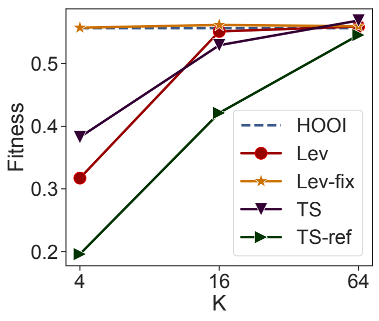
Table 5.4: Relation between the sketch size parameter K and the average relative least squares residual norm error (5.23). We test on Tensor 1, and set $s = 200, R = 5, \alpha = 1.6$. The presented relative residual norm error is the mean of 10 results using leverage score sampling.



(a) Tensor 1 with $s = 200$



(b) Tensor 2 with $s = 200$



(c) Tensor 3 with $s = 1000$

Figure 5.3: Relation between the final fitness and sketch size parameter K for each algorithm with different synthetic tensors. For Tensor 3, $\mathfrak{T}$ is generated based on (5.8). For all the experiments, we set $R = 5, \alpha = 1.6$, and $K = 16$. Each data point is the mean of 10 experimental results with different random seeds. HOSVD/RRF initialization is used for all experiments.

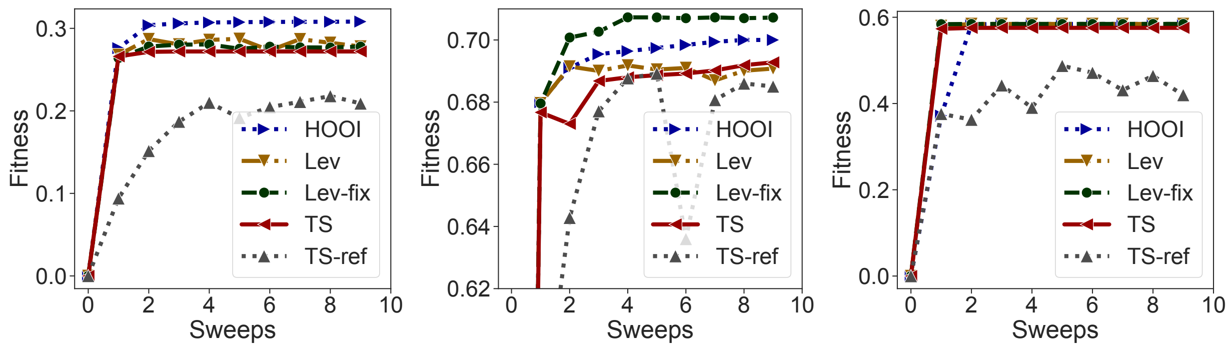
Relation between sketch size and accuracy. In our experiments, we parameterize the sketch size as KR^{N-1}, where K incorporates the effect of ϵ and δ. Here we experimentally show that a moderate K is enough to yield accurate results. Each time we solve a constrained least squares subproblem in HOOI, $\mathbf{X}_r = \arg\min_{\mathbf{X},\text{rank}(\mathbf{X})\leq r} ||\mathbf{AX} - \mathbf{B}||_F$, we calculate the approximate solution $\hat{\mathbf{X}}_r$ using leverage score sampling, and check the relative residual norm error,

$$e = \frac{||\mathbf{A}\hat{\mathbf{X}}_r - \mathbf{B}||_F^2 - ||\mathbf{AX}_r - \mathbf{B}||_F^2}{||\mathbf{AX}_r - \mathbf{B}||_F^2}. \tag{5.23}$$

In our theoretical analysis, this term is bounded by $O(\epsilon)$. As can be seen from Table 5.4, setting K to be 16 or 64 guarantees that each subproblem is accurately solved.

Fig. 5.3 show the relation between the final Tucker decomposition fitness and K. As is expected, increasing K can increase the accuracy of the randomized linear least squares solve, thus improving the final fitness. For leverage score sampling, Fig. 5.3b and Fig. 5.3c show that when the sketch size is small ($K = 4$), the deterministic leverage score sampling scheme outperforms the random sampling scheme for Tensor 2 and Tensor 3. This means that when the tensor has a strong low-rank signal, the deterministic sampling scheme can be better, consistent with the results in [180].

Detailed fitness-sweeps relation. We show the detailed fitness-sweeps relation for different synthetic dense tensors in Fig. 5.4. The reference randomized algorithm suffers from unstable convergence as well as low fitness, while our new randomized ALS scheme,



(a) Tensor 1 with $s = 200$ (b) Tensor 2 with $s = 200$ (c) Tensor 3 with $s = 1000$

Figure 5.4: Detailed fitness-sweeps relation for Tucker decomposition of three dense tensors with different parameters. For Tensor 3, $\mathfrak{T}$ is generated based on (5.8). For all the experiments, we set $R = 5, \alpha = 1.6$. In the plots, Lev, Lev-fix, and TS denote our new sketched Tucker-ALS scheme with leverage score random sampling, leverage score deterministic sampling, and TensorSketch, respectively. TS-ref denotes the reference sketched Tucker-ALS algorithm with TensorSketch. HOOI is initialized with HOSVD, and all other methods are initialized with RRF (Algorithm 5.3). Markers represent the results per sweep.

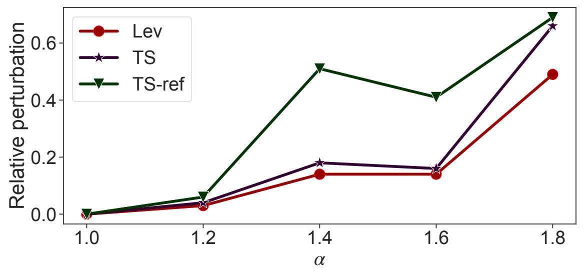
with either leverage score sampling or TensorSketch, converges faster than the reference randomized algorithm and reaches higher accuracy.

Perturbation of factor matrices. We also compare the perturbation of factor matrices for each randomized algorithm relative to the baseline HOOI. Let $\widehat{\mathbf{A}}_i$ be the output ith mode factor matrix from a randomized algorithm, and let $\mathbf{A}_i$ be the output ith mode factor matrix from HOOI. We calculate the relative perturbation of the subspace spanned by $\mathbf{A}_i$,

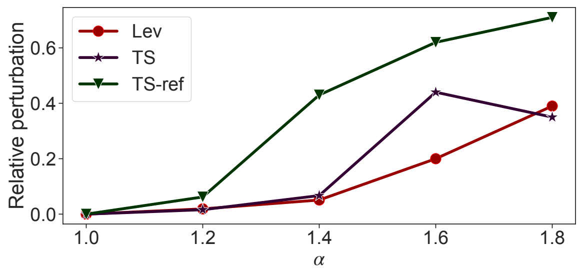
$$p_i = \frac{\left\|\widehat{\mathbf{A}}_i\widehat{\mathbf{A}}_i^T - \mathbf{A}_i\mathbf{A}_i^T\right\|_F}{\left\|\mathbf{A}_i\mathbf{A}_i^T\right\|_F}, \tag{5.24}$$

and report the average relative perturbation acorss the tensor mode i, $p = \frac{1}{N}\sum_{i=1}^{N} p_i$. Smaller perturbation means the output of the randomized algorithm is closer to the HOOI output.

As can be seen from Fig. 5.5, our new sketching algorithms yield less output perturbation compared to the reference [79]. With the increase of The ratio R_{true}/R, denoted as α, all algorithms tend to yield higher perturbation. This is expected, since with large α, the input tensor tends to have non-unique best rank-R decompositions, and a large perturbation in factor matrices can still yield similar fitness. Overall the results show that our sketching algorithms are more accurate than the reference TensorSketch approach [79].



(a) $K = 16$



(b) $K = 64$

Figure 5.5: Relation between the relative perturbation of the subspace spanned by each factor matrix, p, and sketch size parameter K for each algorithm. We test on Tensor 1, and set $s = 200, R = 5, \alpha = 1.6$. Each data point is the mean of 10 experimental results with different random seeds. HOSVD/RRF initialization is used for all experiments.

5.9.2 Results for Tucker Decomposition of Sparse Tensors

We use two synthetic sparse tensors to evaluate different algorithms.

1. **Sparse tensors with specific Tucker rank**. We generate tensors based on (5.8) with each element in the core tensor and factor matrices being an i.i.d normally distributed random variable $\mathcal{N}(0,1)$ with probability p and zero otherwise. Since each element,

$$\mathfrak{T}(i,j,k) = \sum_{x,y,z} \mathbf{B}^{(1)}(i,x) \cdot \mathbf{B}^{(2)}(j,y) \cdot \mathbf{B}^{(3)}(k,z) \cdot \mathfrak{C}(x,y,z), \tag{5.25}$$

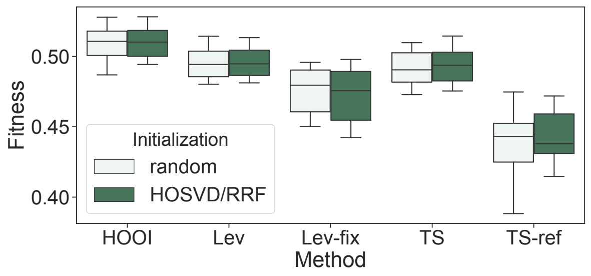
and

$$P\left[\mathbf{B}^{(1)}(i,x) \cdot \mathbf{B}^{(2)}(j,y) \cdot \mathbf{B}^{(3)}(k,z) \cdot \mathfrak{C}(x,y,z) \neq 0\right] = p^4, \tag{5.26}$$

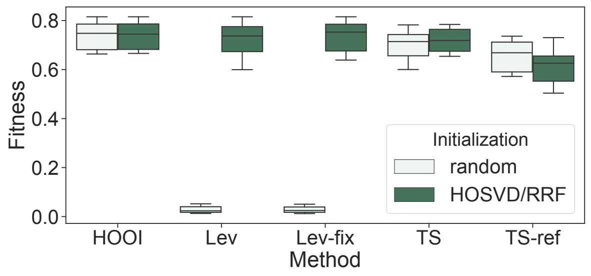
the expected sparsity of $\mathfrak{T}$, which is equivalent to the probability that each element $\mathfrak{T}(i,j,k) = 0$, is bounded below by $1 - R_{\text{true}}^3 p^4$. Through varying p, we generate tensors with different expected sparsity.

2. **Tensors with large coherence**. We also test on tensors with large coherence, $\mathfrak{T}^{(b)} = \mathfrak{T} + \mathfrak{N}$. Tensor $\mathfrak{T}$ is generated based on (5.25), and $\mathfrak{N}$ contains $n \ll s$ elements with random positions and same large magnitude. In our experiments, we set $n = 10$, and each nonzero element in $\mathfrak{N}$ has the i.i.d. normal distribution $\mathcal{N}(\|\mathfrak{T}\|_F/\sqrt{n}, 1)$, which means the expected norm ratio $\mathbb{E}[\|\mathfrak{N}\|_F/\|\mathfrak{T}\|_F] = 1$. This tensor has large coherence and is used to test the robustness problem detailed in Section 5.7.

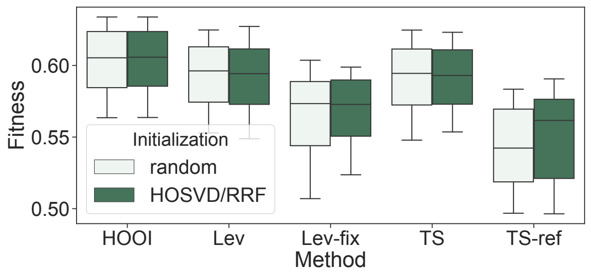
We show our experimental results for sparse tensors in Fig. 5.6. For both Tensor 1 and Tensor 2, we test on tensors with different sparsity via varying the parameter p. When $p = 0.1$



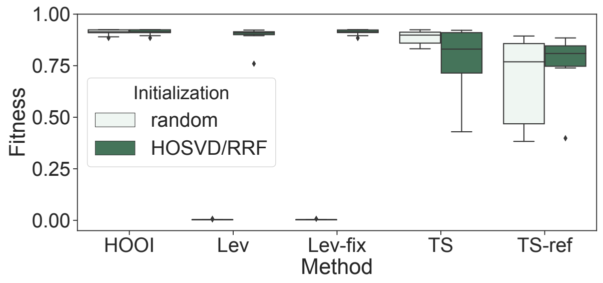
(a) Tensor 1 with $p = 0.5$



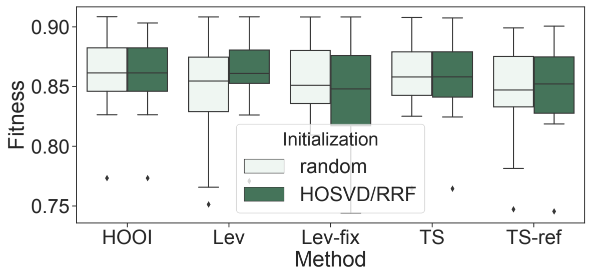
(b) Tensor 2 with $p = 0.5$



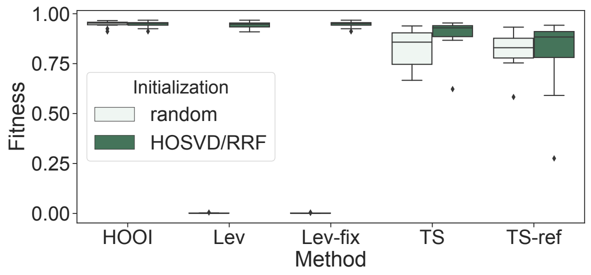
(c) Tensor 1 with $p = 0.1$



(d) Tensor 2 with $p = 0.1$



(e) Tensor 1 with $p = 0.02$



(f) Tensor 2 with $p = 0.02$

Figure 5.6: Experimental results for Tucker decomposition of sparse tensors. For all the experiments, we set $s = 2000, R = 10, \alpha = 1.2$ and $K = 16$. **(a)(c)(e)** Box plots of the final fitness for each algorithm on Tensor 1 with different sparsity parameter p. **(b)(d)(f)** Box plots of the final fitness for each algorithm on Tensor 2 with different sparsity parameter p. Each box is based on 10 experiments with different random seeds.

(Fig. 5.6c, 5.6d), the expected sparsity of the tensor is greater than 0.9. When $p = 0.02$ (Fig. 5.6e, 5.6f), the expected sparsity of the tensor is greater than 0.9998.

The results for Tensor 1 are shown in Fig. 5.6a,5.6c,5.6e. Our new randomized ALS scheme, with either leverage score sampling or TensorSketch, outperforms the reference randomized algorithm with $p = 0.1$ and $p = 0.5$. The relative fitness improvement ranges from 3.6% (Fig. 5.6c) to 12.7% (Fig. 5.6a). The performance of our new scheme is comparable to the reference with $p = 0.02$. The reason for the reduced improvements is that these tensors have high decomposition fitness ($0.8 \sim 0.9$) and each non-zero element has the same distribution, so sophisticated sampling is not needed to achieve high accuracy. Similar to the case of dense tensors shown in Fig. 5.1a, we observe similar behavior for Tensor 1 with random initialization and RRF-based initialization.

The results for Tensor 2 are shown in Fig. 5.6b,5.6d,5.6f. Our new randomized ALS scheme outperforms the reference randomized algorithm for all the cases. Similar to the case of dense tensors (Fig. 5.1c), for leverage score sampling, the random initialization results in approximately zero final fitness, and the RRF-based initialization can greatly improve the output fitness. Therefore, the RRF-based initialization scheme is important for improving the robustness of leverage score sampling.

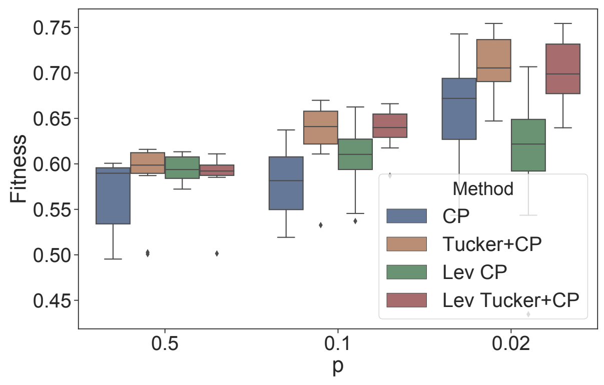
On the contrary, TensorSketch based algorithms are not sensitive to the choice of initialization scheme. Although they perform much better compared to the leverage score sampling with random initialization, the output fitness is still a bit worse than HOOI and can have relatively larger variance (Fig. 5.6d,5.6f). This means TensorSketch is less effective than leverage score sampling with RRF initialization for this tensor.

In summary, we observe the algorithm combining leverage score sampling, the RRF-based initialization and our new ALS scheme achieves the highest accuracy and the most robust performance across test problems among randomized schemes.

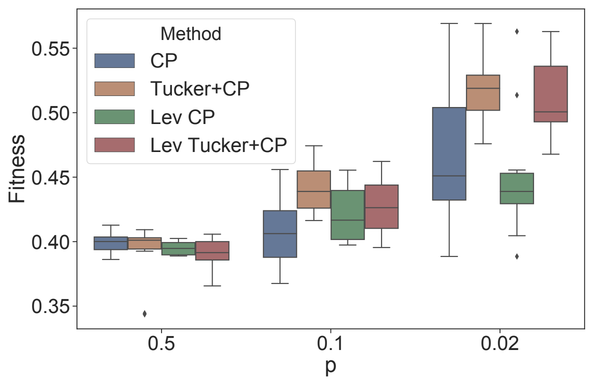
5.9.3 Additional Experiments for CP Decomposition

For (sketched) Tucker + CP algorithms, we run 5 (sketched) Tucker-ALS sweeps first, and then run the CP-ALS algorithm on the core tensor for 25 sweeps. RRF-based initialization is used for Tucker-ALS, and HOSVD on the core tensor is used to initialize the factor matrices of the small CP decomposition problem. For (sketched) CP-ALS algorithms, we also use the RRF-based initialization and run 30 sweeps afterwards, which is sufficient for CP-ALS to converge based on our experiments. This initialization makes sure that leverage score sampling is effective for sparse tensors. We set the sketch size as KR^2 for both algorithms. For the RRF-based initialization, we set the sketch size (k in Algorithm 5.3) as $\sqrt{K}R$.

We show the relation between final fitness and the tensor sparsity parameter, p, in Fig. 5.7. As can be seen, for all the tested tensors, the Tucker + CP algorithms perform similarly, and usually better than directly performing CP decomposition. When the input tensor is sparse ($p = 0.1$ and 0.02), the advantage of the Tucker + CP algorithms is greater. The sketched Tucker-ALS + CP-ALS scheme has a comparable performance compared to Tucker HOOI + CP-ALS, while requiring less computation.



(a) $\alpha = 1.2$



(b) $\alpha = 1.6$

Figure 5.7: Relation between final fitness and sparsity parameter p for CP decomposition. For all the experiments, we set $s = 2000, R = 10$ and $K = 16$. In the plots, CP denotes running CP-ALS, Tucker+CP denotes running the Tucker HOOI + CP-ALS algorithm, Lev CP denotes running leverage score sampling based randomized CP-ALS, and Lev Tucker+CP denotes running the leverage score sampling based Tucker-ALS + CP-ALS algorithm. Each box is based on 10 experiments with different random seeds.

5.10 DETAILED PROOFS FOR SECTION 5.2

In this section, we provide detailed proofs for the sketch size upper bounds of both sketched unconstrained and rank-constrained linear least squares problems. In Section 5.10.1, we define the $(\gamma, \delta, \epsilon)$-accurate sketching matrix, and show the error bound for sketched unconstrained linear least squares, under the assumption that the sketching matrix is a $(1/2, \delta, \epsilon)$-accurate sketching matrix. In Section 5.10.2, we show the error bound for sketched rank-constrained linear least squares. In Section 5.10.3 and Section 5.10.4, we finish the proofs by giving the sketch size bounds that are sufficient for the TensorSketch matrix and leverage score sampling matrix to be the $(1/2, \delta, \epsilon)$-accurate sketching matrix, respectively.

5.10.1 Error Bound for Sketched Unconstrained Linear Least Squares

We define the $(\gamma, \delta, \epsilon)$-accurate sketching matrix in Definition 5.6. In Lemma 5.2, we show the relative error bound for the unconstrained linear least squares problem with a $(1/2, \delta, \epsilon)$-accurate sketching matrix. By $\mathbf{Q}_P$ we denote a matrix whose columns form an orthonormal basis for the column space of $\mathbf{P}$.

Definition 5.6 ($(\gamma, \delta, \epsilon)$-accurate Sketching Matrix)**.** *A random matrix* $\mathbf{S} \in \mathbb{R}^{m \times s}$ *is a* $(\gamma, \delta, \epsilon)$*-accurate sketching matrix for* $\mathbf{P} \in \mathbb{R}^{s \times R}$ *if the following two conditions hold simultaneously.*

1. *With probability at least* $1-\delta/2$*, each singular value* σ *of* $\mathbf{SQ}_P$ *satisfies*

$$1-\gamma \leq \sigma^2 \leq 1+\gamma. \tag{5.27}$$

2. *With probability at least* $1-\delta/2$*, for any fixed matrix* $\mathbf{B}$*, we have*

$$\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{SB}-\mathbf{Q}_P^T\mathbf{B}\|_F^2 \leq \epsilon^2\cdot\|\mathbf{B}\|_F^2. \tag{5.28}$$

Lemma 5.2 (Linear Least Squares with $(1/2,\delta,\epsilon)$-accurate Sketching Matrix [60], [82], [200])**.** *Given a full-rank matrix* $\mathbf{P}\in\mathbb{R}^{s\times R}$ *with* $s\geq R$*, and* $\mathbf{B}\in\mathbb{R}^{s\times n}$*. Let* $\mathbf{S}\in\mathbb{R}^{m\times s}$ *be a* $(1/2,\delta,\epsilon)$*-accurate sketching matrix. Let* $\mathbf{B}^\perp=\mathbf{PX}_{opt}-\mathbf{B}$*, with* $\mathbf{X}_{opt}=\arg\min_{\mathbf{X}}\|\mathbf{PX}-\mathbf{B}\|_F$*, and* $\widetilde{\mathbf{X}}_{opt}=\arg\min_{\mathbf{X}}||\mathbf{SPX}-\mathbf{SB}||_F$*. Then the following approximation holds with probability at least* $1-\delta$*,*

$$\left\|\mathbf{P}\widetilde{\mathbf{X}}_{opt}-\mathbf{PX}_{opt}\right\|_F^2 \leq \mathcal{O}\left(\epsilon^2\right)\left\|\mathbf{B}^\perp\right\|_F^2. \tag{5.29}$$

Proof. Define the reduced QR decomposition, $\mathbf{P}=\mathbf{Q}_P\mathbf{R}_P$. The unconstrained sketched problem can be rewritten as

$$\begin{aligned}\min_{\mathbf{X}}\|\mathbf{SPX}-\mathbf{SB}\|_F &= \min_{\mathbf{X}}\left\|\mathbf{SPX}-\mathbf{S}(\mathbf{PX}_{\text{opt}}+\mathbf{B}^\perp)\right\|_F\\ &= \min_{\mathbf{X}}\left\|\mathbf{SQ}_P\mathbf{R}_P(\mathbf{X}-\mathbf{X}_{\text{opt}})-\mathbf{SB}^\perp\right\|_F,\end{aligned} \tag{5.30}$$

thus the optimality condition is

$$(\mathbf{SQ}_P)^T\mathbf{SQ}_P\mathbf{R}_P(\widetilde{\mathbf{X}}_{\text{opt}}-\mathbf{X}_{\text{opt}})=(\mathbf{SQ}_P)^T\mathbf{SB}^\perp. \tag{5.31}$$

Based on (5.27),(5.28), with probability at least $1-\delta$, both of the following hold,

$$\sigma_{\text{min}}^2(\mathbf{SQ}_P)\geq 1-\gamma=1/2, \tag{5.32}$$

$$\left\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{SB}^\perp\right\|_F^2=\left\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{SB}^\perp-\mathbf{Q}_P^T\mathbf{B}^\perp\right\|_F^2\leq\epsilon^2\cdot\left\|\mathbf{B}^\perp\right\|_F^2, \tag{5.33}$$

where $\sigma_{\text{min}}(\mathbf{SQ}_P)$ is the singular value of $\mathbf{SQ}_P$ with the smallest magnitude. Combining

(5.31), (5.32), and (5.33), we obtain

$$\begin{aligned}\left\|\mathbf{P}\widetilde{\mathbf{X}}_{\text{opt}} - \mathbf{P}\mathbf{X}_{\text{opt}}\right\|_F^2 &= \left\|\mathbf{R}_P\widetilde{\mathbf{X}}_{\text{opt}} - \mathbf{R}_P\mathbf{X}_{\text{opt}}\right\|_F^2 \\ &\overset{(5.32)}{\leq} 4\left\|(\mathbf{S}\mathbf{Q}_P)^T\mathbf{S}\mathbf{Q}_P\mathbf{R}_P(\widetilde{\mathbf{X}}_{\text{opt}} - \mathbf{X}_{\text{opt}})\right\|_F^2 \\ &\overset{(5.31)}{=} 4\left\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{S}\mathbf{B}^{\perp}\right\|_F^2 \\ &\overset{(5.33)}{\leq} 4\epsilon^2 \cdot \left\|\mathbf{B}^{\perp}\right\|_F^2 = \mathcal{O}\left(\epsilon^2\right)\left\|\mathbf{B}^{\perp}\right\|_F^2.\end{aligned} \tag{5.34}$$

Q.E.D.

5.10.2 Error Bound for Sketched Rank-constrained Linear Least Squares

We show in Theorem 5.4 that with at least $1-\delta$ probability, the relative residual norm error for the rank-constrained linear least squares with a $(1/2, \delta, \epsilon)$-accurate sketching matrix is bounded by $\mathcal{O}(\epsilon)$. We first state Mirsky's Inequality below, which bounds the perturbation of singular values when the input matrix is perturbed. We direct readers to the reference for its proof. This bound will be used in Theorem 5.4.

Lemma 5.3 (Mirsky's Inequality for Perturbation of Singular Values [201])**.** *Let* $\mathbf{A}$ *and* $\mathbf{F}$ *be arbitrary matrices (of the same size) where* $\sigma_1 \geq \cdots \geq \sigma_n$ *are the singular values of* $\mathbf{A}$ *and* $\sigma_1' \geq \cdots \geq \sigma_n'$ *are the singular values of* $\mathbf{A}+\mathbf{F}$*. Then*

$$\sum_{i=1}^{n}(\sigma_i - \sigma_i')^2 \leq \|\mathbf{F}\|_F^2. \tag{5.35}$$

Theorem 5.4 (Rank-constrained Linear Least Squares with $(1/2, \delta, \epsilon)$-accurate Sketching Matrix)**.** *Given* $\mathbf{P} \in \mathbb{R}^{s\times R}$ *with orthonormal columns (such that* $\mathbf{P} = \mathbf{Q}_P$*), and* $\mathbf{B} \in \mathbb{R}^{s\times n}$*. Let* $\mathbf{S} \in \mathbb{R}^{m\times s}$ *be a* $(1/2, \delta, \epsilon)$*-accurate sketching matrix. Let* $\widetilde{\mathbf{X}}_r$ *be the best rank-r approximation of the solution of the problem* $\min_{\mathbf{X}} \|\mathbf{S}\mathbf{P}\mathbf{X} - \mathbf{S}\mathbf{B}\|_F$*, and let* $\mathbf{X}_r = \arg\min_{\mathbf{X}, rank(\mathbf{X})=r} \|\mathbf{P}\mathbf{X} - \mathbf{B}\|_F$*. Then the residual norm error bound,*

$$\left\|\mathbf{P}\widetilde{\mathbf{X}}_r - \mathbf{B}\right\|_F^2 \leq (1+\mathcal{O}(\epsilon))\left\|\mathbf{P}\mathbf{X}_r - \mathbf{B}\right\|_F^2, \tag{5.36}$$

holds with probability at least $1-\delta$*.*

Proof. Let $\mathcal{R} = \|\mathbf{P}\mathbf{X}_{\text{r}} - \mathbf{B}\|_F$. In addition, let $\mathbf{X}_{\text{opt}} = \arg\min_{\mathbf{X}} \|\mathbf{P}\mathbf{X} - \mathbf{B}\|_F$ be the optimum solution of the unconstrained linear least squares problem. Since the residual in the true solution for each component of the least-squares problem (column of $\mathbf{B}^{\perp}$) is orthogonal to the error due to low-rank approximation,

$$\begin{aligned}\mathcal{R}^2 = \|\mathbf{P}\mathbf{X}_{\mathrm{r}} - \mathbf{B}\|_F^2 = \|\mathbf{P}\mathbf{X}_{\mathrm{opt}} - \mathbf{B}\|_F^2 + \|\mathbf{P}\mathbf{X}_{\mathrm{r}} - \mathbf{P}\mathbf{X}_{\mathrm{opt}}\|_F^2 \\ = \left\|\mathbf{B}^{\perp}\right\|_F^2 + \|\mathbf{X}_{\mathrm{r}} - \mathbf{X}_{\mathrm{opt}}\|_F^2 .\end{aligned} \tag{5.37}$$

The last equality holds since $\mathbf{P}$ has orthonormal columns. Let $\widetilde{\mathbf{X}}_{\mathrm{opt}} = \arg\min_{\mathbf{X}} \|\mathbf{SPX} - \mathbf{SB}\|_F$ be the optimum solution of the unconstrained sketched problem. We have

$$\begin{aligned}\left\|\mathbf{P}\widetilde{\mathbf{X}}_{\mathrm{r}} - \mathbf{B}\right\|_F^2 &= \left\|\mathbf{P}\widetilde{\mathbf{X}}_{\mathrm{r}} - \mathbf{P}\widetilde{\mathbf{X}}_{\mathrm{opt}}\right\|_F^2 + \left\|\mathbf{P}\widetilde{\mathbf{X}}_{\mathrm{opt}} - \mathbf{B}\right\|_F^2 + 2\left\langle \mathbf{P}\widetilde{\mathbf{X}}_{\mathrm{r}} - \mathbf{P}\widetilde{\mathbf{X}}_{\mathrm{opt}}, \mathbf{P}\widetilde{\mathbf{X}}_{\mathrm{opt}} - \mathbf{B}\right\rangle_F \\ &= \left\|\widetilde{\mathbf{X}}_{\mathrm{r}} - \widetilde{\mathbf{X}}_{\mathrm{opt}}\right\|_F^2 + \left\|\widetilde{\mathbf{X}}_{\mathrm{opt}} - \mathbf{X}_{\mathrm{opt}}\right\|_F^2 + \left\|\mathbf{B}^{\perp}\right\|_F^2 + 2\left\langle \widetilde{\mathbf{X}}_{\mathrm{r}} - \widetilde{\mathbf{X}}_{\mathrm{opt}}, \widetilde{\mathbf{X}}_{\mathrm{opt}} - \mathbf{X}_{\mathrm{opt}}\right\rangle_F .\end{aligned} \tag{5.38}$$

Next we bound the magnitudes of the first, second and the fourth terms. According to Lemma 5.2, with probability at least $1-\delta$, the second term in (5.38) can be bounded as

$$\left\|\widetilde{\mathbf{X}}_{\mathrm{opt}} - \mathbf{X}_{\mathrm{opt}}\right\|_F^2 = \left\|\mathbf{P}\widetilde{\mathbf{X}}_{\mathrm{opt}} - \mathbf{P}\mathbf{X}_{\mathrm{opt}}\right\|_F^2 \leq C\epsilon^2 \left\|\mathbf{B}^{\perp}\right\|_F^2 , \tag{5.39}$$

for some constant $C \geq 1$. Suppose $\widetilde{\mathbf{X}}_{\mathrm{opt}}$ has singular values $\widetilde{\sigma}_i = \sigma_i + \delta\sigma_i$ for i in $\{1, \ldots, \min(R, n)\}$, where σ_i are the singular values of $\mathbf{X}_{\mathrm{opt}}$. Since $\widetilde{\mathbf{X}}_{\mathrm{r}}$ is defined to be the best low rank approximation to $\widetilde{\mathbf{X}}_{\mathrm{opt}}$, we have

$$\left\|\widetilde{\mathbf{X}}_{\mathrm{r}} - \widetilde{\mathbf{X}}_{\mathrm{opt}}\right\|_F^2 = \sum_{i=r+1}^{\min(R,n)} \widetilde{\sigma}_i^2 = \sum_{i=r+1}^{\min(R,n)} (\sigma_i + \delta\sigma_i)^2 = \sum_{i=r+1}^{\min(R,n)} \left(\sigma_i^2 + \delta\sigma_i^2 + 2\sigma_i\delta\sigma_i\right). \tag{5.40}$$

Since $\mathbf{P}$ has orthonormal columns, $\mathbf{X}_{\mathrm{r}}$ is the best rank-r approximation of $\mathbf{X}_{\mathrm{opt}}$,

$$\sum_{i=r+1}^{\min(R,n)} \sigma_i^2 = \|\mathbf{X}_{\mathrm{r}} - \mathbf{X}_{\mathrm{opt}}\|_F^2 . \tag{5.41}$$

In addition, based on Mirsky's inequality (Lemma 5.3),

$$\sum_{i=r+1}^{\min(R,n)} \delta\sigma_i^2 \leq \sum_{i=1}^{\min(R,n)} \delta\sigma_i^2 \overset{(5.35)}{\leq} \left\|\widetilde{\mathbf{X}}_{\mathrm{opt}} - \mathbf{X}_{\mathrm{opt}}\right\|_F^2 \overset{(5.39)}{\leq} C\epsilon^2 \left\|\mathbf{B}^{\perp}\right\|_F^2 , \tag{5.42}$$

and

$$\begin{aligned}\sum_{i=r+1}^{\min(R,n)} |2\sigma_i\delta\sigma_i| &= \epsilon \sum_{i=r+1}^{\min(R,n)} \left|2\sigma_i \frac{\delta\sigma_i}{\epsilon}\right| \le \epsilon \sum_{i=r+1}^{\min(R,n)} \left(\sigma_i^2 + \frac{\delta\sigma_i^2}{\epsilon^2}\right) \\ &\overset{(5.42)}{\le} C\epsilon \left(\|\mathbf{X}_{\mathrm{r}} - \mathbf{X}_{\mathrm{opt}}\|_F^2 + \left\|\mathbf{B}^{\perp}\right\|_F^2\right) = C\epsilon\mathcal{R}^2,\end{aligned} \tag{5.43}$$

thus (5.40) can be bounded as

$$\begin{aligned}\left\|\widetilde{\mathbf{X}}_{\mathrm{r}} - \widetilde{\mathbf{X}}_{\mathrm{opt}}\right\|_F^2 &\le \|\mathbf{X}_{\mathrm{r}} - \mathbf{X}_{\mathrm{opt}}\|_F^2 + C\epsilon^2 \left\|\mathbf{B}^{\perp}\right\|_F^2 + C\epsilon\mathcal{R}^2 \\ &= \|\mathbf{X}_{\mathrm{r}} - \mathbf{X}_{\mathrm{opt}}\|_F^2 + \mathcal{O}(\epsilon)\mathcal{R}^2.\end{aligned} \tag{5.44}$$

Next we bound the magnitude of the inner product term in (5.38),

$$\begin{aligned}\left|\left\langle \widetilde{\mathbf{X}}_{\mathrm{r}} - \widetilde{\mathbf{X}}_{\mathrm{opt}}, \widetilde{\mathbf{X}}_{\mathrm{opt}} - \mathbf{X}_{\mathrm{opt}}\right\rangle_F\right| &\le \left\|\widetilde{\mathbf{X}}_{\mathrm{r}} - \widetilde{\mathbf{X}}_{\mathrm{opt}}\right\|_F \left\|\widetilde{\mathbf{X}}_{\mathrm{opt}} - \mathbf{X}_{\mathrm{opt}}\right\|_F \\ &\overset{(5.39)}{\le} \sqrt{C}\epsilon \left\|\widetilde{\mathbf{X}}_{\mathrm{r}} - \widetilde{\mathbf{X}}_{\mathrm{opt}}\right\|_F \|\mathbf{B}^{\perp}\|_F \\ &\le \sqrt{C}\frac{\epsilon}{2}\left(\left\|\widetilde{\mathbf{X}}_{\mathrm{r}} - \widetilde{\mathbf{X}}_{\mathrm{opt}}\right\|_F^2 + \|\mathbf{B}^{\perp}\|_F^2\right) \\ &\overset{(5.44)}{\le} \sqrt{C}\frac{\epsilon}{2}\left(\|\mathbf{X}_{\mathrm{r}} - \mathbf{X}_{\mathrm{opt}}\|_F^2 + \mathcal{O}(\epsilon)\mathcal{R}^2 + \left\|\mathbf{B}^{\perp}\right\|_F^2\right) \\ &= \mathcal{O}(\epsilon)\mathcal{R}^2.\end{aligned} \tag{5.45}$$

Therefore, based on (5.38),(5.39),(5.44),(5.45), with probability at least $1-\delta$,

$$\left\|\mathbf{P}\widetilde{\mathbf{X}}_{\mathrm{r}} - \mathbf{B}\right\|_F^2 \le (1 + \mathcal{O}(\epsilon))\mathcal{R}^2 = (1 + \mathcal{O}(\epsilon))\left\|\mathbf{P}\mathbf{X}_{\mathrm{r}} - \mathbf{B}\right\|_F^2. \tag{5.46}$$

Q.E.D.

5.10.3 TensorSketch for Unconstrained & Rank-constrained Least Squares

In this section, we first give the sketch size bound that is sufficient for the TensorSketch matrix to be the $(1/2, \delta, \epsilon)$-accurate sketching matrix in Lemma 5.6. The proof is based on Lemma 5.4 and Lemma 5.5, which follows from results derived in previous work [202], [203]. Lemma 5.4 bounds the sketch size sufficient to reach certain matrix multiplication accuracy, while Lemma 5.5 bounds the singular values of the matrix obtained from applying TensorSketch to a matrix with orthonormal columns. We direct readers to prior work for a detailed proof of Lemma 5.4, but provide a simple proof of Lemma 5.5 by application of

Lemma 5.4.

Lemma 5.4 (Approximate Matrix Multiplication with TensorSketch [202])**.** *Given matrices* $\mathbf{P} \in \mathbb{R}^{s^{N-1} \times R^{N-1}}$ *and* $\mathbf{B} \in \mathbb{R}^{s^{N-1} \times n}$*. Let* $\mathbf{S} \in \mathbb{R}^{m \times s^{N-1}}$ *be an order* $N-1$ *TensorSketch matrix. For* $m \geq (2+3^{N-1})/(\epsilon^2 \delta)$*, the approximation error bound,*

$$\|\mathbf{P}^T\mathbf{S}^T\mathbf{S}\mathbf{B} - \mathbf{P}^T\mathbf{B}\|_F^2 \leq \epsilon^2 \cdot \|\mathbf{P}\|_F^2 \cdot \|\mathbf{B}\|_F^2, \tag{5.47}$$

holds with probability at least $1-\delta$*.*

Lemma 5.5 (Singular Value Bound for TensorSketch [203])**.** *Given a full-rank matrix* $\mathbf{P} \in \mathbb{R}^{s^{N-1} \times R^{N-1}}$ *with* $s > R$*, and* $\mathbf{B} \in \mathbb{R}^{s^{N-1} \times n}$*. Let* $\mathbf{S} \in \mathbb{R}^{m \times s^{N-1}}$ *be an order* $N-1$ *TensorSketch matrix. For* $m \geq R^{2(N-1)}(2+3^{N-1})/(\gamma^2\delta)$*, each singular value* σ *of* $\mathbf{S}\mathbf{Q}_P$ *satisfies*

$$1-\gamma \leq \sigma^2 \leq 1+\gamma \tag{5.48}$$

with probability at least $1-\delta$*, where* $\mathbf{Q}_P$ *is an orthonormal basis for the column space of* $\mathbf{P}$*.*

Proof. Since $\mathbf{Q}_P$ is an orthonormal basis for $\mathbf{P}$, $\mathbf{Q}_P^T\mathbf{Q}_P = \mathbf{I}$, and $\|\mathbf{Q}_P\|_F^2 = R^{N-1}$. Based on Lemma 5.4, for $m \geq R^{2(N-1)}(2+3^{N-1})/(\gamma^2\delta)$, with probability at least $1-\delta$, we have

$$\left\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{S}\mathbf{Q}_P - \mathbf{Q}_P^T\mathbf{Q}_P\right\|_F^2 = \left\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{S}\mathbf{Q}_P - \mathbf{I}\right\|_F^2 \leq \frac{\gamma^2}{R^{2(N-1)}} \cdot \|\mathbf{Q}_P\|_F^4 = \gamma^2. \tag{5.49}$$

Therefore,

$$\left\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{S}\mathbf{Q}_P - \mathbf{I}\right\|_2 \leq \left\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{S}\mathbf{Q}_P - \mathbf{I}\right\|_F \leq \gamma, \tag{5.50}$$

which means the singular values of $\mathbf{S}\mathbf{Q}_P$ satisfy $1-\gamma \leq \sigma^2 \leq 1+\gamma$. Q.E.D.

The previous two lemmas can be combined to demonstrate that the TensorSketch matrix provides an accurate sketch within our analytical framework.

Lemma 5.6 ($(1/2,\delta,\epsilon)$-accurate TensorSketch Matrix)**.** *Given the sketch size,*

$$m = \mathcal{O}\left((R^{(N-1)} \cdot 3^{N-1})/\delta \cdot (R^{(N-1)} + 1/\epsilon^2)\right), \tag{5.51}$$

an order $N-1$ *TensorSketch matrix* $\mathbf{S} \in \mathbb{R}^{m \times s^{N-1}}$ *is a* $(1/2,\delta,\epsilon)$*-accurate sketching matrix for any full rank matrix* $\mathbf{P} \in \mathbb{R}^{s^{N-1} \times R^{N-1}}$*.*

Proof. Based on Lemma 5.5 with $\gamma = 1/2$, for

$$m \geq R^{2(N-1)}(2+3^{N-1})/(1/4 \cdot \delta/2) = \mathcal{O}\left((R^{2(N-1)} \cdot 3^{N-1})/\delta\right), \tag{5.52}$$

(5.27) in Definition 5.6 will hold. Based on Lemma 5.4, for $m \geq R^{N-1}(2+3^{N-1})/(\epsilon^2\delta)$,

$$\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{S}\mathbf{B} - \mathbf{Q}_P^T\mathbf{B}\|_F^2 \leq \frac{\epsilon^2}{R^{N-1}} \cdot \|\mathbf{Q}_P\|_F^2 \cdot \|\mathbf{B}\|_F^2 = \epsilon^2\|\mathbf{B}\|_F^2, \tag{5.53}$$

thus (5.28) in Definition 5.6 will hold. Therefore, we need

$$\begin{aligned} m &= \mathcal{O}\left((R^{2(N-1)} \cdot 3^{N-1})/\delta + (R^{(N-1)} \cdot 3^{N-1})/(\epsilon^2\delta)\right) \\ &= \mathcal{O}\left((R^{(N-1)} \cdot 3^{N-1})/\delta \cdot (R^{(N-1)} + 1/\epsilon^2)\right). \end{aligned} \tag{5.54}$$

Q.E.D.

Using Lemma 5.6, we can then easily derive the upper bounds for both unconstrained and rank-constrained linear least squares with TensorSketch.

Theorem 5.5 (TensorSketch for Unconstrained Linear Least Squares)**.** *Given a full-rank matrix* $\mathbf{P} \in \mathbb{R}^{s^{N-1} \times R^{N-1}}$ *with* $s > R$, *and* $\mathbf{B} \in \mathbb{R}^{s^{N-1} \times n}$. *Let* $\mathbf{S} \in \mathbb{R}^{m \times s^{N-1}}$ *be an order* $N-1$ *TensorSketch matrix. Let* $\widetilde{\mathbf{X}}_{opt} = \arg\min_{\mathbf{X}} ||\mathbf{SPX} - \mathbf{SB}||_F$ *and* $\mathbf{X}_{opt} = \arg\min_{\mathbf{X}} \|\mathbf{PX} - \mathbf{B}\|_F$. *With*

$$m = \mathcal{O}\left((R^{(N-1)} \cdot 3^{N-1})/\delta \cdot (R^{(N-1)} + 1/\epsilon)\right), \tag{5.55}$$

the approximation error bound, $\left\|\mathbf{A}\widetilde{\mathbf{X}}_{opt} - \mathbf{B}\right\|_F^2 \leq (1 + \mathcal{O}(\epsilon)) \left\|\mathbf{A}\mathbf{X}_{opt} - \mathbf{B}\right\|_F^2$, *holds with probability at least* $1 - \delta$.

Proof. Based on Lemma 5.2, to prove this theorem, we derive the sketch size m sufficient to make the sketching matrix $(1/2, \delta, \sqrt{\epsilon})$-accurate. According to Lemma 5.6, the sketch size (5.55) is sufficient for being $(1/2, \delta, \sqrt{\epsilon})$-accurate. Q.E.D.

Proof of Theorem 5.1. Based on Theorem 5.4, to prove this theorem, we derive the sketch size m sufficient to make the sketching matrix $(1/2, \delta, \epsilon)$-accurate. According to Lemma 5.6, the sketch size

$$m = \mathcal{O}\left((R^{(N-1)} \cdot 3^{N-1})/\delta \cdot (R^{(N-1)} + 1/\epsilon^2)\right) \tag{5.56}$$

is sufficient for being $(1/2, \delta, \epsilon)$-accurate. Q.E.D.

5.10.4 Leverage Score Sampling for Unconstrained & Rank-constrained Least Squares

In this section, we first give the sketch size bound that is sufficient for the leverage score sampling matrix to be ab $(1/2, \delta, \epsilon)$-accurate sketching matrix according to Lemma 5.9. Using Lemma 5.9, we can then easily derive the upper bounds for both unconstrained and rank-constrained linear least squares with leverage score sampling. To establish these results, we leverage two lemmas. Lemma 5.7 bounds the sketch size sufficient to reach certain matrix multiplication accuracy, while Lemma 5.8 bounds the singular values of the sketched matrix obtained from applying leverage score sampling to a matrix with orthonormal columns. These first two lemmas follow from prior work, and we direct readers to references for detailed proofs of both lemmas.

Lemma 5.7 (Approximate Matrix Multiplication with Leverage Score Sampling [82])**.** *Given matrices* $\mathbf{P} \in \mathbb{R}^{s^{N-1} \times R^{N-1}}$ *consists of orthonormal columns and* $\mathbf{B} \in \mathbb{R}^{s^{N-1} \times n}$*. Let* $\mathbf{S} \in \mathbb{R}^{m \times s^{N-1}}$ *be a leverage score sampling matrix for* $\mathbf{P}$*. For* $m \geq 1/(\epsilon^2 \delta)$*, the approximation error bound,*

$$\|\mathbf{P}^T\mathbf{S}^T\mathbf{S}\mathbf{B} - \mathbf{P}^T\mathbf{B}\|_F^2 \leq \epsilon^2 \cdot \|\mathbf{P}\|_F^2 \cdot \|\mathbf{B}\|_F^2, \tag{5.57}$$

holds with probability at least $1-\delta$*.*

Lemma 5.8 (Singular Value Bound for Leverage Score Sampling [60])**.** *Given a full-rank matrix* $\mathbf{P} \in \mathbb{R}^{s^{N-1} \times R^{N-1}}$ *with* $s > R$*, and* $\mathbf{B} \in \mathbb{R}^{s^{N-1} \times n}$*. Let* $\mathbf{S} \in \mathbb{R}^{m \times s^{N-1}}$ *be a leverage score sampling matrix for* $\mathbf{P}$*. For* $m = \mathcal{O}\left(R^{(N-1)} \log(R^{(N-1)}/\delta)/\gamma^2\right) = \widetilde{O}\left(R^{(N-1)}/\gamma^2\right)$*, each singular value* σ *of* $\mathbf{S}\mathbf{Q}_P$ *satisfies*

$$1 - \gamma \leq \sigma^2 \leq 1 + \gamma \tag{5.58}$$

with probability at least $1-\delta$*, where* $\mathbf{Q}_P$ *is an orthonormal basis for the column space of* $\mathbf{P}$*.*

Lemma 5.9 ($(1/2, \delta, \epsilon)$-accurate Leverage Score Sampling Matrix)**.** *Let the sketch size* $m = \mathcal{O}\left(R^{N-1}/(\epsilon^2\delta)\right)$*, then the leverage score sampling matrix* $\mathbf{S} \in \mathbb{R}^{m \times s^{N-1}}$ *is a* $(1/2, \delta, \epsilon)$*-accurate sketching matrix for the full-rank matrix* $\mathbf{P} \in \mathbb{R}^{s^{N-1} \times R^{N-1}}$*.*

Proof. Based on Lemma 5.8 with $\gamma = 1/2$, for $m = \widetilde{O}\left(R^{(N-1)}\right)$, (5.27) in Definition 5.6 will hold. Based on Lemma 5.7, for $m = \mathcal{O}\left(R^{N-1}/(\epsilon^2\delta)\right)$,

$$\|\mathbf{Q}_P^T\mathbf{S}^T\mathbf{S}\mathbf{B} - \mathbf{Q}_P^T\mathbf{B}\|_F^2 \leq \frac{\epsilon^2}{R^{N-1}} \cdot \|\mathbf{Q}_P\|_F^2 \cdot \|\mathbf{B}\|_F^2 = \epsilon^2\|\mathbf{B}\|_F^2, \tag{5.59}$$

thus (5.28) in Definition 5.6 will hold. Thus we need $m = \widetilde{O}\left(R^{(N-1)}\right) + \mathcal{O}\left(R^{N-1}/(\epsilon^2\delta)\right) = \mathcal{O}\left(R^{N-1}/(\epsilon^2\delta)\right)$. Q.E.D.

Theorem 5.6 (Leverage Score Sampling for Unconstrained Linear Least Squares)**.** *Given a full-rank matrix* $\mathbf{P} \in \mathbb{R}^{s^{N-1} \times R^{N-1}}$ *with* $s > R$*, and* $\mathbf{B} \in \mathbb{R}^{s^{N-1} \times n}$*. Let* $\mathbf{S} \in \mathbb{R}^{m \times s^{N-1}}$ *be a leverage score sampling matrix. Let* $\widetilde{\mathbf{X}}_{opt} = \arg\min_{\mathbf{X}} ||\mathbf{S}\mathbf{P}\mathbf{X} - \mathbf{S}\mathbf{B}||_F$ *and* $\mathbf{X}_{opt} =$

$\arg\min_{\mathbf{X}} \|\mathbf{PX}-\mathbf{B}\|_F$. *With*

$$m=\mathcal{O}\left(R^{N-1}/(\epsilon\delta)\right), \tag{5.60}$$

the approximation error bound, $\left\|\mathbf{A}\widetilde{\mathbf{X}}_{opt}-\mathbf{B}\right\|_F^2 \leq (1+\mathcal{O}(\epsilon))\left\|\mathbf{A}\mathbf{X}_{opt}-\mathbf{B}\right\|_F^2$*, holds with probability at least* $1-\delta$.

Proof. Based on Lemma 5.2, to prove this theorem, we derive the sample size m sufficient to make the sketching matrix $(1/2,\delta,\sqrt{\epsilon})$-accurate. According to Lemma 5.9, the sketch size (5.60) is sufficient for being $(1/2,\delta,\sqrt{\epsilon})$-accurate. Q.E.D.

Proof of Theorem 5.2. Based on Theorem 5.4, to prove this theorem, we derive the sketch size m sufficient to make the sketching matrix $(1/2,\delta,\epsilon)$-accurate. According to Lemma 5.9, the sketch size $\mathcal{O}\left(R^{N-1}/(\epsilon^2\delta)\right)$ is sufficient for being $(1/2,\delta,\epsilon)$-accurate. Q.E.D.

5.11 TENSORSKETCH FOR GENERAL CONSTRAINED LEAST SQUARES

In this section, we provide sketch size upper bound of TensorSketch for general constrained linear least squares problems.

Theorem 5.7 (TensorSketch for General Constrained Linear Least Squares)**.** *Given a full-rank matrix* $\mathbf{P}\in\mathbb{R}^{s^{N-1}\times R^{N-1}}$ *with* $s>R$*, and* $\mathbf{B}\in\mathbb{R}^{s^{N-1}\times n}$*. Let* $\mathbf{S}\in\mathbb{R}^{m\times s^{N-1}}$ *be an order* $N-1$ *TensorSketch matrix. Let* $\widetilde{\mathbf{X}}_{opt}=\arg\min_{\mathbf{X}\in\mathcal{C}}\|\mathbf{SPX}-\mathbf{SB}\|_F$*, and let* $\mathbf{X}_{opt}=\arg\min_{\mathbf{X}\in\mathcal{C}}\|\mathbf{PX}-\mathbf{B}\|_F$*. With*

$$m=\mathcal{O}\left(nR^{2(N-1)}\cdot 3^{N-1}/(\epsilon^2\delta)\right), \tag{5.61}$$

the approximation error bound,

$$\left\|\mathbf{P}\widetilde{\mathbf{X}}_{opt}-\mathbf{B}\right\|_F^2 \leq (1+\mathcal{O}(\epsilon))\left\|\mathbf{P}\mathbf{X}_{opt}-\mathbf{B}\right\|_F^2, \tag{5.62}$$

holds with probability at least $1-\delta$.

Proof. The proof is similar to the analysis performed in [60] for other sketching techniques. Let the ith column of $\mathbf{B},\mathbf{X}$ be denoted $\mathbf{b}_i,\mathbf{x}_i$, respectively. We can express each column in the residual $\mathbf{PX}-\mathbf{B}$ as

$$\mathbf{P}\mathbf{x}_i-\mathbf{b}_i=\begin{bmatrix}\mathbf{P} & \mathbf{b}_i\end{bmatrix}\begin{bmatrix}\mathbf{x}_i\\ -1\end{bmatrix}:=\widetilde{\mathbf{P}}^{(i)}\mathbf{y}_i. \tag{5.63}$$

Based on Lemma 5.5, let $m \geq n(R^{(N-1)}+1)^2(2+3^{N-1})/(\epsilon^2\delta)$, we have with probability at least $1-\delta/n$ that for some $i \in [n]$, each singular value σ of $\mathbf{SQ}_{\widetilde{P}^{(i)}}$ satisfies

$$1-\epsilon \leq \sigma^2 \leq 1+\epsilon. \tag{5.64}$$

This means for any $\mathbf{y}_i \in \mathbb{R}^{R^{N-1}+1}$, we have

$$(1-\epsilon)\left\|\widetilde{\mathbf{P}}^{(i)}\mathbf{y}_i\right\|_2^2 \leq \left\|\mathbf{S}\widetilde{\mathbf{P}}^{(i)}\mathbf{y}_i\right\|_2^2 \leq (1+\epsilon)\left\|\widetilde{\mathbf{P}}^{(i)}\mathbf{y}_i\right\|_2^2. \tag{5.65}$$

Using the union bound, (5.65) implies that with probability at least $1-\delta$,

$$(1-\epsilon)\left\|\mathbf{P}\widetilde{\mathbf{X}}_{\text{opt}}-\mathbf{B}\right\|_F \leq \left\|\mathbf{SP}\widetilde{\mathbf{X}}_{\text{opt}}-\mathbf{SB}\right\|_F \quad \text{and} \quad \left\|\mathbf{SPX}_{\text{opt}}-\mathbf{SB}\right\|_F \leq (1+\epsilon)\left\|\mathbf{PX}_{\text{opt}}-\mathbf{B}\right\|_F. \tag{5.66}$$

Therefore, we have

$$\begin{aligned}\left\|\mathbf{P}\widetilde{\mathbf{X}}_{\text{opt}}-\mathbf{B}\right\|_F &\leq \frac{1}{1-\epsilon}\left\|\mathbf{SP}\widetilde{\mathbf{X}}_{\text{opt}}-\mathbf{SB}\right\|_F \leq \frac{1}{1-\epsilon}\left\|\mathbf{SPX}_{\text{opt}}-\mathbf{SB}\right\|_F \\ &\leq \frac{1+\epsilon}{1-\epsilon}\left\|\mathbf{PX}_{\text{opt}}-\mathbf{B}\right\|_F = (1+\mathcal{O}(\epsilon))\left\|\mathbf{PX}_{\text{opt}}-\mathbf{B}\right\|_F.\end{aligned} \tag{5.67}$$

Therefore, $m = \mathcal{O}\left(nR^{2(N-1)}\cdot 3^{N-1}/(\epsilon^2\delta)\right)$ is sufficient for the approximation in (5.62). Q.E.D.

Chapter 6: SKETCHING FOR TENSOR NETWORKS

The algorithm proposed in Chapter 5 is particularly efficient for sketching a Kronecker product of matrices. In this Chapter, we propose more general algorithms that can efficiently sketch tensor networks with arbitrary structures.

We design algorithms to efficiently sketch general data tensor networks ($\mathbf{x}$) such that each dimension to be sketched has a size lower bounded by the sketch size and is a dimension of only one tensor [204]. One of such data tensor networks is shown in Fig. 6.1. In particular, we look at the following question.

For arbitrary data with a tensor network structure of interest, can we automatically sketch the data into one tensor with Gaussian tensor network embeddings that are accurate, have low sketch size, and also minimize the sketching asymptotic computational cost?

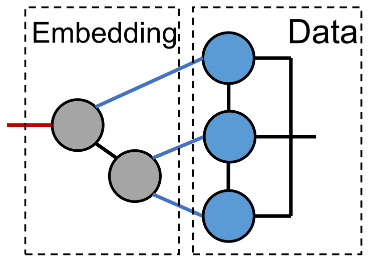


Figure 6.1: Illustration of target data and embedding. Blue edges have larger weights than the red edge.

Previous work Existing works discuss tensor network embeddings with more efficient sketch size than Kronecker and Khatri-Rao product structure, such as tensor train [205] and balanced binary tree [44]. In particular, Ahle et al. [44] designed a balanced binary tree structured embedding and showed that the sketch size sufficient for (ϵ, δ)-accurate embedding can have only linear dependence on N. Using this embedding to sketch Kronecker product structured data yields a sketching cost that only has a polynomial dependence on both N and s. However, for data with other tensor network structures, these embeddings may not be the most computationally efficient. One related work [206] builds tree tensor network embeddings based on Countsketch for getting low-rank tensor network approximation of an input tensor.

Our contributions Different from existing works [44], [81], [172], [206] that construct the embedding based on fast sketching techniques, including Countsketch [175], Tensorsketch [174], and fast Johnson-Lindenstraus (JL) transform using fast Fourier transform [207], we discuss the case where each tensor in the embedding contains i.i.d. Gaussian random elements. Gaussian-based embeddings yield larger computational cost, but have the most efficient sketch size for both unconstrained and constrained optimization problems [208], [209]. This choice also enables us use a simple computational model to analyze the sketching cost, where tensor contractions are performed with classical dense matrix multiplication algorithms. Note that a related work [206] builds tree tensor network embeddings based on Countsketch for getting low-rank tensor network approximation of an input tensor.

While we allow for the data tensor network to be a hypergraph, we consider only graph embeddings, (detailed definition in Section 6.1), which include tree embeddings that have been previously studied [44], [89], [210]. Each one of these embeddings consisting of N_E tensors can be reduced to a sequence of N_E sketches (random sketching matrices). In Section 6.2, we show that if each of these sketches is $(\epsilon/\sqrt{N_E}, \delta)$-accurate, then the embedding is at least (ϵ, δ)-accurate.

In Section 6.3, we provide an algorithm to sketch input data with an embedding that not only satisfies the (ϵ, δ)-accurate sufficient condition, but is computationally efficient and has low sketch size. Given a data tensor network and one data contraction tree T_0, this algorithm outputs a sketching contraction tree that is constrained on T_0. This setting is useful for application of sketching to alternating optimization in tensor-related problems, such as tensor decompositions. In alternating optimization, multiple contraction trees of the data x are chosen in an alternating order to form multiple optimization subproblems, each updating part of the variables [59], [94], [211]. Designing embeddings under the constraint can help reuse contracted intermediates across subproblems.

The sketch size of the embedding used in the algorithm has a linear dependence on the number of sketching dimensions of the input. As to the sketching asymptotic computational cost, within all constrained sketching contraction trees with embeddings satisfying the (ϵ, δ)-accurate sufficient condition and only have one output sketch dimension, this algorithm achieves asymptotic cost within a factor of $O(\sqrt{m})$ of the lower bound, where m is the sketch size. When the input data tensor network structure is a graph, the factor improves to $O(m^{0.375})$. In addition, when each tensor in the input data has a dimension to be sketched, such as Kronecker product input and tensor train input, this algorithm yields the optimal sketching asymptotic cost.

At the end of Section 6.3, we look at cases where the widely discussed tree tensor network embeddings are efficient in terms of the sketching computational cost. We show for input data graphs such that each data tensor has a dimension to be sketched and each contraction in the given data contraction tree T_0 contracts dimensions with size being at least the sketch size, sketching with tree embeddings can achieve the optimal asymptotic cost.

In Section 6.4, we apply our sketching algorithm to two applications, CANDECOMP/-PARAFAC (CP) tensor decomposition [7], [13] and tensor train rounding [12]. We present a new sketching-based alternating least squares (ALS) algorithm for CP decomposition. Compared to existing sketching-based ALS algorithm, this algorithm yields better asymptotic computational cost under several regimes, such as when the CP rank is much lower than each dimension size (the length/number of elements in each dimension) of the input tensor. We also provide analysis on the recently introduced randomized tensor train rounding algorithm [89]. We show that the tensor train embedding used in that algorithm satisfies the accuracy sufficient condition in Section 6.2 and yields the optimal sketching asymptotic cost, implying that this is an efficient algorithm, and embeddings with other structures cannot achieve lower asymptotic cost.

6.1 DEFINITIONS

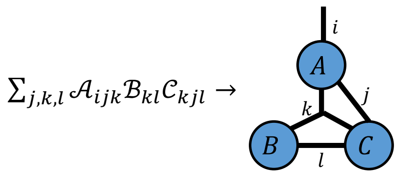

$\sum_{j,k,l} \mathcal{A}_{ijk}\mathcal{B}_{kl}\mathcal{C}_{kjl} \rightarrow$
i
A
k
j
B
C
l

Figure 6.2: An example of tensor diagram notation.

We introduce some tensor network notation here, and provide additional definitions and background in Section 6.7. The structure of a tensor network can be described by an undirected hypergraph $G = (V, E, w)$, also called tensor diagram. Each hyperedge $e \in E$ may be adjacent to either one or at least two vertices, and we refer to hyperedges with a dangling end (one end not adjacent to any vertex) as uncontracted hyperedges, and those without dangling end as contracted hyperedges. We refer to the cardinality of a hyperedge as its number of ends. An example is shown in Fig. 6.2. We use w to denote a function such that for each $e \in E$, $w(e) = \log(s)$ is the natural logarithm of the dimension sizes represented by hyperedge e. For a hyperedge set E, we use $w(E) = \sum_{e\in E} w(e)$ to denote the weighted sum of the hyperedge set.

A tensor network embedding is the matricization of a tensor described by a tensor network, and each embedding can be described by $S = (G_E, \bar{E})$, where $G_E = (V_E, E_E, w)$ shows the embedding graph structure and $\bar{E} \subseteq E_E$ is the edge set connecting data and the embedding. In this work we only discuss the case where G_E is a graph, such that each uncontracted edge in E_E is adjacent to one vertex and contracted edge in E_E is adjacent to two vertices. Let

$E_1 \subset E_E$ be the subset of uncontracted edges, S is a matricization such that uncontracted dimensions in $\bar{E}$ are grouped into the column of the matrix, and dimensions in E_1 are grouped into the row. We use $N = |\bar{E}|$ to denote the order of the embedding, and $m = \exp(w(E_1))$ denotes the output sketch size. We use $G_D = (V_D, E_D, w)$ to represent the data tensor network structure, and use $G = (G_D, G_E)$ to denote the overall tensor network structure.

Within the tensor network $G = (V, E, w)$, the contraction between two tensors represented by $v_i, v_j \in V$ is denoted by (v_i, v_j). The contraction between two tensors that are the contraction outputs of $W_i \subset V$, $W_j \subset V$, respectively, is denoted by (W_i, W_j). A contraction tree on the tensor network $G = (V, E, w)$ is a rooted binary tree $T_B = (V_B, E_B)$ showing how the tensor network is fully contracted. Each vertex in V_B can be represented by a subset of the vertices, $W \subseteq V$, and denotes the contraction output of W. The two children of W, denoted as W_1 and W_2, must satisfy $W_1 \cup W_2 = W$. Each leaf vertex must have $|W| = 1$, and the root vertex is represented by V. Any topological sort of the contraction tree represents a contraction path (order) of the tensor network.

6.2 SUFFICIENT CONDITION FOR ACCURATE EMBEDDING

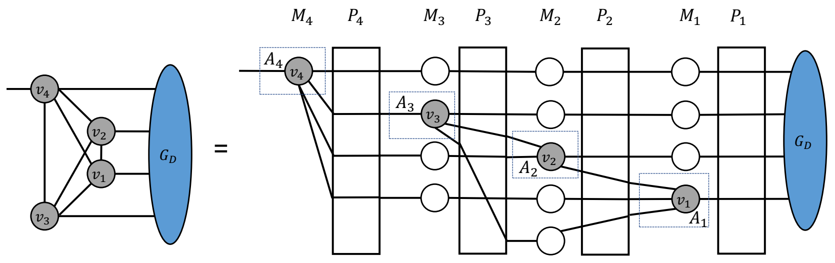


Figure 6.3: Illustration of embedding linearization. Each gray vertex denotes a tensor of the embedding, each white vertex denotes an identity matrix, and each white box denotes a permutation matrix.

We consider the scenario where the data tensor networks have a general hypergraph structure, while the embeddings have a graph structure, thus some embeddings, such as those with a Khatri-Rao product structure [205], [208], are not considered in this work. Such embeddings can be linearized to a sequence of sketches. Let $N_E = |V_E|$ denote the number of vertices in the embedding, in each linearization, each vertex is given an unique index $i \in [N_E]$[7] and denoted v_i. The ith tensor is denoted by $\boldsymbol{\mathcal{A}}_i$, and $\mathbf{A}_i$ denotes its matricization where we combine all uncontracted dimensions and contracted dimensions connected to $\boldsymbol{\mathcal{A}}_j$ with $j > i$ into the row, and other dimensions into the column. The embedding can then be

[7]Throughout the paper we use $[N]$ to denote $\{1, \ldots, N\}$.

represented as a chain of multiplications, $\mathbf{S} = \mathbf{M}_{N_E}\mathbf{P}_{N_E} \cdots \mathbf{M}_1\mathbf{P}_1$, where $\mathbf{M}_i$ is the Kronecker product of identity matrices with $\mathbf{A}_i$ for $i \in [N_E]$, and $\mathbf{P}_i$ is a permutation matrix. We illustrate the linearization in Fig. 6.3 using a fully connected tensor network embedding. We show in Theorem 6.1 a sufficient condition for embeddings to be (ϵ, δ)-accurate.

Theorem 6.1 ((ϵ, δ)-accurate sufficient condition)**.** *Consider a Gaussian tensor network embedding where there exists a linearization such that each $\mathbf{A}_i$ for $i \in [N_E]$ has row size $\Omega(N_E \log(1/\delta)/\epsilon^2)$. Then the tensor network embedding is (ϵ, δ)-accurate.*

Proof. Based on the composition rules of JL moment [212], [213] in Lemma 6.2 and Lemma 6.3 in the appendix, in the linearization all $\mathbf{M}_i\mathbf{P}_i$ satisfy the strong $\left(\frac{\epsilon}{L\sqrt{2N}}, \delta\right)$-JL moment property so $\mathbf{S}$ satisfies the strong (ϵ, δ)-JL moment property. This implies the embedding is (ϵ, δ)-accurate. Q.E.D.

Theorem 6.1 is a sufficient (but not necessary) condition for constructing (ϵ, δ)-accurate embedding. It also implies that specific tree embeddings are (ϵ, δ)-accurate, as we show below.

Corollary 6.1. *Consider a Gaussian embedding containing a tree tensor network structure, where there is only one output sketch dimension with size $m = \Theta\left(N_E \log(1/\delta)/\epsilon^2\right)$, and each dimension within the embedding has size m. Then the embedding is (ϵ, δ)-accurate.*

Proof. Consider the linearization such that vertices are labelled based on the reversed ordering of a breath-first search from the vertex adjacent to the edge associated with the output sketch dimension. Each $\mathbf{A}_i$ has row size $m = \Theta\left(N_E \log(1/\delta)/\epsilon^2\right)$ thus the embedding satisfies Theorem 6.1. Q.E.D.

One special case of Corollary 6.1 is the tensor train [12] (also called matrix product states (MPS) [214]) embedding, where the embedding tensor network has a 1D structure along with an output dimension adjacent to one of the endpoint tensors. Tensor train is widely used to efficiently represent high dimensional tensors in multiple applications, including numerical PDEs [215], [216], quantum physics [37], high-dimensional data analysis [217], [218] and machine learning [17], [219], [220]. Since the tensor train embedding contains N vertices, Corollary 6.1 directly implies that a sketch size of $m = \Theta\left(N \log(1/\delta)/\epsilon^2\right)$ is sufficient for the MPS embedding to be (ϵ, δ)-accurate. This embedding has already been used in applications including tensor train rounding [89] and low rank approximation of matrix product operators [210].

Note that the tensor train embedding introduced in this work and [89] adds an output sketch dimension to the standard tensor train, and restricts the tensor train rank to be the sketch size m. This is different from the recent work by Rakhshan and Rabusseau [205], where they construct an embedding consisting of m independent tensor trains, each one with a tensor train rank of R. A sketch size upper bound of $m = \Theta\left(1/\epsilon^2 \cdot (1 + 2/R)^N \log^{2N}(1/\delta)\right)$ is derived for that embedding to be (ϵ, δ)-accurate. However, this bound has an exponential dependence on N.

6.3 A SKETCHING ALGORITHM WITH EFFICIENT COMPUTATIONAL COST AND SKETCH SIZE

We find Gaussian tensor network embeddings G_E that both have efficient sketch size and yield efficient computational cost. We are given a specific data tensor network G_D that implicitly represents a matrix $\mathbf{M} \in \mathbb{R}^{s_1 s_2 \ldots s_N \times t}$, and want to sketch the row dimension of the matrix. We assume that size of each dimension to be sketched, s_i for $i \in [N]$, is greater than the sketch size m, and each one of these dimensions is adjacent to only one tensor. The goal is to find a Gaussian embedding G_E satisfying the following properties.

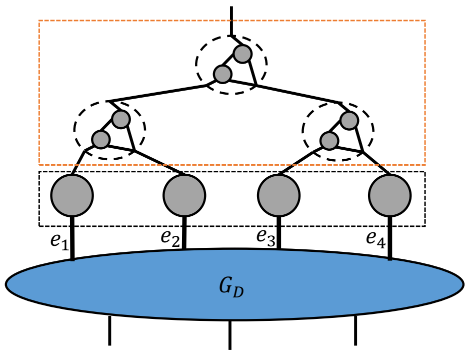


Figure 6.4: Illustration of the embedding. The black box includes the Kronecker product embedding and the orange box includes the embedding containing a binary tree of small tensor networks.

- $G_E \in \mathcal{G}^{(\epsilon,\delta)}$, where $\mathcal{G}^{(\epsilon,\delta)}$ contains all embeddings not only satisfying the (ϵ, δ)-accurate sufficient condition in Theorem 6.1, but also only have one output sketch dimension ($|E_1| = 1$) with size $m = \Theta\left(N_E \log(1/\delta)/\epsilon^2\right)$. This guarantees that the embedding is accurate and the output sketch size is linear w.r.t. the number of vertices in G_E. Note that although the data can be a hypergraph, the embeddings considered in $\mathcal{G}^{(\epsilon,\delta)}$ are defined on graphs.

- To fully contract the tensor network system (G_D, G_E), this embedding yields a contraction tree with the *optimal asymptotic contraction cost* under a fixed data contraction tree. The data contraction tree constraint is useful for application of sketching to alternating optimization algorithms, as we will discuss in Section 6.4. This can be written as an optimization problem below,

$$\min_{G_E}\min_{T_B} C_a(T_B(G_D, G_E)), \quad \text{s.t. } G_E \in \mathcal{G}^{(\epsilon,\delta)}, \quad T_0(G_D) \subset T_B(G_D, G_E), \tag{6.1}$$

where $T_B(G_D, G_E)$ denotes a contraction tree of the tensor network (G_D, G_E), C_a denotes the asymptotic computational cost, and $T_0(G_D) \subset T_B(G_D, G_E)$ means the contraction tree T_B is constrained on T_0 (the detailed definition and a simple example are shown in Definition 6.1 and Section 6.8.1, respectively).

Definition 6.1 (Constrained contraction tree)**.** *Given $G = (G_E, G_D)$ and a contraction tree T_0 of G_D, the contraction tree T_B for G is constrained on T_0 if for each contraction $(A, B) \in T_0$, there must exist one contraction $(\hat{A}, \hat{B}) \in T_B$, such that $\hat{A} \cap V_D = A$ and $\hat{B} \cap V_D = B$.*

Algorithm We propose an algorithm to sketch tensor network data with an embedding containing two parts, a Kronecker product embedding and an embedding containing a binary tree of small tensor networks. The embedding is illustrated in Fig. 6.4. The Kronecker product embedding consists of N Gaussian random matrices and is used to reduce the weight of each edge in $\bar{E}$, the set of edges to be sketched. The binary tree structured embedding consists of $N-1$ *small tensor networks*, each represented by one binary tree vertex. Each small tensor network is used to effectively sketch the contraction of pairs of tensors adjacent to edges in $\bar{E}$. The embedding with a binary tree structure may not be a binary tree tensor network, since each tree vertex is not restricted to represent one tensor. The binary tree is chosen to be consistent with the dimension tree of the data contraction tree T_0, which is a directed binary tree showing the way edges in $\bar{E}$ are merged onto the same tensor in T_0. The detailed definition of dimension tree is in Section 6.8.1.

We first introduce some notation before presenting the algorithm. Consider a given input data tensor network $G_D = (V_D, E_D, w)$ and its given data contraction tree, T_0. Below we let $\bar{E} = \{e_1, e_2, \ldots, e_N\}$ to denote the edges to be sketched. Let $N_D = |V_D|$. Based on the definition we have $N \leq N_D$ and T_0 contains $N_D - 1$ contractions. Let one contraction path of T_0, which is a topological sort of the contractions in T_0, be expressed as

$$\left\{(U_1, V_1), \ldots, (U_{N_D-1}, V_{N_D-1})\right\}, \tag{6.2}$$

where (U_i, V_i) represents the contraction of two intermediate tensors represented by two subset of vertices $U_i, V_i \subset V_D$. The $N_D - 1$ contractions can be categorized into $N + 2$ sets, $\mathcal{D}(e_1), \dots, \mathcal{D}(e_N), \mathcal{S}, \mathcal{I}$, as follows, and these sets are illustrated with an example in Section 6.9.

- Consider contractions (U_i, V_i) such that both U_i and V_i are adjacent to edges in $\bar{E}$. $\mathcal{S}$ contains all contractions with this property.
- Consider contractions (U_i, V_i) such that the only edge in $\bar{E}$ that is adjacent to the contraction output is e_j, $\bar{E}(U_i \cup V_i) = \{e_j\}$. We let $\mathcal{D}(e_j)$ contains contractions with this property. When $\mathcal{D}(e_j)$ is not empty, we let $X(e_j) \subset V$ represent the sub network contracted by $\mathcal{D}(e_j)$. When $\mathcal{D}(e_j)$ is empty, we let $X(e_j) = v_j$, where v_j is the vertex in the data graph adjacent to e_j.
- The remaining contractions in the contraction tree include (U_i, V_i) such that both U_i and V_i are not adjacent to $\bar{E}$, and contractions where U_i or V_i is adjacent to at least two edges in $\bar{E}$, and the other one is not adjacent to any edge in $\bar{E}$. We let $\mathcal{I}$ contain these contractions.

Algorithm 6.1: Sketching algorithm

```
1:  Input: Input data tensor network $G_D$, data contraction tree $T_0$ expressed in (6.2)
2:  for each $e_i \in \bar{E}$ do
3:      // Sketch with Kronecker product embedding
4:      $W \leftarrow$ contract and sketch $X(e_j)$
5:      Replace the contraction output of $X(e_j)$ by $W$ in $T_0$
6:  end for
7:  for each contraction $(U_i, V_i)$ in $\mathcal{S} \cup \mathcal{I}$ do
8:      if $i \in \mathcal{S}$ then
9:          // Sketch with binary tree embedding
10:         $W_i \leftarrow$ contract and sketch $(U_i, V_i)$ (detailed in Section 6.9.1)
11:     else
12:         $W_i \leftarrow$ contract$(U_i, V_i)$
13:     end if
14:     Replace the contraction output of $(U_i, V_i)$ by $W_i$ in $T_0$
15: end for
16: return $W_{N_D - 1}$
```

The sketching algorithm is shown in Algorithm 6.1, and the details are as follows,

- One matrix in the Kronecker product embedding is used to sketch the sub data network $X(e_j)$, which guarantees that two sketch dimensions to be merged onto one tensor will both have size $\Theta(N \log(1/\delta)/\epsilon^2)$. For the case where $\mathcal{D}(e_j) = \emptyset$, we directly sketch

$X(e_j) = v_j$ using an embedding matrix. For the case where $\mathcal{D}(e_j) \neq \emptyset$, we select $k(e_j) \in \mathcal{D}(e_j)$ and apply the sketching matrix during the contraction $(U_{k(e_j)}, V_{k(e_j)})$. The value of $k(e_j)$ is selected via an exhaustive search over all $|\mathcal{D}(e_j)|$ contractions, so that sketching $X(e_j)$ has the lowest asymptotic cost.

- One small tensor network (denoted as Z_i) represented by a binary tree vertex in the binary tree structured embedding is used to sketch the contraction (U_i, V_i) when $i \in \mathcal{S}$, which means that both U_i and V_i are adjacent to $\bar{E}$. Let $\hat{U}_i, \hat{V}_i$ denote the sketched U_i and V_i formed in previous contractions in the sketching contraction tree T_B, such that $\hat{U}_i \cap V_D = U_i$ and $\hat{V}_i \cap V_D = V_i$, the structure of Z_i is determined so that the asymptotic cost to sketch $(\hat{U}_i, \hat{V}_i)$ is minimized under the constraint that Z_i is in $\mathcal{G}^{(\epsilon/\sqrt{N},\delta)}$, so that it satisfies the $(\epsilon/\sqrt{N}, \delta)$-accurate sufficient condition and only has one output dimension. In Section 6.9.1, we provide an algorithm to construct Z_i containing 2 tensors, so that the output sketch size of Z_i is $\Theta(N \log(1/\delta)/\epsilon^2)$.

The total computational cost of Algorithm 6.1 consists of three components: the cost of determining the embedding structure, the cost of determining the sketching contraction tree, and the cost of sketching. The first two components are $O(N)$ and are therefore negligible in comparison to the cost of sketching.

Analysis of the algorithm The embedding constructed during Algorithm 6.1 contains $\Theta(N)$ vertices, and the output sketch size is $m = \Theta(N \log(1/\delta)/\epsilon^2)$. Therefore, the sketching result both has low sketch size and is (ϵ, δ)-accurate. Below we discuss the optimality of Algorithm 6.1 in terms of the sketching asymptotic computational cost. We first discuss the case when each vertex in the data tensor network is adjacent to an edge in $\bar{E}$.

Theorem 6.2. *For data tensor networks where each vertex is adjacent to an edge in $\bar{E}$, the asymptotic cost of Algorithm 6.1 is optimal w.r.t. the optimization problem in (6.1).*

We show the detailed proof of the theorem above in Section 6.10.1. Therefore, Algorithm 6.1 is efficient in sketching multiple widely used tensor network data, including tensor train, Kronecker product, and Khatri-Rao product. As we will discuss in Section 6.4, Algorithm 6.1 can be used to design efficient sketching-based ALS algorithm for CP tensor decomposition.

Note that the embedding in Algorithm 6.1 may not be a tree embedding. As we will show in Section 6.5, for cases including sketching a Kronecker product data, Algorithm 6.1 is more efficient than sketching with tree embeddings. On the other hand, for some data tensor networks, sketching with a tree embedding also yields the optimal asymptotic cost, which we will show in Theorem 6.4.

For general input data where each data vertex may not adjacent to an edge in $\bar{E}$, Algorithm 6.1 may not yield the optimal sketching asymptotic cost, but is within a factor of at most $O(\sqrt{m})$ from the cost lower bound. Below we show the theorem, and the detailed proof is in Section 6.10.2.

Theorem 6.3. *For any data tensor network G_D, the asymptotic cost of Algorithm 6.1 (denoted as c) satisfy $c = O\left(\sqrt{m} \cdot c_{opt}\right)$, where c_{opt} is the optimal asymptotic computational cost for the optimization problem* (6.1) *and $m = \Theta(N \log(1/\delta)/\epsilon^2)$. When G_D is a graph, $c = O\left(m^{0.375} \cdot c_{opt}\right)$.*

Efficiency of tree tensor network embedding We discuss cases where tree tensor network embeddings can be optimal w.r.t. the optimization problem in (6.1). Tree embeddings, in particular the tensor train embedding, have been widely discussed and used in prior work [89], [205], [210]. We design an algorithm to sketch with tree embeddings. The algorithm is similar to Algorithm 6.1, and the only difference is that for each contraction (U_i, V_i) with $i \in \mathcal{S}$, such that both U_i and V_i are adjacent to edges in $\bar{E}$, we sketch it with one tensor rather than a small network. Below, we present the optimality of the algorithm in terms of sketching asymptotic cost.

Theorem 6.4. *Consider G_D with each vertex adjacent to an edge to be sketched and its given contraction tree T_0. If each contraction in T_0 contracts dimensions with size being at least the sketch size, then sketching with tree embedding would yield the optimal asymptotic cost for* (6.1).

We present the proof of Theorem 6.4 in Section 6.11. As we will show in Section 6.5, for tensor network data with relatively large contracted dimension sizes such that the condition in Theorem 6.4 is satisfied, sketching with tree embedding yields a similar performance as Algorithm 6.1. However, for data where the condition in Theorem 6.4 is not satisfied, Algorithm 6.1 is more efficient. For example, when the data is a vector with a Kronecker product structure, sketching with Algorithm 6.1 yields a cost of $\Theta(\sum_{j=1}^{N} s_j m + N m^{2.5})$ and sketching with a tree embedding yields a cost of $\Theta(\sum_{j=1}^{N} s_j m + N m^3)$. We present the detailed analysis in Section 6.9.2 and Section 6.11.

Alternating least squares for CP decomposition On top of Algorithm 6.1, we propose a new sketching-based ALS algorithm for CP tensor decomposition. Throughout analysis we assume the input tensor is dense, and has order N and size $s \times \cdots \times s$, and the CP rank is R. The goal of CP decomposition is to minimize the objective function, $f(\mathbf{A}_1, \ldots, \mathbf{A}_N) = \left\|\boldsymbol{\mathcal{X}} - \sum_{r=1}^{R} \mathbf{A}_1(:,r) \circ \cdots \circ \mathbf{A}_N(:,r)\right\|_F^2$, where $\mathbf{A}_i \in \mathbb{R}^{s\times R}$ for $i \in [N]$ are called factor matrices, and $\boldsymbol{\mathcal{X}}$ denotes the input tensor. In each iteration of ALS, N subproblems are solved sequentially, and the ith subproblem can be formulated as $\mathbf{A}_i = \arg\min_{\mathbf{A}} \left\|L_i \mathbf{A}^T - R_i\right\|_F^2$, where $L_i = \mathbf{A}_1 \odot \cdots \odot \mathbf{A}_{i-1} \odot \mathbf{A}_{i+1} \odot \cdots \odot \mathbf{A}_N$ consists of a chain of Khatri-Rao products, and $R_i = \mathbf{X}_{(i)}^T$ is the transpose of ith matricization of $\boldsymbol{\mathcal{X}}$.

Multiple sketching-based randomized algorithms are proposed to accelerate each subproblem in CP-ALS [80], [82], [88]. The sketched problem can be formulated as $\mathbf{A}_i = \underset{\mathbf{A}}{\operatorname{argmin}} \left\|S_i L_i A^T - S_i R_i\right\|_F^2$, where S_i is an embedding. The goal is to design S_i such that the sketched subproblem can be solved efficiently and accurately. In Table 6.1, we summarize two state-of-the-art sketching methods for CP-ALS. Larsen and Kolda [82] propose a method that sketches the subproblem based on (approximate) leverage score sampling (LSS), but both the per-iteration computational cost and the sketch size sufficient for (ϵ, δ)-accurate solution has an exponential dependence on N, which is inefficient for decomposing high order tensors. Malik [88] proposes a method called recursive leverage score sampling for CP-ALS, where the embedding contains two parts, $S_i = S_{i,1} S_{i,2}$, and $S_{i,2}$ is an embedding with a binary tree structure proposed in [44] with sketch size $\Theta(NR^2/\delta)$, and $S_{i,1}$ performs approximate leverage score sampling on $S_{i,2} L_i$ with sketch size $\tilde{\Theta}(NR/\epsilon^2)$. This sketching method has a better dependence on R in terms of per-iteration cost. For both algorithms, the preparation cost shown in Table 6.1 denotes the cost to go over all elements in the tensor and initialize factor matrices using randomized range finder. As is shown in [82], [172], randomized range finder based initialization is critical for achieving accurate CP decomposition with sampling-based sketched ALS.

We propose a new sketching algorithm for CP-ALS based on Algorithm 6.1. Each S_i is generated on top of the data tensor network L_i and its given data contraction tree T_i, with the sketch size being $m = \Theta(NR\log(1/\delta)/\epsilon^2) = \tilde{\Theta}(NR/\epsilon^2)$. The contraction trees T_i for $i \in [N]$ are chosen in a fixed alternating order, such that the resulting embeddings S_i for $i \in [N]$ have common parts and allow reusing contraction intermediates. We leave the detailed analysis in Section 6.12.2.

The ALS per-iteration cost is $\Theta(N(m^{2.5}R + smR)) = \tilde{\Theta}(N^2(N^{1.5}R^{3.5}/\epsilon^3 + sR^2)/\epsilon^2)$. We present the detailed sketching algorithm and its cost analysis in Section 6.12.3. When

CP-ALS algorithm	Per-iteration cost	Sketch size (m)	Prep cost
Standard ALS	$\Theta(s^N R)$	/	/
LSS [82]	$\tilde{\Theta}(N(R^{N+1} + sR^N)/\epsilon^2)$	$\tilde{\Theta}(R^{N-1}/\epsilon^2)$	$\Theta(s^N)$
Recursive LSS [88]	$\tilde{\Theta}(N^2(R^4 + NsR^3/\epsilon)/\delta)$	$\Theta(NR^2/\delta)$ and $\tilde{\Theta}(R/(\epsilon\delta))$	$\Theta(s^N)$
Algorithm 6.1	$\tilde{\Theta}(N^2(N^{1.5}R^{3.5}/\epsilon^3 + sR^2)/\epsilon^2)$	$\tilde{\Theta}(NR/\epsilon^2)$	$\Theta(s^N m)$

Table 6.1: Comparison of asymptotic algorithmic complexity between standard CP-ALS, CP-ALS with leverage score sampling (LSS), CP-ALS with recursive leverage score sampling (recursive LSS), and sketching CP-ALS with Algorithm 6.1. The third column shows the sketch size sufficient for the sketched linear least squares to be $(1+\epsilon)$-accurate with probability at least $1-\delta$. By using $\tilde{\Theta}$, we neglect logarithmic factors, including $\log(R)$ and $\log(1/\delta)$.

performing a low-rank CP decomposition with $s \gg R^{1.5}$ and ϵ is not too small so that $\epsilon = \Theta(1)$[8], the per-iteration cost is dominated by the term $\tilde{\Theta}(N^2 sR^2/\epsilon^2)$, which is $\Theta(NR\epsilon/\delta) = \Omega(NR)$ times better than the per-iteration cost of the recursive LSS algorithm. For another case of a high-rank CP decomposition with $R \gg s$, which happens when one wants a high-accuracy CP decomposition of high order tensors, the per-iteration cost of our sketched CP-ALS algorithm is dominated by the term $\tilde{\Theta}(N^{3.5}R^{3.5}/\epsilon^5)$, and the cost ratio between this algorithm and the recursive LSS algorithm is $\tilde{\Theta}(N^{1.5}\delta/(\epsilon^5 R^{0.5}))$. For this case, our algorithm is only preferable when $N^{1.5}\delta/\epsilon^5$ is not too large compared to $R^{0.5}$.

Although our proposed sketching algorithm yields better per-iteration asymptotic cost in multiple regimes compared to existing leverage score based sketching algorithms, some preparation computations are needed to sketch right-hand-sides $S_i R_i$ for $i \in [N]$ before ALS iterations, and this cost is non-negligible. On the other hand, this algorithm has better parallelism, since it involves a sequence of matrix multiplications rather than sampling the matrix. We leave the detailed experimental comparison of the computational efficiency of different sketching techniques for future work.

Tensor train rounding Given a tensor train, *tensor train rounding* finds a tensor train with a lower rank to approximate the original representation. Throughout analysis we assume the tensor train has order N with the output dimension sizes equal s, the tensor train rank is $R < s$, and the goal is to round the rank to $r < R$. The standard tensor train rounding algorithm [12] consists of a right-to-left sweep of QR decompositions of the input tensor train (also called orthogonalization), and another left-to-right truncated singular value decompositions (SVD) sweep to perform rank reduction. The orthogonalization step is the

[8]As is shown in [172], in practice, setting ϵ to be 0.1-0.2 will result in accurate sketched least squares with relative residual norm error less than 0.05.

bottleneck of the rounding algorithm, and costs $\Theta(NsR^3)$. Recently, [89] has introduced a randomized rounding algorithm called "*Randomize-then-Orthogonalize*". Let $\mathbf{X}$ denote a matricization of the tensor train data with all except one dimension at the end grouped into the row, the algorithm first sketches X with a tensor train embedding S, then performs a sequence of truncated SVDs on top of SX. The sketch size m of S is r plus some constant, and is assumed to be smaller than R. The bottleneck is to compute SX, which costs $\Theta(NsR^2m)$.

We can recast the problem as finding an embedding satisfying the linearization sufficient condition with sketch size m, such that the asymptotic cost of computing SX is optimal given the data contraction tree that contracts the tensor train from one end to another. Our analysis (detailed in Section 6.13) shows that the sketching cost for the problem is lower bounded by $\Omega(NsR^2m)$, thus the sketching algorithm in [89] attains the asymptotic cost lower bound and is efficient. Note that sketching with Algorithm 6.1 yields the same asymptotic cost, despite using a different embedding.

6.5 EXPERIMENTS

We conduct multiple experiments to demonstrate the efficacy of our proposed embeddings. Below we first justify the theoretical analysis in Theorem 6.2 and Theorem 6.4 via testing the sketching performance on tensor train inputs and Kronecker product inputs. We then perform experiments to demonstrate that the accuracy of our proposed sketching algorithms is comparable to that of state-of-the-art sketching techniques for CP decomposition and tensor train rounding. Our experiments are carried out on an Intel Core i7 2.9 GHz Quad-Core machine using NumPy [194] routines in Python.

Sketching tensor train and Kronecker product inputs We compare the performance of general tensor network embedding used in Algorithm 6.1 (called TN embedding), tree embedding discussed in Theorem 6.4, and the baseline, tensor train embedding [89], in sketching tensor train input data in Fig. 6.5. The input tensor train data has order 6, and the dimension size is 500.

We test the sketching performance under different tensor train ranks. For a given rank, we randomly generate 25 different inputs, with each element in each tensor being an i.i.d. variable uniformly distributed within $[0, 1]$. Additional experiments with Gaussian-distributed input tensor train data are presented in Section 6.14. For each input x and a specific embedding structure, we calculate the relative sketching error twice under different sketch sizes, and record the smallest sketch size such that both of its relative sketching errors are within 0.2, $\frac{\|Sx\|_2}{\|x\|_2} \leq 0.2$. We also calculate the number of floating point operations (FLOPs) for computing

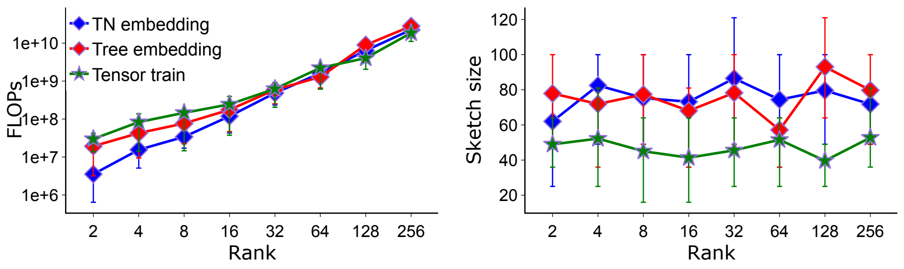


Figure 6.5: Results for sketching tensor train inputs. Each point denotes the mean value across 25 experiments, and each error bar shows the 25th-75th quartiles.

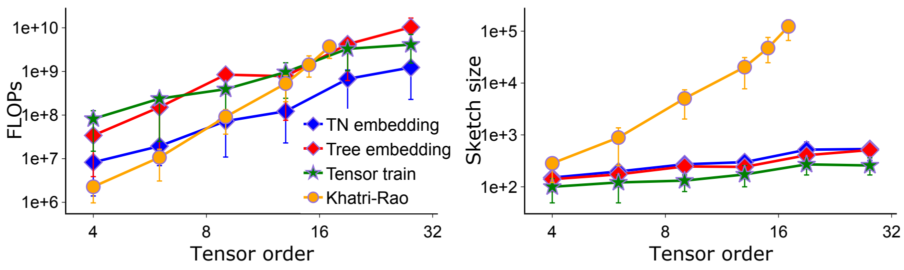


Figure 6.6: Results for sketching Kronecker product inputs.

Sx under the smallest sketch size based on the classical dense matrix multiplication algorithm. As can be seen, tree and tensor train embeddings are as efficient as TN embedding in terms of number of FLOPs under relatively high tensor train rank (when rank is at least 32), but are less efficient than TN embedding when the tensor train rank is lower than 32. The results are consistent with the theoretical analysis in Theorem 6.4, which shows that tree embeddings yield the optimal asymptotic cost when the input tensor train rank is at least the output sketch size, but the asymptotic cost is not optimal when the tensor train rank is low.

We also compare the performance of TN, tree, and two baselines proposed in [205], tensor train and Khatri-Rao product embeddings, in sketching Kronecker product inputs in Fig. 6.6. Each dimension size of the Kronecker product input is fixed to be 1000, and we test the sketching performance under different tensor orders. For each input x and a specific embedding structure, we record the smallest sketch size such that its relative sketching error is within 0.1. As can be seen, compared to Khatri-Rao product embedding, the sketch size of TN, tree and tensor train embeddings all increase slowly with the increase of tensor order, consistent with the theoretical analysis that these embeddings have efficient sketch size. In addition, the cost in FLOPs of TN embedding is smaller than tree and tensor train embeddings. This is consistent with the analysis in Theorem 6.2 and its following discussions, showing that TN embedding yields the optimal asymptotic cost for Kronecker product inputs, but tree and tensor train embeddings do not.

CP decomposition and tensor train rounding We perform experiments to demonstrate that the accuracy of our proposed sketching methods for CP-ALS and tensor train rounding is comparable to that of state-of-the-art sketching techniques. For both applications, we evaluate accuracy based on the final fitness f for each algorithm, defined as $f = 1 - \frac{\|\mathcal{T}-\widetilde{\mathcal{T}}\|_F}{\|\mathcal{T}\|_F}$, where $\mathcal{T}$ is the input tensor and $\widetilde{\mathcal{T}}$ is the reconstructed low-rank tensor.

CP rank	2	5	10
Sketch size	25	64	100
CP-ALS	0.737	0.804	0.838
LSS [82]	0.739	0.773	0.789
Algorithm 6.1	0.737	0.770	0.801

Table 6.2: Comparison of the final fitness of different CP decomposition algorithms under different CP ranks and sketch sizes. 10 ALS iterations are performed for all algorithms before the final fitness are calculated.

TT rounding rank	1	4	11	20
Sketch size	4	9	16	25
TT-SVD [12]	0.734	0.862	0.944	0.981
TT embedding [89]	0.573	0.757	0.882	0.951
Algorithm 6.1	0.527	0.761	0.866	0.948

Table 6.3: Comparison of the final fitness of different tensor train rounding algorithms under different tensor train rounding ranks and sketch sizes.

For CP-ALS, we conduct experiments on a Time-Lapse hyperspectral radiance image [112], which is a 3-D tensor with dimensions of $1024 \times 1344 \times 33$. This input data is used to demonstrate the applicability of our method in real-world scenarios. Standard CP-ALS, sketched CP-ALS using Algorithm 6.1, and sketched CP-ALS with approximate leverage score sampling (LSS) [82] are compared. The output CP decomposition fitness under varying CP ranks and sketch sizes is shown in Table 6.2. As can be seen, sketching with Algorithm 6.1 yields comparable fitness with the algorithm that sketches with approximate leverage score sampling. Note that the computational cost of Algorithm 6.1 is lower that LSS especially when the CP rank is low and the tensor dimension is large, as stated in Table 6.1.

We use 9 images from the Time-Lapse hyperspectral radiance image dataset for tensor train rounding, and reshape the input data to an order 6 tensor with size $9 \times 32 \times 32 \times 28 \times 48 \times 33$.

We use the TensorLy [133] library to truncate the input tensor to a tensor train with a rank of 30. On top of this tensor train, we evaluate the accuracy of various approaches, including tensor train SVD [12], randomized algorithm using tensor train embedding [89], and randomized algorithm using Algorithm 6.1. The fitness of the truncated tensor trains are displayed in Table 6.3 for a variety of rounding rank thresholds and sketch sizes. As can be seen, sketching with Algorithm 6.1 has comparable accuracy with the baseline algorithm (sketching with tensor train embedding). In addition, both sketching algorithms also have similar complexity as is analyzed in Section 6.4.

6.6 CONCLUSIONS

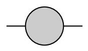
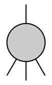
We provide detailed analysis of general tensor network embeddings. For input data such that each dimension to be sketched has size greater than the sketch size, we provide an algorithm to efficiently sketch such data using Gaussian embeddings that can be linearized into a sequence of sketching matrices and have low sketch size. Our sketching method is then used to design state-of-the-art sketching algorithms for CP tensor decomposition and tensor train rounding. We leave the analysis for more general embeddings for future work, including those with each tensor representing fast sketching techniques, such as Countsketch and fast JL transform using fast Fourier transform, and those containing structures cannot be linearized, such as Khatri-Rao product embedding. It would also be of interest to look at other tensor-related applications that could benefit from tensor network embedding, including tensor ring decomposition and simulation of quantum circuits. We also leave the high-performance implementation of the algorithm for general tensor networks as future work.

6.7 BACKGROUND

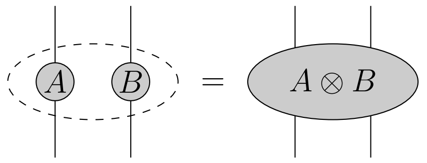
6.7.1 Tensor Algebra and Tensor Diagram Notation

Our analysis makes use of tensor algebra for tensor operations [5]. Vectors are denoted with lowercase Roman letters (e.g., $\mathbf{v}$), matrices are denoted with uppercase Roman letters (e.g., $\mathbf{M}$), and tensors are denoted with calligraphic font (e.g., $\mathcal{T}$). An order N tensor corresponds to an N-dimensional array. For an order N tensor $\mathcal{T} \in \mathbb{R}^{s_1 \times \cdots \times s_N}$, the size of ith dimension is s_i. The ith column of the matrix $\mathbf{M}$ is denoted by $\mathbf{M}(:, i)$, and the ith row is denoted by $\mathbf{M}(i, :)$. Subscripts are used to label different vectors, matrices and tensors (e.g. $\mathcal{T}_1$ and $\mathcal{T}_2$ are unrelated tensors). The Kronecker product of two vectors/matrices is denoted with $\otimes$, and the outer product of two or more vectors is denoted with $\circ$. For matrices $\mathbf{A} \in \mathbb{R}^{m\times k}$ and $\mathbf{B} \in \mathbb{R}^{n\times k}$, their Khatri-Rao product results in a matrix of size $(mn) \times k$ defined by $\mathbf{A} \odot \mathbf{B} = [\mathbf{A}(:,1) \otimes \mathbf{B}(:,1), \ldots, \mathbf{A}(:,k) \otimes \mathbf{B}(:,k)]$. Matricization is the process of unfolding a tensor into a matrix. The dimension-n matricized version of $\mathfrak{T}$ is denoted by $\mathbf{T}_{(n)} \in \mathbb{R}^{s_n \times K}$ where $K = \prod_{m=1,m\neq n}^{N} s_m$.

We introduce the graph representation for tensors, which is also called tensor diagram [3]. A tensor is represented by a vertex with hyperedges adjacent to it, each corresponding to a tensor dimension. A matrix $\mathbf{M}$ and an order four tensor $\mathfrak{T}$ are represented as follows,

$\mathbf{M} \Longrightarrow$  $\qquad \mathfrak{T} \Longrightarrow$  . (6.3)

The Kronecker product of two matrices $\mathbf{A}$ and $\mathbf{B}$ can be expressed as



(6.4)

Connecting two edges means two tensor dimensions are contracted or summed over. One example is shown in Fig. 6.7.

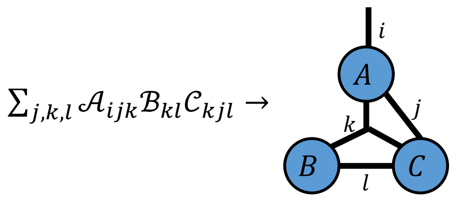

$\sum_{j,k,l} \mathcal{A}_{ijk}\mathcal{B}_{kl}\mathcal{C}_{kjl} \rightarrow$
i
A
j
k
B
C
l

Figure 6.7: An example of tensor diagram notation.

6.7.2 Background on Sketching

In this section, we introduce definitions for sketching used throughout the paper.

Definition 6.2 (Gaussian embedding). *A matrix* $\mathbf{S} = \frac{1}{\sqrt{m}}\mathbf{M} \in \mathbb{R}^{m\times n}$ *is a Gaussian embedding if each element of* $\mathbf{M}$ *is a normalized Gaussian random variable,* $\mathbf{M}(i,j) \sim N(0,1)$.

One key property we would like the tensor network embedding to satisfy is the (ϵ, δ)-accurate property. To achieve this, one central property each tensor in the tensor network embedding needs to satisfy is the Johnson-Lindenstrauss (JL) moment property. The JL moment property captures a bound on the moments of the difference between the vector Euclidean norm and the norm after sketching. We introduce both definitions below.

Definition 6.3 ((ϵ, δ)-accurate embedding)**.** *A random matrix $S \in \mathbb{R}^{m\times n}$ has the (ϵ,δ)-accurate embedding property if for every $\mathbf{x} \in \mathbb{R}^n$ with $\|\mathbf{x}\|_2 = 1$,*

$$\underset{\mathbf{S}}{Pr}\left(\left|\|\mathbf{S}\mathbf{x}\|_2^2 - 1\right| > \epsilon\right) < \delta. \tag{6.5}$$

Definition 6.4 ((ϵ, δ, p)-JL moment [212], [213])**.** *A random matrix $S \in \mathbb{R}^{m\times n}$ has the (ϵ,δ,p)-JL moment property if for every $\mathbf{x} \in \mathbb{R}^n$ with $\|\mathbf{x}\|_2 = 1$,*

$$\underset{\mathbf{S}}{\mathbb{E}}\left|\|\mathbf{S}\mathbf{x}\|_2^2 - 1\right|^p < \epsilon^p\delta \quad \text{and} \quad \mathbb{E}\left[\|\mathbf{S}\mathbf{x}\|_2^2\right] = 1. \tag{6.6}$$

Definition 6.5 (Strong (ϵ, δ)-JL moment [44], [212])**.** *A random matrix $S \in \mathbb{R}^{m\times n}$ has the strong (ϵ,δ)-JL moment property if for every $\mathbf{x} \in \mathbb{R}^n$ with $\|\mathbf{x}\|_2 = 1$, and every integer $p \in [2, \log(1/\delta)]$,*

$$\underset{\mathbf{S}}{\mathbb{E}}\left|\|\mathbf{S}\mathbf{x}\|_2^2 - 1\right|^p < \left(\frac{\epsilon}{e}\right)^p \left(\frac{p}{\log(1/\delta)}\right)^{p/2} \tag{6.7}$$

and $\mathbb{E}\left[\|\mathbf{S}\mathbf{x}\|_2^2\right] = 1$.

Note that the strong (ϵ,δ)-JL moment property directly reveals the $(\epsilon,\delta,\log(1/\delta))$-JL moment property, since letting $p = \log(1/\delta)$, (6.7) becomes

$$\underset{\mathbf{S}}{\mathbb{E}}\left|\|\mathbf{S}\mathbf{x}\|_2^2 - 1\right|^{\log(1/\delta)} < \left(\frac{\epsilon}{e}\right)^{\log(1/\delta)} = \epsilon^p\delta. \tag{6.8}$$

Both the strong (ϵ, δ)-JL moment property and the (ϵ, δ, p)-JL moment property directly imply (ϵ, δ)-accurate embedding via Markov's inequality,

$$\underset{\mathbf{S}}{\Pr}\left(\left|\|\mathbf{S}\mathbf{x}\|_2^2 - 1\right| > \epsilon\right) < \frac{\mathbb{E}\left|\|\mathbf{S}\mathbf{x}\|_2^2 - 1\right|^p}{\epsilon^p} < \delta. \tag{6.9}$$

The lemmas below show that Gaussian embeddings can be used to construct embeddings with the JL moment property.

Lemma 6.1 (Strong JL moment of Gaussian embeddings [212])**.** *Gaussian embeddings with $m = \Omega(\log(1/\delta)/\epsilon^2)$ satisfy the (ϵ,δ)-strong JL moment property.*

Below we review the composition rules of JL moment properties introduced in [44], which are used to prove the (ϵ,δ)-accurate sufficient condition in Theorem 6.1.

Lemma 6.2 (JL moment with Kronecker product)**.** *If a matrix* $\mathbf{S}$ *has the* (ϵ, δ, p)*-JL moment property, then the matrix* $\mathbf{M} = \mathbf{I}_i \otimes \mathbf{S} \otimes \mathbf{I}_j$ *also has the* (ϵ, δ, p)*-JL moment property for identity matrices* $\mathbf{I}_i$ *and* $\mathbf{I}_j$ *with any size. This relation also holds for the strong* (ϵ, δ)*-JL moment property.*

Lemma 6.3 (Strong JL moment with matrix product)**.** *There exists a universal constant* L*, such that for any constants* $\epsilon, \delta \in [0, 1]$ *and any integer* k*, if* $\mathbf{M}_1 \in \mathbb{R}^{d_2 \times d_1}, \cdots, \mathbf{M}_k \in \mathbb{R}^{d_{k+1} \times d_k}$ *are independent random matrices, each having the strong* $\left(\frac{\epsilon}{L\sqrt{k}}, \delta\right)$*-JL moment property, then the product matrix* $\mathbf{M} = \mathbf{M}_k \cdots \mathbf{M}_1$ *satisfies the strong* (ϵ, δ)*-JL moment property.*

6.8 DEFINITIONS AND BASIC PROPERTIES OF TENSOR NETWORK EMBEDDING

In this section, we introduce definitions and basic properties of tensor network embeddings. These properties will be used in Section 6.9 and Section 6.10 for detailed computational cost analysis. The notation defined in the main text is summarized in Table 6.4, which is also used in later analysis.

Notations	Meanings
S, S_i	Embedding matrix
m	Sketch size
$G_E = (V_E, E_E, w)$	Embedding tensor network
$G_D = (V_D, E_D, w)$	Input data tensor network
$\bar{E} = \{e_1, \ldots, e_N\}$	Set of edges to be sketched
s_i	Size of e_i in $\bar{E}$
T_0	Given data contraction tree
$\mathcal{D}(e_1), \ldots, \mathcal{D}(e_N), \mathcal{S}, \mathcal{I}$	Subsets of contractions in T_0
$X(e_i)$	Sub network contracted by $\mathcal{D}(e_i)$

Table 6.4: Notations used throughout the paper.

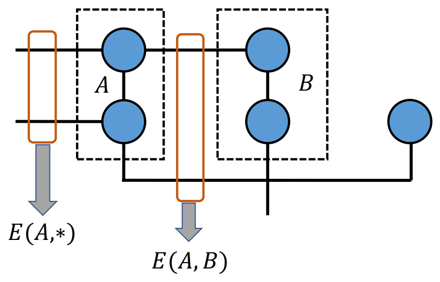


Figure 6.8: An example of $E(A,*)$ and $E(A,B)$, where both A,B are subset of vertices.

6.8.1 Graph Notation for Tensor Network and Tensor Contraction

We use undirected hypergraphs to represent tensor networks. For a given hypergraph $G=(V,E,w)$, V represents the vertex set, E represents the set of hyperedges, and w is a function such that $w(e)$ is the natural logarithm of the tensor dimension size represented by the hyperedge $e \in E$. We use $E(u,v)$ to denote the set of hyperedges adjacent to both u and v, which includes the edge (u,v) and hyperedges adjacent to u,v. We use $E(A,B)$ to denote the set of hyperedges connecting two subsets A,B of V with $A \cap B = \emptyset$. We use $E(A,*)$ to denote all uncontracted edges only adjacent to A, $E(A,*) = \{(u) \in E : u \in A\}$. we illustrate $E(A,B), E(A,*)$ in Fig. 6.8. For any set $A \subseteq V$, we let

$$E(A) = E(A, V \setminus A) \cup E(A,*). \tag{6.10}$$

A tensor network implicitly represents a tensor with a set of (small) tensors and a specific contraction pattern. We use $G[A] = (A, E_A, w)$ to denote a sub tensor network defined on $A \subseteq V$, where E_A contains all hyperedges in E adjacent to any $v \in A$.

Our analysis also use directed graphs to represent tensor network linearizations. We use $E(u,v)$ to denote the edge from u to v, and similarly use $E(A,B)$ to denote the set of edges from A to B.

When representing the contraction tree, we use (v_1, v_2) to denote the contraction of v_1, v_2. This notation is also used to represent multiple contractions. For example, we use $(((v_1,v_4),(v_2,v_5)),v_3)$ to represent the contraction tree shown in Fig. 6.9. The computational cost of a contraction tree is the summation of each contraction's cost. In the discussion throughout the paper, we assume that all tensors in the network are dense. Therefore, the contraction of two general dense tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, represented as vertices v_a and v_b in $G=(V,E,w)$, can be cast as a matrix multiplication, and the overall asymptotic cost is

$$\Theta\left(\exp\left(w(E(v_a)) + w(E(v_b)) - w(E(v_a,v_b))\right)\right) \tag{6.11}$$

with classical matrix multiplication algorithms. In general, contracting tensor networks with arbitrary structure is #P-hard [54], [221].

Here is an example of constrained contraction tree, which is defined in Definition 6.1. Consider a tensor network with three tensors, v_1, v_2, v_3, with a given contraction tree T_0 that is $((v_1, v_2), v_3)$, which indicates that v_1 first contracts with v_2 and subsequently with v_3. Consider an additional tensor network consisting of v_1, v_2, v_3 and another tensors u. Then the contraction tree $(((v_1, v_2), u), v_3)$, $(((v_1, u), v_2), v_3)$ and $(((v_1, v_2), v_3), u)$ are all constrained on T_0, since the contraction ordering of v_1, v_2, v_3 remains unchanged. However, the contraction tree $(((v_1, v_3), u), v_2)$ is not constrained on T_0.

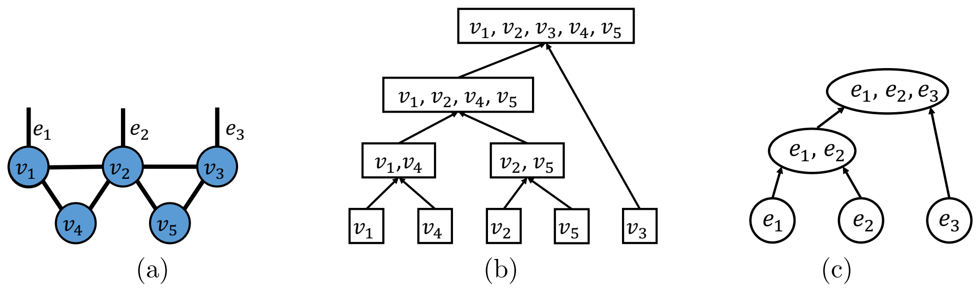


Figure 6.9: Example of a tensor network, its contraction tree and the corresponding dimension tree.

For a given data G_D and its given contraction tree T_0, its dimension tree is a directed binary tree showing the way edges in $\bar{E}$ are merged onto the same tensor. Each vertex in the dimension tree is a subset $E' \subseteq \bar{E}$, and for any two vertices E'_1, E'_2 of the dimension tree with the same parent, there is a contraction in T_0 such that the two input tensors are incident to E'_1, E'_2, respectively. One example is shown in Fig. 6.9.

6.8.2 Definitions Used In the Analysis of Tensor Network Embedding

In this section, we introduce definitions that will be used in later analysis. For a (hyper)graph $G = (V, E, w)$ and two subsets of V denoted as A, B, we define $\mathsf{cut}_G(A, B) = \sum_{e \in E(A,B)} w(e)$. Similarly, we define $\mathsf{cut}_G(A, *) = \sum_{e \in E(A,*)} w(e)$, and define $\mathsf{cut}_G(A) = \sum_{e \in E(A)} w(e)$, where $E(A)$ is expressed in (6.10). When G is a directed hypergraph, $\mathsf{cut}_G(A, B)$ denotes the sum of the weights of edges from A to B. When G is an undirected graph, $\mathsf{cut}_G(A, B)$ denotes the sum of the weights of hyperedges connecting A and B.

For two tensors represented by two subsets $A, B \subset V$ and $A \cap B = \emptyset$, the logarithm of the contraction cost between a tensor represented by A and a tensor represented by B, (A, B), is

$$\mathsf{cost}_G(A, B) = \mathsf{cut}_G(A) + \mathsf{cut}_G(B) - \mathsf{cut}_G(A, B). \tag{6.12}$$

Note that the function cost is only defined on undirected hypergraphs.

Consider a given input data $G_D = (V_D, E_D, w)$ and an embedding $G_E = (V_E, E_E, w)$. Below we let $V = V_E \cup V_D$, $E = E_E \cup E_D$, and $G = (V, E, w)$ denote the hypergraph including both the embedding and the input data. We use $L = (V, E_E, w)$ to denote the graph including V and all edges in the embedding, and use $R = (V, E \setminus E_E, w)$. Note that in this work we focus on the case where L is a graph, and R can be a general hypergraph. We illustrate G, G_D, G_E, L, R in Fig. 6.10. For any $A, B \subset V$ and $A \cap B = \emptyset$, we have

$$\mathsf{cut}_G(A) = \mathsf{cut}_L(A) + \mathsf{cut}_R(A), \tag{6.13}$$

and

$$\mathsf{cut}_G(A, B) = \mathsf{cut}_L(A, B) + \mathsf{cut}_R(A, B). \tag{6.14}$$

Based on (6.13) and (6.14), we have

$$\mathsf{cost}_G(A, B) = \mathsf{cost}_L(A, B) + \mathsf{cost}_R(A, B). \tag{6.15}$$

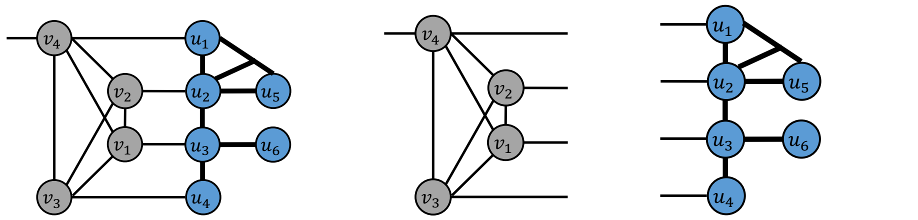
Our analysis of tensor network embedding is based on the linearization of the tensor network graph. Linearization casts an undirected graph into a *directed acyclic graph* (DAG). We define linearization formally below, then specify linearizations of the data and embedding graphs that our analysis considers.

Definition 6.6 (Linearization DAG)**.** *A linearization of the undirected graph $G = (V, E, w)$ is defined by the DAG $G' = (V, E', w)$ induced by a given choice of vertex ordering in V. For each contracted edge in E, E' contains an same-weight edge directing towards the higher indexed vertex. For each uncontracted edge in E, E' contains an edge with the same weight that is directed outward from the vertex it is adjacent to.*

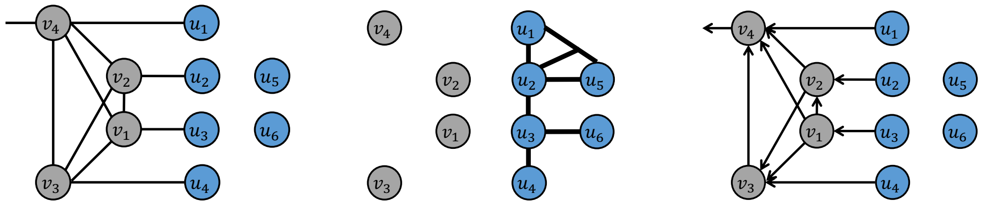
Based on Definition 6.6, we define the *sketching linearization DAG*, $G_S = (V, E_S, w)$, as a DAG defined on top of the graph $L = (V, E_E, w)$, which includes all vertices in both the embedding and the data and all embedding edges. For a given vertex ordering of embedding vertices, G_S is the linearization of L based on the ordering with all data vertices being ordered ahead of embedding vertices.

As discussed in Section 6.2, for a given sketching linearization, the sketching accuracy of each tensor $\boldsymbol{\mathcal{A}}_i$ at v_i is dependent on the row size of its matricization $\mathbf{A}_i$, which is the weighted size of the edge set adjacent to v_i containing all uncontracted edges and contracted edges also adjacent to v_j with $j > i$, which is called *effective sketch dimension* of v_i throughout the paper. Based on the definition, when $v \in V_E$, $\mathsf{cut}_{G_S}(v)$ equals the effective sketch dimension size of v. When $v \in V_D$, $\mathsf{cut}_{G_S}(v)$ represents the size of the sketch dimension adjacent to v. We look at embeddings G_E not only satisfying the (ϵ, δ)-accurate sufficient condition in Theorem 6.1, but also only have one output sketch dimension ($|E_1| = 1$) with the output sketch size $m = \Theta\left(N_E \log(1/\delta)/\epsilon^2\right)$. For each one of these embeddings, there must exist a linearization G_S such that for all $v \in V_E$, we have

$$\mathsf{cut}_{G_S}(v) = \Omega\left(\log(m)\right). \tag{6.16}$$



(a) $G = (V, E_E \cup E_D, w)$ (b) $G_E = (V_E, E_E, w)$ (c) $G_D = (V_D, E_D, w)$



(d) $L = (V, E_E, w)$ (e) $R = (V, E \setminus E_E, w)$ (f) $G_S = (V, E_S, w)$

Figure 6.10: Illustration of graphs and hypergraphs used throughout the paper.

6.8.3 Properties of Tensor Network Embedding

We now derive properties that are used in the sketching computational cost analysis. In Lemma 6.4, we show relations between cuts in the graph L and cuts in the graph G_S. In Lemma 6.5, we show relations between costs in the graph L and cuts in the graph G_S. Lemma 6.5 along with cut lower bounds (6.16) is used to derive lower bounds for cost_L and cost_G in Section 6.10.

Lemma 6.4. *Consider an embedding $G_E = (V_E, E_E, w)$ and a data tensor network $G_D = (V_D, E_D, w)$, and a given sketching linearization $G_S = (V, E_S, w)$, where $V = V_E \cup V_D$. For* *any $A, B \subset V$ and $A \cap B = \emptyset$, the following relations hold,*

$$\mathsf{cut}_L(A) = \mathsf{cut}_{G_S}(A) + \mathsf{cut}_{G_S}(V \setminus A, A), \tag{6.17}$$

$$\mathsf{cut}_L(A, B) = \mathsf{cut}_{G_S}(A, B) + \mathsf{cut}_{G_S}(B, A), \tag{6.18}$$

$$\begin{aligned} \mathsf{cut}_{G_S}(A \cup B) &= \mathsf{cut}_{G_S}(A) + \mathsf{cut}_{G_S}(B) - \mathsf{cut}_{G_S}(A, B) - \mathsf{cut}_{G_S}(B, A) \\ &= \mathsf{cut}_{G_S}(A) + \mathsf{cut}_{G_S}(B) - \mathsf{cut}_L(A, B). \end{aligned} \tag{6.19}$$

Proof. (6.17) and (6.18) hold directly based on the definition of the linearization DAG. For (6.19), based on (6.10), we have

$$\begin{aligned} \mathsf{cut}_{G_S}(A \cup B) &= \mathsf{cut}_{G_S}\left(A \cup B, V \setminus (A \cup B)\right) + \mathsf{cut}_{G_S}(A \cup B, *) \\ &= \mathsf{cut}_{G_S}\left(A, V \setminus (A \cup B)\right) + \mathsf{cut}_{G_S}\left(B, V \setminus (A \cup B)\right) + \mathsf{cut}_{G_S}(A \cup B, *) \\ &= \mathsf{cut}_{G_S}\left(A, V \setminus A\right) - \mathsf{cut}_{G_S}\left(A, B\right) + \mathsf{cut}_{G_S}\left(B, V \setminus B\right) - \mathsf{cut}_{G_S}\left(B, A\right) \\ &+ \mathsf{cut}_{G_S}(A, *) + \mathsf{cut}_{G_S}(B, *) \\ &= \mathsf{cut}_{G_S}(A) + \mathsf{cut}_{G_S}(B) - \mathsf{cut}_{G_S}(A, B) - \mathsf{cut}_{G_S}(B, A). \end{aligned} \tag{6.20}$$

Note that the second and third equalities in (6.20) hold since A and B are disjoint sets. This finishes the proof. Q.E.D.

Lemma 6.5. *Consider any data $G_D = (V_D, E_D, w)$ and embedding $G_E = (V_E, E_E, w)$, and a sketching linearization $G_S = (V, E_S, w)$, where $V = V_D \cup V_E$. For any two subsets $A, B \in V$ such that $A \cap B = \emptyset$, the contraction of two tensors that are the contraction outputs of A and B has a logarithm cost of*

$$\mathsf{cost}_L(A, B) = \mathsf{cut}_{G_S}(A) + \mathsf{cut}_{G_S}(B) + \mathsf{cut}_{G_S}(V \setminus (A \cup B), A \cup B). \tag{6.21}$$

Proof. Based on Lemma 6.4, we have

$$\mathsf{cut}_L(A) \overset{(6.17)}{=} \mathsf{cut}_{G_S}(V \setminus A, A) + \mathsf{cut}_{G_S}(A), \tag{6.22}$$

$$\mathsf{cut}_L(B) \overset{(6.17)}{=} \mathsf{cut}_{G_S}(V \setminus B, B) + \mathsf{cut}_{G_S}(B). \tag{6.23}$$

Based on (6.18), we have

$$\begin{aligned}
&\mathsf{cut}_{G_S}(V \setminus A, A) + \mathsf{cut}_{G_S}(V \setminus B, B) - \mathsf{cut}_{L}(A, B)\\
&= \mathsf{cut}_{G_S}(V \setminus A, A) + \mathsf{cut}_{G_S}(V \setminus B, B) - \mathsf{cut}_{G_S}(A, B) - \mathsf{cut}_{G_S}(B, A)\\
&= \mathsf{cut}_{G_S}(V \setminus (A \cup B), A) + \mathsf{cut}_{G_S}(V \setminus (A \cup B), B)\\
&= \mathsf{cut}_{G_S}(V \setminus (A \cup B), A \cup B).
\end{aligned} \tag{6.24}$$

Based on (6.22),(6.23), (6.24), we have

$$\begin{aligned}
\mathsf{cost}_{L}(A, B) &= \mathsf{cut}_{L}(A) + \mathsf{cut}_{L}(B) - \mathsf{cut}_{L}(A, B)\\
&= \mathsf{cut}_{G_S}(A) + \mathsf{cut}_{G_S}(B) + \mathsf{cut}_{G_S}(V \setminus (A \cup B), A \cup B).
\end{aligned} \tag{6.25}$$

This finishes the proof. Q.E.D.

Lemma 6.6. *Consider any data $G_D = (V_D, E_D, w)$ and an embedding $G_E = (V_E, E_E, w)$, and a sketching linearization $G_S = (V, E_S, w)$ such that the embedding is (ϵ, δ)-accurate. Then for any $U \subseteq V$ such that there exists $v \in U$ and $\mathsf{cut}_{G_S}(v) \geq \log(m)$, we have $\mathsf{cut}_{G_S}(U) \geq \log(m)$.*

Proof. When U is a subset of the data vertices, $U \subseteq V_D$, this holds directly since

$$\mathsf{cut}_{G_S}(U) = \sum_{u \in U} \mathsf{cut}_{G_S}(u) \geq \mathsf{cut}_{G_S}(v) \geq \log(m). \tag{6.26}$$

Next we consider the case where $U \cap V_E \neq \emptyset$. Let $A = U \cap V_E$ and $B = U \cap V_D$. Based on the definition of DAG, there is no directed cycle in the subgraph $G_S[A]$. Therefore, there exists one vertex $s \in A$, such that $\mathsf{cut}_{G_S}(s, A \setminus \{s\}) = 0$. Based on Lemma 6.4, we have

$$\begin{aligned}
\mathsf{cut}_{G_S}(A) &\overset{(6.19)}{=} \mathsf{cut}_{G_S}(s) + \mathsf{cut}_{G_S}(A \setminus \{s\}) - \mathsf{cut}_{G_S}(s, A \setminus \{s\}) - \mathsf{cut}_{G_S}(A \setminus \{s\}, s)\\
&\geq \mathsf{cut}_{G_S}(s) - \mathsf{cut}_{G_S}(s, A \setminus \{s\})\\
&= \mathsf{cut}_{G_S}(s) \overset{(6.16)}{\geq} \log(m),
\end{aligned} \tag{6.27}$$

In addition, we have $\mathsf{cut}_{G_S}(A, B) = 0$ since $A \subseteq V_E$ and $B \subseteq V_D$. Thus we have

$$\begin{aligned}
\mathsf{cut}_{G_S}(U) = \mathsf{cut}_{G_S}(A \cup B) &\overset{(6.19)}{=} \mathsf{cut}_{G_S}(A) + \mathsf{cut}_{G_S}(B) - \mathsf{cut}_{G_S}(A, B) - \mathsf{cut}_{G_S}(B, A)\\
&= \mathsf{cut}_{G_S}(A) + \mathsf{cut}_{G_S}(B) - \mathsf{cut}_{G_S}(B, A)\\
&\geq \mathsf{cut}_{G_S}(A) \geq \log(m).
\end{aligned} \tag{6.28}$$

This finishes the proof. Q.E.D.

6.9 COMPUTATIONALLY-EFFICIENT SKETCHING ALGORITHM

In this section, we introduce the detail of the computationally-efficient sketching algorithm in Algorithm 6.1. Consider a given data tensor network $G_D = (V_D, E_D, w)$ and a given data contraction tree, T_0. Also let $N_D = |V_D|$, and let $\bar{E} \subseteq E_D$ denote the set of edges to be sketched, and $N = |\bar{E}|$. Below we let $\bar{E} = \{e_1, e_2, \ldots, e_N\}$, and let each e_i has weight $\log(s_i) > \log(m)$. Based on the definition we have $N \leq N_D$. Let one contraction path representing T_0 be expressed as a sequence of $N_D - 1$ contractions,

$$\{(U_1, V_1), \ldots, (U_{N_D-1}, V_{N_D-1})\}\,. \tag{6.29}$$

Above we use (U_i, V_i) to represent the contraction of two intermediate tensors represented by two subset of vertices $U_i, V_i \subset V_D$. Below we let

$$\begin{aligned}
a_i &= \exp\left(\mathsf{cut}_R(U_i) - \mathsf{cut}_R(U_i, V_i)\right),\\
c_i &= \exp\left(\mathsf{cut}_R(V_i) - \mathsf{cut}_R(U_i, V_i)\right),\\
d_i &= \exp\left(\mathsf{cut}_R(U_i \cup V_i)\right)/(a_i c_i),\\
b_i &= \exp\left(\mathsf{cut}_R(U_i, V_i)\right)/d_i.
\end{aligned} \tag{6.30}$$

Note that d_i represents the size of uncontracted dimensions adjacent to both U_i and V_i, and b_i represents the size of contracted dimensions between U_i and V_i. We also have $\mathsf{cost}_R(U_i, V_i) = \log(a_i b_i c_i d_i)$, and $\mathsf{cut}_R(U_i \cup V_i) = \log(a_i c_i d_i)$. a_i, b_i, c_i, d_i are visualized in Fig. 6.11.

In Section 6.3, the $N_D - 1$ contractions are categorized into $N + 2$ sets, $\mathcal{D}(e_1), \ldots, \mathcal{D}(e_N)$, $\mathcal{S}$, $\mathcal{I}$, where $\mathcal{D}(e_i)$ contains contractions such that e_i is the only data edge adjacent to the contraction output and in $\bar{E}$, $\mathcal{S}$ contains contractions (U_i, V_i) such that both U_i and V_i are adjacent to edges in $\bar{E}$, and $\mathcal{I}$ includes (U_i, V_i) such that both U_i and V_i are not adjacent to $\bar{E}$. An illustration of these sets is provided below.

Consider an input data consisting of five tensors, v_1, v_2, v_3, v_4, v_5, where v_1 is adjacent to the edge e_1, v_2 is adjacent to e_2, v_3 is adjacent to e_3, and e_1, e_2, e_3 are the edges to be sketched. There are no edges to be sketched adjacent to v_4, v_5. Consider the contraction tree $(((v_1, v_4), (v_2, v_5)), v_3)$, where v_1 contracts with v_4 and outputs $v_{1,4}, v_2$ contracts with v_5 and outputs $v_{2,5}$, and then $v_{1,4}, v_{2,5}$ contract together into $v_{1,2,4,5}$, and $v_{1,2,4,5}$ contracts with v_3. We have $\mathcal{D}(e_1) = \{(v_1, v_4)\}$ and $\mathcal{D}(e_2) = \{(v_2, v_5)\}$, since v_1, v_2 are adjacent to e_1, e_2, respectively. We also have $\mathcal{I} = \emptyset$ and $\mathcal{D}(e_3) = \emptyset$, since each contraction is adjacent to at least one edge in $\{e_1, e_2, e_3\}$, and there is no contraction such that e_3 is the only data edge in the output. All the remaining contractions are in $\mathcal{S}$, so $\mathcal{S} = \{(v_{1,4}, v_{2,5}), (v_{1,2,4,5}, v_3)\}$.

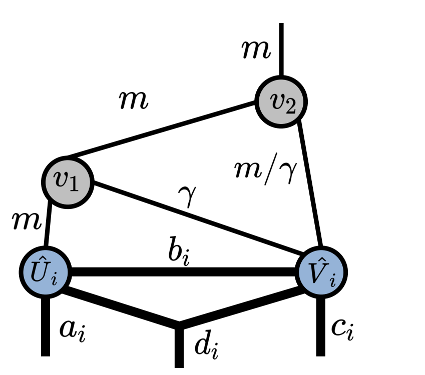
6.9.1 Sketching with the Embedding Containing a Binary Tree of Small Tensor Networks

We now present the details of applying the embedding containing a binary tree of small tensor networks. In Section 6.3, we define $\mathcal{S}$ as the set containing contractions (U_i, V_i) such that both U_i and V_i are adjacent to edges in $\bar{E}$. For each contraction $i \in \mathcal{S}$, one small embedding tensor network (denoted as Z_i) is applied to the contraction. Let $\hat{U}_i, \hat{V}_i$ denote the sketched U_i and V_i formed in previous contractions in the sketching contraction tree T_B, such that $\hat{U}_i \cap V_D = U_i$ and $\hat{V}_i \cap V_D = V_i$. The structure of Z_i is determined so that the asymptotic cost to sketch $(\hat{U}_i, \hat{V}_i)$ is minimized, under the constraint that Z_i is $(\epsilon/\sqrt{N}, \delta)$-accurate and only has one output sketch dimension.

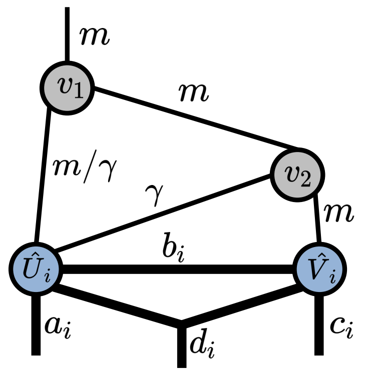
The structure of Z_i is illustrated in Fig. 6.11. For the case $a_i \leq c_i$, the structure is shown in Fig. 6.11a, and sketching is performed via the contraction sequence of contracting $\hat{U}_i$ and v_1 first, then with $\hat{V}_i$, and then with v_2 (also denoted as a contraction sequence of $(((\hat{U}_i, v_1), \hat{V}_i), v_2)$). For the case $a_i > c_i$, the structure of Z_i is shown in Fig. 6.11b, and the sketching is performed via the contraction sequence of $(((\hat{V}_i, v_2), \hat{U}_i), v_1)$. With this algorithm, sketching $(\hat{U}_i, \hat{V}_i)$ yields a computational cost proportional to

$$y_i = a_i b_i c_i d_i m^2 + \min(a_i, c_i) \cdot d_i m^2 \cdot \min_{\gamma \in [1,m]} \left(b_i \gamma + \frac{m \cdot \max(a_i, c_i)}{\gamma} \right). \tag{6.31}$$

We show in Lemma 6.10 that the asymptotic cost lower bound to sketch (U_i, V_i) is also $\Omega(y_i)$.



(a) $\gamma = \min\left(\max\left(\sqrt{\frac{c_i m}{b_i}}, 1\right), m\right)$



(b) $\gamma = \min\left(\max\left(\sqrt{\frac{a_i m}{b_i}}, 1\right), m\right)$

Figure 6.11: Illustration of the small network in the binary tree structured embedding. For each edge e, we show the dimension size of that edge (exponential in $w(e)$).

6.9.2 Computational Cost Analysis

We provide the computational cost analysis of Algorithm 6.1 in this section.

Theorem 6.5. *Algorithm 6.1 has an asymptotic computational cost of*

$$\Theta\left(\sum_{j=1}^{N} t(e_j) + \sum_{i\in\mathcal{S}} y_i + \sum_{i\in\mathcal{I}} z_i\right), \tag{6.32}$$

where $t(e_j)$ is the optimal asymptotic cost to sketch the sub tensor network $X(e_j)$ (defined in Table 6.4) with a matrix in the Kronecker product embedding, y_i is expressed in (6.31), and

$$z_i = a_i b_i c_i d_i \cdot \min\left(\exp(\mathsf{cut}_{G_S}(U_i \cup V_i)), m\right), \tag{6.33}$$

where a_i, b_i, c_i, d_i are expressed in (6.30).

Proof. The terms $\sum_{j\in N} t(e_j) + \sum_{i\in\mathcal{S}} y_i$ can be easily verified based on the analysis in Section 6.3 and Section 6.9.1.

Consider the contractions in $\mathcal{I}$, which include (U_i, V_i) such that both U_i and V_i are not adjacent to $\bar{E}$, and contractions where U_i or V_i is adjacent to at least two edges in $\bar{E}$. The first type of contractions in $\mathcal{I}$ would have a cost of $\Theta(a_i b_i c_i d_i)$, and not be affected by previous sketching steps. For the second type, application of the Kronecker product and binary tree embeddings to U_i and V_i would reduce all adjacent edges in $\bar{E}$ to a single dimension of size m. Consequently, the contraction cost would be $\Theta(a_i b_i c_i d_i \cdot m)$. Summarizing both cases prove the cost in (6.33). The cost in (6.32) follows from combining the terms $\sum_{j\in N} t(e_j) + \sum_{i\in\mathcal{S}} y_i$ and $\sum_{i\in\mathcal{I}} z_i$. Q.E.D.

For the special case where each vertex in the data tensor network is adjacent to an edge to be sketched, we have $\mathcal{D}(e_j) = \emptyset$ for all $j \in [N]$ and $\mathcal{I} = \emptyset$, thus all the contractions are in the set $\mathcal{S}$. Therefore, sketching each e_j has an asymptotic cost of $\Theta(t(e_j)) = \Theta(\exp(\mathsf{cut}_G(v_j)) \cdot m)$, where v_j is the vertex in the data graph adjacent to e_j, and Theorem 6.5 implies that the sketching cost would be

$$\Theta\left(\sum_{j=1}^{N} t(e_j) + \sum_{i\in\mathcal{S}} y_i + \sum_{i\in\mathcal{I}} z_i\right) = \Theta\left(\sum_{j=1}^{N} \exp(\mathsf{cut}_G(v_j)) \cdot m + \sum_{j=1}^{N-1} y_i\right). \tag{6.34}$$

As we will show in Theorem 6.6, this cost matches the asymptotic cost lower bound, when the embedding satisfies the (ϵ, δ)-accurate sufficient condition and only has one output sketch dimension.

When the data has a Kronecker product structure, we have $\mathsf{cut}_G(v_j) = w(e_j) = \log(s_j)$, and $a_i, b_i, c_i, d_i = 1$ for all $i \in \{1, \dots, N-1\}$ for all contraction trees. Therefore,

$$y_i = a_i b_i c_i d_i m^2 + m^2 d_i \sqrt{a_i b_i c_i m} \cdot \min(\sqrt{a_i}, \sqrt{c_i}) = m^2 + m^{2.5}, \tag{6.35}$$

and the sketching cost is

$$\Theta\left(\sum_{j=1}^{N} s_j m + N m^{2.5}\right). \tag{6.36}$$

As we will show in Section 6.11, sketching with tree tensor network embeddings yield an asymptotic cost of $\Theta\left(\sum_{j=1}^{N} s_j m + N m^3\right)$. Therefore, Algorithm 6.1 is more efficient to sketch Kronecker product input data.

6.10 LOWER BOUND ANALYSIS

In this section, we discuss the *asymptotic* computational lower bound for sketching with embeddings satisfying the (ϵ, δ)-accurate sufficient condition and only have one output sketch edge. In Section 6.10.1, we discuss the case where the data has uniform sketch dimensions. In this case, each vertex in the data tensor network is adjacent to an edge to be sketched. In Section 6.10.2, we discuss the sketching computational lower bound for a more general case, when the data tensor network can have arbitrary graph structure, and vertices not adjacent to sketch edges are allowed. For both cases, we assume that the size of each dimension to be sketched is greater than the sketch size.

6.10.1 Sketching Data with Uniform Sketch Dimensions

We now discuss the sketching asymptotic cost lower bound when the data $G_D = (V_D, E_D, w)$ has uniform sketch dimensions, where each $v \in V_D$ is adjacent to an edge to be sketched with size lower bounded by the target sketch size, m. We have $N = |\bar{E}| = |V_D|$, and we let the size of each $e_i \in \bar{E}$ be denoted $s_i > m$. We let $V = V_E \cup V_D$ denote the set of all vertices in both the data and the embedding. Below, we show the main theorem using lemmas and notations introduced in Section 6.8.

Theorem 6.6. *For any embedding G_E satisfying the (ϵ, δ)-accurate sufficient condition and only has one output sketch dimension, and any contraction tree T_B of (G_D, G_E) constrained on the data contraction tree T_0 expressed in (6.2), the sketching asymptotic cost is lower* *bounded by*

$$\Omega\left(\sum_{j=1}^{N}\exp(\mathsf{cut}_G(v_j))\cdot m+\sum_{j=1}^{N-1}y_i\right), \tag{6.37}$$

where $m=\Omega(N\log(1/\delta)/\epsilon^2)$ *represents the embedding sketch size,* v_j *is the vertex in* V_D *adjacent to* e_j*, and* $\exp(\mathsf{cut}_G(v_j))$ *denotes the size of the tensor at* v_j*, and* y_i *is expressed in* (6.31)*.*

We present the proof of Theorem 6.6 at the end of Section 6.10.1. Note that the first term in (6.37), $\sum_{v\in V_D}\exp(\mathsf{cut}_G(v))\cdot m$, is a term independent of the data contraction tree, while the second term is dependent of the data contraction tree.

Proof of Theorem 6.2. The asymptotic cost of of Algorithm 6.1 in (6.34) matches the lower bound shown in Theorem 6.6, thus proving the statement. Q.E.D.

Theorem 6.6 also yields an asymptotic lower bound for sketching data with a Kronecker product structure. We state the results below.

Corollary 6.2. *Consider an input data* G_D *representing a vector with a Kronecker product structure and each* v_j *for* $j\in[N]$ *is adjacent to an edge to be sketched with size* s_j*. For any embedding* G_E *satisfying the* (ϵ,δ)*-accurate sufficient condition with only one output sketch dimension and any contraction tree* T_B *of* (G_D,G_E)*, the asymptotic cost must be lower bounded by*

$$\Omega\left(\sum_{j=1}^{N}s_jm+Nm^{2.5}\right), \tag{6.38}$$

where $m=\Omega(N\log(1/\delta)/\epsilon^2)$*.*

Below, we present some lemmas needed to prove Theorem 6.6.

Lemma 6.7. *Consider an* (ϵ,δ)*-accurate embedding* $G_E=(V_E,E_E,w)$ *with a sketching linearization* $G_S=(V,E_S,w)$*. Then for any subset of the embedding and data graph vertex set,* $W\subseteq V$*, we have* $\mathsf{cut}_{G_S}(W)\geq\log(m)$*.*

Proof. Since each vertex in the data graph is adjacent to an edge to be sketched, and the edge dimension size is greater than m, we have $\mathsf{cut}_{G_S}(w)\geq\log(m)$ for all $w\in V_D$. Since the embedding satisfies the (ϵ,δ)-accurate sufficient condition, we have $\mathsf{cut}_{G_S}(w)\geq\log(m)$ for all $w\in V_E$. Therefore, $\mathsf{cut}_{G_S}(w)\geq\log(m)$ for all $w\in V$. Based on Lemma 6.6, $\mathsf{cut}_{G_S}(W)\geq\log(m)$ for all $W\subseteq V$. Q.E.D.

Lemma 6.8. *Consider an (ϵ, δ)-accurate embedding G_E with a sketching linearization $G_S = (V, E_S, w)$. Consider any contraction tree T_B for (G_D, G_E). If there exists a contraction output of $U \subset V$ formed in T_B and $\mathsf{cut}_{G_S}(U) > \log(m)$, then the asymptotic cost for the contraction tree T_B must be lower bounded by $\Omega\left(\exp\left(\mathsf{cut}_R(U) + \mathsf{cut}_{G_S}(U)\right) \cdot m\right)$.*

Proof. Since $\mathsf{cut}_{G_S}(U) > \log(m)$, there must exist a contraction $(U, W) \in T_B$ with W containing some vertices in $V \setminus U$. Based on Lemma 6.7, $\mathsf{cut}_{G_S}(W) \geq \log(m)$. Based on Lemma 6.5, we have

$$\begin{aligned}\mathsf{cost}_G(U, W) &= \mathsf{cost}_R(U, W) + \mathsf{cost}_L(U, W)\\ &= \mathsf{cost}_R(U, W) + \mathsf{cut}_{G_S}(U) + \mathsf{cut}_{G_S}(W) + \mathsf{cut}_{G_S}(V \setminus (U \cup W), U \cup W).\end{aligned} \tag{6.39}$$

Further, since $\mathsf{cost}_R(U, W) \geq \mathsf{cut}_R(U)$,

$$\begin{aligned}\mathsf{cost}_G(U, W) &\geq \mathsf{cut}_R(U) + \mathsf{cut}_{G_S}(U) + \mathsf{cut}_{G_S}(W)\\ &\geq \mathsf{cut}_R(U) + \mathsf{cut}_{G_S}(U) + \log(m).\end{aligned} \tag{6.40}$$

This proves the lemma since the contraction cost is $\Theta\left(\exp(\mathsf{cost}_G(U, W))\right)$. Q.E.D.

In Lemma 6.10, we show that when the data contraction tree T_0 contains the contraction (U_i, V_i), then any contraction tree T_B of (G_D, G_E) that is constrained on T_0 will yield a contraction cost of $\Omega(y_i)$. To show that, we first discuss the case where T_B also contains the contraction (U_i, V_i) in Lemma 6.9. The more general case where the contraction (U_i, V_i) need not be in T_B is discussed in Lemma 6.10.

Lemma 6.9. *Consider a specific contraction tree T_0 for G_D, where the contraction (U_i, V_i) is in T_0. For any embedding G_E satisfying the (ϵ, δ)-accurate sufficient condition with only one output sketch dimension and any contraction tree T_B of (G_D, G_E) constrained on T_0, if (U_i, V_i) is also in T_B, the sketching asymptotic cost must be lower bounded by $\Omega\left(a_i b_i c_i d_i m^2 + a_i c_i d_i m^3\right)$, where a_i, b_i, c_i, d_i are defined in (6.30).*

Proof. Consider any sketching linearization $G_S = (V, E_S, w)$ such that the embedding satisfies the (ϵ, δ)-accurate sufficient condition with only one output sketch dimension. Based on Lemma 6.7, we have $\mathsf{cut}_{G_S}(U_i), \mathsf{cut}_{G_S}(V_i) \geq \log(m)$. Based on Lemma 6.5, we have

$$\begin{aligned}\mathsf{cost}_G(U_i, V_i) &= \mathsf{cost}_R(U_i, V_i) + \mathsf{cost}_L(U_i, V_i)\\ &= \mathsf{cost}_R(U_i, V_i) + \mathsf{cut}_{G_S}(U_i) + \mathsf{cut}_{G_S}(V_i) + \mathsf{cut}_{G_S}(V \setminus (U_i \cup V_i), U_i \cup V_i)\\ &\geq \mathsf{cost}_R(U_i, V_i) + \mathsf{cut}_{G_S}(V_i) + \mathsf{cut}_{G_S}(V_i)\\ &\geq \log(a_i b_i c_i d_i) + 2\log(m).\end{aligned} \tag{6.41}$$

Thus this contraction has a cost of $\Omega\left(a_ib_ic_id_i \cdot m^2\right)$. In addition, since U_i, V_i are subsets of the data vertices, $\mathsf{cut}_L(U_i, V_i) = 0$. Therefore, based on (6.19),

$$\mathsf{cut}_{G_S}(U_i \cup V_i) = \mathsf{cut}_{G_S}(U_i) + \mathsf{cut}_{G_S}(V_i) \geq 2\log(m). \tag{6.42}$$

Based on Lemma 6.8, the cost needed to sketch $U_i \cup V_i$ is $\Omega\left(a_ic_id_im^3\right)$. Thus the overall asymptotic cost is lower bounded by $\Omega\left(a_ib_ic_id_im^2 + a_ic_id_im^3\right)$. This finishes the proof. Q.E.D.

Lemma 6.10. *Consider a specific contraction tree T_0 for G_D, where the contraction (U_i, V_i) is in T_0. For any embedding G_E satisfying the (ϵ, δ)-accurate sufficient condition with only one output sketch dimension and any contraction tree T_B of (G_D, G_E) constrained on T_0, the sketching asymptotic cost must be lower bounded by*

$$\Omega(y_i) = \Omega\left(a_ib_ic_id_im^2 + \min(a_i, c_i) \cdot d_im^2 \cdot \min_{\gamma \in [1,m]}\left(b_i\gamma + \frac{m \cdot \max(a_i, c_i)}{\gamma}\right)\right), \tag{6.43}$$

where a_i, b_i, c_i, d_i are defined in (6.30), and y_i is defined in (6.31).

Proof. Consider any sketching linearization $G_S = (V, E_S, w)$ such that the embedding satisfies the (ϵ, δ)-accurate sufficient condition with only one output sketch dimension. We first consider the case where the contraction (U_i, V_i) exists in T_B. Based on Lemma 6.9, the overall asymptotic cost is lower bounded by

$$\begin{aligned}&\Omega\left(a_ib_ic_id_im^2 + a_ic_id_im^3\right)\\ &= \Omega\left(a_ib_ic_id_im^2 + \min(a_i, c_i) \cdot d_im^2 \cdot \left(b_i \cdot 1 + \frac{m \cdot \max(a_i, c_i)}{1}\right)\right) = \Omega(y_i),\end{aligned} \tag{6.44}$$

and hence it satisfies (6.43).

We next consider the other case where the contraction (U_i, V_i) is not performed directly in T_B. Since T_B is constrained on T_0, there must exist a contraction $(\hat{U}_i, \hat{V}_i) \in T_B$ with either $\hat{U}_i$ or $\hat{V}_i$ containing embedding vertices, and $\hat{U}_i \cap V_D = U_i$, $\hat{V}_i \cap V_D = V_i$. Let x be the last embedding vertex (based on the linearization order) applied in T_B to $\hat{U}_i \cup \hat{V}_i$, so that $\mathsf{cut}_{G_S}(x, (\hat{U}_i \cup \hat{V}_i) \setminus \{x\}) = 0$. For the case where $x \in \hat{U}_i \setminus U_i$, we show below that the sketching asymptotic cost is lower bounded by

$$\Omega\left(a_ib_ic_id_im^2 + a_id_im^2 \cdot \min_{\gamma \in [1,m]}\left(b_i\gamma + \frac{mc_i}{\gamma}\right)\right) \tag{6.45}$$

For the other case where $x \in \hat{V}_i \setminus V_i$, we have the cost is lower bounded by

$$\Omega\left(a_i b_i c_i d_i m^2 + c_i d_i m^2 \cdot \min_{\gamma \in [1,m]} \left(b_i \gamma + \frac{m a_i}{\gamma}\right)\right) \tag{6.46}$$

by symmetry. Together, these two results prove the lemma.

Detailed proof of (6.45) Since $|\hat{U}_i| > |U_i|$, there must exist a contraction (Y_1, Y_2), for which the output is $\hat{U}_i = Y_1 \cup Y_2$. Based on Lemma 6.5, we have

$$\begin{aligned}
\mathsf{cost}_G(Y_1, Y_2) &= \mathsf{cost}_R(Y_1, Y_2) + \mathsf{cost}_L(Y_1, Y_2) \\
&= \mathsf{cost}_R(Y_1, Y_2) + \mathsf{cut}_{G_S}(Y_1) + \mathsf{cut}_{G_S}(Y_2) + \mathsf{cut}_{G_S}(V \setminus (Y_1 \cup Y_2), Y_1 \cup Y_2) \\
&\geq \mathsf{cost}_R(Y_1, Y_2) + \mathsf{cut}_{G_S}(Y_1) + \mathsf{cut}_{G_S}(Y_2) + \mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i) \\
&\geq \log(a_i b_i d_i) + 2\log(m) + \mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i).
\end{aligned} \tag{6.47}$$

Thus, the cost of the contraction (Y_1, Y_2) is lower bounded by

$$\Omega\left(a_i b_i d_i m^2 \cdot \exp\left(\mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i)\right)\right). \tag{6.48}$$

In addition, since

$$\begin{aligned}
\mathsf{cost}_G(\hat{U}_i, \hat{V}_i) &= \mathsf{cost}_R(\hat{U}_i, \hat{V}_i) + \mathsf{cut}_{G_S}(\hat{U}_i) + \mathsf{cut}_{G_S}(\hat{V}_i) + \mathsf{cut}_{G_S}(V \setminus (\hat{U}_i \cup \hat{V}_i), \hat{U}_i \cup \hat{V}_i) \\
&\geq \log(a_i b_i c_i d_i) + 2\log(m),
\end{aligned} \tag{6.49}$$

the contraction $(\hat{U}_i, \hat{V}_i)$ yields a cost lower bounded by

$$\Omega\left(a_i b_i c_i d_i \cdot m^2\right). \tag{6.50}$$

Combining (6.48) and (6.50), we have that the contractions (Y_1, Y_2) and $(\hat{U}_i, \hat{V}_i)$ have a cost of

$$\Omega\left(a_i b_i d_i m^2 \cdot \exp\left(\mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i)\right) + a_i b_i c_i d_i \cdot m^2\right). \tag{6.51}$$

When $\mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i) = \log(m)$, (6.51) implies that the overall asymptotic cost is lower bounded by

$$\begin{aligned}\Omega\left(a_ib_ic_id_im^2 + a_ib_id_im^3\right) &= \Omega\left(a_ib_ic_id_im^2 + a_id_im^2 \cdot \left(b_i \cdot m + \frac{mc_i}{m}\right)\right) \\ &= \Omega\left(a_ib_ic_id_im^2 + a_id_im^2 \cdot \min_{\gamma\in[1,m]}\left(b_i\gamma + \frac{mc_i}{\gamma}\right)\right).\end{aligned} \tag{6.52}$$

When $\mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i) < \log(m)$, based on Lemma 6.4, the effective sketch dimensions of $\hat{U}_i \cup \hat{V}_i$ satisfy

$$\begin{aligned}\mathsf{cut}_{G_S}(\hat{U}_i \cup \hat{V}_i) &= \left(\mathsf{cut}_{G_S}(\hat{U}_i) - \mathsf{cut}_{G_S}(\hat{U}_i, \hat{V}_i)\right) + \mathsf{cut}_{G_S}(\hat{V}_i) - \mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i) \\ &\geq \mathsf{cut}_{G_S}(x) + \mathsf{cut}_{G_S}(\hat{V}_i) - \mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i) \\ &\geq 2\log(m) - \mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i),\end{aligned} \tag{6.53}$$

where the first inequality holds since

$$\begin{aligned}\mathsf{cut}_{G_S}(\hat{U}_i) - \mathsf{cut}_{G_S}(\hat{U}_i, \hat{V}_i) &= \mathsf{cut}_{G_S}(\hat{U}_i, V \setminus (\hat{U}_i \cup \hat{V}_i)) + \mathsf{cut}_{G_S}(\hat{U}_i, *) \\ &\geq \mathsf{cut}_{G_S}(x, V \setminus (\hat{U}_i \cup \hat{V}_i)) + \mathsf{cut}_{G_S}(x, *) = \mathsf{cut}_{G_S}(x),\end{aligned} \tag{6.54}$$

and the second inequality in (6.53) holds since $\mathsf{cut}_{G_S}(x), \mathsf{cut}_{G_S}(\hat{V}_i) \geq \log(m)$ based on Lemma 6.7. Based on the condition $\mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i) < \log(m)$ as well as (6.53), we have $\mathsf{cut}_{G_S}(\hat{U}_i \cup \hat{V}_i) > \log(m)$.

Based on Lemma 6.8, since $\mathsf{cut}_{G_S}(\hat{U}_i \cup \hat{V}_i) > \log(m)$, there must exist another contraction in T_B to sketch $\hat{U}_i \cup \hat{V}_i$ with a cost of

$$\begin{aligned}\Omega\left(\exp\left(\mathsf{cut}_R(\hat{V}_i \cup \hat{U}_i) + \mathsf{cut}_{G_S}(\hat{V}_i \cup \hat{U}_i)\right) \cdot m\right) &= \Omega\left(a_ic_id_i \cdot \exp\left(\mathsf{cut}_{G_S}(\hat{V}_i \cup \hat{U}_i)\right) \cdot m\right) \\ &\overset{(6.53)}{=} \Omega\left(a_ic_id_i \cdot \frac{m^3}{\exp(\mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i))}\right).\end{aligned} \tag{6.55}$$

Let $\gamma = \exp\left(\mathsf{cut}_{G_S}(\hat{V}_i, \hat{U}_i)\right)$, we have $\gamma \geq 1$. In addition, since $\mathsf{cut}_{G_S}(\hat{U}_i \cup \hat{V}_i) \geq 2\log(m) - \log(\gamma) > \log(m)$, we have $\gamma \leq m$. Therefore, the asymptotic cost is then lower bounded by

$$\Omega\left(a_ib_ic_id_im^2 + \min_{\gamma\in[1,m]}\left(a_ib_id_im^2 \cdot \gamma + a_ic_id_im^3 \cdot \frac{1}{\gamma}\right)\right). \tag{6.56}$$

This finishes the proof.

Q.E.D.

Proof of Theorem 6.6. Based on Lemma 6.10, the cost of $\Omega(y_i)$ is needed to sketch the contraction (U_i, V_i). Since T_0 contains contractions (U_i, V_i) for $i \in [N-1]$, the asymptotic cost of T_B must be lower bounded by $\Omega\left(\sum_{i=1}^{N-1} y_i\right)$. In addition, in the analysis of Lemma 6.10, at least one embedding vertex is needed to sketch each contraction (U_i, V_i), thus $N_E = \Omega(N)$ and $m = \Omega(N \log(1/\delta)/\epsilon^2)$ for the lower bound $\Omega\left(\sum_{i=1}^{N-1} y_i\right)$ to hold.

In addition, each $v_j \in V_D$ for $j \in [N]$ is adjacent to e_j and each $w(e_j) > \log(m)$. Based on Lemma 6.8, the asymptotic cost must be lower bounded by

$$\Omega\left(\sum_{j=1}^{N} \exp\left(\mathsf{cut}_R(v_j) + \mathsf{cut}_{G_S}(v_j)\right) \cdot m\right) = \Omega\left(\sum_{j=1}^{N} \exp\left(\mathsf{cut}_G(v_j)\right) \cdot m\right). \tag{6.57}$$

The above holds since v_j is a vertex in the data graph, thus $\mathsf{cut}_{G_S}(v_j) = \mathsf{cut}_L(v_j)$ and $\mathsf{cut}_R(v_j) + \mathsf{cut}_{G_S}(v_j) = \mathsf{cut}_G(v_j)$. This finishes the proof. Q.E.D.

6.10.2 Sketching General Data

In this section, we look at general tensor network data G_D, where each vertex in G_D can either be adjacent to an edge to be sketched with weight greater than $\log(m)$ or not adjacent to any edge to be sketched. Below we consider any data contraction tree T_0 containing $\mathcal{D}(e_1), \ldots, \mathcal{D}(e_N), \mathcal{S}, \mathcal{I}$ defined in Section 6.3. We also let $X(e_j) \subset V$ represent the sub network contracted by $\mathcal{D}(e_j)$. We present the asymptotic sketching lower bound in Theorem 6.7.

Lemma 6.11. *Consider G_D with a data contraction tree T_0 containing $\mathcal{D}(e_j)$, which is a set containing contractions such that e_j is the only data edge adjacent to the contraction output and in $\bar{E}$ (set of data edges to be sketched). For any embedding G_E satisfying the (ϵ, δ)-accurate sufficient condition with only one output sketch dimension and any contraction tree T_B of (G_D, G_E) constrained on the data contraction tree T_0, the sketching asymptotic cost must be lower bounded by $\Omega(t(e_j))$, where $t(e_j)$ is the optimal asymptotic cost to sketch the sub tensor network $X(e_j)$ (defined in Table 6.4) with an adjacent matrix in the Kronecker product embedding.*

Proof. When $\mathcal{D}(e_j) = \emptyset$, $X(e_j) = \{v_j\}$, where v_j is the vertex in the data graph adjacent to e_j. As is analyzed in the proof of Theorem 6.6, the asymptotic cost must be lower bounded by $\Omega\left(\exp\left(\mathsf{cut}_G(v_j)\right) \cdot m\right)$, which equals the asymptotic cost to contract v_j with the adjacent embedding matrix.

Now we discuss the case where $\mathcal{D}(e_j) \neq \emptyset$. We first consider the case where there is a contraction $(X(e_j), W)$ in T_B. We show that under this case, the cost is lower bounded by $\Omega(t(e_j))$. We then show that for the case where there is no contraction $(X(e_j), W)$ in T_B, meaning that some sub network of $X(e_j)$ is sketched, the cost is also lower bounded by $\Omega(t(e_j))$. Summarizing both cases prove the lemma.

Consider the case where there exists a contraction $(X(e_j), W)$ in T_B. Contracting $X(e_j)$ yields a cost of $\Omega(\sum_{i\in\mathcal{D}(e_j)} a_ib_ic_id_i \cdot s_j)$. Next we analyze the contraction cost of $(X(e_j), W)$. Since $X(e_j)$ is the contraction output of $\mathcal{D}(e_j)$, W must either contain embedding vertices, or contain some data vertex adjacent to edges in $\bar{E}$ (edges to be sketched). Therefore, W contains some vertex v with $\mathsf{cut}_{G_S}(v) \geq \log(m)$. Based on Lemma 6.6, we have $\mathsf{cut}_{G_S}(W) \geq \log(m)$. Therefore,

$$\begin{aligned}\mathsf{cost}_L(X(e_j), W) &= \mathsf{cut}_{G_S}(X(e_j)) + \mathsf{cut}_{G_S}(W) + \mathsf{cut}_{G_S}(V \setminus (X(e_j) \cup W), X(e_j) \cup W) \\ &\geq \log(s_j) + \log(m).\end{aligned} \tag{6.58}$$

Let $l \in \mathcal{D}(e_j)$ denote the last contraction in $\mathcal{D}(e_j)$, then we have $\mathsf{cut}_R(X(e_j)) = \log(a_lc_ld_l)$. Thus, we have

$$\begin{aligned}\mathsf{cost}_G(X(e_j), W) &= \mathsf{cost}_L(X(e_j), W) + \mathsf{cost}_R(X(e_j), W) \\ &\geq \mathsf{cost}_L(X(e_j), W) + \mathsf{cut}_R(X(e_j)) \geq \log(a_lc_ld_ls_jm),\end{aligned} \tag{6.59}$$

making the cost lower bounded by $\Omega\left(\sum_{i\in\mathcal{D}(e_j)} a_ib_ic_id_i \cdot s_j + a_lc_ld_ls_jm\right)$. Note that contracting $X(e_j)$ and sketching the contraction output with a matrix in the Kronecker product embedding yields a cost of $\Theta\left(\sum_{i\in\mathcal{D}(e_j)} a_ib_ic_id_i \cdot s_j + a_lc_ld_ls_jm\right)$, which upper-bounds the value of $t(e_j)$ based on definition. Thus the sketching cost is lower bounded by $\Omega(t(e_j))$.

Below we analyze the case where there is no contraction $(X(e_j), W)$ in T_B. Without loss of generality, for each contraction (U_i, V_i) with $i \in \mathcal{D}(e_j)$, assume that U_i is adjacent to e_j. When $X(e_j)$ is not formed in T_B, there must exist U_k with $k \in \mathcal{D}(e_j)$, and a contraction (U_k, X) with $X \subset V_E$ is in T_B. All contractions before k yield a cost of

$$\Omega\left(\sum_{i\in\mathcal{D}(e_j), i<k} a_ib_ic_id_i \cdot s_j\right). \tag{6.60}$$

Similar to the analysis for the contraction $(X(e_j), W)$ in (6.59), the contraction (U_k, X) yields a cost of

$$\Omega(a_kb_kd_ks_jm). \tag{6.61}$$

For other contractions in T_0, (U_i, V_i) with $i \in \mathcal{D}(e_j), i \geq k$, there must exist some contractions $(\hat{U}_i, \hat{V}_i)$ in T_B with $\hat{U}_i \cap V_D = U_i$, $\hat{V}_i \cap V_D = V_i$, since T_B is constrained on T_0. Therefore, we have

$$\begin{aligned}\mathsf{cost}_G(\hat{U}_i, \hat{V}_i) &= \mathsf{cost}_R(\hat{U}_i, \hat{V}_i) + \mathsf{cost}_L(\hat{U}_i, \hat{V}_i) = \mathsf{cost}_R(U_i, V_i) + \mathsf{cost}_L(\hat{U}_i, \hat{V}_i) \\ &\geq \mathsf{cost}_R(U_i, V_i) + \mathsf{cut}_{G_S}(\hat{U}_i) \geq \log(a_i b_i c_i d_i m).\end{aligned} \tag{6.62}$$

In the last inequality in (6.62) we use the fact that there exists a vertex $v \in U_i \subseteq \hat{U}_i$ with $\mathsf{cut}_{G_S}(v) = \log(s_j) \geq \log(m)$, then based on Lemma 6.6, $\mathsf{cut}_{G_S}(\hat{U}_i) \geq \log(m)$.

Combining (6.60), (6.61) and (6.62), we have the cost is lower bounded by

$$\Omega\left(f(k)\right) = \Omega\left(\sum_{i \in \mathcal{D}(e_j), i<k} a_i b_i c_i d_i \cdot s_j + a_k b_k d_k s_j m + \sum_{i \in \mathcal{D}(e_j), i \geq k} a_i b_i c_i d_i \cdot m\right), \tag{6.63}$$

where $f(k)$ represents the asymptotic cost to contract $X(e_j)$ with an embedding matrix adjacent to e_j, when sketching is performed at kth contraction with $k \in \mathcal{D}(e_j)$. Based on the definition of $t(e_j)$, we have $f(k) = \Omega(t(e_j))$, thus finishing the proof. Q.E.D.

Lemma 6.12. *Consider any data G_D. For any embedding G_E satisfying the (ϵ, δ)-accurate sufficient condition with only one output sketch dimension and any contraction tree T_B of (G_D, G_E), the sketching asymptotic cost must be lower bounded by $\Omega(Nm^{2.5})$, where $m = \Omega(N \log(1/\delta)/\epsilon^2)$.*

Proof. Let G'_D be the data with the same set of sketching edges $(\bar{E})$ as G_D, but G'_D is a Kronecker product data. For any given contraction tree T_B of (G_D, G_E), there must exist a contraction tree of (G'_D, G_E) whose asymptotic cost is upper bounded by the cost of T_B. Therefore, the asymptotic cost lower bound to contract (G'_D, G_E) must also be the asymptotic cost lower bound to contract (G_D, G_E). Based on Corollary 6.2, the asymptotic cost of T_B must be lower bounded by

$$\Omega\left(\sum_{j=1}^{N} s_j m + N m^{2.5}\right) = \Omega(Nm^{2.5}). \tag{6.64}$$

Q.E.D.

Theorem 6.7. *For any embedding G_E satisfying the (ϵ, δ)-accurate sufficient condition and any contraction tree T_B of (G_D, G_E) constrained on the data contraction tree T_0 expressed in*

(6.2), *the sketching asymptotic cost must be lower bounded by*

$$\Omega\left(\sum_{j=1}^{N} t(e_j) + \sum_{i\in\mathcal{S}} a_i b_i c_i d_i m^2 + Nm^{2.5} + \sum_{i\in\mathcal{I}} z_i\right), \tag{6.65}$$

where $m = \Omega(N\log(1/\delta)/\epsilon^2)$, a_i, b_i, c_i, d_i *are expressed in* (6.30), $t(e_j)$ *is the optimal asymptotic cost to sketch the sub tensor network* $X(e_j)$ *(the sub network contracted by* $\mathcal{D}(e_j)$*, also defined in Table 6.4) with an adjacent matrix in the Kronecker product embedding, and* z_i *is expressed in* (6.33).

Proof. The term $\sum_{j=1}^{N} t(e_j)$ can be proven based on Lemma 6.11, and the term $Nm^{2.5}$ with $m = \Omega(N\log(1/\delta)/\epsilon^2)$ can be proven based on Lemma 6.12. Below we show the asymptotic cost is also lower bounded by $\Omega(\sum_{i\in\mathcal{S}} a_i b_i c_i d_i m^2 + \sum_{i\in\mathcal{I}} z_i)$, thus finishing the proof.

For each contraction (U_i, V_i) in T_0 with $i \in \mathcal{S}\cup\mathcal{I}$, there must exist a contraction $(\hat{U}_i, \hat{V}_i)$ in T_B, and $\hat{U}_i \cap V_D = U_i$, $\hat{V}_i \cap V_D = V_i$. For the case where $i \in \mathcal{S}$, since both U_i and V_i contain edges to be sketched, we have $\mathsf{cut}_{G_S}(\hat{U}_i) \geq \log(m)$ and $\mathsf{cut}_{G_S}(\hat{V}_i) \geq \log(m)$ based on Lemma 6.6. Therefore, we have

$$\begin{aligned}
\sum_{i\in\mathcal{S}} \mathsf{cost}_G(\hat{U}_i, \hat{V}_i) &= \sum_{i\in\mathcal{S}} \mathsf{cost}_R(\hat{U}_i, \hat{V}_i) + \mathsf{cost}_L(\hat{U}_i, \hat{V}_i) \\
&= \sum_{i\in\mathcal{S}} \mathsf{cost}_R(U_i, V_i) + \mathsf{cost}_L(\hat{U}_i, \hat{V}_i) \\
&\geq \sum_{i\in\mathcal{S}} \mathsf{cost}_R(U_i, V_i) + \mathsf{cut}_{G_S}(\hat{U}_i) + \mathsf{cut}_{G_S}(\hat{V}_i) \\
&\geq \sum_{i\in\mathcal{S}} \log(a_i b_i c_i d_i m^2),
\end{aligned} \tag{6.66}$$

where the first inequality above holds based on Lemma 6.5. This shows the cost is lower bounded by $\Omega(\sum_{i\in\mathcal{S}} a_i b_i c_i d_i m^2)$.

Now consider the case where $i \in \mathcal{I}$. In this case, either $\mathsf{cut}_{G_S}(U_i \cup V_i) = 0$ or $\mathsf{cut}_{G_S}(U_i \cup V_i) \geq \log(m)$. When $\mathsf{cut}_{G_S}(U_i \cup V_i) = 0$, we have $\mathsf{cut}_{G_S}(\hat{U}_i \cup \hat{V}_i) \geq \mathsf{cut}_{G_S}(U_i \cup V_i)$. When $\mathsf{cut}_{G_S}(U_i \cup V_i) \geq \log(m)$, based on Lemma 6.6, we have $\mathsf{cut}_{G_S}(\hat{U}_i \cup \hat{V}_i) \geq \log(m)$. To summarize, we have

$$\mathsf{cut}_{G_S}(\hat{U}_i \cup \hat{V}_i) \geq \min\left(\mathsf{cut}_{G_S}(U_i \cup V_i), \log(m)\right), \tag{6.67}$$

thus

$$\begin{aligned}\sum_{i\in\mathcal{I}} \mathsf{cost}_G(\hat{U}_i,\hat{V}_i) &= \sum_{i\in\mathcal{I}} \mathsf{cost}_R(\hat{U}_i,\hat{V}_i) + \mathsf{cost}_L(\hat{U}_i,\hat{V}_i) \\ &\geq \sum_{i\in\mathcal{I}} \mathsf{cost}_R(U_i,V_i) + \mathsf{cut}_{G_S}(\hat{U}_i) + \mathsf{cut}_{G_S}(\hat{V}_i) \\ &\geq \sum_{i\in\mathcal{I}} \mathsf{cost}_R(U_i,V_i) + \mathsf{cut}_{G_S}(\hat{U}_i \cup \hat{V}_i) \\ &\geq \sum_{i\in\mathcal{I}} \log(a_ib_ic_id_i) + \min\left(\mathsf{cut}_{G_S}(U_i\cup V_i), \log(m)\right) \\ &= \sum_{i\in\mathcal{I}} \log(z_i).\end{aligned} \tag{6.68}$$

This shows the sketching cost is lower bounded by $\Omega\left(\sum_{i\in\mathcal{I}} z_i\right)$, thus finishing the proof. Q.E.D.

Proof of Theorem 6.3. Based on Theorem 6.5, the computational cost of Algorithm 6.1 is

$$\alpha = \Theta\left(\sum_{j=1}^{N} t(e_j) + \sum_{i\in\mathcal{S}} y_i + \sum_{i\in\mathcal{I}} z_i\right). \tag{6.69}$$

Let β equals the expression in (6.65). We have

$$\begin{aligned}\frac{\alpha}{\beta} &= \frac{\Theta\left(\sum_{j=1}^{N} t(e_j) + \sum_{i\in\mathcal{S}} y_i + \sum_{i\in\mathcal{I}} z_i\right)}{\Omega\left(\sum_{j=1}^{N} t(e_j) + \sum_{i\in\mathcal{S}} (a_ib_ic_id_im^2 + m^{2.5}) + \sum_{i\in\mathcal{I}} z_i\right)} = \mathcal{O}\left(\frac{\sum_{i\in\mathcal{S}} y_i}{\sum_{i\in\mathcal{S}}(a_ib_ic_id_im^2+m^{2.5})}\right) \\ &= \mathcal{O}\left(\max_{i\in\mathcal{S}} \frac{y_i}{a_ib_ic_id_im^2+m^{2.5}}\right) \\ &= \mathcal{O}\left(\max_{i\in\mathcal{S}} \frac{a_ib_ic_id_im^2 + \min(a_i,c_i)\cdot d_im^2\cdot \min_{\gamma\in[1,m]}\left(b_i\gamma + \frac{m\cdot\max(a_i,c_i)}{\gamma}\right)}{a_ib_ic_id_im^2+m^{2.5}}\right) \\ &= O(1) + \mathcal{O}\left(\max_{i\in\mathcal{S}} \frac{\min(a_i,c_i)\cdot d_im^2\cdot \min_{\gamma\in[1,m]}\left(b_i\gamma + \frac{m\cdot\max(a_i,c_i)}{\gamma}\right)}{a_ib_ic_id_im^2+m^{2.5}}\right).\end{aligned} \tag{6.70}$$

Below we derive asymptotic upper bound of the term

$$\theta = \frac{\min(a_i,c_i)\cdot d_im^2\cdot \min_{\gamma\in[1,m]}\left(b_i\gamma + \frac{m\cdot\max(a_i,c_i)}{\gamma}\right)}{a_ib_ic_id_im^2+m^{2.5}}. \tag{6.71}$$

We analyze the case below with $a_i \leq c_i$, and the other case with $a_i > c_i$ can be analyzed in a similar way based on the symmetry of a_i, c_i in θ.

When $a_i \leq c_i$, we have

$$\theta = \frac{a_i d_i m^2 \cdot \min_{\gamma \in [1,m]} \left(b_i \gamma + \frac{m c_i}{\gamma} \right)}{a_i b_i c_i d_i m^2 + m^{2.5}}. \tag{6.72}$$

We consider two cases, one satisfies $\sqrt{b_i c_i m} \leq b_i c_i$ and the other satisfies $\sqrt{b_i c_i m} > b_i c_i$.

When $\sqrt{b_i c_i m} \leq b_i c_i$, we have $\theta \leq 1$, thus $\frac{\alpha}{\beta} = O(1)$, thus satisfying the theorem statement.

When $\sqrt{b_i c_i m} > b_i c_i$, which means that $m > b_i c_i$, we have $\min_{\gamma \in [1,m]} \left(b_i \gamma + \frac{m c_i}{\gamma} \right) = \Theta(\sqrt{b_i c_i m})$, and

$$\theta = \frac{a_i m^2 d_i \sqrt{b_i c_i m}}{a_i b_i c_i d_i m^2 + m^{2.5}} \leq \min \left(\frac{\sqrt{m}}{\sqrt{b_i c_i}}, a_i d_i \sqrt{b_i c_i} \right) \leq \sqrt{m}, \tag{6.73}$$

thus $\frac{\alpha}{\beta} \leq O\left(\sqrt{m}\right)$. In addition, when G_D is a graph, we have $d_i = 1$ for all i. Therefore,

$$\begin{aligned} \theta &\leq \min \left(\frac{\sqrt{m}}{\sqrt{b_i c_i}}, a_i d_i \sqrt{b_i c_i} \right) = \min \left(\frac{\sqrt{m}}{\sqrt{b_i c_i}}, a_i \sqrt{b_i c_i} \right) \\ &\leq \min \left(\frac{\sqrt{m}}{\sqrt{b_i c_i}}, c_i \sqrt{b_i c_i} \right) \leq \min \left(\frac{\sqrt{m}}{(b_i c_i)^{1/2}}, (b_i c_i)^{3/2} \right) \leq m^{0.375}. \end{aligned} \tag{6.74}$$

Therefore, in this case we have $\frac{\alpha}{\beta} \leq O\left(m^{0.375}\right)$, thus finishing the proof. Q.E.D.

6.11 ANALYSIS OF TREE TENSOR NETWORK EMBEDDINGS

In this section, we provide detailed analysis of sketching with tree embeddings. The algorithm to sketch with tree embedding is similar to Algorithm 6.1, and the only difference is that for each contraction (U_i, V_i) with $i \in \mathcal{S}$, such that both U_i and V_i are adjacent to edges in $\bar{E}$, we sketch it with one embedding tensor z_i rather than a small network. Let $\hat{U}_i, \hat{V}_i$ denote the sketched U_i and V_i formed in previous contractions in the sketching contraction tree T_B, such that $\hat{U}_i \cap V_D = U_i$ and $\hat{V}_i \cap V_D = V_i$, we sketch $(\hat{U}_i, \hat{V}_i)$ via the contraction path $((\hat{U}_i, \hat{V}_i), z_i)$. For the case where each vertex in the data tensor network is adjacent to an edge to be sketched, the sketching cost would be

$$\Theta \left(\sum_{j=1}^{N} \exp(\mathsf{cut}_G(v_j)) \cdot m + \sum_{j=1}^{N-1} (a_i b_i c_i d_i m^2 + a_i c_i d_i m^3) \right), \tag{6.75}$$

where v_j is the vertex in the data graph adjacent to e_j, a_i, b_i, c_i, d_i are defined in (6.30), and we replace the term y_i in (6.34) with $a_ib_ic_id_im^2 + a_ic_id_im^3$.

Proof of Theorem 6.4. Since each contraction in T_0 contracts dimensions with size being at least the sketch size m, we have $b_i \geq m$ for $i \in \mathcal{S}$. Let $\theta = \min(a_i, c_i) \cdot d_im^2 \cdot \min_{\gamma\in[1,m]} \left(b_i\gamma + \frac{m\cdot\max(a_i,c_i)}{\gamma}\right)$. Below we analyze the three cases.

- When $\sqrt{\frac{m\cdot\max(a_i,c_i)}{b_i}} \in [1, m]$, we have

$$\begin{aligned}\theta &= m^2d_i\sqrt{a_ib_ic_im}\cdot\min(\sqrt{a_i},\sqrt{c_i})\\ &= O\left(a_im^2d_i\sqrt{b_ic_im}\right) = O\left(a_ib_id_im^2\sqrt{c_i}\right) = O\left(a_ib_ic_id_im^2\right).\end{aligned} \tag{6.76}$$

- When $\sqrt{\frac{m\cdot\max(a_i,c_i)}{b_i}} \leq 1$, we have

$$\begin{aligned}\theta &= \min(a_i,c_i)\cdot d_im^2\cdot(b_i + m\cdot\max(a_i,c_i))\\ &= O\left(\min(a_i,c_i)\cdot b_id_im^2\right) = O\left(a_ib_ic_id_im^2\right).\end{aligned} \tag{6.77}$$

- When $\sqrt{\frac{m\cdot\max(a_i,c_i)}{b_i}} \geq m$, we have

$$\begin{aligned}\theta &= \min(a_i,c_i)\cdot d_im^2\cdot(b_im + \max(a_i,c_i))\\ &= O\left(a_ic_id_im^2\right) = O\left(a_ib_ic_id_im^2\right).\end{aligned} \tag{6.78}$$

Therefore, $\theta = O\left(a_ib_ic_id_im^2\right)$, and the asymptotic cost in (6.34) would be

$$\Theta\left(\sum_{j=1}^{N}\exp(\mathsf{cut}_G(v_j))\cdot m + \sum_{j=1}^{N-1}a_ib_ic_id_im^2\right). \tag{6.79}$$

Based on Theorem 6.6, (6.79) matches the sketching asymptotic cost lower bound for this data. Since $a_ic_id_im^3 \leq a_ib_ic_id_im^2$ so (6.75) equals (6.79), sketching with tree embeddings also yield the optimal asymptotic cost. Q.E.D.

When the data has a Kronecker product structure, sketching with tree tensor network embedding is less efficient compared to Algorithm 6.1. As is shown in (6.80), Algorithm 6.1 yields a cost of $\Theta\left(\sum_{j=1}^{N} s_jm + Nm^{2.5}\right)$ to sketch the Kronecker product data. However, for tree embeddings, the asymptotic cost (6.75) is equal to

$$\Theta\left(\sum_{j=1}^{N} s_j m + N m^3\right). \tag{6.80}$$

6.12 COMPUTATIONAL COST ANALYSIS OF SKETCHED CP-ALS

In this section, we provide detailed computational cost analysis of the sketched CP-ALS algorithm based on Algorithm 6.1. We are given a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{s\times\cdots\times s}$, and aim to decompose that into N factor matrices, $A_i \in \mathbb{R}^{s\times R}$ for $i \in [N]$. Let $L_i = \mathbf{A}_1 \odot \cdots \odot \mathbf{A}_{i-1} \odot \mathbf{A}_{i+1} \odot \cdots \odot \mathbf{A}_N$ and $R_i = \mathbf{X}_{(i)}^T$. In each iteration, we aim to update A_i via solving a sketched linear least squares problem, $\mathbf{A}_i = \underset{\mathbf{A}}{\operatorname{argmin}} \left\|S_i L_i A^T - S_i R_i\right\|_F^2$, where S_i is an embedding constructed based on Algorithm 6.1.

Below we first discuss the sketch size of S_i sufficient to make each sketched least squares problem accurate. We then discuss the contraction trees of L_i, on top of which embedding structures are determined. We select contraction trees such that contraction intermediates can be reused across subproblems. Finally, we present the detailed computational cost analysis of the sketched CP-ALS algorithm.

6.12.1 Sketch Size Sufficient for Accurate Least Squares Subproblem

Since the tensor network of L_i contains N output dimensions and L_i contains R columns, we show below that a sketch size of $\Theta(NR\log(1/\delta)/\epsilon^2) = \tilde{\Theta}(NR/\epsilon^2)$ is sufficient for the least squares problem to be (ϵ,δ)-accurate.

Theorem 6.8. *Consider the sketched linear least squares problem* $\min_{\mathbf{A}} \left\|S_i L_i A^T - S_i R_i\right\|_F^2$. *Let* $\mathbf{S}_i$ *be an embedding constructed based on Algorithm 6.1, with the sketch size* $m = \Theta(NR\log(1/\delta)/\epsilon^2)$*, solving the sketched least squares problem gives us an* $(1+\epsilon)$*-accurate solution with probability at least* $1-\delta$.

Proof. Algorithm 6.1 outputs an embedding with $\Theta(N)$ vertices. Based on Theorem 6.1, a sketched size of $\Theta(NR\log(1/\delta)/\epsilon^2)$ will make the embedding $(\epsilon, \frac{\delta}{e^R})$-accurate. Based on the ϵ-net argument [60], $\mathbf{S}$ is the (ϵ,δ)-accurate subspace embedding for a subspace with dimension R. Therefore, we can get an $(1+\epsilon)$-accurate solution with probability at least $1-\delta$ for the least squares problem. Q.E.D.

6.12.2 Data Contraction Trees and Efficient Embedding Structures

The structures of embeddings $S_1, \ldots, S_N$ also depend on the data contraction trees for $L_1, \ldots, L_N$. We denote the contraction tree of L_i as T_i. We construct T_i for $i \in [N]$ such that resulting embeddings $S_1, \ldots, S_N$ have common parts, which yields more efficient sketching computational cost via reusing contraction intermediates.

Let the vertex v_i represent the matrix A_i. We also let $V_L^{(i)} = \{v_1, \ldots, v_i\}$ denote the set of all first i vertices, and let $V_R^{(i)} = \{v_i, \ldots, v_N\}$ denote the set of vertices from v_i to v_N. In addition, we let $\mathcal{C}_L^{(i)}$ denote a contraction tree to fully contract $V_L^{(i)}$, from v_1 to v_i. Let $\mathcal{C}_L^{(1)} = \emptyset$, we have for all $i \geq 1$, $\mathcal{C}_L^{(i+1)} = \mathcal{C}_L^{(i)} \cup \left\{(V_L^{(i)}, v_{i+1})\right\}$. Similarly, we let $\mathcal{C}_R^{(i)}$ denote a contraction tree to fully contract $V_R^{(i)}$, from v_N to v_i. Let $\mathcal{C}_R^{(N)} = \emptyset$, we have for all $i \leq N$, $\mathcal{C}_R^{(i-1)} = \mathcal{C}_R^{(i)} \cup \left\{(V_R^{(i)}, v_{i-1})\right\}$.

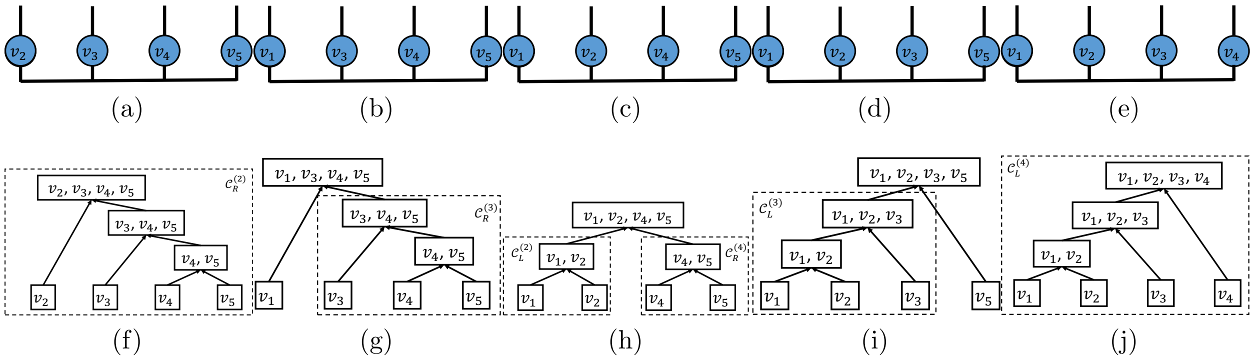


Figure 6.12: (a)-(e): Representations of $L_1, \ldots, L_5$ for the CP decomposition of an order 5 tensor. (f)-(j): Data dimension trees $T_1, \ldots, T_5$.

Note that the vertex set of the tensor network of L_i is $V_L^{(i-1)} \cup V_R^{(i+1)}$. Each T_i is constructed so that $V_L^{(i-1)}, V_R^{(i+1)}$ are first contracted via the contraction trees $\mathcal{C}_L^{(i-1)}$, $\mathcal{C}_R^{(i+1)}$, respectively, then a contraction of $(V_L^{(i-1)}, V_R^{(i+1)})$ is used to contract them into a single tensor. We illustrate T_i for the CP decomposition of an order 5 tensor in Fig. 6.12.

These tree structures allow us to reuse contraction intermediates during sketching. On top of T_1, sketching L_1 using Algorithm 6.1 yields a cost of $\Theta(N(smR + m^{2.5}R))$, where the term $\Theta(NsmR)$ comes from sketching with the Kronecker product embedding, and the term $\Theta(Nm^{2.5}R)$ comes from sketching each data contraction in $\mathcal{C}_R^{(2)}$. Since $\mathcal{C}_R^{(2)} = \mathcal{C}_R^{(3)} \cup \left\{(V_R^{(3)}, v_2)\right\}$, all contractions in $\mathcal{C}_R^{(3)}$ are sketched, and we obtain the sketching output of $V_R^{(3)}$, which is denoted as $\hat{V}_R^{(3)}$ below.

We use $\hat{V}_R^{(3)}$ formed during sketching L_1 to sketch L_2. Since T_2 contains contractions

$$T_2 = \mathcal{C}_R^{(3)} \cup \mathcal{C}_L^{(1)} \cup \left\{ (V_L^{(1)}, V_R^{(3)}) \right\} = \mathcal{C}_R^{(3)} \cup \left\{ (v_1, V_R^{(3)}) \right\}, \tag{6.81}$$

through reusing $\hat{V}_R^{(3)}$, we only need to sketch $(v_1, \hat{V}_R^{(3)})$ to compute $S_2 L_2$, which only costs $\Theta(smR + m^{2.5}R)$. Similarly, sketching each L_i for $i \geq 2$ only costs $\Theta(smR + m^{2.5}R)$, thus making the overall cost of sketching $L_1, \dots, L_N$ being $\Theta(N(smR + m^{2.5}R))$.

6.12.3 Detailed Algorithm and the Overall Computational Cost

Algorithm 6.2: Sketched-ALS: Sketched ALS for CP decomposition

```
1: Input: Input tensor $\mathcal{X}$, initializations $\mathbf{A}_1, \dots, \mathbf{A}_N$, maximum number of iterations $I_{\max}$
2: $G_D(L_i) \leftarrow$ structure of the data $L_i = \mathbf{A}_1 \odot \cdots \odot \mathbf{A}_{i-1} \odot \mathbf{A}_{i+1} \odot \cdots \odot \mathbf{A}_N$ for $i \in [N]$
3: $T_i \leftarrow$ contraction tree of $G_D(L_i)$ for $i \in [N]$ constructed based on Section 6.12.2
4: Build tensor network embeddings $S_i$ on $G_D(L_i)$ and $T_i$ based on Algorithm 6.1 for $i \in [N]$
5: Compute $\hat{R}_i \leftarrow S_i \mathbf{X}_{(i)}^T$ for $i \in [N]$
6: for $t \in [I_{\max}]$ do
7:     for $i \in [N]$ do
8:         Compute $\hat{L}_i \leftarrow S_i L_i$
9:         $\mathbf{A}_i \leftarrow \operatorname{argmin}_{\mathbf{X}} \left\| \hat{L}_i X - \hat{R}_i \right\|_F^2$
10:    end for
11: end for
12: Return: $\mathbf{A}_1, \dots, \mathbf{A}_N$
```

We present the detailed sketched CP-ALS algorithm in Algorithm 6.2. Here we analyze the overall computational cost of the algorithm.

Line 5 yields a preparation cost of the algorithm. Note that we construct S_i based on Section 6.12.2, where they share common tensors. Contracting $S_1 X_{(1)}^T$ yields a cost of $\Theta(s^N m)$. On top of that, contracting $S_i X_{(i)}^T$ for $i \geq 2$ also only yields a cost of $\Theta(s^N m)$, making the overall preparation cost $\Theta(s^N m)$.

Within each ALS iteration (Lines 7-10), based on Section 6.12.2, computing $S_i L_i$ for $i \in [N]$ costs $\Theta(N(smR + m^{2.5}R))$. For each $i \in [N]$, line 9 costs $\Theta(mR^2)$, making the cost of per-iteration least squares solves $\Theta(NmR^2)$. Based on Section 6.12.1, a sketch size of $m = \tilde{\Theta}(NR/\epsilon^2)$ is sufficient for the least squares solution to be $(1+\epsilon)$-accurate with probability at least $1-\delta$. Overall, the per-iteration cost is $\Theta(N(smR + m^{2.5}R)) = \tilde{\Theta}(N^2(N^{1.5}R^{3.5}/\epsilon^3 + sR^2)/\epsilon^2)$.

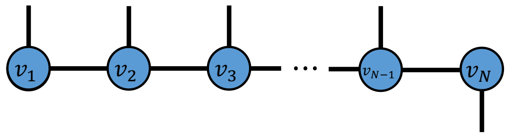


Figure 6.13: Illustration of the matricization of the tensor train (X) to be sketched. The $N-1$ uncontracted edges adjacent to $v_1, \ldots, v_{N-1}$ are to be sketched.

6.13 COMPUTATIONAL COST ANALYSIS OF SKETCHING FOR TENSOR TRAIN ROUNDING

We provide the computational cost lower bound analysis of computing SX, where $\mathbf{X}$ denotes a matricization of the tensor train data shown in Fig. 6.13. This step is the computational bottleneck of the tensor train randomized rounding algorithm proposed in [89]. As is discussed in Section 6.4, we assume the tensor train has order N with the output dimension sizes equal s, the tensor train rank is $R < s$, and the goal is to round the rank to $r < R$. The sketch size m of S is r plus some constant, and is assumed to be smaller than R. The lower bound is derived within all embeddings satisfying the sufficient condition in Theorem 6.1 and only have one output sketch dimension with size m.

For the data contraction tree that contracts the tensor train shown in Fig. 6.13 from left to right, we have $a_i = 1, b_i = R, c_i = R, d_i = 1$ for $i \in [N-2]$, where a_i, b_i, c_i, d_i are expressed in (6.30). Based on Theorem 6.7, the sketching asymptotic cost lower bound is

$$\begin{aligned}\Omega\left(\sum_{j=1}^{N-1} t(e_j) + \sum_{i\in\mathcal{S}} a_i b_i c_i d_i m^2 + N m^{2.5} + \sum_{i\in\mathcal{I}} z_i\right) &= \Omega\left(\sum_{j=1}^{N-1} t(e_j) + \sum_{i\in\mathcal{S}} a_i b_i c_i d_i m^2\right)\\ &= \Omega\left(\sum_{j=1}^{N-1} \exp(\mathsf{cut}_G(v_j))\cdot m + \sum_{j=1}^{N-2} a_i b_i c_i d_i m^2\right)\\ &= \Omega\left(NsR^2m + NR^2m^2\right) = \Omega\left(NsR^2m\right).\end{aligned} \tag{6.82}$$

Above we use the fact that $\exp(\mathsf{cut}_G(v_1)) = sR$, and for $j \in \{2, \ldots, N-1\}$, we have $\exp(\mathsf{cut}_G(v_j)) = sR^2$. Sketching with Algorithm 6.1, tree embedding and tensor train embedding all would yield this optimal asymptotic cost.

6.14 ADDITIONAL EXPERIMENTS

In this section, we experimentally verify that the sketch size of embeddings to get the same sketching accuracy trends similarly for both uniform and Gaussian input tensor distributions.

	Uniform	Gaussian
Tensor network embedding (Algorithm 6.1)	85.0	78.1
Tree embedding (Theorem 6.4)	75.4	68.1
Tensor train embedding [89]	49.3	45.4

Table 6.5: Comparison of the mean sketch sizes with different input data distribution when sketching a tensor train input. The tensor train order is chosen to be 6 and the dimension size is chosen to be 500. Each reported sketch size is the mean of 25 experiments. Variables in the uniform distribution are within the interval of $[0, 1]$, and variables in the Gaussian distribution have the same mean as the uniform distribution and have the unit variance.

We compare the performance of general tensor network embedding used in Algorithm 6.1, tree embedding discussed in Theorem 6.4, and the baseline, tensor train embedding [89], in sketching tensor train input data in Table 6.5. For each input tensor train x and a specific embedding structure, we calculate the relative sketching error twice under different sketch sizes, and record the smallest sketch size such that both of its relative sketching errors are within 0.2, $\frac{\|Sx\|_2}{\|x\|_2} \leq 0.2$. As can be seen in the table, for both distributions, tensor network embedding produces a slightly larger sketch size than tree embedding, and tensor train embedding yields the lowest sketch size.

Part III

APPROXIMATE TENSOR NETWORK CONTRACTION ALGORITHMS

Chapter 7: TENSOR NETWORK CONTRACTION WITH AN EFFICIENT SWAP-BASED ALGORITHM

For the next two Chapters, we change the focus from tensor decompositions to tensor network contractions. In this Chapter, we present an algorithm that can efficiently approximate the contraction of arbitrary tensor networks using low-rank approximations.

Our algorithm builds on the previous approach called Contracting Arbitrary Tensor Network (CATN) [41]. Given a specific contraction tree (a rooted binary tree representing the complete contraction of the tensor network), CATN proceeds by approximating each input tensor, as well as each intermediate tensor produced during the contraction as a matrix product state (MPS [33], also called tensor train [12]) with a bounded rank. To contract two tensors, the algorithm combines two MPSs by swapping/permuting the dimensions that connect both MPSs to the boundaries. Then, it contracts these dimensions to obtain the output MPS. For this algorithm, the adjacent dimension swaps are the bottleneck for complexity.

CATN is a highly general and flexible MPS-based approximation algorithm, yet there is still room for improvement in its performance. Specifically, the algorithm's efficiency is dependent on the selection of the input contraction tree, and the optimal method of selecting such trees to minimize the computational cost is currently unknown. Furthermore, the overall computational cost can also be affected by the choice of the dimension ordering in the MPS when approximating each tensor.

We offer an algorithm called CATN with global ordering (CATN-GO), an approach that combines the CATN algorithm with routines for determining efficient orderings and near-optimal contraction trees. CATN-GO utilizes a provided vertex ordering of the tensor network to guide the selection of MPS dimension orderings and the construction of the contraction tree. The algorithm's effectiveness is supported by theoretical analysis and extensive experimental evaluation.

Since the adjacent dimension swaps are the bottleneck for complexity, we establish a lower bound on the number of swaps required in CATN and show that our proposed algorithm attains the lower bound. To be precise, for a tensor network defined on $G = (V, E)$, we prove in Section 7.3 that the minimum number of swaps required during contraction is lower bounded by the convex crossing number of G [222]–[225], which is the minimum number of edge crossings over all vertex linear ordering of G, denoted by $\min_{\sigma^V} \operatorname{cr}(G, \sigma^V)$. A vertex linear ordering $\sigma^V : V \to \{1, \ldots, |V|\}$ assigns each vertex a unique number, and two edges with incident vertex orders $(i, j), (k, l)$ cross if $\min(i, j) < \min(k, l) < \max(i, j) < \max(k, l)$. In addition, for a vertex ordering σ^V that minimizes the edge crossings, the number of swaps used in CATN-GO equals the lower bound, $\min_{\sigma^V} \mathrm{cr}(G, \sigma^V)$.

In addition to minimizing the number of swaps, it is also important to perform swaps with low computational cost. Inspired by the algorithm presented in [226] for exact tensor network contractions, CATN-GO includes a dynamic programming algorithm to select the contraction tree under a given vertex ordering. This algorithm aims to minimize the overall computational cost, under the assumption that all MPSs have a uniform rank. As is detailed in Section 7.5, the uniform rank assumption makes the problem equivalent to minimizing the total length of the MPSs generated during the contractions. This algorithm runs in $O(n^3 m)$, where n is the number of vertices and m is the number of edges in the tensor network graph.

We evaluate the performance of CATN-GO in Section 7.6. Experimental results demonstrate that when applied to tensor networks defined on 3D lattices and random regular graphs, the utilization of vertex orderings that minimize edge crossings, along with the selection of efficient contraction trees, substantially reduces the overall execution time. These results align consistently with our theoretical analysis. Moreover, when considering tensor networks defined on 3D lattices using the Ising model, our algorithm outperforms both the CATN algorithm proposed in [41] and SweepContractor proposed in [86] in terms of speed. Specifically, our approach achieves a 5.9X speed-up in execution time while maintaining the same accuracy. This improvement in speed demonstrates the efficiency of the CATN-GO algorithm.

7.1 DEFINITIONS AND THE BACKGROUND

7.1.1 Definitions

We use undirected graphs to represent tensor networks. For a given graph $G = (V, E)$, V represents the vertex set and E represents the set of edges. Throughout the Chapter, we assume that each edge in the network is incident to one or two vertices. Scalars, vectors, matrices, and tensors are represented by vertices with zero, one, two, and at least three adjacent edges, respectively. We refer to edges with a dangling end (one end not incident to any vertex) as uncontracted edges, and those without dangling end as contracted edges. We use $E(A, B)$ to denote the set of edges connecting two subsets A, B of V with $A \cap B = \emptyset$. $E(A)$ represents the union of $E(A, V \setminus A)$ and all uncontracted edges adjacent to vertices in A.

A contraction tree of the tensor network graph G is a directed binary tree showing how vertices in V are contracted, and it is denoted T^V. Each leaf of T^V is a vertex in V, and each non-leaf vertex in T^V can be represented by a subset of the vertices, $W_1 \cup W_2$, where its

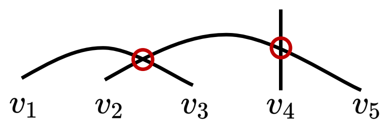


Figure 7.1: Illustration of a crossing of two contracted edges and a crossing of a contracted and an uncontracted edge.

two children are represented by W_1 and W_2, respectively.

Our analysis leverages the linear ordering $\sigma^S : S \to \{1, \ldots, |S|\}$, which assigns a unique value to each item in S. We use $\sigma^{S_1} \oplus \sigma^{S_2}$ to denote the concatenation of two orderings $\sigma^{S_1}, \sigma^{S_2}$, where each $x \in S_1$ is mapped to $\sigma^{S_1}(x)$ and each $x \in S_2$ is mapped to $\sigma^{S_2}(x) + |S_1|$.

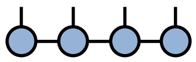
For a given tensor network $G = (V, E)$, we use σ^V to denote a specific vertex ordering, and use $\sigma^{E(v)}$ to denote a specific ordering of edges incident to the vertex v. The number of edge crossings in G on top of σ^V is denoted by $\mathrm{cr}(G, \sigma^V)$. When two edges are uncontracted, they will never cross. On the other hand, if two edges are contracted with incident vertex orders (i, j) and (k, l), then they will cross if $\min(i, j) < \min(k, l) < \max(i, j) < \max(k, l)$. In the case where one contracted edge is connected to vertices with orders (i, j) and the other edge is an uncontracted edge connected to the vertex k, the two edges will cross if $\min(i, j) < k < \max(i, j)$. We illustrate these two cases in Fig. 7.1.

7.1.2 MPS and the Swap Operation

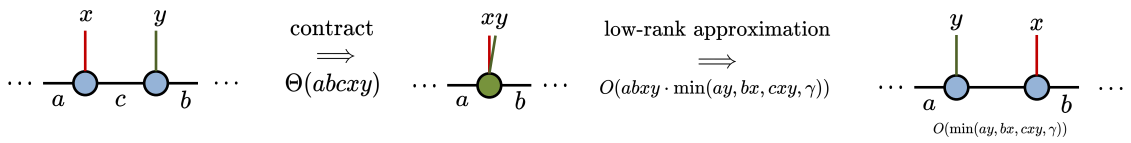
An MPS is a tensor network with a linear structure illustrated in Fig. 7.2a, where each tensor has one uncontracted dimension. Each vertex in the MPS tensor network is called a site, and the MPS ranks are the sizes of the dimensions connecting adjacent sites.

In tensor network contraction algorithms, one commonly used operation is the swapping of adjacent MPS sites, as depicted in Fig. 7.2. The standard swap operation involves combining/contracting two sites into a single tensor and subsequently performing a low-rank factorization to split the tensor into two parts. When the uncontracted dimensions 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 $abxy$[9]. Without truncation, the output rank of the low-rank factorization operation would be upper-bounded by the minimum among ay, bx, cxy. In practice, it is common to set an upper bound γ for the MPS ranks. When using a standard low-rank factorization algorithm that employs a rank-revealing QR factorization [184], the asymptotic cost is $O(abxy \min(ay, bx, cxy, \gamma))$.

[9]In the complexity analysis throughout the paper, we assume the classical matrix multiplication algorithm rather than fast algorithms such as Strassen's algorithm [61] are employed.

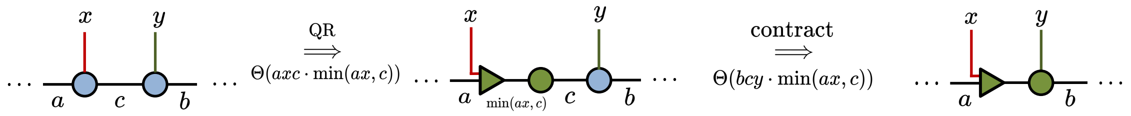


(a) The tensor diagram of an MPS



x
y
a
c
b
contract
$\Theta(abcxy)$
xy
low-rank approximation
$O(abxy \cdot \min(ay, bx, cxy, \gamma))$
$O(\min(ay, bx, cxy, \gamma))$

(b) Illustration of the swap operation.



x
y
a
c
b
QR
$\Theta(axc \cdot \min(ax, c))$
$\min(ax, c)$
contract
$\Theta(bcy \cdot \min(ax, c))$

(c) Illustration of the operation to orthogonalize one site in the MPS.

Figure 7.2: Visualization of MPS and the adjacent swap in an MPS.

Prior to each swap operation, a commonly-employed procedure to enhance the accuracy of low-rank factorization is canonicalization. During this step, all tensors except the two tensors involved in the swap are orthogonalized, and we illustrate the orthogonalization step and its costs in Fig. 7.2c. This ensures that the non-orthogonal parts of MPS are taken into account during low-rank truncation. In practice, canonicalization is achieved through a sequence of QR factorizations. For an MPS of order N and rank r, where each physical dimension has a size of s, the computational cost of canonicalization is $\Theta(Nsr^2)$.

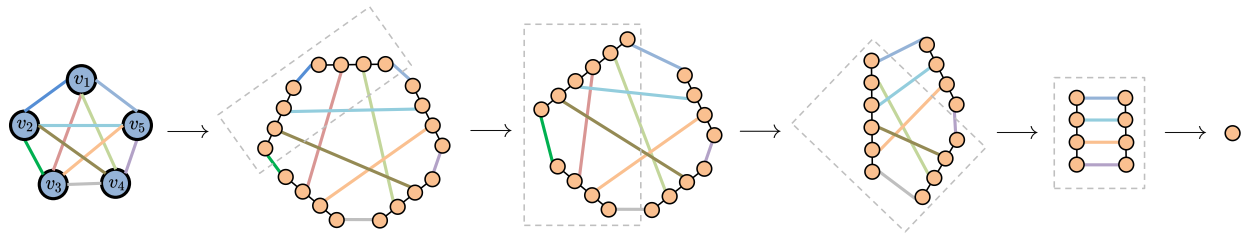
7.1.3 The Recursive Bisection Algorithm for Ordering Vertices

Recursive bisection is a simple divide-and-conquer heuristic adopted in linear ordering problems [227], [228]. The algorithm proceeds via first applying an approximate 1/3-balanced cut to separate the vertices V into two parts S and $V \setminus S$, then placing all vertices of S before all vertices not in S, and then recursing on both S and $V \setminus S$.

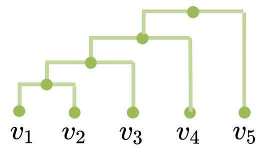
The effectiveness of the recursive bisection algorithm in minimizing edge crossings in the resulting vertex ordering has been discussed in [229]. Specifically, it is shown that the number of crossings in the output ordering obtained through recursive bisection is within a multiplicative factor of $O\left(\log^2 |V| \cdot \alpha\right)$ from the optimal convex crossing number for any graph with $|E| \geq 4|V|$ and $\Delta = O\left(\delta^{1.5}\right)$. Here, Δ denotes the maximum degree, δ represents the minimum degree of any vertex, and α represents the approximation factor of the minimum bisection algorithm.

In Section 7.6, we empirically demonstrate that utilizing the vertex ordering derived from recursive bisection can significantly reduce the number of swaps in the CATN algorithm compared to random vertex orderings.

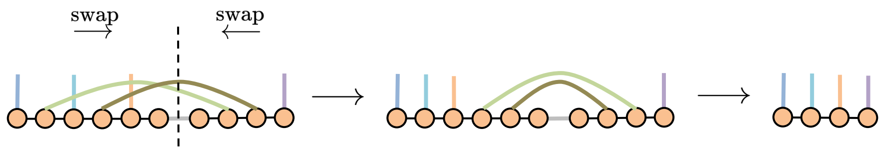
7.1.4 The CATN Algorithm



(a) Visualization of the contraction intermediates



(b) Contraction tree



(c) Steps to contract two MPSs into an output MPS

Figure 7.3: Illustration of the CATN algorithm. The input tensor network is a clique with 5 vertices. In (a), the contraction intermediates are shown following the contraction tree in (b). Each dashed box denotes the part of the tensor network that is approximated as an MPS. In (c), we visualize the steps to contract the two MPSs in the dashed box of the fourth diagram in (a) into an output MPS. Swaps are performed to move the contracted edges to the MPS boundaries, and then sites adjacent to contracted edges are eliminated.

Algorithm 7.1: The CATN algorithm

```
1: Input: Tensor network $G = (V, E)$, the contraction tree $T^V$, rank threshold $\gamma$
2: for each $v \in V$ do
3:     $\sigma^{E(v)} \leftarrow$ an arbitrary MPS site ordering for $v$
4:     $\mathfrak{T}^v \leftarrow \texttt{MPS}(v, \sigma^{E(v)})$
5: end for
6: for contraction $(U_i, V_i)$ in a topological sort of $T^V$ do
7:     $\sigma^{E(U_i \cup V_i)} \leftarrow$ an MPS site ordering for $U_i \cup V_i$
8:     $\mathfrak{T}^{U_i \cup V_i} \leftarrow$ contract $\mathfrak{T}^{U_i}, \mathfrak{T}^{V_i}$ into an MPS with the site ordering $\sigma^{E(U_i \cup V_i)}$ and maximum
   rank $\gamma$
9: end for
10: Return: $\mathfrak{T}^V$
```

Here we review the CATN algorithm. To contract a tensor network $G = (V, E)$, the algorithm relies on a given contraction tree T^V. In the algorithm, each initial and intermediate tensor generated in the contraction tree that is the contraction output of $S \subset V$ is approximated as an MPS with a given edge ordering $\sigma^{E(S)}$. The algorithm is presented in Algorithm 7.1 and is visualized in Fig. 7.3.

For each contraction that contracts vertices U_i and V_i (Line 8 in Algorithm 7.1), adjacent swaps are first used to reorder the MPS that represents U_i, V_i so that the contracted edges $E(U_i, V_i)$ are on the boundaries of the two MPSs, and then these two MPSs are concatenated with the contracted edges in the middle. These edges are then contracted/eliminated in the new MPS, and adjacent swaps are used to change the ordering of the MPS to $\sigma^{E(U_i \cup V_i)}$. These steps are visualized in Fig. 7.3c.

The input contraction tree and the MPS concatenation ordering could affect the total number of swaps. For example, the contraction tree specifies which pair of MPSs are concatenated at each contraction step, but it does not specify the concatenation ordering, and concatenating two MPSs with ordering $\sigma^{E_1}, \sigma^{E_2}$ can either yield an ordering of $\sigma^{E_1} \oplus \sigma^{E_2}$ or $\sigma^{E_2} \oplus \sigma^{E_1}$, which can have a different number of swaps. The number of swaps could also be affected by the site orderings for both the input and the intermediate MPSs.

Since the MPS ranks are non-uniform, it is also important to perform swaps with low computational costs. In Section 7.2, we propose the CATN-GO algorithm that focuses on achieving both a small number of swaps and low computational costs.

7.2 CATN WITH A GLOBAL ORDERING

In this section, we introduce the CATN-GO approach. The algorithm's pseudocode is outlined in Algorithm 7.2. It takes as input a given tensor network $G = (V, E)$, a specified vertex ordering σ^V over V, and a rank threshold γ. The threshold guarantees that the rank of every MPS involved in the contractions is at most γ. As we will explain in Section 7.3, the input σ^V can be obtained through multiple heuristics to minimize $\mathrm{cr}(G, \sigma^V)$. Within the algorithm, both the contraction tree T^V and the orderings of MPS sites are determined based on the input vertex ordering σ^V. Specifically, the MPS site orderings are optimized to minimize the number of MPS swaps required by the algorithm. Meanwhile, the contraction tree T^V is designed to minimize the computational cost under the fixed σ^V, which will be detailed in Section 7.5.

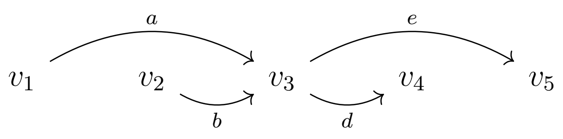


Figure 7.4: Illustration of the canonical ordering of the vertex v_3 with a vertex ordering of v_1, v_2, v_3, v_4, v_5. The canonical ordering is $\sigma^{E(v_3)} = (b, a, e, d)$.

Algorithm 7.2: CATN-GO: CATN with a global ordering

```
1: Input: Tensor network $G = (V, E)$, vertex ordering $\sigma^V$, rank threshold $\gamma$
2: for each $v \in V$ do
3:     $\sigma^{E(v)} \leftarrow$ canonical_ordering$(v, \sigma^V, G)$
4:     $\mathfrak{T}^v \leftarrow$ MPS$(v, \sigma^{E(v)})$
5: end for
6: $T^V \leftarrow$ contraction_tree$(\sigma^V, G)$                ▷ detailed in Section 7.5
7: for contraction $(U_i, V_i)$ in a topological sort of $T^V$ do
8:     $\mathfrak{T}^{U_i \cup V_i} \leftarrow$ MPS_times_MPS$(\mathfrak{T}^{U_i}, \mathfrak{T}^{V_i}, \gamma)$
9: end forreturn $\mathfrak{T}^V$
```

7.2.1 The MPS Site Orderings in CATN-GO

For each vertex $v \in V$, the MPS site ordering is selected as the *canonical edge ordering* of v on top of σ^V, which is illustrated in Fig. 7.4. The canonical ordering contains two parts. The first part includes edges connecting v with the vertices on the left of v, and the second part includes edges connecting v with vertices on the right. For an edge $e = (u_1, u_2)$, we define the length of e as $|\sigma^V(u_1) - \sigma^V(u_2)|$. In the first part of the canonical ordering, edges are sorted based on the length from the shortest to the longest. In the second part, edges are sorted from the longest to the shortest.

Let $\Phi = \{\sigma^{E(v)} : v \in V\}$ denote the set of MPS site orderings, and let V be ordered as $v_1, \dots, v_{|V|}$ in σ^V. We use $\mathsf{site\_ordering}\left(\sigma^V, \Phi\right) = \sigma^{E(v_1)} \oplus \cdots \oplus \sigma^{E(v_{|V|})}$ to denote the concatenation of all MPS site orderings based on σ^V. For each intermediate MPS generated during the contractions, Line 5 in Algorithm 7.3 guarantees that its site ordering is chosen to be a sub-ordering of $\mathsf{site\_ordering}\left(\sigma^V, \Phi\right)$.

7.2.2 The MPS-times-MPS Algorithm

The MPS-times-MPS algorithm is presented in Algorithm 7.3. During each contraction, the algorithm eliminates one contracted edge e at a time. The edge e is greedily picked when it has the smallest absolute difference of the incident vertex orders and yields the lowest elimination cost. We perform swaps between the pair of vertices incident to e until they become neighboring vertices, after which e is eliminated. As will be shown in Lemma 7.4, this ensures that each swap executed in the algorithm eliminates one edge crossing. In Section 7.4, we demonstrate that Algorithm 7.3, combined with the canonical edge orderings of MPSs, guarantee that the number of swaps employed in Algorithm 7.2 equals the lower bound, $\mathrm{cr}(G, \sigma^V)$.

When eliminating a specific contracted edge $e = (u_1, u_2)$ in the MPS $\mathfrak{T}^{U_i \cup V_i}$ (Line 9), we

Algorithm 7.3: MPS_times_MPS

```
1: Input: The two input MPSs $\mathfrak{T}^{U_i}, \mathfrak{T}^{V_i}$, rank threshold $\gamma$
2: $\sigma^{E(U_i)}, \sigma^{E(V_i)} \leftarrow$ the site ordering of $\mathfrak{T}^{U_i}, \mathfrak{T}^{V_i}$
3: $\sigma^{E(U_i \cup V_i)} \leftarrow \sigma^{E(U_i)} \oplus \sigma^{E(V_i)}$
4: $S \leftarrow E(U_i) \cap E(V_i)$                    ▷ Edges to be contracted
5: $\mathfrak{T}^{U_i \cup V_i} \leftarrow$ concatenate $\mathfrak{T}^{U_i}, \mathfrak{T}^{V_i}$ with the site ordering being $\sigma^{E(U_i \cup V_i)}$
6: while $S \neq \emptyset$ do
7:     $W \leftarrow$ subset of $S$ with each $e \in W$ has the smallest distance of the incident site orders in $\sigma^{E(U_i \cup V_i)}$
8:     $e \leftarrow$ edge with the lowest elimination cost in $W$
9:     $\mathfrak{T}^{U_i \cup V_i} \leftarrow$ eliminate_edge$(e, \mathfrak{T}^{U_i \cup V_i}, \gamma)$
10:    $\sigma^{E(U_i \cup V_i)} \leftarrow$ edge ordering of the updated $\mathfrak{T}^{U_i \cup V_i}$
11:    $S \leftarrow S \setminus \{e\}$
12: end while
13: Return: $\mathfrak{T}^{U_i \cup V_i}$
```

can perform a combination of right swaps and left swaps to bring the corresponding sites adjacent to each other. The number of swaps required equals $d = |\sigma^{E(U_i \cup V_i)}(u_2) - \sigma^{E(U_i \cup V_i)}(u_1)|$. Specifically, we can apply x right swaps, where $x \in \{0, \ldots, d\}$, along with $d - x$ left swaps. Since the MPS ranks are non-uniform, selecting different values of x results in varying computational costs. In our algorithm, we upper-bound the MPS swap cost with each x using the computational cost model presented in Section 7.1.2, and choose the value of x that minimizes such cost. The process of selecting the optimal x requires a time complexity of $\Theta(d) = O(|E(U_i \cup V_i)|)$, where $|E(U_i \cup V_i)|$ equals the number of sites in the MPS.

Using the computational cost model in Section 7.1.2, we are able to establish an upper bound on the cost of the two-MPS contraction in Algorithm 7.3. Let $\mathfrak{T}^{U_i}, \mathfrak{T}^{V_i}$ be the input two MPSs, we use

$$\mathsf{contraction\_cost}\left(\mathfrak{T}^{U_i}, \mathfrak{T}^{V_i}\right) \tag{7.1}$$

to denote such contraction cost upper bound. (7.1) will be used in Section 7.5 and Section 7.8 to construct contraction cost-efficient contraction trees.

Evaluating (7.1) yields a cost/running time of $O\left(|E(U_i, V_i)|^2 \cdot |E(U_i \cup V_i)|\right)$. Specifically, to eliminate each edge $e \in S$ (defined in Line 4), we evaluate the elimination cost of all remaining $O(|E(U_i, V_i)|)$ contracted edges, and this evaluation has a complexity of $O\left(|E(U_i, V_i)| \cdot |E(U_i \cup V_i)|\right)$. Since there are $|E(U_i, V_i)|$ edges to be eliminated, the overall running time for evaluating (7.1) becomes $O\left(|E(U_i, V_i)|^2 \cdot |E(U_i \cup V_i)|\right)$.

We derive the lower bound of the number of swaps of adjacent MPS sites needed in the CATN algorithm. For a given tensor network graph $G = (V, E)$ and a set of MPS site orderings $\Phi = \{\sigma^{E(v)} : v \in V\}$, we let $\mathsf{MPS\_graph}\,(G, \Phi)$ denote the tensor network graph consisting of all tensors in the MPSs. Each MPS has a site ordering of $\sigma^{E(v)}$ and is the MPS decomposition of the tensor represented by v in G. To differentiate between the vertices in the original graph G and the MPS graph, we refer to each vertex in the MPS graph as a site in the MPS.

For a fixed vertex ordering σ^V and a fixed site ordering set Φ and let $\hat{G} = \mathsf{MPS\_graph}\,(G, \Phi)$, in Lemma 7.1, we show that the number of swaps necessary in the CATN algorithm is lower bounded by $\mathrm{cr}\left(\hat{G}, \mathsf{site\_ordering}(\sigma^V, \Phi)\right)$, where $\mathsf{site\_ordering}$ is defined in Section 7.2. This result, together with Lemma 7.2, is used to prove Theorem 7.1, which establishes a lower bound on the number of adjacent swaps for any σ^V and Φ.

Lemma 7.1. *Consider a given MPS graph $\hat{G} = \left(\hat{V}, \hat{E}\right)$ and a given site ordering $\sigma^{\hat{V}}$. Then the number of adjacent swaps in the CATN algorithm with any choice of contraction tree $T^{\hat{V}}$ to remove all edge crossings cr $\left(\hat{G}, \sigma^{\hat{V}}\right)$ is at least the number of edge crossings.*

Proof. Given that $\hat{G}$ is a graph of tensors in MPSs, $\hat{G}$ has the property that each site is connected to only one edge that is not incident on its neighboring sites, and we call such edge "physical edge". If the two physical edges remain uncontracted, then the swapping will not change the number of crossings. Otherwise, each adjacent swap can either induce a crossing between the physical edges of the swapped sites or eliminate such crossing. Therefore, each adjacent swap in $\sigma^{\hat{V}}$ can eliminate at most one edge crossing. This directly implies the lemma statement. Q.E.D.

Lemma 7.2. *Consider a given tensor network $G = (V, E)$ and a vertex ordering σ^V. Then for any site ordering set Φ, we have the inequality,*

$$cr\left(G, \sigma^V\right) \leq cr\left(\mathsf{MPS\_graph}\,(G, \Phi), \mathsf{site\_ordering}(\sigma^V, \Phi)\right). \tag{7.2}$$

Proof. Let $\hat{G} = \mathsf{MPS\_graph}\,(G, \Phi)$ and let $\sigma^{\hat{V}} = \mathsf{site\_ordering}(\sigma^V, \Phi)$. According to the definition, if there is a crossing of contracted edges in G, σ^V with incident vertex orders (i, j) and (k, l), there must be a corresponding crossing of edges in $\hat{G}, \sigma^{\hat{V}}$. The sites incident on this crossing will be located in the ith, jth, kth, and lth MPS, respectively. The statement also holds when one of the edges is uncontracted. Therefore, it follows that

$\mathrm{cr}(G, \sigma^V) \leq \mathrm{cr}(\hat{G}, \sigma^{\hat{V}})$. If there are edge crossings for sites within one MPS in $\hat{G}$, then we will have $\mathrm{cr}(G, \sigma^V) < \mathrm{cr}(\hat{G}, \sigma^{\hat{V}})$. Q.E.D.

Theorem 7.1 (Lower bound for the number of adjacent swaps in CATN)**.** *The number of adjacent swaps necessary during the CATN algorithm to contract a tensor network $G = (V, E)$ is lower-bounded by the convex crossing number $\min_{\sigma^V} cr\left(G, \sigma^V\right)$.*

Proof. Consider a fixed vertex ordering σ^V and a fixed MPS site ordering set Φ, and let $\hat{G} = \mathsf{MPS\_graph}\left(G, \Phi\right)$ and let $\sigma^{\hat{V}} = \mathsf{site\_ordering}(\sigma^V, \Phi)$. Based on Lemma 7.1, the number of swaps is lower bounded by $\mathrm{cr}\left(\hat{G}, \sigma^{\hat{V}}\right)$. In addition, based on Lemma 7.2, for any Φ we have $\mathrm{cr}\left(\hat{G}, \sigma^{\hat{V}}\right) \geq \mathrm{cr}\left(G, \sigma^V\right)$, thus the number of swaps is lower bounded by $\min_{\sigma^V} \mathrm{cr}\left(G, \sigma^V\right)$ for any σ^V, Φ. This finishes the proof. Q.E.D.

7.4 THE NUMBER OF SWAPS IN CATN-GO

We show in Theorem 7.2 that the number of swaps employed in CATN-GO (Algorithm 7.2) equals the lower bound $\mathrm{cr}(G, \sigma^V)$. We first show in Lemma 7.3 that assigning each ordering in $\Phi = \{\sigma^{E(v)} : v \in V\}$ the canonical ordering can make the inequality in Lemma 7.2 an equality. In addition, we show in Lemma 7.4 that each swap in CATN-GO always eliminates an edge crossing.

It is important to note that, in addition to CATN-GO, any algorithm that performs swaps to eliminate edge crossings will have a number of swaps equal to the lower bound, as long as each swap executed in the algorithm eliminates a single edge crossing.

Lemma 7.3. *The equality $cr\left(\hat{G}, \sigma^{\hat{V}}\right) = cr\left(G, \sigma^V\right)$ holds when Φ is a set of canonical site orderings and $\hat{G} = \mathsf{MPS\_graph}\left(G, \Phi\right), \sigma^{\hat{V}} = \mathsf{site\_ordering}(\sigma^V, \Phi)$.*

Proof. Based on the definition, if there is a crossing of contracted edges in graph G with vertex orders (i, j) and (k, l), it implies that there must be a corresponding crossing of edges in graph $\hat{G}$ with the incident sites being the i-th, j-th, k-th, and l-th MPS. Moreover, the canonical ordering ensures that no edge crossing that is incident to a pair of sites belonging to the same MPS will occur. As a result, this condition ensures that the equality holds. Q.E.D.

Lemma 7.4. *Let $\hat{G} = \mathsf{MPS\_graph}\left(G, \Phi\right)$ and $\sigma^{\hat{V}} = \mathsf{site\_ordering}(\sigma^V, \Phi)$. Each swap in Algorithm 7.2 will always eliminate one edge crossing of $\hat{G}$ on $\sigma^{\hat{V}}$.*

Proof. Swaps are used in Line 9 to eliminate each contracted edge $e \in E(U_i, V_i)$. For a specific contracted edge e, we consider a sequence of swaps that swaps both adjacent sites of e towards the middle, and show that each swap will always eliminate one edge crossing.

Let the two sites incident on e have orders j and n, and $j < n$. In addition, consider the operation that swaps jth site to the right. Let the edge incident on the $j+1$th site be denoted as e'.

For the case where the $j+1$th site is the only site in the MPS that is adjacent to e', we can easily see that the swap between j and $j+1$th site eliminates an edge crossing.

Consider the other case where e' is incident on two sites in the MPS with orders $j+1$ and m. Assuming there is no edge crossing elimination in such case, then we must have $j+1 < m < n$, meaning that e' has a smaller absolute difference of the incident vertex orders. This contradicts Line 7, which means that e has the smallest absolute difference of the incident vertex orders. This finishes the proof.

Q.E.D.

Lemma 7.4 implies that the number of swaps in CATN-GO equals

$$\mathrm{cr}\left(\mathsf{MPS\_graph}\left(G, \Phi\right), \mathsf{site\_ordering}(\sigma^V, \Phi)\right). \tag{7.3}$$

Below we show the main theorem of the section.

Theorem 7.2. *Consider a given tensor network $G = (V, E)$ and a given vertex ordering σ^V. Then CATN-GO (Algorithm 7.2) performs a number of swaps that matches the lower bound $cr\left(G, \sigma^V\right)$.*

Proof. Let $\hat{G} = \mathsf{MPS\_graph}\left(G, \Phi\right)$ and let $\sigma^{\hat{V}} = \mathsf{site\_ordering}(\sigma^V, \Phi)$. Based on Lemma 7.3, since each ordering in Φ is set as a canonical ordering, we have $\mathrm{cr}\left(G, \sigma^V\right) = \mathrm{cr}\left(\hat{G}, \sigma^{\hat{V}}\right)$. Based on Lemma 7.4, each swap eliminates one edge crossing, so the number of swaps equals $\mathrm{cr}\left(\hat{G}, \sigma^{\hat{V}}\right)$. This finishes the proof.

Q.E.D.

7.5 FINDING AN EFFICIENT CONTRACTION TREE VIA DYNAMIC PROGRAMMING

In this section, we present a dynamic programming algorithm to find the contraction tree that yields efficient computational cost. Given a tensor network $G = (V, E)$ and a vertex linear ordering σ^V, the goal is to solve the objective

$$\min_{T \in \Pi(\sigma^V)} \sum_{(U_i, V_i) \in T} \mathsf{contraction\_cost}\left(\mathfrak{T}^{U_i}, \mathfrak{T}^{V_i}\right), \tag{7.4}$$

where $\Pi(\sigma^V)$ is the set of contraction trees constrained by σ^V, $\mathsf{contraction\_cost}$ is defined in Eq. (7.1) and denotes the cost to contract two MPSs, and (U_i, V_i) denotes a contraction with $U_i, V_i \subset V$. As described in Section 7.2, for each subset $W \subseteq V$, the site ordering of the MPS, $\mathfrak{T}^W$, is a sub-ordering of $\mathsf{site\_ordering}(\sigma^V, \Phi)$, with each $\sigma^{E(v)} \in \Phi$ corresponding to the canonical ordering of the edges incident to the vertex $v \in V$.

Let the vertices in V be ordered as $v_1, v_2, \ldots, v_{|V|}$ in σ^V. Additionally, for $1 \leq i < j \leq |V|$, we define $S(i,j) = \{v_i, \ldots, v_j\}$. The challenge in solving (7.4) arises from the potential dependence of the MPS ranks of $\mathfrak{T}^{S(i,j)}$ on the contraction path used to construct the MPS. This dependence hinders the decomposition of the optimization algorithm into independent subproblems.

In Section 7.5.1, we present a dynamic programming algorithm to derive an upper bound for the solution of (7.4). The algorithm has a time complexity of $O\left(|V|^3|E|\right)$. This algorithm assumes that all the MPSs involved in the contractions have a uniform rank γ, resulting in (7.4) being equivalent to minimizing the total length of the MPSs generated during the contractions. We present another dynamic programming algorithm that provides an upper bound of (7.4) without the uniform rank assumption, both with greater time complexity in Section 7.8.

7.5.1 Minimizing the Computational Cost for Uniform MPS Ranks

We introduce a dynamic programming algorithm to provide an upper bound for the solution of (7.4). Specifically, the contraction cost between $\mathfrak{T}^{U_i}$ and $\mathfrak{T}^{V_i}$ can be upper-bounded by considering the scenario where the ranks of both MPSs are constrained to the threshold γ. In this scenario, the computational cost is asymptotically identical for each swap operation and every MPS site orthogonalization, respectively.

We use α to denote the cost of each swap and β to denote the cost to orthogonalize each MPS site. The asymptotic computational cost can then be expressed as

$$\Theta\left(n_i \cdot \alpha + \left(|E(U_i \cup V_i)| + n_i\right) \cdot \beta\right), \tag{7.5}$$

where $|E(U_i \cup V_i)|$ denotes the number of uncontracted edges incident on $U_i \cup V_i$ and represents the length of $\mathfrak{T}^{U_i \cup V_i}$. n_i is the number of swaps involved when contracting $\mathfrak{T}^{U_i}$ and $\mathfrak{T}^{V_i}$.

We can then upper-bound the asymptotic solution of (7.4) by solving the following problem,

$$\min_{T \in \Pi(\sigma^V)} \sum_{(U_i, V_i) \in T} n_i \cdot \alpha + \left(|E(U_i \cup V_i)| + n_i\right) \cdot \beta. \tag{7.6}$$

Given that $\sum_i n_i = \mathrm{cr}(G, \sigma^V)$ holds true for all contraction trees constrained by σ^V, the problem is equivalent to

$$\min_{T \in \Pi(\sigma^V)} \sum_{(U_i, V_i) \in T} |E(U_i \cup V_i)|, \tag{7.7}$$

where the objective is to minimize the total length of the MPSs generated during the contractions.

Let $d(i, j)$ represents the sum of MPS lengths for contracting $S(i, j)$, we can then derive $d(1, |V|)$ using the following recursive equation,

$$d(i, j) = \min_{k \in \{i, \dots, j-1\}} d(i, k) + d(k + 1, j) + |E(S(i, j))|, \tag{7.8}$$

with the base cases $d(i, i) = |E(v_i)|$ for any $i \in \{1, \dots, |V|\}$.

We can compute $d(1, |V|)$ by employing memoization. Given $E(S(i, k))$ and $E(S(k+1, j))$, computing $E(S(i, j))$ requires a cost of $O(|E(S(i, k))| + |E(S(k + 1, j))|)$. Therefore, the overall time complexity of the dynamic programming algorithm is

$$O\left(\sum_{i=1}^{|V|-1} \sum_{j=i+1}^{|V|} \sum_{k=i}^{j-1} |E(S(i, k))| + |E(S(k + 1, j))| \right) = O\left(|V|^3 |E|\right). \tag{7.9}$$

Vertex ordering	$5 \times 5 \times 5$ lattice			$8 \times 8 \times 8$ lattice			$(3, 300)$-rand regular graph			$(6, 300)$-rand regular graph		
	# swaps	Time	GFlops	# swaps	Time	GFlops	# swaps	Time	GFlops	# swaps	Time	GFlops
Baseline	3090	86.3s	421	34608	2247s	9404	32864	2458s	11665	133019	10800s	52060
Recursive bisection	2080	53.7s	267	16872	1019s	4603	2522	88.5s	446	37560	2817s	13843
Relative improvements	1.5X	1.6X	1.6X	2.1X	2.2X	2.1X	13X	28X	26X	3.5X	3.8X	3.8X

Table 7.1: Comparison of the CATN performance using different vertex orderings. Experiments are performed on four different tensor network structures: a $5 \times 5 \times 5$ 3D lattice, an $8 \times 8 \times 8$ 3D lattice, a random regular graph with 300 vertices and a degree of 3, and a random regular graph with 300 vertices and a degree of 6. In the case of the 3D lattices, the baseline vertex ordering followed a sequential traversal of the 3D array. For the random regular graphs, the baseline ordering is a random ordering of the vertices. For each tensor network structure, the data collected represents the average performance over 5 runs.

It is worth noting that the dynamic programming algorithm proposed can be adapted to accommodate other metrics as well. For instance, if the objective is to minimize the tensor size represented by the MPS, one may consider replacing the term $|E(U_i \cup V_i)|$ in (7.7) with $\exp(|E(U_i \cup V_i)|)$. In such cases, a simple modification can be made to the dynamic programming algorithm by substituting the term $|E(S(i, j))|$ in (7.8) with $\exp(|E(S(i, j))|)$.

7.6 EXPERIMENTS

In this section, we conduct experiments to assess the performance of the proposed CATN-GO. We first apply CATN-GO to contract random tensor networks defined on 3D lattices and random regular graphs. By utilizing vertex orderings that minimize edge crossings, in conjunction with our algorithm for selecting the contraction tree, we significantly reduce the overall execution time. Furthermore, we evaluate CATN-GO, the original CATN algorithm [41][10], and another algorithm called SweepContractor [86][11], in contracting Ising model tensor networks defined on 3D lattices. We demonstrate a remarkable 5.9X speedup while maintaining the same level of contraction accuracy.

For implementation, we provide public access to our code repository at `https://github.com/LinjianMa/CATN-GO.jl`. This implementation is built on top of ITensors.jl [91], a publicly available Julia [230] package specifically designed for tensor network computations. ITensors.jl provides comprehensive support for MPS operations, including swapping adjacent sites and performing canonicalization. All our experiments were conducted on an Intel Core i7 2.9 GHz Quad-Core machine.

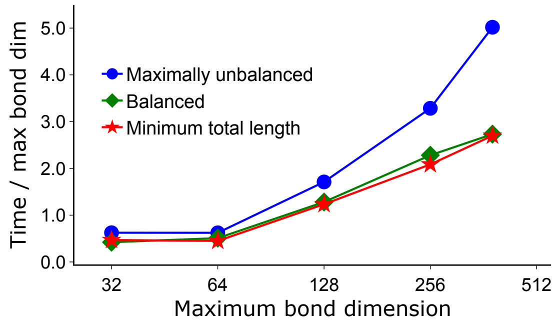
7.6.1 Comparison of Different Vertex Orderings

Our findings demonstrate that utilizing a vertex ordering that minimizes edge crossings and reduces the number of adjacent swaps can significantly decrease both the execution time and the required number of GFlops (giga floating-point operations). The results of these improvements are presented in Table 7.1. For the considered tensor networks, each element within the tensors is an i.i.d. variable uniformly distributed in the range of $[-1, 1]$. These particular tensor networks have been utilized in previous research [87] as benchmarks for evaluating contraction algorithms. In all our experiments, we set the MPS rank threshold γ to a value of 256. Additionally, we employ the dynamic programming algorithm outlined in Section 7.5.1 to generate the contraction trees.

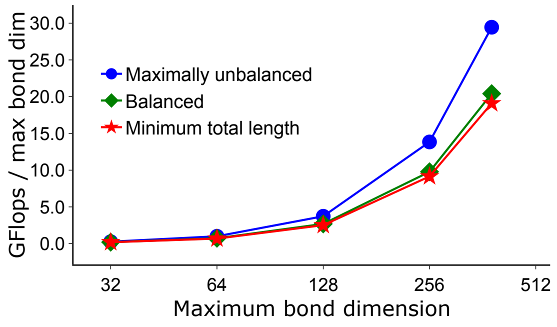
The results presented in Table 7.1 illustrate that the vertex orderings generated using the recursive bisection algorithm, as discussed in Section 7.1.3, result in substantially reduced running time compared to the baseline orderings. The baseline vertex ordering follows a sequential traversal of the 3D array for 3D lattices, and is a random ordering of the vertices for random regular graphs. Moreover, the analysis reveals that the improvements in terms of both time and GFlops are generally larger than the improvement observed in the number of

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

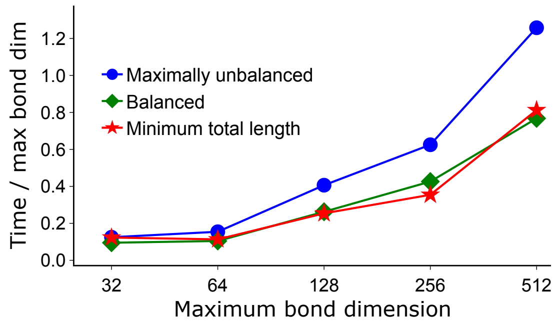
[11]We use the SweepContractor implementation at `https://github.com/chubbc/SweepContractor.jl`.



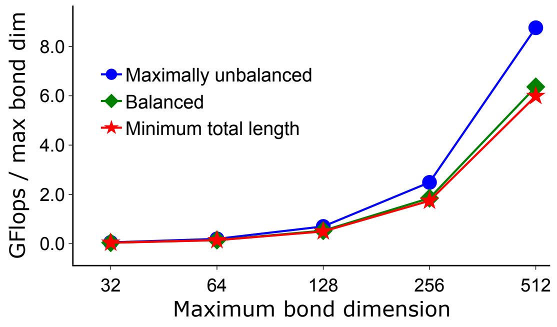
(a) $7 \times 7 \times 7$ grid



(b) $7 \times 7 \times 7$ grid



(c) Random regular graph with degree 3 and 300 vertices



(d) Random regular graph with degree 3 and 300 vertices

Figure 7.5: Comparison of three different contraction trees, the maximally-unbalanced contraction tree, the balanced contraction tree, and the tree generated via the algorithm in Section 7.5.1. (a)(b) show the results on the tensor network defined on a $7 \times 7 \times 7$ lattice, and (c)(d) show the results on random regular graphs.

swaps. This observation can be attributed to the fact that performing swaps increases the MPS ranks. Consequently, a higher number of swaps increases the average cost associated with each individual swap. While Table 7.1 does not consider the accuracy of the contraction result obtained with different vertex orderings, we empirically observed that using vertex orderings that minimize edge crossings typically results in higher accuracy for the tensor network contraction, as they lead to fewer low-rank truncations.

7.6.2 Comparison of Different Contraction Trees

To justify our proposed algorithm in Section 7.5.1, we conducted a performance comparison between the algorithm and alternative contraction trees. The results are depicted in Fig. 7.5. As baselines, we considered two approaches: a contraction tree with a balanced binary tree structure, and a maximally-unbalanced contraction tree. In the latter, the contraction process starts from the first vertex in the given vertex ordering, and sequentially contracts the previously-contracted section with the neighboring vertex. For all experiments, the vertex orderings are generated via recursive bisection.

As depicted in the figure, the contraction tree that minimizes the sum of MPS lengths consistently exhibits the lowest number of flops across all experiments. In comparison to the maximally-unbalanced contraction tree, our proposed contraction tree demonstrates a notable speed improvement of up to 1.85 times. Furthermore, the performance of the balanced contraction tree is similar to that of our proposed contraction tree. This similarity can be attributed to the fact that the tested tensor networks contain tensors of approximately equal order.

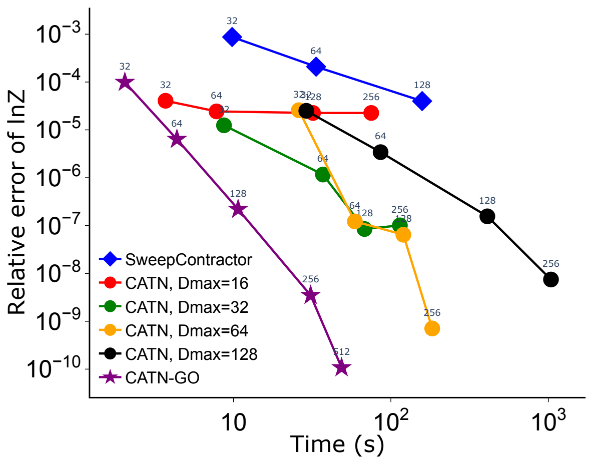


Figure 7.6: Comparison among CATN-GO, the CATN algorithm in [41], and the SweepContractor algorithm in [86]. In the reference CATN algorithm, "Dmax" is an additional input parameter of the original CATN algorithm that controls the size of the MPS uncontracted dimensions. The value on top of each point is the MPS rank threshold γ.

7.6.3 Comparison Among CATN-GO and Previous Works

Finally, we conduct a comparison among CATN-GO, the CATN algorithm introduced in [41], and SweepContractor introduced in [86] for contracting the ferromagnetic Ising model tensor network defined on a $5 \times 5 \times 5$ lattice. SweepContractor is another algorithm proposed recently for contraction arbitrary tensor networks. In the algorithm, each intermediate tensor is also approximated as an MPS, but the the algorithm only supports the maximally-unbalanced contraction trees. In our experiments, we use the same vertex ordering in both CATN-GO and SweepContractor.

The contraction output of the ferromagnetic Ising model tensor network, denoted as Z

and referred to as the partition function, can be expressed as follows,

$$Z = \sum_{\sigma_i,\sigma_j\in\{-1,1\}} \prod_{(i,j)\in E} \exp(\beta\sigma_i\sigma_j). \tag{7.10}$$

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

$$t^v_{E(v)} = \sum_i \prod_{e\in E(v)} W_{i,e}, \tag{7.11}$$

where

$$W = \frac{1}{\sqrt{2}}\begin{bmatrix}\sqrt{\cosh(\beta)}+\sqrt{\sinh(\beta)} & \sqrt{\cosh(\beta)}-\sqrt{\sinh(\beta)}\\ \sqrt{\cosh(\beta)}-\sqrt{\sinh(\beta)} & \sqrt{\cosh(\beta)}+\sqrt{\sinh(\beta)}\end{bmatrix} \tag{7.12}$$

and β is an input parameter to the model.

We show the relation between the relative error of $\ln Z$ and the running time in Fig. 7.6. The quantity $\ln Z$ is an important measure that is proportional to the free energy of the system. With the increase of the MPS rank threshold, the relative error decreases. When executing CATN-GO, we use recursive bisection to generate the vertex ordering, and use the algorithm in Section 7.5.1 to generate the contraction tree. The figure clearly demonstrates that our algorithm outperforms both the CATN algorithm and SweepContractor in terms of efficiency across all relative errors. In particular, our algorithm is 5.9X faster than both baseline algorithms to reach a relative error of less than 10^{-8}.

7.7 CONCLUSIONS

We introduce an efficient algorithm to contract tensor networks with arbitrary structures. Our work is built on top of the recently-proposed CATN algorithm [41], and we provide theoretical analysis and accelerate multiple components of the algorithm. In particular, we show the relation between the number of swap operations in the algorithm and the number of edge crossings in the graph, and we present two dynamic programming algorithms to generate efficient contraction trees. We find that the strategies proposed in the paper significantly improve the running time of tensor network contractions.

We envision numerous aspects presented in the study that have the potential for further enhancement in future research. Specifically, there remain opportunities to devise heuristics aimed at finding vertex orderings that result in fewer edge crossings. Additionally, both this study and the preceding work that introduced the CATN algorithm assume that, during the contraction of two MPSs, all contracted edges must be initially swapped towards the

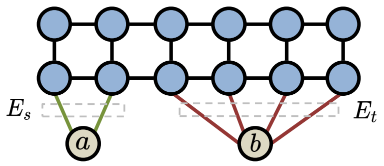


Figure 7.7: Illustration of the graph used to define $\mathsf{mincut}_G(E_s, E_t)$.

boundary. However, this assumption is suboptimal in various scenarios, leaving room for its reconsideration.

7.8 A DYNAMIC PROGRAMMING ALGORITHM BASED ON THE MPS RANK UPPER BOUND FOR EFFICIENT CONTRACTION TREE

We introduce a dynamic programming algorithm that provides an upper bound for the solution of (7.4). This algorithm runs in $O(n^3m^3)$, where n is the number of vertices and m is the number of edges in the tensor network graph. The dynamic programming algorithm is based on the observation that the resulting MPS from contracting a group of MPSs will have a consistent upper bound on MPS ranks across all contraction trees involving these input MPSs. Moreover, this upper bound can be determined by computing a sequence of minimum cuts on top of the input graph, which is detailed in Section 7.8.1.

We define the edge weights and cuts on the tensor network graph G. We use w to denote a function such that for each $e \in E$, the edge weight $w(e) = \log(s)$ is the natural logarithm of the dimension size represented by 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. We define $\mathsf{cut}_G(X, Y) = \sum_{e\in E(X,Y)} w(e)$, where $E(X, Y)$ denotes the set of edges connecting X, Y. For two vertices $s, t \in V$, we define the minimum s-t cut between s, t in G as

$$\mathsf{mincut}_G(s, t) = \min_{\substack{A,B\subset V\\ s\in A, t\in B}} \mathsf{cut}_G(A, B). \tag{7.13}$$

Let E_s, E_t be the two different subsets of the uncontracted edges of G, we define

$$\mathsf{mincut}_G(E_s, E_t) = \mathsf{mincut}_{\hat{G}}(a, b), \tag{7.14}$$

where $\hat{G} = (V \cup \{a, b\}, E)$ contains both G and two new vertices a and b, where a, b are adjacent to E_1, E_2, respectively. We illustrate the graph $\hat{G}$ used to define (7.14) in Fig. 7.7.

7.8.1 An Upper Bound of MPS Ranks

Given a tensor network $G = (V, E)$ and the set of uncontracted edges $\hat{E} \subseteq E$, if we transform the tensor network into an MPS with uncontracted edges $\hat{E}$ and a corresponding site orderings $\sigma^{\hat{E}}$, we can establish an upper bound on the ranks of the resulting MPS by analyzing the graph cuts of G. Such upper bound is presented in Theorem 7.3. This upper bound will play an important role in Section 7.8.2, where we devise efficient contraction trees for CATN based on this information.

Theorem 7.3. *Consider transforming the tensor network G with uncontracted edges $\hat{E}$ into an MPS with the same uncontracted edges and a site ordering $\sigma^{\hat{E}}$. If we order the edges in $\hat{E}$ as $e_1, \ldots, e_{|\hat{E}|}$ based on $\sigma^{\hat{E}}$ and define $L(i) = \{e_1, \ldots, e_i\}$, the rank of the bond dimension connecting the i-th and $(i+1)$-th sites in the MPS can be upper-bounded as*

$$R_G(i) = \exp\left(\mathsf{mincut}_G\left(L(i), \hat{E} \setminus L(i)\right)\right), \tag{7.15}$$

where mincut_G is defined in (7.14).

Proof. Let $\mathbf{A}_i\mathbf{B}_i$ be one exact matrix decomposition of the tensor network G, where dimensions in $L(i)$ are combined into the row of $\mathbf{A}_i$, and the dimensions in $\hat{E} \setminus L(i)$ are combined into the column of $\mathbf{B}_i$. We can observe that the size of the MPS dimension that is connected to the ith and $i + 1$th sites is upper-bounded by the column size of $\mathbf{A}_i$. Below we show that there exists such decomposition with the column size of $\mathbf{A}_i$ being $R_G(i)$, thus finishing the proof.

To achieve this, we define $\hat{V}$ and $V \setminus \hat{V}$ as the two parts that are separated by the cut of edges whose weight sum equals $\mathsf{mincut}_G\left(L(i), \hat{E} \setminus L(i)\right)$. The matrix $\mathbf{A}_i$ can then be constructed by contracting tensors within $\hat{V}$, and all the uncontracted dimensions from $L(i)$ are combined to form the rows, while the other uncontracted dimensions from $G[\hat{V}]$ are combined to form the columns, resulting in a matrix with a column size equal to $R_G(i)$.

Q.E.D.

Utilizing the state-of-the-art max flow algorithm described in [231] alongside the max flow-min cut theorem [232], the computation of each minimum s-t cut results in a cost of $O(|E|^{1+o(1)} \log(U))$, where U represents the largest edge weight. Since, in our scenario, U is generally a constant value, the cost for each cut reduces to $O(|E|^{1+o(1)})$. Consequently, the computation of all $R_G(i)$ for $i \in \{1, \ldots, |\hat{E}| - 1\}$ in (7.15) incurs a cost of $O(|V| \cdot |E|^{1+o(1)})$.

7.8.2 Minimizing the Computational Cost

For each $\mathfrak{T}^{S(i,j)}$ introduced in Section 7.5, the MPS ranks can be upper-bounded based on the technique described in Section 7.8.1. Specifically, let $G_{(i,j)}$ denote the tensor network containing all MPSs $\mathfrak{T}^{v_i}, \dots, \mathfrak{T}^{v_j}$, and let γ denote the rank threshold. The rank of the bond dimension connecting the i-th and $(i+1)$-th sites in $\mathfrak{T}^{\hat{V}(i,j)}$ can then be upper-bounded as $\min\left(R_{G_{(i,j)}}(i), \gamma\right)$, where $R_{G_{(i,j)}}(i)$ is defined in (7.15). Importantly, this upper bound is independent of the contraction tree used to construct $\mathfrak{T}^{S(i,j)}$. Based on Section 7.8.1, for each i, j, computing the upper bound of the ranks of $\mathfrak{T}^{S(i,j)}$ has a cost of $O(|V| \cdot |E|^{1+o(1)})$. This makes the cost to compute all MPS's ranks $O\left(|V|^3 \cdot |E|^{1+o(1)}\right)$.

Considering each $S(i,j)$, we define $\mathfrak{X}^{S(i,j)}$ as an MPS with ranks that correspond to the aforementioned upper bounds. Let $c(i,j)$ denote the optimal cost for contracting $S(i,j)$. To obtain an upper bound on the optimal cost $c(1,|V|)$, we employ the following recursive equation,

$$c(i,j) \leq \min_{k \in \{i,\dots,j-1\}} c(i,k) + c(k+1,j) + \mathsf{contraction\_cost}\left(\mathfrak{X}^{S(i,k)}, \mathfrak{X}^{S(k+1,j)}\right), \tag{7.16}$$

where $c(i,i) = 0$ is the base case for any $i \in \{1, \dots, |V|\}$.

We can compute the optimal upper bound for $c(1,|V|)$ by utilizing memoization. Based on the analysis in Section 7.2, computing $\mathsf{contraction\_cost}\left(\mathfrak{X}^{S(i,k)}, \mathfrak{X}^{S(k+1,j)}\right)$ requires a cost of

$$\begin{aligned} &O\left(|E\left(S(i,k), S(k+1,j)\right)|^2 \cdot |E\left(S(i,k) \cup S(k+1,j)\right)|\right) \\ &= O\left(|E\left(S(i,k), S(k+1,j)\right)|^2 \cdot |E\left(S(i,j)\right)|\right) .end \end{aligned} \tag{7.17}$$

Therefore, the overall time complexity of the dynamic programming algorithm using memoization is

$$O\left(\sum_{i=1}^{|V|-1} \sum_{j=i+1}^{|V|} |E\left(S(i,j)\right)| \cdot \sum_{k=i}^{j-1} |E\left(S(i,k), S(k+1,j)\right)|^2\right) = O\left(|V|^3 |E|^3\right). \tag{7.18}$$

To summarize, the overall cost for both computing the ranks and the dynamic programming algorithm is

$$O\left(|V|^3 \cdot |E|^{1+o(1)} + |V|^3 |E|^3\right) = O\left(|V|^3 |E|^3\right). \tag{7.19}$$

Chapter 8: TENSOR NETWORK CONTRACTION WITH A FLEXIBLE AND COST-EFFICIENT DENSITY MATRIX ALGORITHM FOR TREE APPROXIMATION

In this Chapter, we introduce a new 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. Compared to previous works and the algorithm proposed in Chapter 7, the algorithm in this Chapter has the flexibility to incorporate a larger portion of the environment when performing low-rank approximations. 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. In addition, the algorithm includes a cost-efficient density matrix algorithm [90], [91] 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.

8.1 PREVIOUS WORKS

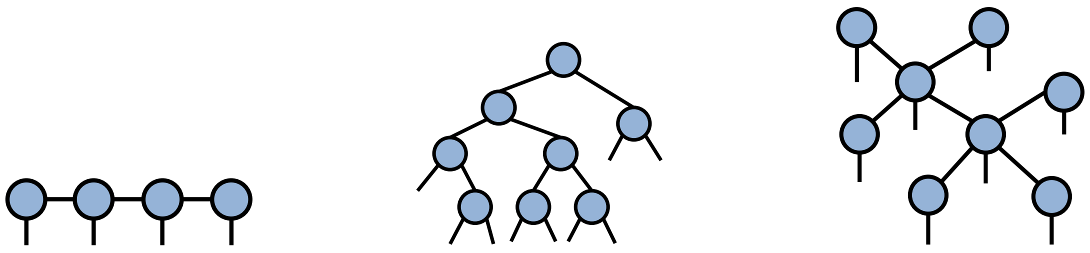

(a) MPS (b) Binary tree tensor network (c) TTNS

Figure 8.1: Illustration of the matrix product state (MPS), the (full) binary tree tensor network, and the tree tensor network state (TTNS). An MPS is a maximally-unbalanced binary tree tensor network if contracting the tensor at one end with its neighbor. Both an MPS and a binary tree tensor network are special cases of a TTNS, in which 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 matrix product state (MPS [33], also called tensor train [12]), binary tree tensor network [36], and tree tensor network state (TTNS) [233]–[235], which are visualized in Fig. 8.1. For tensor network contractions defined on regular structures, such as the 2D lattice structure, projected entangled pair states (PEPS) [33], [35], many efficient approximate contraction algorithms based on MPS approximations [83], [84] have been proposed. However, many of these methods have not been extended to other general tensor network structures.

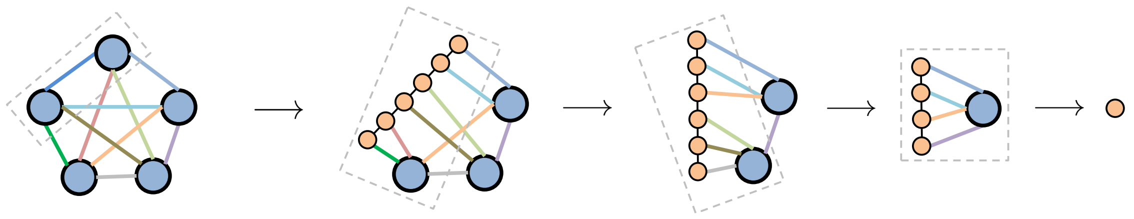


Figure 8.2: Illustration of the approximate contraction technique. Each intermediate is approximated as an MPS, which has an unbalanced binary tree structure. The left diagram is the tensor diagram of the input tensor network. Each dashed box denotes the part of the tensor network that is approximated as an MPS.

Recent works have proposed automated approximation algorithms for contracting tensor networks with more general graph structures [41], [85]–[87], [236], [237], and many of these methods employ low-rank approximation/truncation techniques. In [41], [85], [86], each intermediate tensor produced during the contraction is approximated as a binary tree tensor network, and we illustrate this approach in Fig. 8.2. In particular, [85] approximates each intermediate tensor as a general binary tree tensor network, while the algorithm proposed in [41] called Contracting Arbitrary Tensor Network (CATN) approximates each intermediate tensor as an MPS. When contracting two MPSs, CATN swaps/permutes the dimensions that connect both MPSs to the boundaries. Then, it contracts these dimensions to obtain the output MPS. The adjacent dimension swaps are the bottleneck for complexity in CATN. In another algorithm proposed in [86] 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. [87] 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 [41] and SweepContractor [86], it is essential to carefully select the binary tree/MPS structures and permutations (i.e., a mapping from tensor modes onto binary tree vertices) [238]. These choices should yield accurate low-rank approximations while enabling efficient subsequent contractions. However, previous works such as [41], [85], [86] have made arbitrary selections for these structures.

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. 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 various problems. Hence, it becomes crucial to provide the 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 [41], [86], the selection of environments was implicitly determined by the algorithm. For instance, in the CATN algorithm [41], truncation takes place during adjacent swaps of MPS dimensions, with the environment consisting of all tensors in the target MPS. Similarly, the SweepContractor algorithm [86] 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 [87] introduces user-specified environment sizes. The method utilized tree-structured environments $\hat{\mathbf{E}}$, which is constructed by including a spanning tree of tensors around the pair of tensors to be truncated. [87] 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.

8.2 OUR CONTRIBUTIONS

We propose a new approach for performing approximate contractions of arbitrary tensor networks. This approach follows the technique used in [41], [85], [86], 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.

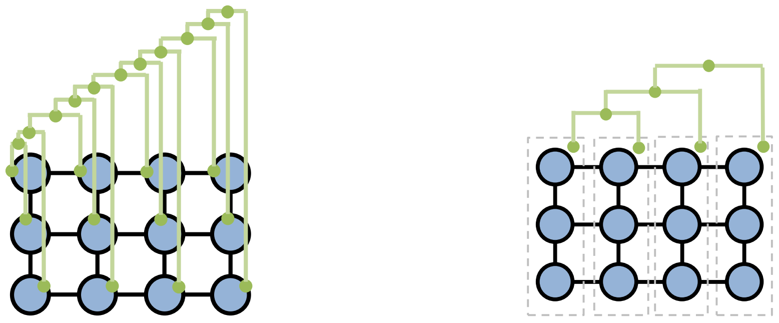

(a) Complete contraction tree (b) Contraction tree on the partitioned network

Figure 8.3: 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 [87]. In (b), each dotted box denotes a partition of the tensor network.

Firstly, it efficiently handles the environment with different sizes and provides the flexibility to incorporate larger environments compared to the methods employed in [41], [85], [86]. Unlike prior work that contracts 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 the two contraction trees in Fig. 8.3. 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 [239] and Metis [240] for partitioning the tensor network, or manually selecting suitable partitions for specific problems. In Section 8.4.2, we will demonstrate how the utilization of the partial contraction tree abstraction enables the straightforward extension of various contraction algorithms designed for the 2D grid with different environments, including those that have not been automated in the prior work [41], [85], [86].

Secondly, we provide a new approach to approximate a given tensor network into a binary tree structure, as depicted in Fig. 8.4. This approach is composed of the following three novel components.

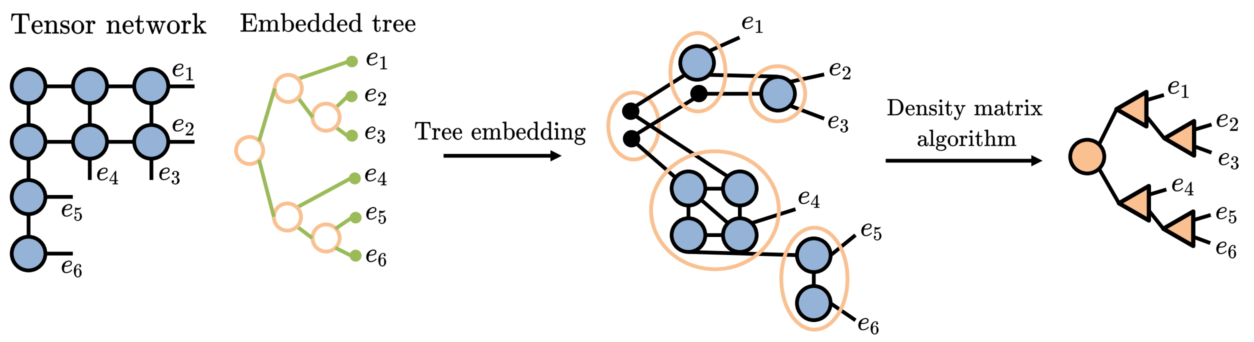


Figure 8.4: Illustration of the process to approximate the input tensor network (left diagram) into a binary tree tensor network (right diagram). The embedding tree is a rooted binary tree that represents the output tree structure. The tree embedding step maps a partition of the input tensor network to each non-leaf (orange) vertex in the embedding tree. Finally, the density matrix algorithm approximates the embedded tensor network into a binary tree tensor network. Each black dot in the diagrams represents an identity matrix.

- It encompasses a new heuristic for generating binary tree structures and permutations (i.e., a mapping from tensor modes onto binary tree vertices [238]) of intermediate tensor networks. The binary tree structure is also called the embedding tree in Fig. 8.4 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 dimensions. 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 [241]–[245], allowing for an accurate approximation with low ranks in the resulting tree tensor network. The details of the algorithm can be found in Section 8.6.

- 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 input tensor network as the environment. When compared to the canonicalization-based algorithm that employs the same environment, the density matrix algorithm exhibits lower or the same 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 8.7.

- In scenarios where 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 the swap-based algorithm to approximate the tree. The swap-based algorithm, extensively utilized in MPS-based tensor network contraction algorithms such as CATN and SweepContractor, uses a sequence of adjacent swaps of MPS dimensions to change the MPS permutation. Within our algorithm, a sequence of density matrix algorithms is performed, each one 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 8.8.

In Section 8.9, 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 [41] and the SweepContractor proposed in [86] when considering tensor networks defined on lattices using the Ising model. Specifically, our approach achieves a 9.2X speed-up in execution time while maintaining the same level of accuracy. This improvement in speed demonstrates the efficiency of our approach.

8.3 DEFINITIONS AND THE COMPUTATIONAL COST MODEL

8.3.1 Tensor Network Definitions

We introduce the tensor network notation here. The structure of a tensor network can be described by an undirected graph $G = (V, E, w)$, also called a tensor diagram. 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 a function such that for each $e \in E$, $w(e) = \log(s)$ is the natural logarithm of the dimension sizes represented by 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. For a vertex $v \in V$, we use $N(v)$ to denote the set of vertices adjacent to v.

We use $G[S] = (S, E_S, w)$ 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 subsets of V denoted as X, Y, we let $E(X, Y)$ denote the set of edges connecting X, Y. We use $E(X, *)$ to denote all uncontracted edges only adjacent to X, $E(X, *) = \{(u) \in E : u \in X\}$. We let $E(X)$ denote the set of uncontracted edges of $G[X]$, $E(X) = E(X, V \setminus X) \cup E(X, *)$.

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

$$\mathsf{mincut}_G(u, v) = \min_{\substack{A,B \subset V \\ u \in A, v \in B}} \mathsf{cut}_G(A, B). \tag{8.1}$$

Let E_1, E_2 be the two different subsets of the uncontracted edges of G. $\mathsf{mincut}_G(E_1, E_2)$ is defined 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.

For the tensor network represented by $G = (V, E, w)$, 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 partitioning 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 $V' \subset V$, we use $\mathsf{path}(T^{(\mathcal{V})}, V')$ to 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 $\mathcal{V}'$ as well as the children of these ancestors. To illustrate, we provide an example of the sub-contraction path in Fig. 8.5.

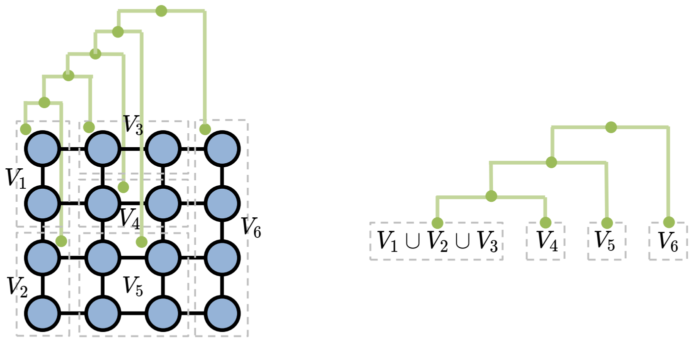


Figure 8.5: Illustration of the sub-contraction path. The left diagram denotes the graph partitioning and the contraction tree $T^{(\mathcal{V})}$, and the right diagram denotes the sub-contraction path $\mathsf{path}(T^{(\mathcal{V})}, V_4)$.

8.3.2 The Computational Cost Model

We summarize the computational cost model used throughout the paper. 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 $G = (V, E, w)$, can be cast as a matrix multiplication, and the overall asymptotic cost is

$$\Theta\left(\exp\left(w(E(v_a)) + w(E(v_b)) - w(E(v_a, v_b))\right)\right). \tag{8.2}$$

Above we assume the classical matrix multiplications rather than fast algorithms such as Strassen's algorithm [61] are employed.

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 dimension 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 [184]. The asymptotic cost of this algorithm is $O(mnr)$.

8.4 BACKGROUND

This section offers background for the proposed approach. In Section 8.4.1, we provide a short survey of several common tensor networks discussed in the paper. In Section 8.4.2, 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 8.7. Section 8.4.3 covers the standard swap-based algorithm used to permute MPS dimensions, which serves as a motivation for our algorithm that combines the density matrix algorithm and the swap-based algorithm, as outlined in Section 8.8. Furthermore, in Section 8.4.4, we delve into the definition and heuristics of the graph embedding problem, which is utilized in Section 8.6 to select an efficient binary tree structure.

8.4.1 A Survey of Common Tensor Network Structures

We survey both tree tensor networks and tensor networks defined on lattices. The MPS [12], [33], binary tree tensor network [36], and the tree tensor network state (TTNS) [233]–[235] are illustrated in Fig. 8.1. An MPS is a tensor network with a linear structure, with each tensor having one uncontracted dimension. The binary tree tensor network has a rooted binary tree structure, and all non-root vertices have an order of three. In a TTNS, each tensor has an uncontracted dimension, 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 TTNS, especially when considering a fixed rank r. When the uncontracted dimension size 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 $O(r^3)$.

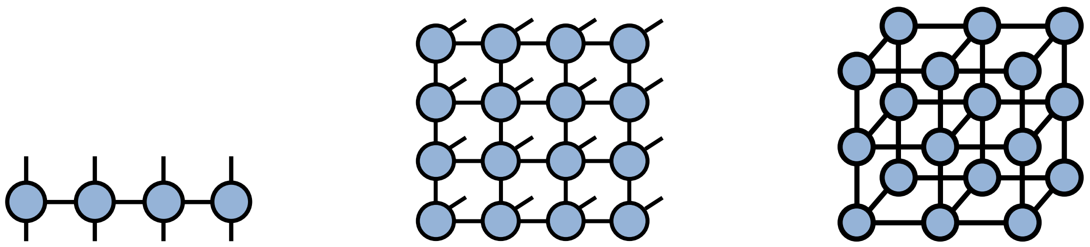

(a) MPO (b) PEPS (c) 3D lattice tensor network

Figure 8.6: 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. 8.6 and Fig. 8.3 provide visual representations of other tensor networks, including the Matrix Product Operator (MPO), the Projected Entangled Pair States (PEPS) [33], [35], and the tensor network defined on lattices. 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 is adjacent to two uncontracted edges.

8.4.2 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 envionment tensor network. The low-rank approximation problem that is widely used in this work can be framed as

$$\min_{\hat{\mathbf{A}} \in \mathbb{R}^{b \times r}, \mathbf{V} \in \mathbb{R}^{c \times r}} \left\| \mathbf{EAB} - \mathbf{E}\hat{\mathbf{A}}\mathbf{V}^T \right\|_F, \quad \text{s.t. } \mathbf{V}^T\mathbf{V} = \mathbf{I}, \tag{8.3}$$

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 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 8.3.2. 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 for cases when the number of rows in $\mathbf{E}$ is large.

In Section 8.4.2, we review the canonicalization-based algorithm to reduce the dimension sizes of tree tensor networks. We will show in Section 8.7.2 that the cost of the density matrix algorithm is upper-bounded by the canonicalization-based algorithm. In Section 8.4.2, we provide a review of existing algorithms employed in truncating the MPO-MPS contraction.

The canonicalization-based algorithm for truncating tree tensor networks

We review the canonicalization-based algorithm to truncate the tree tensor network [246]. We first introduce the canonical form in Definition 8.1. 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.

Definition 8.1 (Canonical form)**.** *Consider a tensor network with a tree structure $T = (V_T, E_T, w)$. For a given vertex $u \in V_T$ and an edge (u, v), let $S \subseteq V_T$ 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. 8.7.*

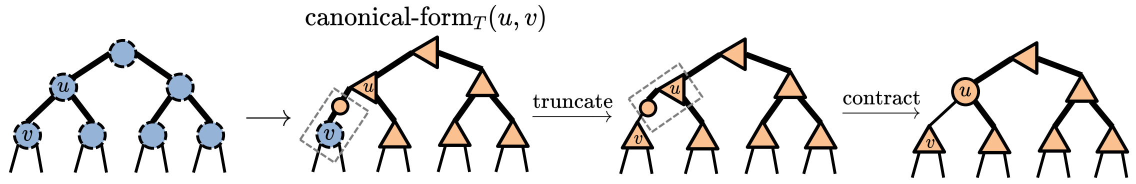


Figure 8.7: Illustration of truncating the dimension represented by the edge (u, v) through canonicalization.

The canonicalization-based algorithm is shown in Algorithm 8.1. It proceeds by computing the truncated network through a post-order 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 persists until only the root vertex remains, which contains the comprehensive non-orthogonal information of the entire network.

Algorithm 8.1: The canonicalization-based algorithm for truncating the tree tensor network

```
 1: **Input:** The tree tensor network $T = (V_T, E_T, w)$, the maximum bond dimension $\chi$, the
    root vertex $r$
 2: $T_d \leftarrow$ a directed tree of $T$ with the root being $r$
 3: **for** each $v \in V_T \setminus \{r\}$ based on a post-order DFS traversal of $T_d$ **do**
 4:     $u \leftarrow \texttt{parent}(T_d, v)$
 5:     Change the tree tensor network to $\texttt{canonical\_form}_T(u, v)$ with the non-orthogonal
        matrix denote $\mathbf{R}_u$
 6:     $\mathbf{M}_v \leftarrow$ matricization of the tensor at $v$ with the dimension connecting $u, v$ combined
        into the column
 7:     $\mathbf{U}_v\hat{\mathbf{R}}_u \leftarrow$ rank-$\chi$ approximation of $\mathbf{M}_v\mathbf{R}_u$ with $\mathbf{U}_v$ being orthogonal
 8:     Update the tensor at $u$ as $\hat{\mathbf{R}}_u\mathbf{M}_u$
 9: **end for**
10: **return** the tree tensor network that contains all $\mathbf{U}_v$ and the root tensor $\mathbf{M}_r$
```

Existing algorithms for truncating the MPO-MPS multiplication

We provide a review of a set of algorithms to truncate the output of the MPO-MPS multiplication. These algorithms are widely used in the boundary-based algorithm to approximately contract the 2D lattice tensor network surveyed in Section 8.4.1. The contraction algorithm initiates the process with a boundary MPS of the 2D network (e.g., the leftmost MPS in Fig. 8.3b), and at each step, it applies the adjacent MPO to it and approximate the resulting output as a low-rank MPS. The 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 [47], [90] have explored various algorithms for MPO-MPS multiplication. These algorithms include approaches based on canonicalization [47], [247], the density matrix algorithm [90], [91], and the iterative fitting algorithm [247]. 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 they offer theoretical guarantees for the resulting output.

Algorithms that use canonicalization We review two different canonicalization-based algorithms, the zip-up algorithm [247] and the canonicalization algorithm with full environment [47]. 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 other dimensions. The computational cost comparison between the algorithms is summerized in Table 8.1.

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 8.1: Comparison of asymptotic algorithmic complexity between the zip-up algorithm, the canonicalization-based algorithm that uses the full environment, and the density matrix algorithm.

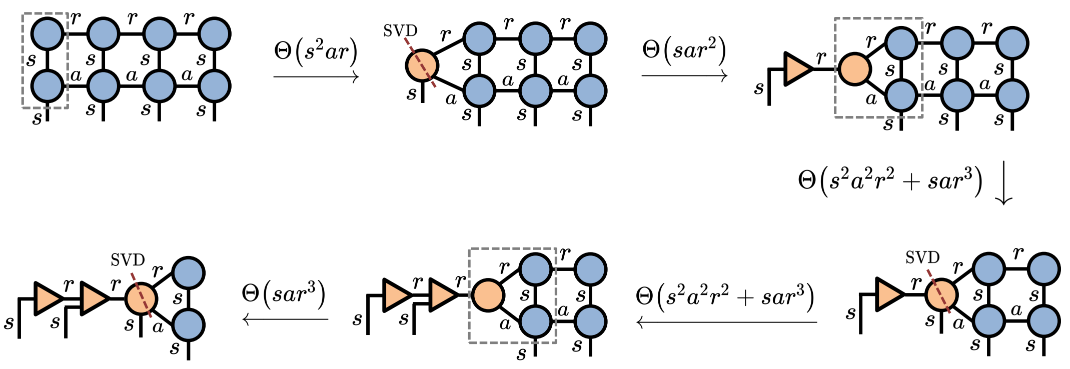


Figure 8.8: Illustration of the zip-up algorithm. Each dashed block includes the tensors to be contracted at a given step. Each tensor represented by a triangular vertex denotes a tensor with an orthogonality property.

The zip-up algorithm [47], [247] is illustrated in Fig. 8.8. 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. 8.8, 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).

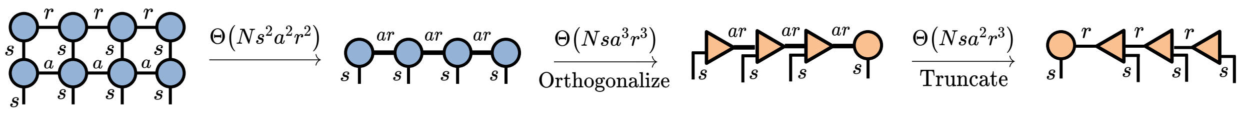


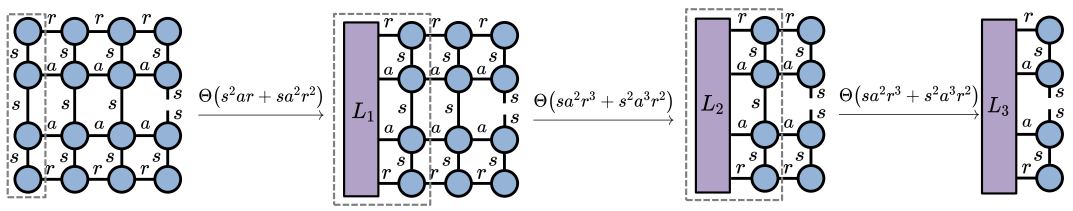
Figure 8.9: Illustration of the application and truncation algorithm.

The canonicalization-based algorithm that uses the full environment is illustrated in Fig. 8.9. 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 8.4.2. 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 [12].

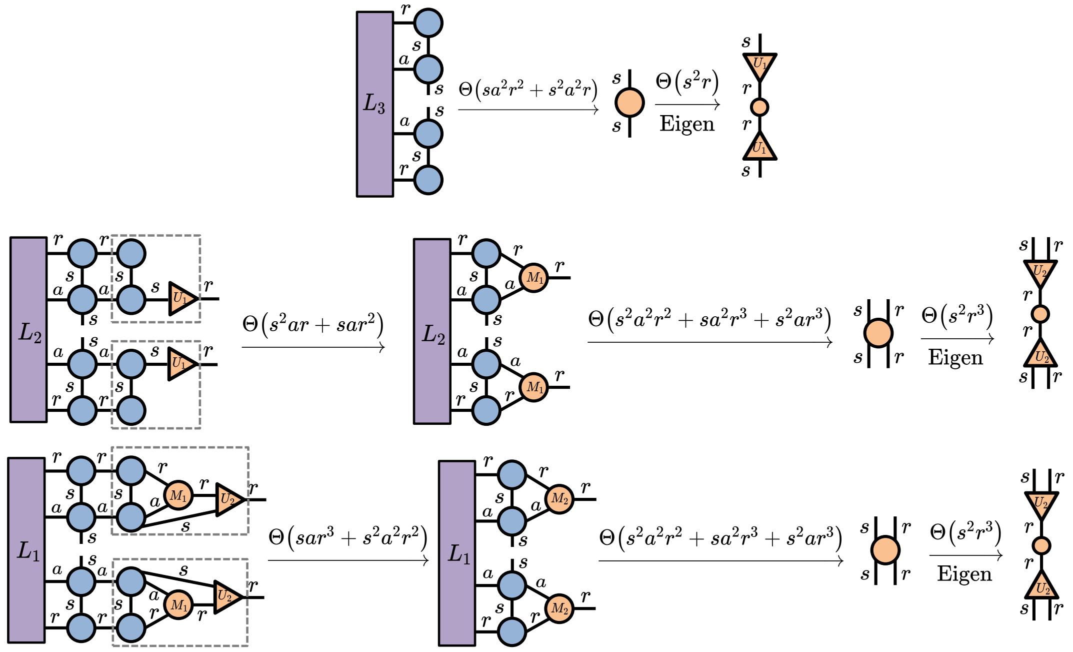
The density matrix algorithm The density matrix algorithm produces an equivalent truncated MPS as the application and truncation algorithm, and we illustrate the algorithm in Fig. 8.10. The algorithm contains three steps,

1. Computing matrices $\mathbf{L}_i$, as is shown in Fig. 8.10a. These matrices are computed by sequentially contracting the network from left to right, and intermediates $\mathbf{L}_i$ are saved during the contractions.

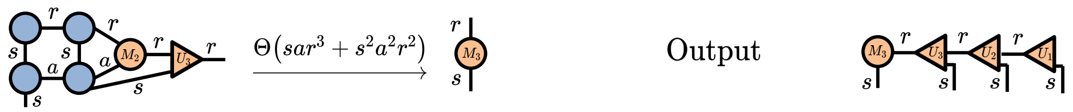
2. Performing a sweep of contractions from right to left and use $\mathbf{L}_i$ to compute all the leading singular vectors $\mathbf{U}_i$ for $i \in \{1, \ldots, N-1\}$. Specifically, $\mathbf{L}_N$ is firstly used to compute the density matrix with the last pair of uncontracted dimensions left open, 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 dimensions left open. Additionally, the


$\Theta(s^2ar + sa^2r^2)$
L_1
$\Theta(sa^2r^3 + s^2a^3r^2)$
L_2
$\Theta(sa^2r^3 + s^2a^3r^2)$
L_3

(a) The first step


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

(b) The second step



(c) The third step

Figure 8.10: Illustration of the density matrix algorithm. Each triangular vertex represents a tensor with an orthogonal property.

basis of this density matrix is transformed by applying $\mathbf{U}_1$, as shown in Fig. 8.10b. This process is repeated until $N-1$ tensors $\mathbf{U}_i$ are obtained.

3. Getting the leftmost matrix $\mathbf{M}_N$ that encompasses all the non-orthogonal information through the contraction depicted in Fig. 8.10c, 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 8.1, 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 [86] 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 to reduce costs becomes less straightforward. In order to address this issue, we have introduced a strategy that utilizes graph partitioning in Section 8.7. 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 8.6, we propose algorithms to select efficient tree structures for general graphs.

8.4.3 The Swap-based Algorithm to Reorder MPS Dimensions

In the MPS-based automated tensor network contraction algorithms including CATN and SweepContractor, an important step is to reorder the sites in an MPS. The reordering changes the 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 $d_{\mathrm{KT}}(\sigma, \tau)$ number of swaps, where d_{KT} denotes the Kendall-Tau distance defined in Definition 8.2.

Definition 8.2. *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),*

$$d_{KT}(\sigma, \tau) = \sum_{(c,c')\in S} \Big|\sigma(c, c') - \tau(c, c')\Big|, \tag{8.4}$$

where $\sigma(c, c') := \mathbb{1}\Big(\sigma(c) < \sigma(c')\Big)$ 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. 8.11. 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 dimensions 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 8.3.2. To reduce the truncation error, canonicalization is commonly performed on the MPS to orthogonalize all other sites.

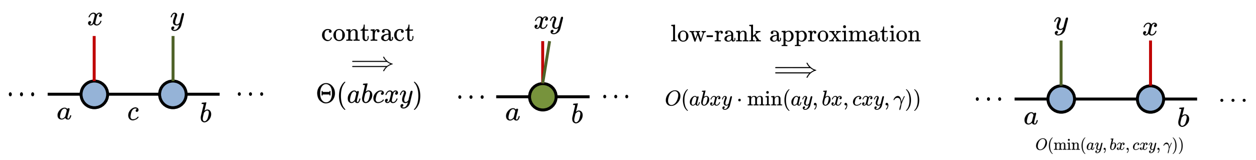


Figure 8.11: Illustration of the swap operation and the asymptotic computational cost.

8.4.4 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 `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 [241]–[245]. 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 [248]. When the objective is to minimize $\sum_{e\in E_t} \texttt{congestion}(e)$, the problem has been called the minimum linear arrangement problem [249], [250], and multiple approximation algorithms with bounded complexity have been proposed [251]–[255].

Recursive bisection is a simple yet effective divide-and-conquer heuristic widely adopted in both linear ordering problems [227], [228] and balanced graph partitioning [239], [256]. 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 $O(\gamma \log n)$ [228], [257]. The approximation factor for the 1/3-balanced cut is improved from $\gamma = O(\log n)$ [258] to $\gamma = O(\sqrt{\log n})$ [259], making the approximation factor of the recursive bisection $O(\log^{1.5} n)$.

In Sections 8.6 and 8.7, 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.

8.5 THE PROPOSED TENSOR NETWORK CONTRACTION ALGORITHM

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

8.5.1 An Overview of the Algorithm

We provide an overview of the algorithm for approximating tensor network contractions, and the corresponding pseudocode is presented in Algorithm 8.2. 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.

For each contraction involving a pair of partitions, an embedding tree is chosen to approximate the partitions. We define the embedding tree for the network G' as a full binary tree. This tree, denoted as $T^{(\text{openedges}(G'))}$, has each leaf vertex representing an edge in

Algorithm 8.2: `partitioned_contract`: approximate tensor network contraction based on its given partition

```
1: Input: The tensor network $\mathfrak{T}$ with graph $G = (V, E, w)$, its partition $\mathcal{V} = \{V_1, \dots, V_N\}$
   and its contraction path $T^{(\mathcal{V})}$, ansatz $A$, maximum bond dimension $\chi$, and swap batch
   size $r$                                   ▷ The ansatz $A$ can be either "MPS" or "Comb"
2: tn ← a function used to map each vertex set to its approximated tensor network
3: for $U \in \mathcal{V}$ do
4:     tn($U$) ← part of the tensor network $\mathfrak{T}$ represented by $G[U]$
5: end for
6: $\mathcal{E} \leftarrow \{E(V_i, V_j) : i, j \in \{1, \dots, N\}\}$
7: $\left\{\sigma^{(E')} : E' \in \mathcal{E}\right\} \leftarrow$ assigning each edgeset in $\mathcal{E}$ an ordering
8: for each contraction $(U_s, W_s) \in T^{(\mathcal{V})}$ do
9:     $\mathcal{E}_s \leftarrow \{E' \cap E(U_s \cup W_s) : E' \in \mathcal{E}\}$
10:    $\sigma^{(\mathcal{E}_s)} \leftarrow$ embedded_tree_ordering$\left(G[U_s \cup W_s], \mathrm{path}\left(T^{\mathcal{V}}, U_s \cup W_s\right), \mathcal{E}_s\right)$
11: end for
12: for each contraction $(U_s, W_s) \in T^{(\mathcal{V})}$ do
13:    $\mathfrak{X}_s \leftarrow$ approx_tensor_network$\left(\mathrm{tn}(U_s) \cup \mathrm{tn}(V_s), \sigma^{(\mathcal{E}_s)}, \{\sigma^{(E')} : E' \in \mathcal{E}_s\}, \chi, r, A\right)$   ▷
   Approximate the input tensor network tn($U_s$) ∪ tn($V_s$) as a binary tree tensor network $\mathfrak{X}_s$
14:    tn($U_s \cup W_s$) ← $\mathfrak{X}_s$
15: end for
16: return the final approximated tensor network $\mathfrak{X}_{|\mathcal{V}|-1}$
```

openedges(G'). 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. 8.4.

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 7-11. 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 8.5.2.

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 dimension remains below χ. This process is performed in lines 12-15. 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 8.7 and Section 8.8. 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.

8.5.2 Determination of the Embedding Tree

We explain the embedding tree structure used in Algorithm 8.2. As is defined in Section 8.5.1, an embedding tree is a rooted full binary tree, with each leaf vertex representing an uncontracted edge in the tensor network.

Let $\mathcal{E} = \{E(V_i, V_j) : i, j \in \{1, \ldots, N\}\}$, so that each element in $\mathcal{E}$ is an edge subset connecting two different partitions. For a specific contraction (U_s, W_s), 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 dimensions.

Two structures we use for the embedding tree are the MPS (maximally-unbalanced full binary tree) and the comb [260], [261]. 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 7-11 of Algorithm 8.2. Below we formally define the embedding tree with an MPS and a comb structure in Definition 8.4 and Definition 8.5. Both definitions are based on the MPS tree, which is defined in Definition 8.3. We visualize both the embedding tree with an MPS structure and with a comb structure in Fig. 8.12.

Definition 8.3 (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 $i \in \{2, \ldots, |S| - 1\}$ is connected to the $i - 1$th non-leaf node and x_{i+1}. An example is shown in Fig. 8.12a.*

Definition 8.4 (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 8.5 (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 $\hat{\sigma} : \{r_1, \ldots, r_{n_s}\} \rightarrow \{1, \ldots, n_s\}$, where $\hat{\sigma}(r_i) = i$.*

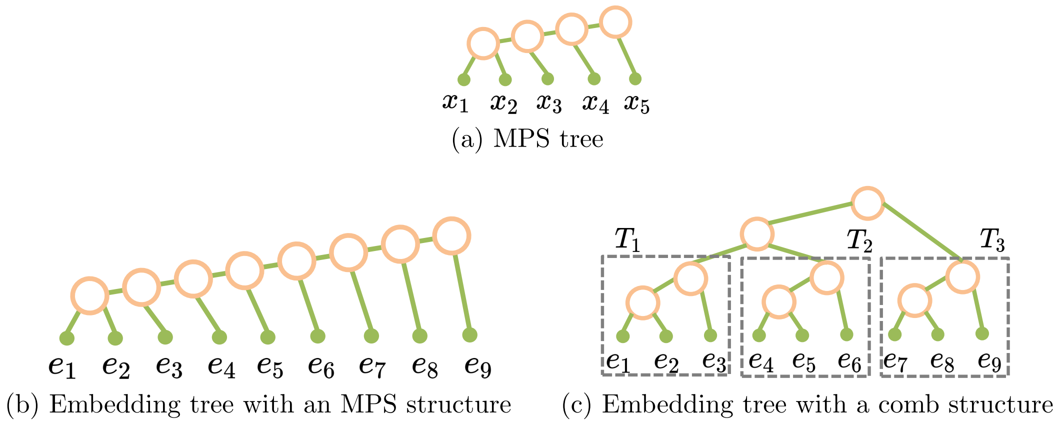

x_1 x_2 x_3 x_4 x_5
(a) MPS tree
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9
(b) Embedding tree with an MPS structure
T_1 T_2 T_3
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9
(c) Embedding tree with a comb structure

Figure 8.12: (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 dimension. This is larger than the maximum tensor size of the MPS structure, which is $\Theta(s\chi^2)$, where s represents the uncontracted dimension size and is typically much smaller than χ. In Section 8.9, 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 Section 8.4.4, 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 8.6.

8.6 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 $T = \mathsf{path}\left(T^{(\mathcal{V})}, V_s\right)$, 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 in $E(U_k, W_k)$ 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. 8.13. In Section 8.6.1, 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 8.6.2, we detail the algorithm to find the ordering not only consistent with the constraint tree, but also to minimize the Kendall-Tau distance between the chosen ordering and another reference ordering whose corresponding line structure is similar to G_s.

8.6.1 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 6 of Algorithm 8.2, 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

$$\hat{\mathcal{E}}_i = (\mathcal{E}_s \cap \mathcal{E}(U_i)) \bigcup_{j \in S_i} \hat{\mathcal{E}}_j, \tag{8.5}$$

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. 8.13, we use an example to illustrate the constraint tree construction algorithm, and each $\hat{\mathcal{E}}_i$ is also shown in the figure.

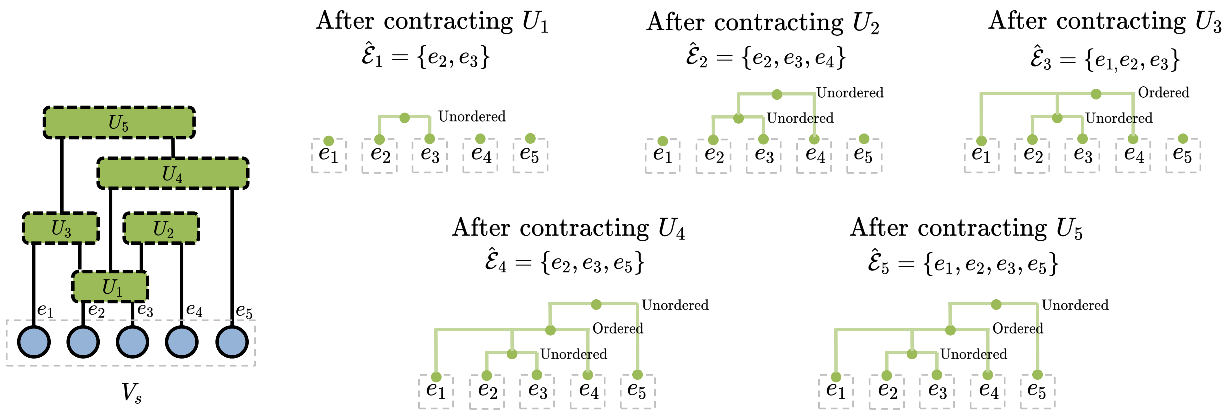

After contracting U_1
$\hat{\mathcal{E}}_1 = \{e_2, e_3\}$
After contracting U_2
$\hat{\mathcal{E}}_2 = \{e_2, e_3, e_4\}$
After contracting U_3
$\hat{\mathcal{E}}_3 = \{e_1, e_2, e_3\}$
After contracting U_4
$\hat{\mathcal{E}}_4 = \{e_2, e_3, e_5\}$
After contracting U_5
$\hat{\mathcal{E}}_5 = \{e_1,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

Figure 8.13: 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. 8.13 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. 8.13, 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. 8.13, 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. 8.13, the constraint trees after the third contraction belongs to this case.

8.6.2 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 8.3. 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.

Algorithm 8.3: `linear_ordering_under_constraint_tree`: Algorithm to get the edge set ordering that minimizes the Kendall-Tau distance with the reference orderings under the adjacency constraint

```
 1: Input: the edge set $\mathcal{E}_s$, the constraint tree $T^{(\mathcal{E}_s)}$, the tensor network graph $G_s = (V_s, E_s, w)$
 2: $\tau \leftarrow$ linear_ordering$(\mathcal{E}_s, G_s)$          ▷ Ordering generated via recursive bisection
 3: $f \leftarrow$ a function that maps each vertex in $T^{(\mathcal{E}_s)}$ to its edge set ordering
 4: for each leaf vertex $v$ that represents $E_i$ in $T^{(\mathcal{E}_s)}$ do
 5:     $f(v) \leftarrow$ the ordering that contains the single edge set $E_i$
 6: end for
 7: 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
 8:     $u_1, \ldots, u_{n_v} \leftarrow$ children of $v$
 9:     if $v$ is labeled as ordered then
10:         $\sigma_1 \leftarrow f(u_1) \oplus f(u_2) \oplus \cdots \oplus f(u_{n_v})$          ▷ Concatenate all $f(u_i)$ in order
11:         $\mathcal{S} \leftarrow \{\sigma_1, \texttt{reverse}(\sigma_1)\}$
12:     else
13:         $\mathcal{S} \leftarrow$ a set of all permutations of $\{f(u_1), f(u_2), \ldots, f(u_{n_v})\}$
14:     end if
15:     $\tau_v \leftarrow$ a partial ordering of $\tau$ over the subset $\hat{\mathcal{E}}_i$
16:     $f(v) \leftarrow \arg\min_{\sigma \in \mathcal{S}} d_{\mathrm{KT}}(\sigma, \tau_v)$
17: end for
18: return $f\left(\texttt{root}\left(T^{(\mathcal{E}_s)}\right)\right)$
```

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 8.1, we establish that the output ordering produced by Algorithm 8.3 aims to minimize the Kendall-Tau distance, as defined in Definition 8.2, between itself and the reference ordering τ,

$$\sigma^{(\mathcal{E}_s)} = \arg \min_{\sigma \in \mathcal{P}\left(T^{(\mathcal{E}_s)}\right)} d_{\mathrm{KT}}\left(\sigma, \tau\right). \tag{8.6}$$

Before the presentation of Theorem 8.1, we first present Lemma 8.1 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 8.2.

Lemma 8.1. *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*

$$d_{KT}\left(\tau^{(C)}, \sigma^{(C_1)} \oplus \sigma^{(C_2)}\right) = d_{KT}\left(\tau^{(C)}, \tau^{(C_1)} \oplus \tau^{(C_2)}\right) + d_{KT}\left(\tau^{(C_1)}, \sigma^{(C_1)}\right) + d_{KT}\left(\tau^{(C_2)}, \sigma^{(C_2)}\right), \tag{8.7}$$

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

Theorem 8.1. *Given a reference ordering τ and a guide tree $T^{(\mathcal{E}_s)}$, the output ordering of Algorithm 8.3 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 8.3, 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 $v \in T^{(\mathcal{E}_s)}$,

$$f(v) = \arg \min_{\sigma \in \mathcal{P}\left(\texttt{subtree}\left(v, T^{(\mathcal{E}_s)}\right)\right)} d_{\mathrm{KT}}\left(\sigma, \tau_v\right), \tag{8.8}$$

where $f(v)$ is defined in Line 16 of Algorithm 8.3. 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, (8.8) 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 (8.8) holds for its children, u_1 and u_2. Consider the case where

$$d_{\mathrm{KT}}\left(f(u_1) \oplus f(u_2), \tau_v\right) < d_{\mathrm{KT}}\left(f(u_2) \oplus f(u_1), \tau_v\right), \tag{8.9}$$

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

$$\begin{aligned} d_{\mathrm{KT}}\left(f(v), \tau_v\right) &= d_{\mathrm{KT}}\left(f(u_1) \oplus f(u_2), \tau_v\right) \\ &\overset{\textit{Lemma } 8.1}{=} d_{\mathrm{KT}}\left(\tau_v, \tau_{u_1} \oplus \tau_{u_2}\right) + d_{\mathrm{KT}}\left(f(u_1), \tau_{u_1}\right) + d_{\mathrm{KT}}\left(f(u_2), \tau_{u_2}\right). \end{aligned} \tag{8.10}$$

The first term in (8.10) reaches the minimum since (8.9) holds. Moreover, the last two terms also reach the minimum since (8.8) holds for u_1 and u_2. These conditions imply $f(v)$ satisfies (8.8). 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.

Q.E.D.

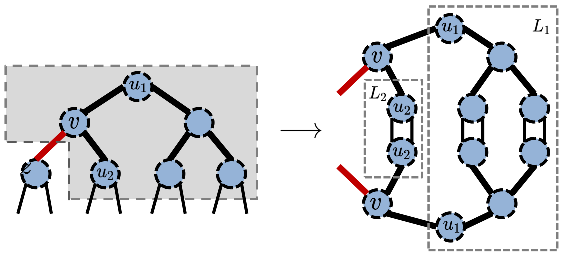
8.7 THE DENSITY MATRIX ALGORITHM FOR TREE APPROXIMATIONS

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

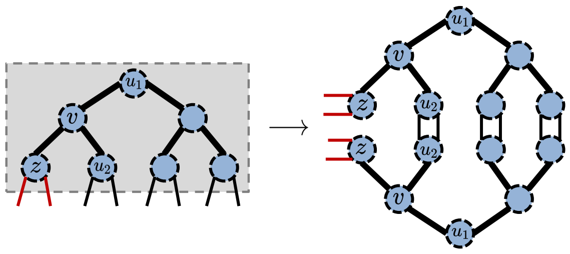
Our proposed density matrix algorithm builds upon the density matrix algorithm originally designed for MPO-MPS multiplication, which is discussed in Section 8.4.2. Given a tree embedding of the input tensor network, our algorithm eliminates the need to explicitly construct the untruncated tree tensor 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 8.7.2 that the asymptotic computational cost of the algorithm is upper-bounded by the cost of the canonicalization-based algorithm, and we show in Section 8.9 that for many input tensor networks, the proposed algorithm substantially reduces the overall execution time.

Within the algorithm, we use $\texttt{density\_matrix}_T(v)$ and $\texttt{density\_matrix}_T(v, z)$ introduced in Definition 8.6. 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. 8.14.



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



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

Figure 8.14: 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 8.6). 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.

Definition 8.6 (Density matrix). *Consider a given embedding tree $T = (V_T, E_T, w)$ 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 dimensions 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 $\tilde{E}_v = E_T(u, v)$.*

The density matrix algorithm is summarized in Algorithm 8.4. 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. 8.15.

In Algorithm 8.4, we initially construct an embedding ϕ utilizing the recursive bisection technique outlined in Algorithm 8.5. 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 Section 8.4.4, recursive bisection is a standard heuristic to find embeddings with low congestion. It is worth noting that Algorithm 8.5 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 situation 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 8.4 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 dimensions 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 [40] 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 represenentatin 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 8.7.2, we demonstrate that Algorithm 8.4 provides a guarantee that its asymptotic computational cost remains upper-bounded by that of the canonicalization-based algorithm.

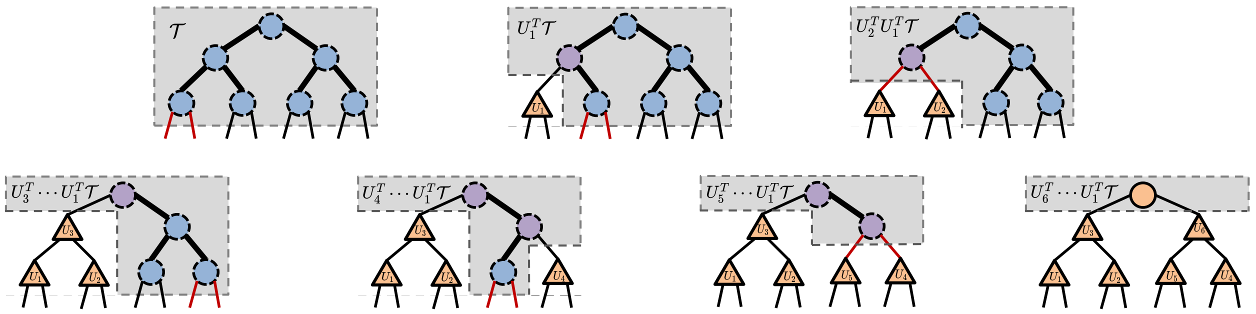

$\mathcal{T}$
$U_1^T\mathcal{T}$
$U_2^TU_1^T\mathcal{T}$
$U_3^T \cdots U_1^T\mathcal{T}$
$U_4^T \cdots U_1^T\mathcal{T}$
$U_5^T \cdots U_1^T\mathcal{T}$
$U_6^T \cdots U_1^T\mathcal{T}$
U_1
U_2
U_3
U_4
U_5
U_6

Figure 8.15: 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 8.4 and is used to compute the density matrix, and red edges denote the open edges of the density matrix.

Algorithm 8.4: `density_matrix_alg`: The density matrix algorithm for tree approximation

```
1: Input: The tensor network $G = (V, E, w)$, its embedding tree $T^{(\mathsf{openedges}(G))} = (V_T, E_T)$, and maximum bond dimension $\chi$
2: $\phi \leftarrow$ tree_embedding$(G, T^{(\mathsf{openedges}(G))})$        ▷ Constructed based on Algorithm 8.5
3: $r \leftarrow$ root vertex in $T^{(\mathsf{openedges}(G))}$
4: $T' \leftarrow$ a tree with the same structure as $T^{(\mathsf{openedges}(G))}$ and $T'(v)$ for $v \in V_T$ denotes all tensors embedded to $v$ in $\phi$
5: for each $v \in V_T \setminus \{r\}$ based on a post-order DFS traversal of $T^{(\mathsf{openedges}(G))}$ do
6:     $A_v \leftarrow$ uncontracted_edges$(T', v)$
7:     $B_v \leftarrow$ contracted_edges$(T', v)$
8:     $u \leftarrow$ parent$(T', v)$
9:     if $w(A_v) = O(w(B_v))$ then
10:        $\mathbf{L}_v \leftarrow$ density_matrix$_{T'}(v)$        ▷ Defined in Definition 8.6
11:        $\mathbf{U}_v \leftarrow$ leading_singular_vectors$(\mathbf{L}_v, \chi)$
12:    else
13:                ▷ Perform QR-SVD [40] to reduce the asymptotic cost
14:        $\mathbf{L}_u \leftarrow$ density_matrix$_{T'}(u, v)$
15:        $\mathbf{M}_v \leftarrow$ the matricized contraction output of $T(v)$ with $A_v$ combined into row
16:        $\mathbf{Q}_v, \mathbf{R}_v \leftarrow$ QR$(\mathbf{M}_v)$
17:        $\hat{\mathbf{U}}_v \leftarrow$ leading_singular_vectors$(\mathbf{R}_v \mathbf{L}_u \mathbf{R}_v^T, \chi)$
18:        $\mathbf{U}_v \leftarrow \mathbf{Q}_v \hat{\mathbf{U}}_v$
19:    end if
20:    Add both $T'(v)$ and a vertex that represents $\mathbf{U}_v^T$ to $T'(u)$, and remove $v$ from $T'$
21: end for
22: $\mathbf{M}_r \leftarrow$ contraction output of $T(r)$
23: return the tree tensor network that contains all $\mathbf{U}_v$ and the root tensor $\mathbf{M}_r$
```

8.7.1 The Density Matrix Algorithm with Memoization

As can be seen from Fig. 8.15, 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 8.4.

The memoization strategy uses the following recursive relation for `density_matrix`$_T(v)$

Algorithm 8.5: `tree_embedding`: embedding a graph into the embedding tree via recursive bisection

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

and `density_matrix`$_T\,(v, z)$,

$$\begin{aligned}
\texttt{density\_matrix}_T\,(v) &= \mathbf{M}^{(v)}_{E_T(v,*)}\left(\bigotimes_{u\in N(v)} \texttt{density\_matrix}_T(u, v)\right)\mathbf{M}^{(v)T}_{E_T(v,*)},\\
\texttt{density\_matrix}_T\,(v, z) &= \mathbf{M}^{(v)}_{E_T(v,z)}\left(\bigotimes_{u\in N(v)\setminus\{z\}} \texttt{density\_matrix}_T(u, v)\right)\mathbf{M}^{(v)T}_{E_T(v,z)},
\end{aligned} \tag{8.11}$$

where $\otimes$ 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 dimensions incident on v are combined into the row. $\mathbf{M}^{(v)}_{E_T(v,z)}$ denote a matricization of $T(v)$ where the dimension represented by the edge $E_T(v, z)$ is the matrix row.

To compute the density matrix `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 (8.11). 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 [262]. 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 `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. 8.14a. The same strategy is used to compute $\texttt{density\_matrix}_T(v)$.

In Algorithm 8.4, 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.

8.7.2 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 8.4.2. Firstly, we demonstrate in Lemma 8.4 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 dimension sizes in the tree tensor network.

Subsequently, in Theorem 8.2, 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.

The Lemma 8.2 and Lemma 8.3 below are used to prove Lemma 8.4.

Lemma 8.2. *Consider a tensor network with a tree structure $T = (V_T, E_T, w)$. Assuming that changing a tree tensor network into the canonical form will not change any dimension size of the network. For two adjacent vertices z, v, forming $\texttt{canonical\_form}_T(v, z)$ has the same asymptotic cost as forming $\texttt{density\_matrix}_T(v, z)$.*

Proof. For each edge set $E' \subseteq E_T$, we let $s(E') = \exp(w(E'))$ denote the dimension 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 $\texttt{canonical\_form}_T(u, v)$ already exist for all $u \in N(v) \setminus \{z\}$. Let $\mathbf{R}_u$ denote the non-orthogonal core of $\texttt{canonical\_form}_T(u, v)$. To construct the form $\texttt{canonical\_form}_T(v, z)$, we first contract $\mathbf{M}_v$ with $\mathbf{R}_u$ for each $u \in N(v) \setminus \{z\}$, which yields a cost of $\Theta\left(\sum_{u\in N(v)\setminus\{z\}} s(E_T(v))s(E_T(u, v))\right)$, 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

$$\Theta\left(\sum_{u\in N(v)} s(E_T(v))s(E_T(u,v))\right). \tag{8.12}$$

We now consider the computation of $\texttt{density\_matrix}_T(v,z)$ under the assumption that for all $u \in N(v)\setminus\{z\}$, $\mathbf{L}_u = \texttt{density\_matrix}_T(u,v)$ 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 $\texttt{canonical\_form}_T(v,z)$ and $\texttt{density\_matrix}_T(v,z)$ have the same recursive relation, computing $\texttt{canonical\_form}_T(u,v)$ has the same cost as that of the $\texttt{density\_matrix}_T(u,v)$ for $u \in N(v)\setminus\{z\}$. This finishes the proof.

Q.E.D.

Lemma 8.3. *Consider a tensor network with a tree structure $T = (V_T, E_T, w)$, 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 dimensions and b_v denotes the size of the contracted dimensions 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 8.4) has a cost of $\Theta\left(a_vb_v^2\right)$.*

Proof. For the case where $a_v = O(b_v)$, the algorithm first computes $\mathbf{L}_v = \texttt{density\_matrix}_T(v)$ with a cost of $\Theta\left(a_vb_v^2 + a_v^2b_v\right)$, 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 $r = O(a_v)$, which costs $\Theta\left(a_v^2r\right)$. The overall cost is $\Theta\left(a_vb_v^2 + a_v^2b_v + a_v^2r\right) = \Theta\left(a_vb_v^2\right)$.

For the case where $a_v = \Omega(b_v)$, 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 $\Theta\left(a_vb_v^2\right)$, 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 $\Theta(a_vb_vr)$. Overall the cost is $\Theta\left(a_vb_v^2 + b_v^3 + a_vb_vr\right) = \Theta\left(a_vb_v^2\right)$. This finishes the proof.

Q.E.D.

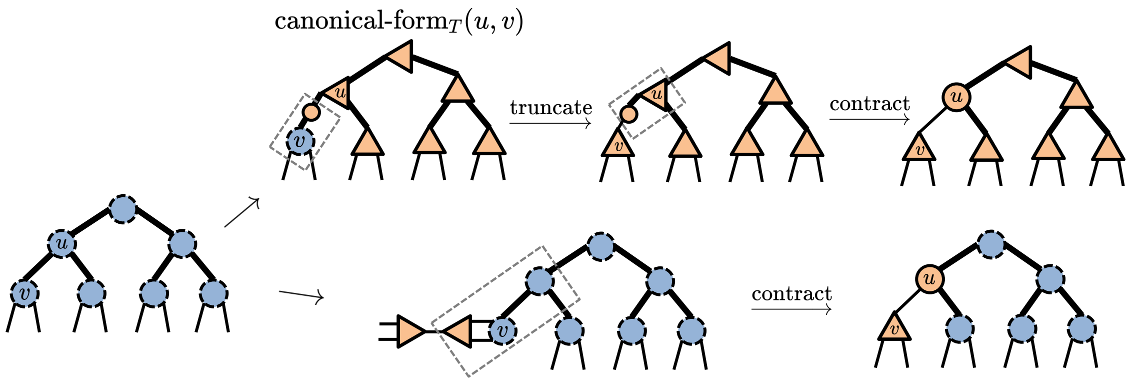


Figure 8.16: Illustration of the difference between the canonicalization-based algorithm and the density matrix algorithm. The upper path denotes truncating the edge (u, v) using canonicalization, and the lower path uses the density matrix algorithm. In the lower path, the orthogonal matrix is calculated as the leading singular vectors/eigenvectors of the density matrix $\texttt{density\_matrix}_T(z)$.

Lemma 8.4. *Consider a given tree tensor network $T = (V_T, E_T, w)$. 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 dimension size of the network, the asymptotic cost of the density matrix algorithm (Algorithm 8.4) for truncating the dimensions in T is the same as that of the canonicalization-based algorithm (Algorithm 8.1) if both algorithms use the same update ordering σ, and the same maximum bond dimension 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 dimensions and b_v denote the size of the contracted dimensions of $\mathbf{M}_v$. Also let $r = \min(a_v, b_v, \chi)$ and $u = \texttt{parent}(T, v)$. We break down the cost of Algorithm 8.4 and Algorithm 8.1 into 3 parts, and show that for each of the three parts, the costs of the two algorithms are asymptotically equal.

In Algorithm 8.1, 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 8.4 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. 8.16. It can be seen that the third step of both algorithms have the same asymptotic cost. For the first step, we show in Lemma 8.2 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 8.3.2. In addition, we show in Lemma 8.3 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. Q.E.D.

In Theorem 8.2 below we show that the asymptotic cost of the density matrix algorithm is upper-bounded by that of the canonicalization-based algorithm.

Theorem 8.2. *Consider a given tensor network $G = (V, E, w)$, an embedding tree $T = (V_T, E_T)$, and an embedding ϕ that embeds G into 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 dimension size of the network, the asymptotic cost of the density matrix algorithm (Algorithm 8.4) is upper-bounded by that of the canonicalization-based algorithm (Algorithm 8.1) if both algorithms use the same embedding ϕ, the same update ordering σ, and the same maximum bond dimension 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 (8.11). 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 8.4, 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 8.1. This finishes the proof.

Q.E.D.

8.8 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 $G_s = (V_s, E_s, w)$ into an embedding

Algorithm 8.6: `approx_tensor_network`: approximate a tensor network into an embedding tree

```
1: Input: The tensor network $\mathfrak{T}$ with graph $G_s = (V_s, E_s, w)$, the edge set ordering $\sigma^{(\mathcal{E}_s)}$, the edge orderings $\{\sigma^{(E')} : E' \in \mathcal{E}_s\}$, the maximum bond dimension $\chi$, the swap batch size $r$, and ansatz $A$    ▷ The ansatz $A$ can be either "MPS" or "Comb"
2: $\tau^{(\mathcal{E}_s)} \leftarrow$ linear_ordering $(\mathcal{E}_s, G_s)$    ▷ Ordering generated via recursive bisection
3: $d \leftarrow d_{\mathrm{KT}}\left(\tau^{(\mathcal{E}_s)}, \sigma^{(\mathcal{E}_s)}\right)$    ▷ Number of adjacent edge set swaps needed to change $\tau^{(\mathcal{E}_s)}$ to $\sigma^{(\mathcal{E}_s)}$
4: $n \leftarrow \lceil d/r \rceil$    ▷ The number of density matrix algorithms to be performed
5: $\hat{\sigma}_1 \ldots, \hat{\sigma}_n \leftarrow n$ equally-spaced inverval orderings that separate $\tau^{(\mathcal{E}_s)}$ and $\sigma^{(\mathcal{E}_s)}$
6: $\mathfrak{X}_0 \leftarrow \mathfrak{T}$
7: for $i \in \{1, \ldots, n\}$ do
8:     $T \leftarrow$ embedded_tree $\left(\hat{\sigma}_i, \{\sigma^{(E')} : E' \in \mathcal{E}\}, A\right)$    ▷ construct the embedding tree based on Definition 8.4 and Definition 8.5
9:     $\mathfrak{X}_i \leftarrow$ density_matrix_alg($\mathfrak{X}_{i-1}, T, \chi$)
10: end for
11: return the output tensor network $\mathfrak{X}_n$
```

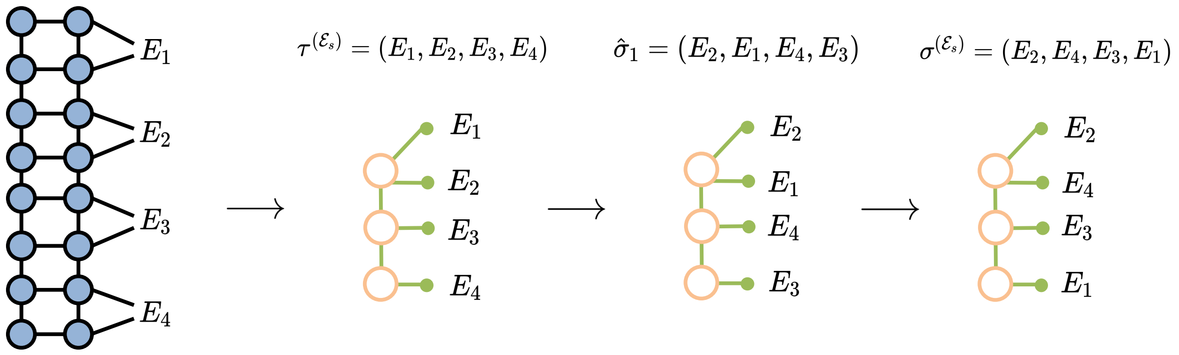

E_1
E_2
E_3
E_4
$\tau^{(\mathcal{E}_s)} = (E_1, E_2, E_3, E_4)$
$\hat{\sigma}_1 = (E_2, E_1, E_4, E_3)$
$\sigma^{(\mathcal{E}_s)} = (E_2, E_4, E_3, E_1)$

Figure 8.17: Illustration of Algorithm 8.6 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 $\sigma^{(\mathcal{E}_s)}$.

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 8.6, and an illustration is shown in Fig. 8.17. 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, 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 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.

8.9 EXPERIMENTAL RESULTS

In this section, we conducted 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 8.9.1, we introduce our implementations, the tensor networks and models tested in our experiments. In Section 8.9.2, 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 8.9.3, we justify the `partitioned_contract` algorithm presented in Algorithm 8.2. 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 [41][12], and SweepContractor [86][13], in contracting tensor networks defined on lattices and random regular graphs. 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.

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

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

8.9.1 Implementations, Tested Tensor Networks, and the Evaluation

The proposed algorithms in the paper have been implemented at ITensorNetworks.jl[14], which is a publicly available Julia [230] package built for manipulating tensor networks of arbitrary geometry, and is built on top of ITensors.jl [91]. The library also provides an interface to OMEinsumContractionOrders.jl[15], which implements multiple heuristics introduced in [262], [263] 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) [263] to generate contraction paths for exact tensor network contractions.

Our experiments consider tensor networks generated based on two models, the random model and the Ising model. In the random model, 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 [87] 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.

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,

$$Z = \sum_{\sigma_i, \sigma_j \in \{-1,1\}} \prod_{(i,j)\in E} \exp(\beta \sigma_i \sigma_j). \tag{8.13}$$

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

$$t^{(v)}_{E(v)} = \sum_i \prod_{e \in E(v)} W_{i,e}, \tag{8.14}$$

where

$$W = \frac{1}{\sqrt{2}} \begin{bmatrix} \sqrt{\cosh(\beta)} + \sqrt{\sinh(\beta)} & \sqrt{\cosh(\beta)} - \sqrt{\sinh(\beta)} \\ \sqrt{\cosh(\beta)} - \sqrt{\sinh(\beta)} & \sqrt{\cosh(\beta)} + \sqrt{\sinh(\beta)} \end{bmatrix} \tag{8.15}$$

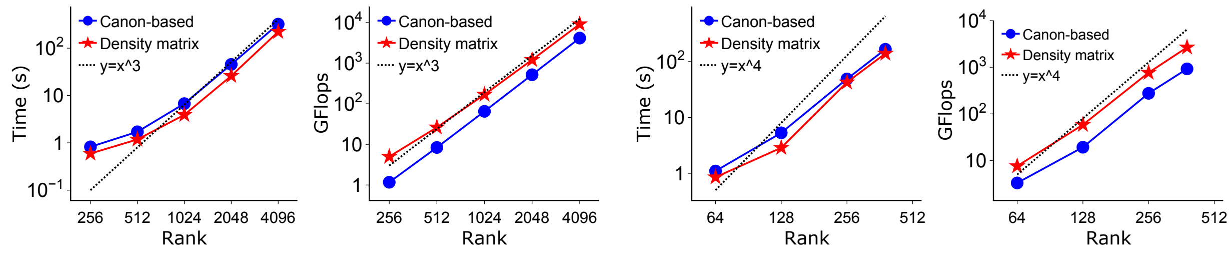
and β is an input parameter to the model. We show the relation between the relative error of $\ln Z$ and the running time of `partitioned_contract` and the baselines in Section 8.9.3. The quantity $\ln Z$ is an important measure that is proportional to the free energy of the system.

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

[14]The implementation is at `https://github.com/mtfishman/ITensorNetworks.jl`.
[15]The library is implemented at `https://github.com/TensorBFS/OMEinsumContractionOrders.jl`

GFlops calculations encompass tensor contractions, QR factorization, and low-rank approximations, as outlined in the model detailed in Section 8.3.2. It's 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 its contribution to the overall running time is negligible when the ranks are high.

8.9.2 Comparion Between the Density Matrix Algorithm and the Canonicalization-based Algorithm

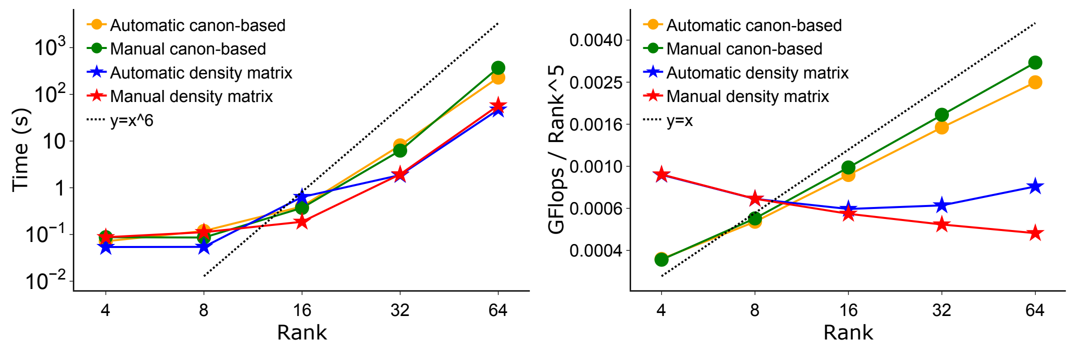


(a) MPS, $\chi = 100$ (b) MPS, $\chi = 100$ (c) BBT, $\chi = 50$ (d) BBT, $\chi = 50$

Figure 8.18: 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 dimensions 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. 8.18, 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 number of flops.

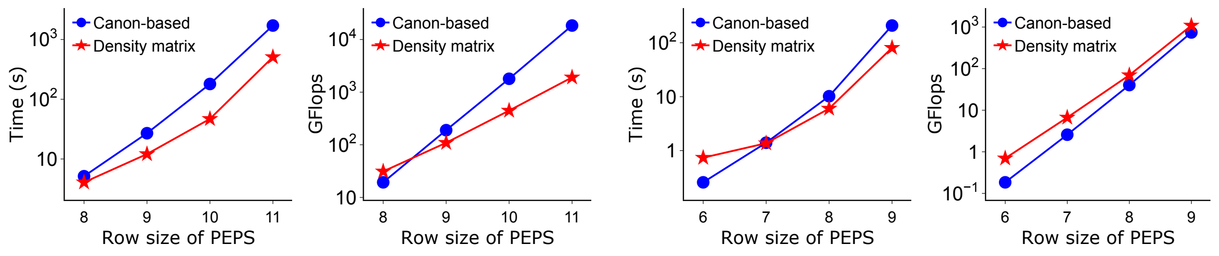


(a) (b)

Figure 8.19: 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 8.4.2 that use a manually-determined memoization strategy.

In Fig. 8.19, 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 8.4.2. In the reference algorithms, the memorization 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. 8.19b, the scaling behavior of both algorithms aligns with our analysis. The density matrix algorithm outperforms the canonicalization-based algorithm and has a 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.

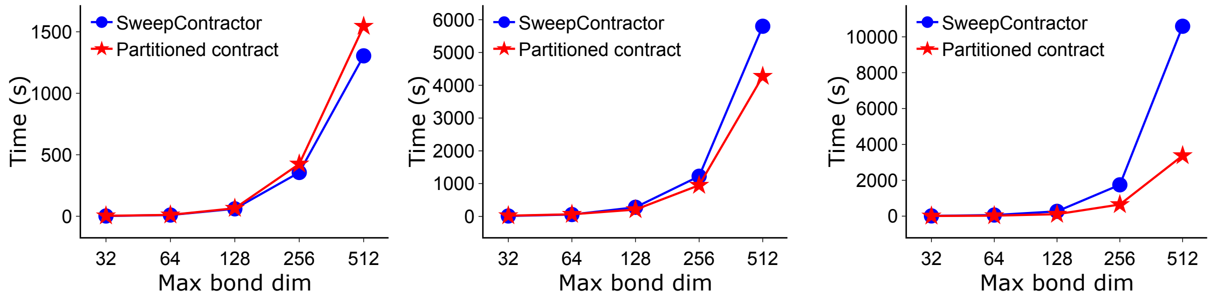
In Fig. 8.20, we compare the performance of approximating a PEPS into the MPS and the comb binary tree structure. Both structures are defined in Section 8.5.2. As can be seen, the density matrix algorithm is more efficient when the row and column size of PEPS is large. The inefficiency of the canonicalization-based 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.



(a) MPS, $\chi = 250$ (b) MPS, $\chi = 250$ (c) Comb, $\chi = 50$ (d) Comb, $\chi = 50$

Figure 8.20: 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 number of columns 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 sequential 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.

8.9.3 Benchmark of the `partitioned_contract` Algorithm



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

Figure 8.21: 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 dimension 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 We justify our embedding tree selection algorithm in `partitioned_contract`. In Fig. 8.21, we compare our `partitioned_contract` with SweepContractor, for tensor networks defined on three different structures using the random model in Section 8.9.1. For all the experiments, our `partitioned_contract` uses the MPS ansatz, and both algorithms use the same maximally-unbalanced contraction tree. 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.

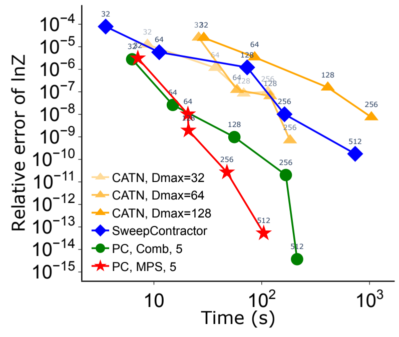
As can be seen, both algorithms have a similar performance when contracting a 2D grid, while `partitioned_contract` significantly outperforms SweepContract for the other two graph structures. This difference in performance arises from the fact that different embedding trees result in varying numbers of adjacent swaps of MPS dimensions. 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 3D lattices and random regular graphs, and the results are shown in Fig. 8.22 and Fig. 8.23.

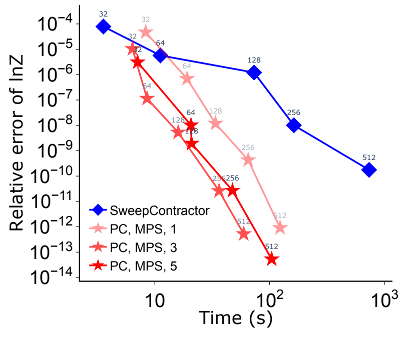
In both 3D lattices and random regular graphs, we employ the maximally-unbalanced partial contraction path for the contraction process. This path initiates from one partition and progressively combines the previously-contracted section with a new partition. For 3D lattices, each partition represents a portion or the whole fiber of the lattice. The contraction path is determined through a sequential traversal of the 2D array mapping of all the fibers. Regarding random regular graphs, we draw inspiration from [226] 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 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 considering 3D lattices with the MPS ansatz, the results presented in Fig. 8.22b and Fig. 8.22e reveal that employing a partition size of 3 or 5 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 density matrix algorithms required, offsetting any overhead from using larger environments. Regarding accuracy, we can see that under the same maximum bond dimension, utilizing a partition size of 3 or 5 yields lower relative errors compared to using a partition size of 1. This observation validates the efficacy of the environment in enhancing accuracy. For the comb ansatz, Fig. 8.22c and Fig. 8.22f show that employing a partition size of 3 results in the lowest running time. Similarly to the MPS ansatz, using a partition size of 3 or 5 exhibits better accuracy compared to a partition size of 1.

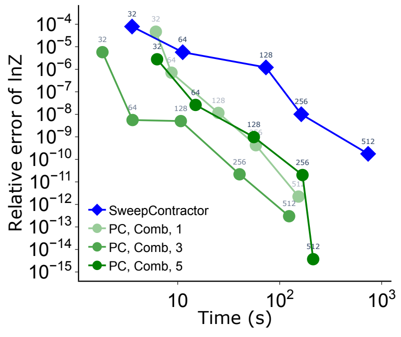
Regarding random regular graphs, the results displayed in Figs. 8.23b, 8.23c, 8.23e



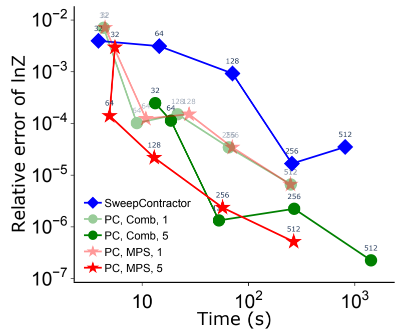
(a) Ising Model, $\beta = 0.3$



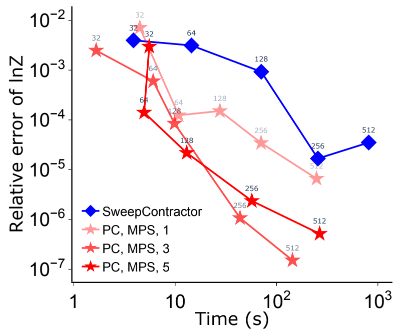
(b) Ising Model, $\beta = 0.3$



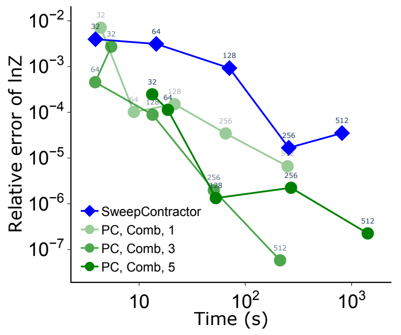
(c) Ising Model, $\beta = 0.3$



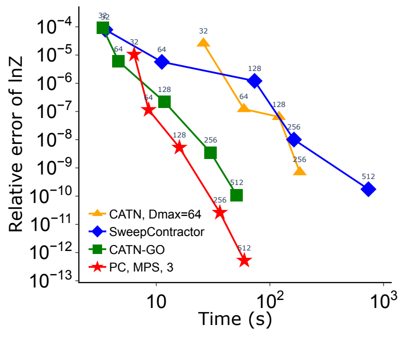
(d) Random Model, $\alpha = -0.4$



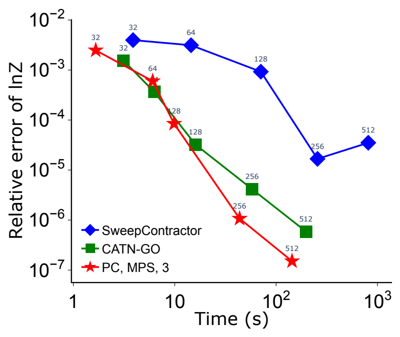
(e) Random Model, $\alpha = -0.4$



(f) Random Model, $\alpha = -0.4$



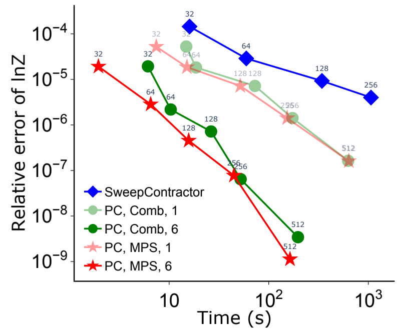
(g) Ising Model, $\beta = 0.3$



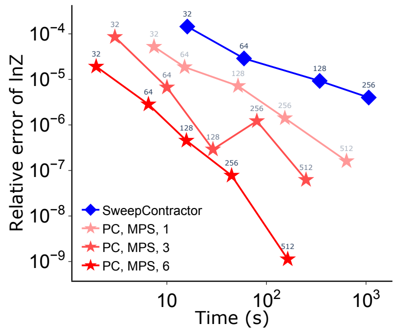
(h) Random Model, $\alpha = -0.4$

Figure 8.22: Performance comparison between `partitioned_contract`, SweepContractor [86], CATN [41], and CATN-GO in Chapter 7 in contracting $5 \times 5 \times 5$ 3D lattices. 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. The number shown on top of each point is the maximum bond dimension χ. In CATN, "Dmax" is an additional input parameter of the algorithm that controls the size of the MPS uncontracted dimensions.

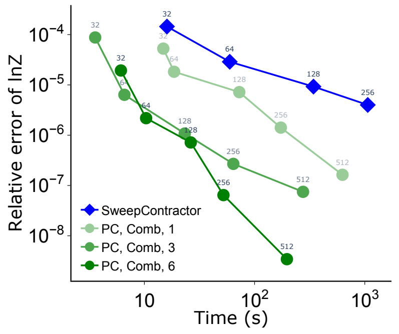
and 8.23f indicate that using a partition size of 6 results in the best combination of efficiency and accuracy. To summarize, employing a larger partition leads to a larger environment size, generally reducing the contraction error under the same rank. However, when it comes to efficiency, the optimal partition size depends on the specific problem. Factors such as the



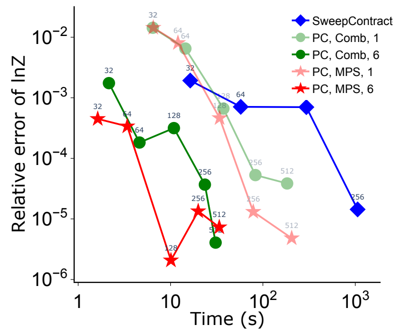
(a) Ising Model, $\beta = 0.65$



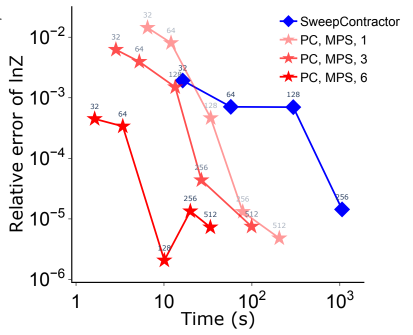
(b) Ising Model, $\beta = 0.65$



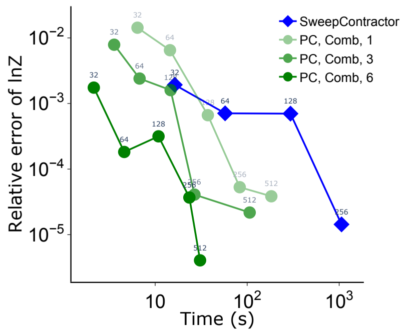
(c) Ising Model, $\beta = 0.65$



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



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



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

Figure 8.23: Performance comparison between `partitioned_contract` and SweepContractor [86] 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. The number shown on top of each point is the maximum bond dimension χ.

number of density matrix algorithms 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 We conduct a comparison between the MPS and the comb ansatz. When contracting 3D lattices, the results in Figs. 8.22a and 8.22d demonstrate that both MPS and comb structures exhibit similar performance when the maximum bond dimension is small. However, as the maximum bond dimension increases, using the comb ansatz becomes slower in comparison to MPS. On the other hand, when contracting random regular graphs, the results in Figs. 8.23a and 8.23d 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 We conducted evaluations of our proposed `partitioned_contract` algorithm, along with the CATN algorithm [41] and SweepContractor [86], on contracting tensor networks defined on lattices and random regular graphs. As shown from the results in Figs. 8.22a and 8.23a, our algorithm outperforms both CATN and SweepContractor in terms of efficiency across all relative errors. 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}. These significant speed improvements clearly demonstrate the efficiency of our approach over the compared algorithms.

We also compare `partitioned_contract` with CATN-GO in Chapter 7 on contracting tensor networks defined on lattices. The illustrations in Fig. 8.22g and Fig. 8.22h demonstrate that with an environment size of 3, `partitioned_contract` outperforms CATN-GO in execution time in achieving a relative error below 10^{-8} for the Ising model and 10^{-6} for the random model. This shows the advantage of employing a relatively larger environment size for enhancing the efficiency of approximate contraction.

8.10 CONCLUSION

We introduce 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.

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 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 [44], [172], [204], [205] to speed up randomized SVD [177], which may enhance the efficiency of the tree approximation process. Finally, integrating the proposed algorithm into automatic differentiation libraries [135], [211], [264] 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.

Part IV

APPLICATIONS OF TENSOR DECOMPOSITIONS IN QUANTUM COMPUTING

Chapter 9: LOW-RANK APPROXIMATION IN SIMULATIONS OF QUANTUM ALGORITHMS

In this Chapter, we examine the possibility of simulating a few quantum algorithms by using low-rank CP decomposition to represent the input and all intermediate states of these algorithms.

9.1 BACKGROUND

A quantum algorithm is often expressed by a unitary transformation U applied to a quantum state $|\psi\rangle$. On a quantum computer, $|\psi\rangle$ can be efficiently encoded by n qubits, effectively representing 2^n amplitudes simultaneously, and U is implemented as a sequence of one or two-qubit gates that are themselves 2×2 or 4×4 unitary transformations. When $|\psi\rangle$ is viewed as an order n tensor, there are several ways to represent it efficiently. One of them is known as a canonical polyadic (CP) decomposition [7], [13] written as

$$|\psi\rangle = \sum_{i_1,\ldots,i_n \in \{0,1\}} \sum_{k=1}^{R} A^{(1)}_{i_1 k} A^{(2)}_{i_2 k} \cdots A^{(n)}_{i_n k} |i_1 i_2 \ldots i_n\rangle, \tag{9.1}$$

where $A^{(i)} \in \mathbb{C}^{2\times R}$ and R is known as the rank of the CP decomposition. The second representation is known as matrix product state (MPS) [214] in the physics literature or tensor train (TT) [12] in the numerical linear algebra literature, which is a special tensor networks representation of a high dimensional tensor [265]. In this representation, the quantum state can be written as

$$|\psi\rangle = \sum_{i_1,\ldots,i_n \in \{0,1\}} \sum_{k_1,\ldots,k_{n-1}} \boldsymbol{\mathcal{A}}^{(1)}_{i_1 k_0 k_1} \boldsymbol{\mathcal{A}}^{(2)}_{i_2 k_1 k_2} \cdots \boldsymbol{\mathcal{A}}^{(n)}_{i_n k_{n-1} k_n} |i_1 i_2 \ldots i_n\rangle, \tag{9.2}$$

where $\boldsymbol{\mathcal{A}}^{(j)}$ is a tensor of dimension $2 \times R_{j-1} \times R_j$, with $R_0 = R_n = 1$. The rank of an MPS is often defined to be the maximum of R_j for $j \in \{1, 2, ..., n-1\}$. The memory requirements for CP and MPS representations of $|\psi\rangle$ are $\mathcal{O}(Rn)$ and $\mathcal{O}(R^2 n)$, respectively. When R is relatively small, such requirement is much less than the $\mathcal{O}(2^n)$ requirement for storing $|\psi\rangle$ as an vector, which allows us to simulate a quantum algorithm with a relatively large n on a classical computer that stores and manipulates $|\psi\rangle$ in these compact forms.

For several quantum algorithms, the rank of the CP or MPS representation of the input $|\psi\rangle$ is low. However, when $U^{(i)}$'s are successively applied to $|\psi\rangle$, the rank of the intermediate tensors (the tensor representation of the intermediate states) can start to increase. When the rank of an intermediate tensor becomes too high, we may not be able to continue the simulation for a large n. One way to overcome this difficulty is to perform rank reductions on intermediate tensors when their ranks exceed a threshold. When a CP decomposition is used to represent $|\psi\rangle$, we can take, for example, (9.1) as the input and use the alternating least squares (ALS) [6], [266] algorithm to obtain an alternative CP decomposition that has a smaller R. The rank reduction of an MPS can be achieved by performing a sequence of truncated singular value decomposition (SVD).

Performing rank reduction on intermediate tensors can introduce truncation error. For some quantum algorithms, this error is zero or small, thus not affecting the final outcome of the quantum algorithm. For other algorithms, the truncation error can be large, which results in significant deviation of the computed result from the exact solution. For a specific quantum algorithm, understanding whether the intermediate tensors can be accurately approximated through low rank truncation is valuable for assessing the difficulty of simulating the algorithm on a classical computer. We attempt to investigate such difficulty for a few well known quantum algorithms in this paper both analytically and numerically.

9.2 OUR CONTRIBUTIONS

We examine the use of low-rank approximation via CP decomposition to simulate several quantum algorithms. We choose to focus on using CP decomposition instead of MPS or general tensor networks to represent the input and intermediate tensors, because the (low) rank product structure of the input and intermediate tensors in the quantum algorithm are relatively easy to see and interpret in CP terms. Furthermore, some of the unitary operations such as swapping two qubits are relatively easy to implement for a CP decomposed tensor. The use of low rank MPS and more general tensor networks in quantum circuit simulation can be found in [39], [40], [262], [267], [268].

The algorithms we examine include the quantum Fourier transform (QFT) [269] and quantum phase estimation [270], which are the building blocks of other quantum algorithms, the Grover's search algorithm [271], [272], and quantum walk [273], [274] algorithms, which are quantum extensions of classical random walks on graphs.

For both QFT and phase estimation, we show that we can accurately approximate the intermediate states by low rank CP decomposition when the input states have special structures. For general input states, low-rank approximation can yield a large truncation error. For the Grover's search algorithm, we show analytically that CP ranks of all the intermediate states are bounded by $a+1$, where a is the size of the marked set to be searched. Therefore, Grover's algorithm can, in principle, always be simulated efficiently by using low-rank CP decomposition when the size of the marked set is small. For quantum walks, we show that accurate low-rank approximation is possible when the walk is performed on some graphs. However, rank reduction can be difficult when the walk is performed on a general graph.

We discuss two numerical algorithms for performing rank reduction for intermediate tensors produced in the simulation of the quantum circuit, CP-ALS and an alternative algorithm called direct elimination of scalar multiples (DESM). CP-ALS is a general and widely used algorithm for performing CP decomposition, but it may suffer from numerical issues when the initial amplitudes associated with some of the terms in CP decomposition are significantly smaller than those associated with other terms. In this case, the direct elimination of scalar multiples is more effective.

We perform numerical experiments to test the feasibility of simulating these quantum algorithms using CP decomposition. Our results show that, by using CP decomposition and low rank representation/approximation, we can indeed simulate some quantum algorithms with a many-qubit input on a classical computer with high accuracy. Other quantum algorithms such as quantum walks on a general graph are more difficult to simulate, because the CP rank of the intermediate tensors grows rapidly as we move along the depth of the quantum circuit representation of the quantum algorithm.

9.3 NOTATIONS FOR QUANTUM STATES, GATES AND CIRCUITS

Our analysis makes use of tensor algebra in both element-wise equations and specialized notation for tensor operations [5]. For vectors, lowercase Roman letters are used, e.g., $\mathbf{v}$. For matrices and quantum gates, uppercase Roman letters are used, e.g., $\mathbf{M}$. For tensors, calligraphic fonts are used, e.g., $\boldsymbol{\mathcal{T}}$. An order n tensor corresponds to an n-dimensional array with dimensions $s_1 \times \cdots \times s_n$. In the following discussions, we assume that $s_1 = \cdots = s_n = 2$. Elements of tensors are denotes in subscripts, e.g., $\boldsymbol{\mathcal{T}}_{ijkl}$ for an order 4 tensor $\boldsymbol{\mathcal{T}}$. For a matrix A, a_i denotes the ith column of A. Matricization is the process of unfolding a tensor into a matrix. Given a tensor $\boldsymbol{\mathcal{T}}$ the mode-i matricized version is denoted by $\mathbf{T}_{(i)} \in \mathbb{C}^{2\times 2^{n-1}}$, where all the modes except the ith mode are combined into the column. We use parenthesized superscripts to label different tensors. The Hadamard product of two matrices $\mathbf{U}, \mathbf{V}$ is denoted by $\mathbf{W} = \mathbf{U} * \mathbf{V}$. The outer product of n vectors $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(n)}$ is denoted by $\boldsymbol{\mathcal{T}} = \mathbf{u}^{(1)} \circ \cdots \circ \mathbf{u}^{(n)}$. The Kronecker product of matrices $\mathbf{A} \in \mathbb{C}^{m\times n}$ and $\mathbf{B} \in \mathbb{C}^{p\times q}$ is denoted by $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ where $\mathbf{C} \in \mathbb{C}^{mp\times nq}$. For matrices $\mathbf{A} \in \mathbb{C}^{m\times k}$ and $\mathbf{B} \in \mathbb{C}^{n\times k}$, their Khatri-Rao product results in a matrix of size $mn \times k$ defined by $\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1, \ldots, \mathbf{a}_k \otimes \mathbf{b}_k]$. We use $\mathbf{A}^\dagger$ and $\mathbf{A}^+$ to denote the conjugate and the pseudo-inverse of the matrix $\mathbf{A}$, respectively.

The quantum state $|\psi\rangle$ with n qubits is a unit vector in $\mathbb{C}^{2^n}$. It can be viewed as an order n tensor $\boldsymbol{\mathfrak{T}}^{(\psi)} \in \mathbb{C}^{2\times\cdots\times 2}$,

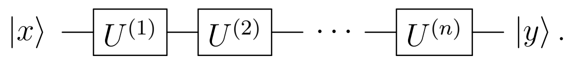
$$|\psi\rangle = \sum_{i_1,\ldots,i_n\in\{0,1\}} \boldsymbol{\mathfrak{T}}^{(\psi)}_{i_1 i_2\ldots i_n} |i_1 i_2 \ldots i_n\rangle. \tag{9.3}$$

The Kronecker product of two quantum states $|\psi\rangle, |\phi\rangle$ can be written as $|\psi\rangle \otimes |\phi\rangle$ or $|\psi\rangle|\phi\rangle$. We use the quantum circuit diagram [275] to represent the unitary transformation on a n-qubit system. In the quantum circuit, the unitary transformation is decomposed into simpler unitaries. Each factor $U^{(i)}$ corresponds to one layer of the circuit, which consists of Kronecker products of 2×2 or 4×4 unitary matrices known as one-qubit and two-qubit gates. Some commonly used one-qubit gates are:

$$H := \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, \quad X := \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}, \quad Z := \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}, \tag{9.4}$$

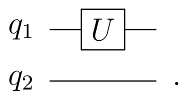
$$R_n := \begin{bmatrix}1 & 0\\ 0 & e^{-\frac{2\pi i}{2^n}}\end{bmatrix}, \quad R_y(\theta) := \begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}. \tag{9.5}$$

Graphically, applying n 2×2 operators $U^{(1)}, U^{(2)}, \ldots, U^{(n)}$ successively to a one-qubit state $|x\rangle$ yields $|y\rangle = U^{(n)}\cdots U^{(2)}U^{(1)}|x\rangle$. This operation can be drawn as



(9.6)

The application of a 4×4 operator $U\otimes I$ to two qubits q_1 and q_2, where U denotes an arbitrary 2×2 unitary matrix, can be drawn as



(9.7)

A controlled gate controlled-U is a 4×4 operator whose expression is

$$\begin{bmatrix}I & O\\ O & U\end{bmatrix} = E_1\otimes I + E_2\otimes U, \quad \text{where} \quad E_1 = \begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}, \quad E_2 = \begin{bmatrix}0 & 0\\ 0 & 1\end{bmatrix}. \tag{9.8}$$

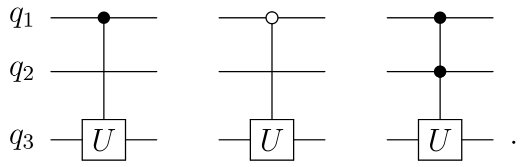
The control-on-zero gate is similar to the controlled gate and is expressed as

$$\begin{bmatrix}U & O\\ O & I\end{bmatrix} = E_1\otimes U + E_2\otimes I. \tag{9.9}$$

The generalized controlled gate controls the behavior of one qubit based on multiple qubits. For a 3-qubit system where U operates on the third qubit, the controlled-controlled-U gate is expressed as

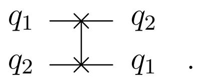
$$E_1 \otimes E_1 \otimes U + (I \otimes I - E_1 \otimes E_1) \otimes I = I \otimes I \otimes I + E_1 \otimes E_1 \otimes (U - I). \tag{9.10}$$

The diagrammatic representations for these three gates are shown respectively as follows,



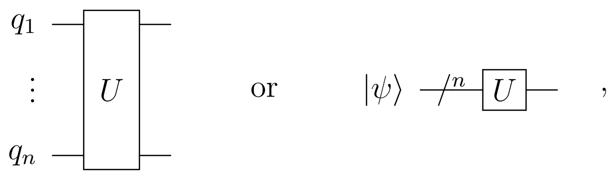
(9.11)

The controlling qubit is denoted with a solid circle when it's control-on-one, and is denoted with an empty circle when it's control-on-zero. The gate U applies on the controlled qubit with a line connected to the controlling qubits. The SWAP gate is defined as $\text{SWAP}(|x\rangle \otimes |y\rangle) = |y\rangle \otimes |x\rangle$, and is graphically denoted by



(9.12)

In general, a unitary transformation $U \in \mathbb{C}^{2^n \times 2^n}$ applied to an n-qubit state $|\psi\rangle$ is denoted as



(9.13)

where '$/^n$' indicates the state contains n qubits.

9.4 SIMULATION OF QUANTUM ALGORITHMS

Although tremendous progress has been made in the development of quantum computing hardware [276], [277], enormous engineering challenges remain in producing reliable quantum computers with a sufficient number of qubits required for solving practical problems. However, these challenges should not prevent us from developing quantum algorithms that can be deployed once reliable hardware becomes available. Our understanding of many quantum algorithms can be improved by simulating these algorithms on classical computers. Furthermore, classical simulations of quantum algorithms also provide a validation tool for testing quantum hardware on which quantum algorithms are to be executed.

Although we can in principle simulate quantum algorithms on a classical computer by constructing an unitary transformation as a matrix U and the input state $|\psi\rangle$ as a vector explicitly, and performing $U|\psi\rangle$ as a matrix vector multiplication, this approach quickly becomes infeasible as the number of simulated qubits increases.

In many quantum algorithms, the input to the quantum circuit $|\psi\rangle$ has a low CP rank, i.e., we can rewrite $|\psi\rangle$ as

$$|\psi\rangle = \sum_{j=1}^{R} a_j^{(1)} \otimes a_j^{(2)} \cdots \otimes a_j^{(n)}, \tag{9.14}$$

where $R \ll n$ is an integer that is relatively small, and $a_j^{(i)} \in \mathbb{C}^2$ is a vector of length 2. The storage requirement for keeping $|\psi\rangle$ in a rank-R CP format is $\mathcal{O}(2nR)$, which is significantly less that the $\mathcal{O}(2^n)$ requirement for representing $|\psi\rangle$ as an order n tensor or a single vector.

Because the unitary transformation encoded in a quantum algorithm is implemented by a quantum circuit that consists of a sequence of one- and two-qubit quantum gates as discussed in Section 9.3, the transformation can be implemented efficiently using local transformations that consist of multiplications of 2×2 matrices with vectors of length 2, and we may be able to keep the intermediate states produced in the quantum circuit low rank also. Let us consider the case where the input state to a quantum circuit is rank-1, i.e.,

$$|\psi\rangle = a^{(1)} \otimes a^{(2)} \otimes \cdots \otimes a^{(n)}, \tag{9.15}$$

where $a^{(i)} \in \mathbb{C}^2$. It is easy to see that applying a one-qubit gate, such as the Hadamard gate H, or a two-qubit SWAP gate does not change the CP rank of $|\psi\rangle$. For example, if H is applied to the first qubit of $|\psi\rangle$, and the first and the last qubit are swapped, the resulting states become

$$Ha^{(1)} \otimes a^{(2)} \otimes \cdots \otimes a^{(n)}, \quad \text{and} \quad a^{(n)} \otimes a^{(2)} \otimes \cdots \otimes a^{(n-1)} \otimes a^{(1)}, \tag{9.16}$$

respectively. Both are still rank-1 tensors. Unfortunately, not all two-qubit gate can keep the intermediate output in rank-1. A commonly used two-qubit gate, the controlled-U gate defined by (9.8), doubles the CP rank when it is applied to a rank-1 tensor, as we can see from the simple algebraic expressions below:

$$\begin{aligned}
&(E_1 \otimes I \otimes \cdots \otimes I + E_2 \otimes U \otimes \cdots \otimes I)\,|\psi\rangle \\
&= E_1 a^{(1)} \otimes a^{(2)} \otimes \cdots \otimes a^{(n)} + E_2 a^{(1)} \otimes U a^{(2)} \otimes \cdots \otimes a^{(n)} \\
&= \alpha\,|0\rangle \otimes a^{(2)} \otimes \cdots \otimes a^{(n)} + \beta\,|1\rangle \otimes U a^{(2)} \otimes \cdots \otimes a^{(n)},
\end{aligned} \tag{9.17}$$

where α, β are the first and second components of $a^{(1)}$ respectively. As along as neither α or β is zero, (9.17) is rank-2.

Successive applications of controlled-unitary gates where the controlling qubit vary can rapidly increase the CP rank of the output tensor. In the worst case, the rank of the output tensor can reach 2^n after n controlled unitary gates are applied. This rapid increase in CP rank clearly diminishes the benefit of the low-rank representation. However, the output of several quantum algorithms are expected to have only a few large amplitude components, i.e., they can be approximated by low rank tensors. Therefore, the rapid increase in the CP rank of the intermediate tensors produced at successive stages of the quantum circuit may be due to the sub-optimal representation of the tensor. Because the CP decomposition of a tensor is not unique, it may be possible to find an alternative CP decomposition that has a lower rank.

When such a decomposition does not exist, we seek to find a low rank approximation that preserves the main feature of the quantum algorithm to be simulated and its output. We discuss low rank approximation techniques in the next section and examine the effects of these low rank approximation techniques on several examples in Sections Section 9.6 through 9.8 and Section 9.10. As we will see, for some quantum algorithms where this low rank approximation yields relatively small intermediate state truncation errors, the simulation output can still be accurate.

9.5 LOW-RANK APPROXIMATION IN QUANTUM ALGORITHM SIMULATION

In this section, we discuss two techniques for reducing the rank of a CP decomposition of the tensor in the context of quantum algorithm simulation. Before we describe the details of these techniques, we first outline the basic procedure of using low rank approximation in the simulation of a quantum algorithm represented by a quantum circuit in Algorithm 9.1. Rank reduction techniques are used in Line 6 of the algorithm.

We should note that for some quantum algorithms, the unitary transformation U can be decomposed as

$$U = \sum_{r=1}^{R_u} \mathbf{A}_r^{(1)} \otimes \cdots \otimes \mathbf{A}_r^{(n)}, \tag{9.18}$$

where $U \in \mathbb{C}^{2^n \times 2^n}$, $R_u \ll 2^n$, $\mathbf{A}_r^{(i)} \in \mathbb{C}^{2\times 2}$, $i \in \{1, \ldots, n\}$. In this case, the multiplication of U with $|\psi\rangle$ results in a low-rank tensor if $|\psi\rangle$ is low-rank also. It is sometimes possible to obtain a good low-rank approximation of U even when U is not strictly low-rank [278]. Although seeking a low rank approximation of U can enable efficient simulations of quantum

Algorithm 9.1: Quantum Algorithm Simulation with Low-rank Approximation

```
1: Input: An input state $|\psi\rangle$ represented in CP format (9.14). A quantum circuit with
   $D$ layers one- or two-qubit gates, i.e., $U = U^{(1)}U^{(2)}\cdots U^{(D)}$, where $U^{(i)}$ is a Kronecker
   product of one- or two-qubit unitaries with $2 \times 2$ identities; maximum CP rank allowed
   $r_{\max}$ for any intermediate state produced in the simulation.
2: Output: Approximation to $|\phi\rangle = U|\psi\rangle$.
3: for $k \in \{1, 2, ..., D\}$ do
4:     Compute $|\phi\rangle = U^{(k)}|\psi\rangle$;
5:     if the number of rank-1 components of $|\phi\rangle$ ($R$ in (9.14)) exceeds $r_{\max}$ then
6:         Apply a rank reduction procedure to $|\phi\rangle$ to reduce the CP rank of $|\phi\rangle$ to at most
   $r_{\max}$;
7:     end if
8:     $|\psi\rangle \leftarrow |\phi\rangle$;
9: end for
10: Return: $|\phi\rangle$ in CP decomposed form.
```

algorithms with any low-rank input states, it is a harder problem to solve than finding a low-rank approximation of intermediate tensor. In this paper, we will not discuss this approach.

To simplify our discussion, we define the matrix

$$A^{(i)} = \left[a_1^{(i)}\ a_2^{(i)}\ \cdots\ a_R^{(i)}\right], \tag{9.19}$$

for $i = 1, 2, ..., n$, where $a_i^{(j)}$'s are 2×1 vectors that appear in (9.14), and sometimes use the short-hand notation

$$\left\{\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)}\right\} \tag{9.20}$$

to denote the tensor $|\psi\rangle$ (9.14) in CP representation. The pth term of (9.14) is denoted by

$$|\psi_p\rangle = \left\{a_p^{(1)}, a_p^{(2)}, ..., a_p^{(n)}\right\}. \tag{9.21}$$

9.5.1 Low-rank Approximation via Alternating Least Squares

We now discuss methods for performing rank reduction in Line 6 of Algorithm 9.1. A general way to reduce the rank of a tensor in CP format is to formulate the rank reduction problem as an optimization problem and solve the problem using a numerical optimization technique. To reduce the rank of $|\psi\rangle$ denoted by (9.20), from R to $s < R$, we seek the solution to the following nonlinear least squares problem

$$\min_{\mathbf{B}^{(1)},\cdots,\mathbf{B}^{(n)}} \frac{1}{2} \left\| \left\{ \mathbf{B}^{(1)}, \cdots, \mathbf{B}^{(n)} \right\} - \left\{ \mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)} \right\} \right\|_F^2, \tag{9.22}$$

where the rank of $B^{(i)}$ is s.

Note that the target CP low-rank decomposition scenario has several differences from the standard CP decomposition scenario, where the goal is to approximate a given large tensor with low order (3 or 4). We summarize the detailed differences as follows.

- We consider the case where the input tensor order is high, and each tensor mode size is small (equals 2). The CP rank can be much higher compared to the mode size.
- We consider the case where both inputs and outputs can have complex numbers.
- In our case, the input to each CP decomposition routine is an implicit tensor expressed in the CP format with rank higher than the decomposition rank, rather than an explicit large tensor.

A widely used method for solving (9.22) is the alternating least squares (ALS) [6], [266] method, which we will refer to as the CP-ALS method. Given a starting guess of $\{\mathbf{B}^{(1)}, \cdots, \mathbf{B}^{(n)}\}$, CP-ALS seeks to update one component $B^{(i)}$ at a time while $B^{(j)}$'s are fixed for $j \neq i$. Such an update can be obtained by solving a linear least squares problem. The solution of the linear least squares problem satisfies the normal equation

$$\mathbf{B}^{(i)}\Gamma^{(i)} = \mathbf{T}^{(\psi)}{}_{(i)}\mathbf{P}^{(i)}, \tag{9.23}$$

where $\mathbf{T}^{(\psi)}{}_{(i)} \in \mathbb{C}^{2\times 2^{n-1}}$ is $\mathfrak{T}^{(\psi)}$ (the tensor view of $|\psi\rangle$) matricized along the ith mode, the matrix $\mathbf{P}^{(i)} \in \mathbb{C}^{2^{n-1}\times r}$ is formed by Khatri-Rao products of $B^{(j)}$'s for $j \neq i$, i.e.,

$$\mathbf{P}^{(i)} = \mathbf{B}^{(1)\dagger} \odot \cdots \odot \mathbf{B}^{(i-1)\dagger} \odot \mathbf{B}^{(i+1)\dagger} \odot \cdots \odot \mathbf{B}^{(n)\dagger}, \tag{9.24}$$

and $\mathbf{\Gamma}^{(i)} \in \mathbb{C}^{R\times R}$ can be computed via a sequence of Hadamard products,

$$\Gamma^{(i)} = \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(i-1)} * \mathbf{S}^{(i+1)} * \cdots * \mathbf{S}^{(n)}, \tag{9.25}$$

with each $\mathbf{S}^{(i)} = \mathbf{B}^{(i)T}\mathbf{B}^{(i)\dagger}$. The *Matricized Tensor Times Khatri-Rao Product* or MTTKRP computation $\mathbf{M}^{(i)} = \mathbf{T}^{(\psi)}{}_{(i)}\mathbf{P}^{(i)}$ is the main computational bottleneck of CP-ALS. The computational cost of MTTKRP is $\Theta(2^n R)$. Because $\mathfrak{T}^{(\psi)}$ is already decomposed in CP format (9.20), the MTTKRP computation used in (9.23) can be computed efficiently via

$$\mathbf{M}^{(i)} = \mathbf{A}^{(i)}\Big(\mathop{\Huge *}_{j\in\{1,\dots,n\},j\neq i}(\mathbf{A}^{(j)T}\mathbf{B}^{(j)\dagger})\Big), \tag{9.26}$$

with complexity $\mathcal{O}(nsR)$. The algorithm is described in Algorithm 9.2. Consider the case where t iterations are performed in the ALS procedure, Algorithm 9.2 has the memory cost of $\mathcal{O}(Rn + s^2)$ and the computational cost of $\mathcal{O}(t(Rsn^2 + s^3n))$ (the term Rsn^2 is the cost of (9.25),(9.26) and the term s^3n is the cost of performing linear system solves), yielding an asymptotic computational cost of $\mathcal{O}(Rsn^2 + s^3n)$ considering t is usually a constant.

Algorithm 9.2: ALS procedure for CP decomposition of an implicit tensor

```
 1: **Input**: $\{\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)}\}$, compression rank $s$
 2: Initialize $\mathbf{B}^{(i)}$ for $i \in \{1,\dots,n\}$ as uniform random matrices within $[0,1]$
 3: $\mathbf{S}^{(i)} \leftarrow \mathbf{B}^{(i)T}\mathbf{B}^{(i)\dagger}$ for $i \in \{1,\dots,n\}$
 4: while not converge do
 5:     for $i \in \{1,\dots,n\}$ do
 6:         $\Gamma^{(i)} \leftarrow \mathbf{S}^{(1)} * \cdots * \mathbf{S}^{(i-1)} * \mathbf{S}^{(i+1)} * \cdots * \mathbf{S}^{(n)}$
 7:         Update $\mathbf{M}^{(i)}$ based on (9.26)
 8:         $\mathbf{B}^{(i)} \leftarrow \mathbf{M}^{(i)}\Gamma^{(i)+}$,    $\mathbf{S}^{(i)} \leftarrow \mathbf{B}^{(i)T}\mathbf{B}^{(i)\dagger}$
 9:     end for
10: end while
11: **Return**: $\{\mathbf{B}^{(1)},\dots,\mathbf{B}^{(n)}\}$
```

9.5.2 Direct Elimination of Scalar Multiples

If the pth term in (9.14) is a scalar multiple of the qth term, for $p \neq q$, these two terms can be combined. As a result, the effective rank of $|\psi\rangle$ can be lowered. As we will see in subsequent sections, scalar multiples of the same rank-1 tensor do appear in intermediate states of a quantum circuit for some quantum algorithms. Therefore, detecting such scalar multiples and combining them is an effective strategy for reducing the CP rank of intermediate states in Line 6 of Algorithm 9.1.

One way to check whether the pth term in (9.14) is a scalar multiple of the qth term is to compute the cosine of the angle between these two rank-1 tensors defined by

$$\cos(\theta_{p,q}) = \frac{\langle\psi_p|\psi_q\rangle}{\|\psi_p\|\cdot\|\psi_q\|}, \tag{9.27}$$

where the inner product $\langle\psi_p|\psi_q\rangle$ can be easily computed as

$$\langle\psi_p|\psi_q\rangle = \langle a_p^{(1)}, a_q^{(1)}\rangle \cdot \langle a_p^{(2)}, a_q^{(2)}\rangle \cdots \langle a_p^{(n)}, a_q^{(n)}\rangle\,, \tag{9.28}$$

and $\|\psi_p\|$ is the 2-norm of $|\psi_p\rangle$ defined as

$$\|\psi_p\| = \sqrt{\langle\psi_p|\psi_p\rangle}. \tag{9.29}$$

If $|\cos(\theta_{p,q})|$ is 1.0, $|\psi_q\rangle$ is a scalar multiple of $|\psi_p\rangle$. It can be combined with $|\psi_p\rangle$ as

$$|\psi_p\rangle \leftarrow \left(1 + \frac{\cos(\theta_{p,q})\,\beta}{\alpha}\right)|\psi_p\rangle\,, \tag{9.30}$$

where $\alpha = \|\psi_p\|$ and $\beta = \|\psi_q\|$.

Algorithm 9.3 gives a procedure of detecting and combining scalar multiples of rank-1 terms in a tensor $|\psi\rangle$ in CP format. Note that Algorithm 9.3 essentially computes the Gram matrix G associated with all rank-1 terms in the CP decomposition of $|\psi\rangle$, where the (p,q)th element of G is the cosine of the angle between the ith and jth terms. If G is rank deficient, which can be determined by performing singular value decomposition of G, $|\psi\rangle$ can be expressed as a linear combination of fewer tensors (viewed as vectors). However, each one of these tensor may not have a rank-1 CP form. Therefore, this approach does not necessarily yield a rank reduction in CP format.

9.5.3 Fidelity Estimation

Consider two states $|\psi_P\rangle$ and $|\psi_T\rangle$, where $|\psi_P\rangle$ denotes the perfect/accurate state and $|\psi_T\rangle$ denotes the truncated (low-rank approximated) state. The fidelity $\mathcal{F}$ of $|\psi_T\rangle$ in approximating $|\psi_P\rangle$ is defined as

$$\mathcal{F}(|\psi_P\rangle, |\psi_T\rangle) = |\langle\psi_P|\psi_T\rangle|^2, \tag{9.31}$$

which is a metric for measuring the accuracy of the truncated state, $|\psi_T\rangle$. For a general quantum algorithm, measuring the true fidelity of a low-rank approximation is generally difficult, since $|\psi_P\rangle$ is generally not available or costly to calculate. We introduce a fidelity estimation scheme below to approximate the fidelity with much lower computational cost. This estimation scheme is used in our experiments.

Consider a circuit consisting of D layers of quantum gates. Each layer is denoted by $U^{(i)}$, where $i \in \{1, \ldots, D\}$. Let the truncated state resulting from the application of the first i layers of gates be $|\psi_T(i)\rangle$, i.e., $|\psi_T(i)\rangle$ is the output of performing rank reduction on the state

Algorithm 9.3: Detect and Combine rank-1 terms in a tensor $|\psi\rangle$ in a CP format

```
1: **Input**: $\{\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)}\}$, where $\mathbf{A}^{(i)} \in \mathbb{C}^{2\times R}$,
2: **Output**: $\{\mathbf{B}^{(1)}, \cdots, \mathbf{B}^{(n)}\} = \{\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)}\}$, where $\mathbf{B}^{(i)} \in \mathbb{C}^{2\times s}$, with $s \le R$.
3: Initialize $\mathbf{B}^{(i)} \leftarrow \mathbf{A}^{(i)}$ for $i \in \{1, \dots, n\}$
4: $K \leftarrow R$
5: $p \leftarrow 1$
6: while $p \le K$ do
7:     $l \leftarrow \{\}$
8:     for $q \in \{p+1, \dots, K\}$ do
9:         Calculate $\cos(\theta_{p,q})$ based on (9.27)
10:        if $|\cos(\theta_{p,q})| = 1$ then
11:            Update $|\psi_p\rangle := \{b_p^{(1)}, b_p^{(2)}, ..., b_p^{(n)}\}$ based on (9.30)
12:            Append $q$ to $l$
13:        end if
14:    end for
15:    Remove the columns $b_i^{(k)}$ from $B^{(k)}$ when indices $i$ appear in $l$, for $k \in \{1, \dots, n\}$
16:    $K \leftarrow$ number of columns of $\mathbf{B}^{(1)}$
17:    $p \leftarrow p+1$
18: end while
19: **Return**: $\{\mathbf{B}^{(1)}, \dots, \mathbf{B}^{(n)}\}$
```

$U^{(i)}|\psi_T(i-1)\rangle$. Define the local fidelity f_i as the fidelity of this rank reduction:

$$f_i = |\langle\psi_T(i)|U^{(i)}|\psi_T(i-1)\rangle|^2, \tag{9.32}$$

the global fidelity $\mathcal{F}$ can be approximated by the products of all the local fidelity:

$$\mathcal{F}(|\psi_P(D)\rangle, |\psi_T(D)\rangle) = |\langle\psi_P(D)|\psi_T(D)\rangle|^2 \approx \prod_{i=1}^{D} f_i. \tag{9.33}$$

Note that this approximation is not restricted to a specific low-rank approximation format. Our experimental results show that this approximation is accurate when approximating the state with the CP representation. Reference [39] showed that it's accurate when performing the approximation with the MPS representation.

To check the convergence of CP-ALS, we calculate the local fidelity of the CP decomposition after each ALS iteration. Consider that $U^{(i)}|\psi_T(i-1)\rangle$ is represented in the CP format by $\{\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)}\}$ and $|\psi_T(i)\rangle$ is represented in the CP format by $\{\mathbf{B}^{(1)}, \dots, \mathbf{B}^{(n)}\}$,

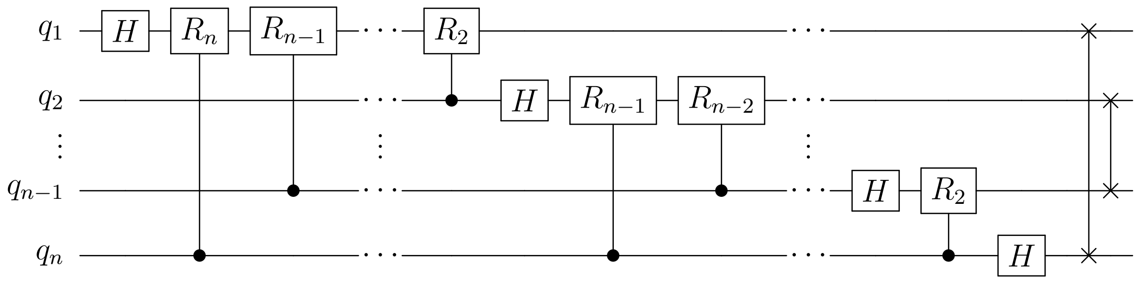


Figure 9.1: Circuit representation for quantum Fourier transform.

the fidelity f_i can be efficiently calculated by

$$f_i = \left[e^T \Big(\mathop{\scalebox{2}{$\ast$}}_{j\in\{1,\dots,n\}} (\mathbf{B}^{(j)T}\mathbf{A}^{(j)\dagger}) \Big) e \right]^2, \tag{9.34}$$

where e is an all ones vector. In this way, the computational cost of both the inner product and the fidelity calculation is $\mathcal{O}(Rsn)$. In addition, (9.34) can also be used to calculate the Frobenius-norm of a tensor $\mathfrak{T}$ represented in the CP representation, $\|\mathfrak{T}\|_F = \sqrt{|\langle \mathfrak{T}|\mathfrak{T}\rangle|}$.

9.6 QUANTUM FOURIER TRANSFORM AND PHASE ESTIMATION

9.6.1 Quantum Fourier Transform

The quantum Fourier transform (QFT) uses a special decomposition [269], [279] of the discrete Fourier transform $F^{(N)}$ define by

$$F^{(N)} := \frac{1}{\sqrt{N}} \begin{bmatrix} \omega_N^0 & \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \omega_N^2 & \cdots & \omega_N^{N-1} \\ \omega_N^0 & \omega_N^2 & \omega_N^4 & \cdots & \omega_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \cdots & \omega_N^{(N-1)(N-1)} \end{bmatrix} \in \mathbb{C}^{N\times N}, \tag{9.35}$$

where $N = 2^n$, and the output is $y = F^{(N)}x$ for the input vector $x \in \mathbb{C}^N$. We show the quantum circuit for QFT in Fig. 9.1. As is shown in the figure, a n-qubit QFT circuit consists of n 1-qubit Hadamard gates, $\lfloor N/2 \rfloor$ SWAP gates, and $n-1$ controlled unitary (R_i) gates. Without rank reduction, applying each controlled-R_i gate can double the rank of the input CP tensor as shown in (9.17). The successive application of all controlled-R_i gates in a QFT circuit can ultimately increase the CP rank of the output tensor exponentially.

However, for some specific input states, it is possible to represent or approximate the output tensors at different layers of the circuit with low-rank tensors. In the following theorem we show that, if the input state is a standard basis, all the intermediate states in the QFT circuit are rank 1 tensors. By a standard basis, we mean a unit vector of the form

$$|i_1\rangle \otimes |i_2\rangle \otimes \cdots \otimes |i_n\rangle, \tag{9.36}$$

where $i_j \in \{0, 1\}$ for $j \in \{1, 2, ..., n\}$.

Theorem 9.1. *All the intermediate states in a QFT circuit are rank 1 if the input to the circuit is a standard basis.*

Proof. The proof relies on the observation that the input factor to each controlling qubit in the QFT circuit is always either $|0\rangle$ or $|1\rangle$, if the input to the circuit is a standard basis. A controlled unitary does not change that factor and keeps the output as a rank-1 tensor. For example, when the first factor of the input rank-1 tensor is $|0\rangle$ or $|1\rangle$, only one of the two terms in (9.17) is retained, and the rank of the output tensor remains rank-1. In addition, because the 1-qubit Hadamard (H) gate and SWAP gate do not change the CP rank of an input tensor, all the intermediate output tensors at each layer of the circuit remain rank 1. Q.E.D.

Theorem 9.1 suggests that the simulation of QFT with an standard basis as the input can be simulated with $\mathcal{O}(n)$ memory. Because the application of each 1-qubit and 2-qubit gate costs $\mathcal{O}(1)$ operations, the overall computational cost of the simulation is $\mathcal{O}(n^2)$. When the input state is a linear combination of l standard basis, the CP ranks of all intermediate states are bounded by l. The memory cost for simulating such a QFT is $\mathcal{O}(ln)$ and the computational cost of the simulation is $\mathcal{O}(ln^2)$.

Note that the analysis above can be extended to the analysis for the inverse of QFT (QFT^{-1}), which inverts the input and the output of the QFT circuit shown in Fig. 9.1. If the output of QFT^{-1} is a standard basis, then all the intermediate states will have rank 1. This is because the QFT^{-1} circuit can be expressed as

$$U = U^{(D)-1} U^{(2)-1} \cdots U^{(1)-1}, \tag{9.37}$$

where $U^{(1)} U^{(2)} \cdots U^{(D)}$ makes the QFT circuit. Therefore, the intermediates of QFT^{-1} are the same as those in QFT, but in a reversed order.

9.6.2 Phase Estimation

One of the main applications of QFT is phase estimation [270]. The goal of phase estimation is to estimate an eigenvalue of a unitary operator U corresponding to a specific eigenvector $|\psi\rangle$. All eigenvalues of U are on the unit circle, which can be represented by $e^{i2\pi\theta}$ for some phase angle θ. We assume that $|\psi\rangle$ can be prepared somehow, and there exists an "oracle" that performs $U^{2^j}|\psi\rangle$ for $j \in \{0, \cdots, n-1\}$.

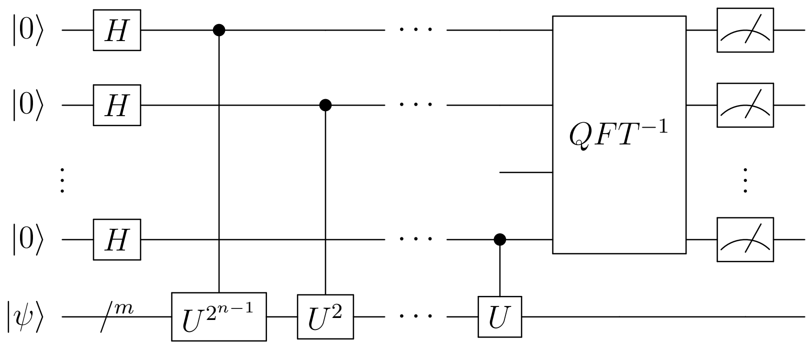


Figure 9.2: Circuit representation for phase estimation.

The quantum circuit that performs phase estimation takes two registers as the input. The first register is initialized as $|0\cdots0\rangle$. The number of qubits (n) contained in the register depends on how accurately we want to represent θ as a binary. The second register is used to prepare the target eigenvector $|\psi\rangle$. The circuit consists of a set of Hadamard gates applied to the first register, followed by a sequence of controlled-U^{2^j} gates as shown in Fig. 9.2. After applying these gates, we obtain the following rank-1 tensor in the first register,

$$\frac{1}{2^{n/2}}\left(|0\rangle + e^{i2\pi 2^{n-1}\theta}|1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + e^{i2\pi 2^{1}\theta}|1\rangle\right) \otimes \left(|0\rangle + e^{i2\pi 2^{0}\theta}|1\rangle\right). \tag{9.38}$$

When the above state is used as the input to QFT^{-1}, we obtain the following output,

$$\frac{1}{2^n}\sum_{x=0}^{2^n-1}\sum_{k=0}^{2^n-1} e^{-\frac{2\pi i k}{2^n}(x-2^n\theta)}|x\rangle. \tag{9.39}$$

If θ satisfies

$$x = 2^n\theta, \tag{9.40}$$

the amplitude of $|x\rangle$ is 1 and the amplitude of $|y\rangle$ is zero for $y \neq x$. As a result, (9.39) is a standard basis and is rank-1. Furthermore, because the inverse QFT circuit is identical to the QFT circuit, but with the input and output reversed, the intermediate output at each layer of the inverse QFT circuit should be rank-1 when (9.38) is the input and (9.40) holds exactly. In this case, the phase estimation algorithm can be simulated efficiently.

When (9.40) holds only approximately, rank reduction will be needed during the simulation of the inverse QFT to keep the intermediate output at each layer of the inverse QFT circuit low rank. We can use the techniques discussed in Section 9.5 to perform rank reduction. In Section 9.12 we provide an analysis that shows all the intermediate states on the first register of the phase estimation circuit can be approximated by a low-rank state whose CP rank is bounded by $\mathcal{O}(1/\epsilon)$, and the output state fidelity is at least $1-\epsilon$.

When the input state of the second register is a linear combination of l eigenvectors, for $l \ll n$, which occurs in applications such as noisy phase estimation [280] and NMRS quantum walk based search [281], the output of the circuit has l large amplitudes. This type of phase estimation can also be simulated efficiently.

9.7 GROVER'S ALGORITHM

Search is a common problem in information science. Grover's algorithm [271] achieves quadratic speed-up compared to the classical search algorithms. We first examine the possibility to simulate the Grover's algorithm with only one marked item using the CP representation of the tensor, then generalize the analysis to the cases with multiple marked items.

9.7.1 Search with One Marked Item

The goal of the search problem is to find a particular item x^* called the marked item from a set (X) of $N = 2^n$ items that contains x^*. On a classical computer, the worst case complexity of finding x^* is $\mathcal{O}(N)$. On a quantum computer, one can use the Grover's algorithm to find x^* in $\mathcal{O}(\sqrt{N})$ steps. In this algorithm, each item in the set to be searched is mapped to a basis state in an 2^n-dimensional Hilbert space. The algorithm involves applying an unitary transformation of the form,

$$U^{(g)} = U^{(o)}U^{(d)}, \tag{9.41}$$

successively to a superposition of all basis states, $|h\rangle = H^{\otimes n}|0^n\rangle$, where $U^{(o)}$ is known as an *oracle* that recognizes the item to be searched but does not provide the location of the item, and $U^{(d)}$ is known as a diffusion operator, which is a reflector to be defined below. It is well known that after $\mathcal{O}(\sqrt{N})$ successive applications of $U^{(g)}$ to $|h\rangle$, the amplitude associated with $|x^*\rangle$ becomes close to 1, while the amplitudes associated with other basis states become close to 0.

The oracle can be defined as

$$U^{(o)}|x\rangle = (-1)^{f(x)}|x\rangle, \quad \text{where} \quad f(x) = \begin{cases} 1 & \text{if } \ x = x^*, \\ 0 & \text{otherwise.} \end{cases} \tag{9.42}$$

This oracle is a unitary, which can be implemented as a generalized controlled-NOT gate show in Fig. 9.3. The diffusion operator is defined as

$$U^{(d)} = 2|h\rangle\langle h| - I. \tag{9.43}$$

Because $|h\rangle$ can be obtained by applying Kronecker products of Hadamard matrices to the standard basis state $|0^n\rangle$, we can write (9.43) as

$$U^{(d)} = H^{\otimes^n}\left(2|0\rangle^{\otimes^n}\langle 0|^{\otimes^n} - I\right)H^{\otimes^n}. \tag{9.44}$$

This unitary can be implemented by a layer of Hadamard gates followed by a generalized controlled-NOT, followed by another layer of Hadamard gates as shown in Fig. 9.3.

We show below that all the intermediate states produced at different layers of the circuit for Grover's algorithm can be represented by a linear combination of $|h\rangle$ and $|x^*\rangle$, which are both rank-1, if the input to the circuit is $|h\rangle$.

Without loss of generality, let us assume that $U^{(g)}$ is applied to the input $|x\rangle = \alpha|h\rangle+\beta|x^*\rangle$. When $|x\rangle$ is the input to the entire circuit, we have $\alpha = 1$ and $\beta = 0$. Applying $U^{(o)}$ to $|x\rangle$ yields

$$U^{(o)}|x\rangle = \alpha\left(|h\rangle - \frac{1}{\sqrt{N}}|x^*\rangle\right) - \alpha\frac{1}{\sqrt{N}}|x^*\rangle - \beta|x^*\rangle = \alpha|h\rangle - \left(\alpha\frac{2}{\sqrt{N}} + \beta\right)|x^*\rangle, \tag{9.45}$$

which remains in the span of $|h\rangle$ and $|x^*\rangle$. The application of U_d to $U_o\,|x\rangle$ starts with the application of $H^{\otimes^n}$ to last expression in (9.45). This yields $\alpha|0^n\rangle-(\alpha\frac{2}{\sqrt{N}}+\beta)H^{\otimes^n}|x^*\rangle$, which is in the span of $|0^n\rangle$ and $H^{\otimes^n}|x^*\rangle$. The subsequent application of the reflector $2|0^n\rangle\langle 0^n| - I$ still keeps the result in span$\{|0\rangle\,, H^{\otimes^n}|x^*\rangle\}$. Applying $H^{\otimes^n}$ again to $|0\rangle$ and $H^{\otimes^n}|x^*\rangle$ respectively brings them back to $|h\rangle$ and $|x^*\rangle$. Therefore, the CP ranks of all intermediate states are at most 2.

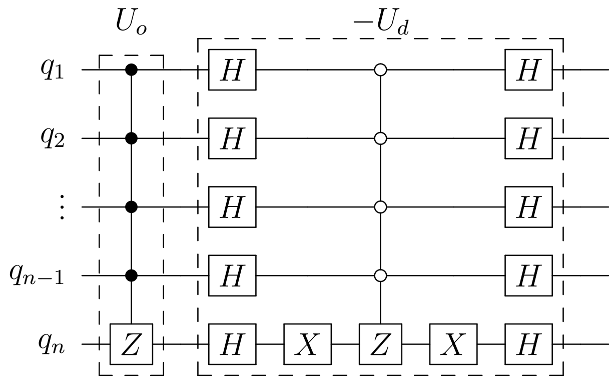


Figure 9.3: Circuit implementation for the operator $U^{(g)}$ in Grover's algorithm. The implementation assumes that the marked state is $|x^*\rangle = |11\dots1\rangle$.

9.7.2 Search with Multiple Marked Items

We now generalize the analysis to the cases with multiple marked items. The problem can be cast as the problem of finding an arbitrary element in the non-empty marked subset, A, from the set X. When mapped to the quantum circuit, the set A and the unmarked set B can be expressed as

$$\begin{aligned} A = \{x \in \{0,1\}^n \quad &: \quad f(x) = 1\}, \\ B = \{x \in \{0,1\}^n \quad &: \quad f(x) = 0\}. \end{aligned} \tag{9.46}$$

Let $a = |A|$ and $b = |B|$, we have $a + b = N = 2^n$. Grover's algorithm starts from $|h\rangle$, and applies the operator $U^{(g)}$ for $\lfloor \frac{\pi}{4}\sqrt{\frac{N}{a}} \rfloor$ times. Let

$$|A\rangle = \frac{1}{\sqrt{a}} \sum_{x \in A} |x\rangle \quad \text{and} \quad |B\rangle = \frac{1}{\sqrt{b}} \sum_{x \in B} |x\rangle, \tag{9.47}$$

as is shown in [282], application of $U^{(g)}$ on $|A\rangle$ and $|B\rangle$ results in

$$U^{(g)}|A\rangle = (1 - \frac{2a}{N})|A\rangle - \frac{2\sqrt{ab}}{N}|B\rangle \quad \text{and} \quad U^{(g)}|B\rangle = \frac{2\sqrt{ab}}{N}|A\rangle - (1 - \frac{2b}{N})|B\rangle. \tag{9.48}$$

Because the initial state is in the space S spanned by $|A\rangle$ and $|B\rangle$, all the intermediate states are all in the space S.

It is clear that $|A\rangle$ is low rank if $a \ll 2^n$ because it can be written as a linear combination of a standard bases. Although $|B\rangle$ may not appear to be low rank because it is a linear combination of many standard bases, it is actually low rank because $|B\rangle$ can be written as

$|B\rangle = (2^{n/2}|h\rangle - \sqrt{a}|A\rangle)/\sqrt{b}$. Since $|h\rangle$ is rank-1, the rank of $|B\rangle$ is $a+1$. As a result, all intermediate states in the Grover's algorithm are low rank.

The observation we made above suggests that the states that emerge from the applications of $U^{(g)}$ are low-rank. However, since both $U^{(o)}$ and $U^{(d)}$ are implemented with controlled-unitary gates, applying $U^{(g)}$ directly will increase the rank of the intermediate states. Rank reduction techniques described in Section 9.5 need to be used to keep the ranks of these tensors low. The complexity of the rank reduction procedure is thus closely related to the rank of the intermediate states emerging from direct applications of $U^{(g)}$ to the input. A detailed analysis, which is presented in Section 9.13, shows that the rank of the state input to the rank reduction procedure is at most $2(a+1)$ times the optimal rank $(a+1)$, making the procedure still efficient because a is small. For example, consider a 5-qubit system and $a = 2$. The two marked items are $|11111\rangle$ and $|00000\rangle$. Then

$$\begin{aligned} U^{(o)} &= (I - 2E_1\otimes E_1\otimes E_1\otimes E_1\otimes E_1)(I - 2E_0\otimes E_0\otimes E_0\otimes E_0\otimes E_0) \\ &= I - 2E_1\otimes E_1\otimes E_1\otimes E_1\otimes E_1 - 2E_0\otimes E_0\otimes E_0\otimes E_0\otimes E_0. \end{aligned} \tag{9.49}$$

If the input state to $U^{(o)}$ has rank R, the output from applying $U^{(o)}$ to the input has rank $(a+1)R = 3R$. The subsequent application of the gate $U^{(d)}$ increases the rank by at most a factor of two. We outline the simulation of Grover's algorithm using rank reduction in Algorithm 9.4.

We can use either a direct elimination of scalar multiples (DESM) or CP-ALS to reduce the rank of an intermediate tensor. For Grover's algorithm, DESM is much more efficient. This is because the output that emerge from the direct application of $U^{(g)}$ contains several terms that are scalar multiples of each other. For the example system shown above, applying $U^{(o)}$ to a state that's in the space S yields

$$\begin{aligned} &U^{(o)}(\alpha|00000\rangle + \beta|11111\rangle + \gamma|h\rangle) \\ &= \alpha(|00000\rangle - 2|00000\rangle) + \beta(|11111\rangle - 2|11111\rangle) + \gamma(|h\rangle - \frac{1}{2\sqrt{2}}|00000\rangle - \frac{1}{2\sqrt{2}}|11111\rangle). \end{aligned} \tag{9.50}$$

We can see that applying DESM can reduce the rank of the output tensor to 3. A similar reduction can be achieved after $U^{(d)}$ is applied.

The analysis above assumes that a is know in advance. In cases where a is unknown, it may not be easy to set the rank threshold. In Section 9.13, we also show that for any marked set that is small in size, we can approximate the intermediate states by rank-2 states, and the simulation can still yield one marked state will high probability after $\mathcal{O}(\sqrt{N})$ steps of

Algorithm 9.4: Simulating Grover's algorithm with rank reduction

```
1: Input: Input state $|h\rangle$ represented in CP format $\{\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(n)}\}$, maximum rank allowed $r_{\max}$
2: for iter $= 1, 2, ..., \lfloor \frac{\pi}{4}\sqrt{N} \rfloor$ do
3:     $\{\mathbf{B}^{(1)}, \cdots, \mathbf{B}^{(n)}\} \leftarrow$ Apply $U^{(g)}$ to the state represented by $\{\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)}\}$
4:     $\{\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)}\} \leftarrow$ Rank_Reduction($\{\mathbf{B}^{(1)}, \cdots, \mathbf{B}^{(n)}\}, r_{\max}$)
5:     Normalize the state represented by $\{\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)}\}$
6: end for
7: Return: $\{\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(n)}\}$
```

Grover's algorithm.

9.8 QUANTUM WALKS

Quantum walks [283], [284] play an important role in the development of many quantum algorithms, including quantum search algorithms [285] and the quantum page rank algorithm [286]. A quantum walk operator is the quantum extension of a classical random walk operator that has been studied extensively in several scientific disciplines. A classical random walk is characterized by an $N \times N$ Markov chain stochastic matrix P associated a graph with N vertices. There is an edge from the jth vertex to the ith vertex if the (i, j)th element of P, denoted by P_{ij}, is nonzero. A random walk is a process in which a walker randomly moves from vertex j to vertex i with probability P_{ij}. If v is a vector that gives a probability distribution of the initial position of the walker, then $w = P^t v$ gives the probability distribution of the walker's position after t steps of the random walk are taken. One of the key results in the classical random walk theory is that the standard deviation of the walker's position with respect to its mean position after taking t steps is $\mathcal{O}(\sqrt{t})$. In contrast, the standard deviation is known to be $\mathcal{O}(t)$ in a corresponding quantum walk.

The simplest type of coined quantum walk on a one dimensional cyclic lattice can be represented by the following unitary matrix

$$U = (I \otimes H)(L \otimes E_1 + R \otimes E_2), \tag{9.51}$$

where H, E_1 and E_2 are defined in (9.4),(9.8), and L and R are left and right shift permutation matrices defined by

$$L = \begin{pmatrix} 0 & 0 & \dots & 0 & 1 \\ 1 & 0 & \ddots & \ddots & 0 \\ 0 & 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 1 & 0 \end{pmatrix}, \quad R = \begin{pmatrix} 0 & 1 & \dots & \dots & 0 \\ 0 & 0 & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 1 & 0 & \dots & \ddots & 0 \end{pmatrix}. \tag{9.52}$$

The permutation matrices L and R correspond to the stochastic matrix P in a classical random walk. The Hadamard matrix H is known as a quantum coin operator that introduces an additional degrees of freedom in determining how the walker should move on the 1D lattice.

A quantum walk on a more general graph defined by a vertex set V and edge set E can be described by a formalism established by Szegedy [273], [281]. Szegedy's quantum walk is defined on the edges of the bipartite cover of the original graph (V, E), i.e. the graph is mapped to a Hilbert space $\mathcal{H}^{|V|^2} = \mathcal{H}^{|V|} \otimes \mathcal{H}^{|V|}$, with the orthonormal computational basis defined as follows:

$$\{|x, y\rangle := |x\rangle \otimes |y\rangle : x \in V, y \in V\}. \tag{9.53}$$

For each $x \in V$, $|\psi^{(x)}\rangle$ is defined as the weighted superposition of the edges emanating from x,

$$|\psi^{(x)}\rangle = |x\rangle \otimes \sum_{y \in V} \sqrt{P_{yx}} |y\rangle = |x\rangle \otimes |\phi^{(x)}\rangle. \tag{9.54}$$

By making use of the SWAP operator S and the reflection operator $U^{(d)}$ associated with $\{|\psi^{(x)}\rangle : x \in V\}$, defined as

$$S = \sum_{x,y \in V} |y, x\rangle\langle x, y|, \tag{9.55}$$

$$U^{(d)} = 2 \sum_{x \in V} |\psi^{(x)}\rangle\langle\psi^{(x)}| - I = \sum_{x \in V} |x\rangle \langle x| \otimes \left(2 |\phi^{(x)}\rangle \langle\phi^{(x)}| - I\right), \tag{9.56}$$

we can define a quantum walk operator as

$$U^{(w)} = SU^{(d)}. \tag{9.57}$$

A Szegedy's quantum walk can be used as a building block for searching a marked vertex in a graph. Let x^* be the vertex to be searched. The oracle associated with x^* is defined as

$$U^{(o)} = I - 2\sum_{y\in V} |x^*, y\rangle\langle x^*, y|. \tag{9.58}$$

Using such an oracle, we can perform the search by applying the following unitary [287],

$$U^{(s)} = U^{(o)}U^{(w)}U^{(w)}, \tag{9.59}$$

to an initial state.

In the following, we will show quantum circuits for implementing $U^{(s)}$ associated with a few Markov transition matrices induced by structured graphs. We also assume the transition probability from vertex x to vertex y is defined by

$$P_{yx} = \frac{A_{yx}}{\text{outdeg}(x)}, \tag{9.60}$$

where A is the adjacent matrix of the graph and outdeg(x) is the number of edges emanating from x. We will show that for some structured graphs, the quantum search can be simulated efficiently if the initial state is rank-1.

9.8.1 Quantum Walk on a Complete Graph with Self-loops

We first consider a complete graph with $|V| = N = 2^n$ vertices. We assume that a random step from a vertex can return to the vertex itself, i.e., the graph contains self-loops. The transition probability (Markov chain) matrix P associated with such a random walk can be expressed as

$$P = \frac{1}{N}ee^T, \tag{9.61}$$

where e is a vector of all ones. In this case, the vector $|\phi^{(x)}\rangle$ defined in (9.54) is the same for all $x \in V$, and can be set to $|h\rangle = \frac{1}{\sqrt{N}}\sum_{x\in V}|x\rangle$. As a result, the diffusion operator in (9.56) can be simplified to

$$U^{(d)} = 2\sum_{x\in V}|x\rangle\langle x| \otimes |h\rangle\langle h| - I = I \otimes (2|h\rangle\langle h| - I)\,. \tag{9.62}$$

Likewise, the oracle operator in (9.58) can be simplified to

$$U^{(o)} = I - 2|x^*\rangle\langle x^*| \otimes (\sum_{y\in V}|y\rangle\langle y|) = (I - 2|x^*\rangle\langle x^*|) \otimes I. \tag{9.63}$$

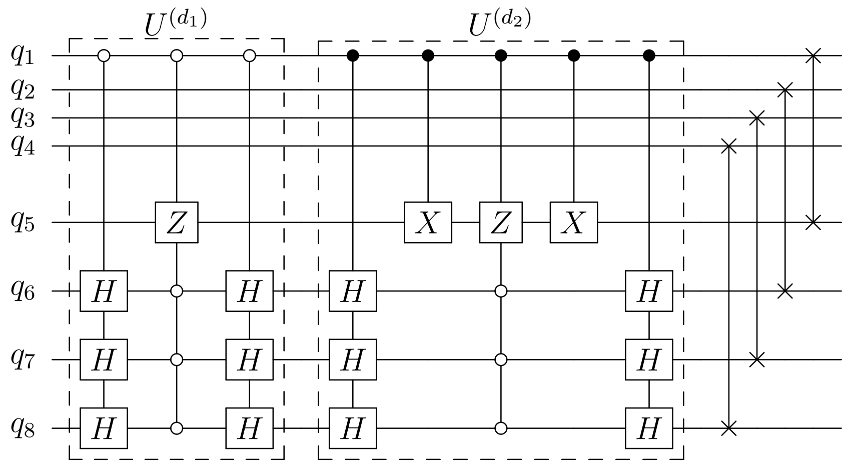


Figure 9.4: Circuit implementation of the operator $U^{(d)}$ for a quantum walk on a complete bipartite graph. The implementation assumes that $n_1 = n_2 = 3$.

With these simplified expression for $U^{(d)}$ and $U^{(o)}$, the search operator defined in (9.59) has the form

$$\begin{aligned} U^{(s)} &= \Big((I - 2|x^*\rangle\langle x^*|) \otimes I\Big)\Big((2|h\rangle\langle h| - I) \otimes I\Big)\Big(I \otimes (2|h\rangle\langle h| - I)\Big) \\ &= \Big((I - 2|x^*\rangle\langle x^*|)(2|h\rangle\langle h| - I)\Big) \otimes \Big(2|h\rangle\langle h| - I\Big). \end{aligned} \tag{9.64}$$

The circuit implementation of $U^{(s)}$ is similar to the implementation for the Grover's algorithm, since the operator on the first n qubits is $U^{(g)}$ (expressed in (9.41) and shown in Fig. 9.3), and the operator on the last n qubits is (9.43). It is easy to verify that, when the $U^{(s)}$ is applied to the rank-1 tensor $|h\rangle|h\rangle$, it produces a tensor with a rank that is at most 2. As a result, the search for x^* can be simulated on a classical computer with $\mathcal{O}(n)$ complexity in flops and memory.

9.8.2 Quantum Walk on a Complete Bipartite Graph

We now consider a quantum walk on a complete bipartite graph K_{N_1,N_2}, where the set of vertices V consists of two subsets V_1 and V_2, where $N_1 = |V_1| = 2^{n_1}$, $N_2 = |V_2| = 2^{n_2}$. Each vertex in V_1 is connected to all vertices in V_2 and vice versa. In this case vector $|\phi^{(x)}\rangle$ is defined by $|h^{(V_1)}\rangle = \frac{1}{\sqrt{N_2}} \sum_{x\in V_2} |x\rangle$, for $x \in V_1$, and by $|h^{(V_2)}\rangle = \frac{1}{\sqrt{N_1}} \sum_{x\in V_1} |x\rangle$ for $x \in V_2$.

The diffusion operator can be rewritten as

$$U^{(d)} = \sum_{x\in V_1} |x\rangle\langle x| \otimes \left(2|h^{(V_1)}\rangle\langle h^{(V_1)}| - I\right) + \sum_{x\in V_2} |x\rangle\langle x| \otimes \left(2|h^{(V_2)}\rangle\langle h^{(V_2)}| - I\right). \tag{9.65}$$

If we start from a rank-1 quantum state, all intermediate quantum states in a quantum walk based search on a complete bipartite graphs can be low-rank. Below we provide analysis for $n_1 = n_2$, and the analysis can be generalized to the cases where $n_1 \neq n_2$.

We can map all $2\cdot 2^{n_1}$ vertices to quantum states described by n_1+1 qubits. The quantum circuit for the corresponding quantum walk operates on $2n+2$ qubits. Each vertex $x \in V_1$ can be represented by $|0\rangle\,|x\rangle$, and each vertex $x \in V_2$ can be represented by $|1\rangle\,|x\rangle$. Using this representation and the definition $|h\rangle \equiv H\,|0\rangle$, we obtain

$$|h^{(V_1)}\rangle = |1\rangle\,|h\rangle^{\otimes n_1}, \quad \text{and} \quad |h^{(V_2)}\rangle = |0\rangle\,|h\rangle^{\otimes n_1}, \tag{9.66}$$

$$\sum_{x\in V_1} |x\rangle\,\langle x| = |0\rangle\,\langle 0| \otimes I^{\otimes n_1}, \quad \text{and} \quad \sum_{x\in V_2} |x\rangle\,\langle x| = |1\rangle\,\langle 1| \otimes I^{\otimes n_1}. \tag{9.67}$$

Assuming that $x^* \in V_1$, the oracle operator in (9.58) can be simplified to

$$U^{(o)} = I - 2\,|0\rangle\,\langle 0|\,|x^*\rangle\langle x^*| \otimes (\sum_{y\in V_2} |y\rangle\langle y|) = I - 2\,|0\rangle\,\langle 0| \otimes |x^*\rangle\langle x^*| \otimes |1\rangle\,\langle 1| \otimes I^{\otimes n_1}. \tag{9.68}$$

Based on (9.67), $U^{(d)}$ can be rewritten as

$$\begin{aligned} U^{(d)} = {} & \underbrace{|0\rangle\,\langle 0| \otimes I^{\otimes n_1} \otimes \left(2\,|1\rangle\,\langle 1| \otimes |h\rangle^{\otimes n_1}\,\langle h|^{\otimes n_1} - I^{\otimes(n_1+1)}\right)}_{U^{(d_1)}} \\ & + \underbrace{|1\rangle\,\langle 1| \otimes I^{\otimes n_1} \otimes \left(2\,|0\rangle\,\langle 0| \otimes |h\rangle^{\otimes n_1}\,\langle h|^{\otimes n_1} - I^{\otimes(n_1+1)}\right)}_{U^{(d_2)}}. \end{aligned} \tag{9.69}$$

As a result, we can rewrite $U^{(d_1)}$ and $U^{(d_2)}$ as

$$U^{(d_1)} = |0\rangle\,\langle 0| \otimes I^{\otimes n_1} \otimes \left[\left(I \otimes H^{\otimes n_1}\right)\left(2\,|1\rangle\,\langle 1| \otimes |0\rangle^{\otimes n_1}\,\langle 0|^{\otimes n_1} - I^{\otimes(n_1+1)}\right)\left(I \otimes H^{\otimes n_1}\right)\right], \tag{9.70}$$

$$U^{(d_2)} = |1\rangle\,\langle 1| \otimes I^{\otimes n_1} \otimes \left[\left(I \otimes H^{\otimes n_1}\right)\left(2 \otimes |0\rangle^{\otimes(n_1+1)}\,\langle 0|^{\otimes(n_1+1)} - I^{\otimes(n_1+1)}\right)\left(I \otimes H^{\otimes n_1}\right)\right]. \tag{9.71}$$

The circuit implementations of these operators are shown in Fig. 9.4.

The input state to the quantum walk based search algorithm is the superposition of all

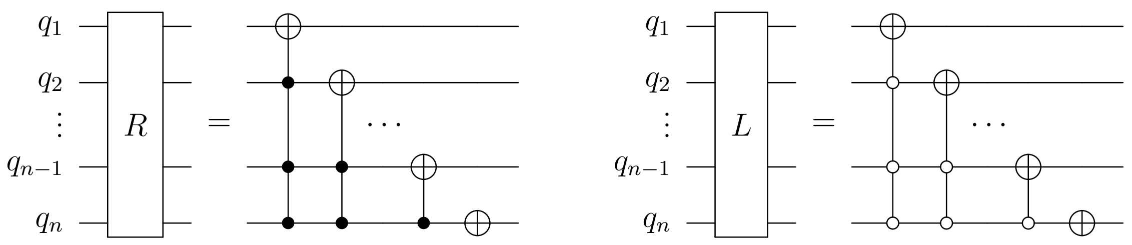


Figure 9.5: Circuit implementation for the one-element rotation operators [288].

edge states,

$$\begin{aligned}&\frac{1}{\sqrt{2|V_1|}}\sum_{x\in V_1}|x\rangle|\phi^{(x)}\rangle+\frac{1}{\sqrt{2|V_2|}}\sum_{x\in V_2}|x\rangle|\phi^{(x)}\rangle\\&=\frac{1}{\sqrt{2}}|0\rangle|h\rangle^{\otimes n_1}|1\rangle|h\rangle^{\otimes n_1}+\frac{1}{\sqrt{2}}|1\rangle|h\rangle^{\otimes n_1}|0\rangle|h\rangle^{\otimes n_1}.\end{aligned}\tag{9.72}$$

Applying the search operator $U^{(s)}=U^{(o)}U^{(w)}U^{(w)}=U^{(o)}SU^{(d)}SU^{(d)}$ amounts to applying $U^{(d)}$, $SU^{(d)}S$ and $U^{(o)}$ successively to the input state. Similar to the analysis for Grover's algorithm, it can be verified that the intermediate states emerging from the application of these unitaries implemented in quantum circuit are in the space spanned by

$$\left\{|0\rangle|h\rangle^{\otimes n_1}|1\rangle|h\rangle^{\otimes n_1},|1\rangle|h\rangle^{\otimes n_1}|0\rangle|h\rangle^{\otimes n_1},|0\rangle|x^*\rangle|1\rangle|h\rangle^{\otimes n_1},|1\rangle|h\rangle^{\otimes n_1}|0\rangle|x^*\rangle\right\}.\tag{9.73}$$

The CP rank of these intermediate states is bounded by 4. As a result, both the overall memory and computational costs are bounded by $\mathcal{O}(n)$.

9.8.3 Quantum Walk on Cyclic Graphs

The last type of quantum walk we examine is performed on a cyclic graph. The transition probability matrix associated with a cyclic graph is a circulant matrix. We focus on a subset of cyclic graphs in which each vertex is connected to $N-a$ other vertices, where $N=2^n$ and $a=2^m-1$ for some natural number $m<n$. The matrix elements of the transition matrix P are defined as

$$P_{yx}=\begin{cases}\frac{1}{\sqrt{N-a}} & \text{if } (y-x) \bmod N\geq a,\\ 0 & \text{otherwise.}\end{cases}\tag{9.74}$$

When $a=1$, the cyclic graph becomes a complete graph. To simplify the notation, we use $|x\rangle$ to denote the state $|x \bmod N\rangle$. We also let $|v^{(x)}\rangle$ be the superposition of vertices that

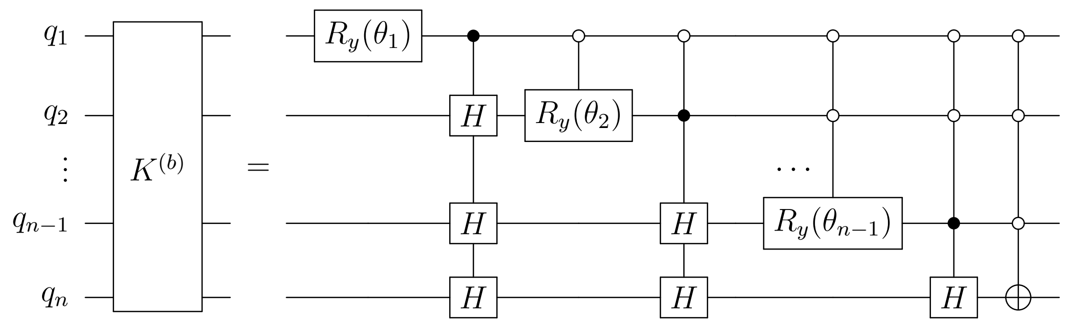


Figure 9.6: Circuit implementation for $K^{(b)}$ for the complete graph without self-loops. θ_i is defined as $\theta_i = \arccos\sqrt{\frac{2^{n-i}-1}{2^{n-i+1}-1}}$.

are not adjacent to x,

$$|v^{(x)}\rangle = \frac{1}{\sqrt{a}}\sum_{i=0}^{a-1}|x+i\rangle. \tag{9.75}$$

It follows that the vector $|\psi^{(x)}\rangle$ defined in (9.54) can be rewritten as

$$|\psi^{(x)}\rangle = |x\rangle\left(\sqrt{\frac{N}{N-a}}|h\rangle - \sqrt{\frac{a}{N-a}}|v^{(x)}\rangle\right). \tag{9.76}$$

Consequently, the diffusion operator $U^{(d)}$ has the form

$$U^{(d)} = (2\frac{N}{N-a}I\otimes|h\rangle\langle h|-I)+\sum_{x\in V}\frac{2a}{N-a}|x\rangle\langle x|\otimes\left(-\sqrt{\frac{N}{a}}\left(|v^{(x)}\rangle\langle h|+|h\rangle\langle v^{(x)}|\right)+|v^{(x)}\rangle\langle v^{(x)}|\right). \tag{9.77}$$

When $a \ll N$, $U^{(d)}$ is dominated by the first term, making it behave like a diffusion operator that appears in the Grover's algorithm discussed in Section 9.7. However, the presence of the second term can introduce entanglement in the intermediate states in a quantum circuit implementation of $U^{(d)}$. Our experimental results in Section 9.10.4 shows that low-rank approximation for quantum walk based search on cyclic graphs is not accurate.

To implement $U^{(d)}$ as a quantum circuit, we use the technique presented in [289] to construct a unitary operator $U^{(t)}$ that diagonalizes $U^{(d)}$. It has been shown that, for cyclic graphs with a circulant P, we can construct a unitary $U^{(t)}$ so that

$$U^{(t)}U^{(d)}U^{(t)+} = I\otimes(2|b\rangle\langle b|-I), \tag{9.78}$$

for some computational basis $|b\rangle$. The diagonal unitary $D = 2|b\rangle\langle b| - I$ can be efficiently

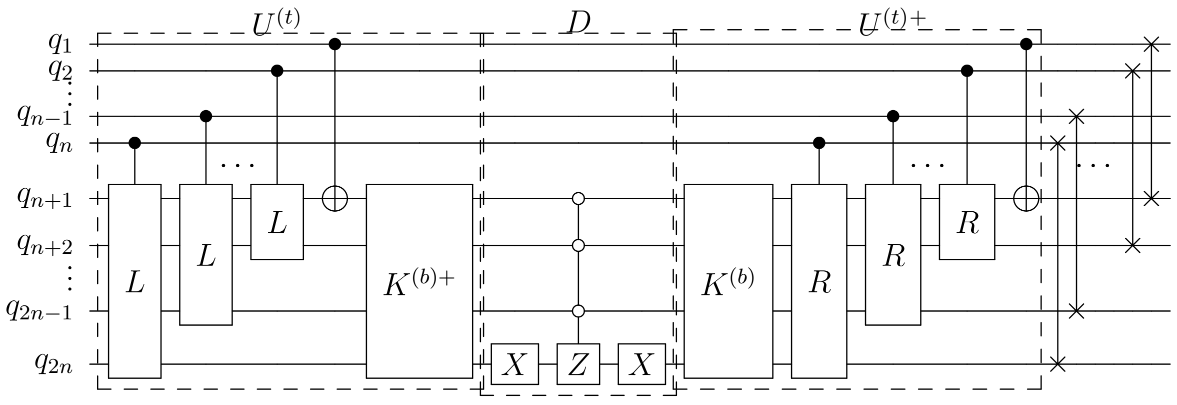


Figure 9.7: Circuit implementation for the quantum walk $U^{(w)}$ on cyclic graphs.

implemented by a quantum circuit. The unitary $U^{(t)}$ can be written as a product of shift permutation matrices L of different sizes, tensor product with identities, as well a unitary $K^{(b)}$ that maps the first column of P to $|b\rangle$. Shift permutation can be implemented efficiently by circuits shown in Fig. 9.5. When each column of P is sparse or structured, $K^{(b)}$ can also be implemented by an efficient quantum circuit (for the complete graph without self-loops, the implementation is shown in Fig. 9.6). As a result, $U^{(d)} = U^{(t)+}DU^{(t)}$ can be implemented by an efficient circuit shown in Fig. 9.7.

To summarize, quantum walk based search can be accurately low-rank simulated only on specific structure graphs, including complete graphs with self-loops, and complete bipartite graphs. For general graphs, such as cyclic graphs, the low-rank simulation will not be accurate.

9.9 SUMMARY OF COMPUTATIONAL COST

The use of low rank CP decomposition to represent the input and intermediate states in the simulation of a quantum algorithm allows us to significantly reduce the memory requirement of the simulation. If the rank of all intermediate states can be bounded by a small constant, then the memory requirement of the simulation is linear with respect to the number of qubits n. This is significantly less than the memory required to simulate a quantum algorithm directly, which is exponential with respect to n, by performing a matrix-vector multiplication.

By keeping the input and intermediate states in low rank CP form, we can also significantly reduce the number of floating point operations (FLOPs) in the simulation. Because a quantum gate in each layer of the quantum circuit is typically local, meaning that it is a 2×2 matrix operating on one factor of a CP term, the number of FLOPs required to multiply quantum gates with the input states is proportional to nrD, where r is the maximum rank of all intermediate states and D is the depth of the quantum circuit. Therefore, as long as D and r are not too large, the cost of applying a unitary transformation in decomposed form (i.e. a quantum circuit) to a low rank input is relatively low also. However, to keep intermediate states in low rank CP decompositions, rank reduction through CP-ALS or direct elimination of scalar multiples (DESM, Algorithm 9.3) may need to be used. The cost of the rank reduction computation can be higher than applying the unitary transformation associated with the quantum algorithm.

We summarize the computational costs of simulating the quantum algorithms analyzed above using CP decomposition in Table 9.1. To lower the rank of intermediate tensors in the simulation, the cost of DESM (Algorithm 9.3) is lower than CP-ALS (Algorithm 9.2). However, as can be seen in the table, DESM cannot always be effectively applied to the simulation of a quantum algorithm in which an intermediate tensor does not contain CP terms that are multiple of each other. The CP-ALS is a more general method to compress intermediate tensors emerging from successive layers of a quantum circuit. It can be used for the simulation of all quantum algorithms. When both DESM and CP-ALS can be applied, the cost of CP-ALS is typically much higher even when the number of ALS iterations is fixed at a small constant. Therefore, whenever possible, we should try using DESM first before using CP-ALS.

Algorithm	DESM	CP-ALS
QFT w/ standard basis input state	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$
QFT w/ random rank-1 input state and CP rank limit r	/	$\mathcal{O}(r^2n^3 + r^3n^2)$
Phase estimation w/ CP rank limit r	/	$\mathcal{O}(r^2n^3 + r^3n^2)$
Grover's Algorithm w/ a marked items	$\mathcal{O}(a^3n)$	$\mathcal{O}(a^3n^2)$
Quantum walks based search w/ complete graph with loops	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Quantum walks based search w/ complete bipartite graph	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$

Table 9.1: The asymptotic computational cost of simulating different quantum algorithms with CP low-rank approximation. The computational cost shown for Grover's algorithm is the cost of applying each $U^{(g)}$ defined in (9.41), and for quantum walks is the cost of applying each $U^{(w)}$ defined in (9.57).

9.10 EXPERIMENTAL RESULTS

In this section, we demonstrate the efficacy of using low rank approximation in the simulation of four quantum algorithms: QFT, phase estimation, Grover's algorithm and quantum walks. We implemented our algorithms on top of an open-source Python library, "Koala" [40], which is a quantum circuit/state simulator and provides interface to several numerical libraries, including NumPy [194] for CPU executions and CuPy [290] for GPU executions. All of our code is available at `https://github.com/LinjianMa/koala`. Most of our experiments were carried out on an Intel Core i7 2.9 GHz Quad-Core machine using NumPy routines. For some QFT simulation experiments with large number of qubits and large CP rank limits, the experiments were carried out on an NVIDIA Titan X GPU using CuPy routines to accelerate the execution.

In each of these simulations, the input to the simulated quantum circuit is chosen to be a rank-1 or low rank state in the CP representation. We limit the CP rank of the intermediate states to a fixed integer that varies from one algorithm to another. This limit may also change with respect to the number of simulated qubits employed in the quantum algorithm. Typically, a higher limit is required for simulations that employ more qubits.

We measure the fidelity of the simulation output using the metric defined in Section 9.5.3. To improve the accuracy of the CP-ALS rank reduction procedure, we repeat the procedure several times using different initial guesses of the approximate low-rank CP tensors. The approximation that yields the highest fidelity is chosen as the input to the next stage of the simulation.

9.10.1 Quantum Fourier Transform

We first simulate the QFT algorithm when the input is a standard basis. As is discussed in Theorem 9.1, all the intermediate states can be represented by a rank-1 state. The results are shown in in Table 9.2. As can be seen, we can accurately simulate the large circuits with as many as 60 qubits.

Number of qubits	18	20	22	24	26	28	30	32	40	60
Fidelity estimation (9.33)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 9.2: Numerical experimental results for QFT with DESM when the input is a standard basis. The rank limit (r) is set as 1 for all the experiments.

We also simulate the QFT algorithm in which the input to the QFT circuit is a randomly generated rank-1 state in the CP representation. Each element of each CP factor is drawn from a uniformly distributed random number between 0 and 1. As we explained in Section 9.6.1, even though the input state to the QFT circuit has rank 1, the output of the circuit may not be low rank. Furthermore, the rank of the intermediate states resulting from the application of a sequence of one or two-qubit gates may increase rapidly if no rank reduction procedure is applied.

We simulate the QFT algorithm performed on circuits with at least 16 qubits and at most 40 qubits, with the results presented in Table 9.3. The experiments with 40 qubits are run on one GPU, and all the other experiments are run on a CPU. These simulations correspond to Fourier transforms of vectors of dimensions between $2^{16} = 65,536$ and $2^{40} \approx 1.1 \times 10^{12}$. For all the experiments, we use CP-ALS to reduce the rank of the intermediate states when the CP rank of the state exceeds the limit r reported in the second row of Table 9.3. Three different random initial guesses are used in each attempt to reduce the rank of the intermediate state.

We report the approximation fidelity of the simulation output in the third row of Table 9.3. As can be seen from this table, fidelity beyond 90% can be achieved for 16 to 24-qubit QFT simulations when the CP rank of all intermediate states are limited to 256. As the number of qubits increase to 26, 27 or 28, the fidelity of the output drops below 90%. Higher CP ranks are necessary to maintain high fidelity. The largest system we have tested has 40 qubits. Even when we increase the limit of the CP rank to 2048, we can only achieve 58% fidelity. We didn't further increase the rank limit, since the simulation time will be too long (the experiment with rank limit being 2048 took around 10 hours to finish).

Number of qubits	16	20	24	26	27	28	40	40
Rank limit (r)	256	256	256	256	256	256	1024	2048
Fidelity estimation (9.33)	0.998	0.975	0.918	0.784	0.845	0.788	0.534	0.580

Table 9.3: Numerical experimental results for QFT with CP-ALS when the input is a random state with CP rank 1.

9.10.2 Phase Estimation

We next simulate phase estimation by constructing a rank-1 state according to (9.38) as the input to an inverse QFT quantum circuit. We set the phase angle θ in (9.38) to $\theta = 1/2(1 + 1/2^n)$. It follows from (9.83) that this particular choice of θ results in an output state that is not simply an elementary basis. According to Theorem 9.2, the algorithm can still be efficiently low-rank approximated.

The CP rank of the intermediate state resulting from the application of a two-qubit gate doubles. We use CP-ALS to reduce the rank of the state when the rank becomes larger than the limit of 20. Three random initial guesses are used in each CP-ALS rank reduction step, and the best approximation is used to continue the simulation.

The fidelity of the simulation is shown in Table 9.4. Because all intermediate states can be well approximated by rank-20 states, we are able to simulate large circuits with as many as 60 qubits. As can be seen from the table, high fidelity (> 0.999) can be achieved for all simulations.

Number of qubits	18	20	22	24	26	28	30	32	40	60
Rank limit (r)	20	20	20	20	20	20	20	20	20	20
Fidelity estimation (9.33)	1.0	0.9997	0.9998	0.9998	0.9995	0.9993	0.9993	0.9997	0.9972	0.9994

Table 9.4: The fidelity of phase estimation simulation.

9.10.3 Grover's Algorithm

In Section 9.7, we showed that Grover's algorithm is intrinsically low-rank, i.e., when the input to the Grover's quantum circuit is a particular rank-1 state, all intermediate states produced at each layer of the circuit have low ranks. Therefore, we should be able to simulate the algorithm by keeping the rank of all intermediate states in the simulation low using CP-ALS.

However, in practice, the use of CP-ALS to reduce the rank of an intermediate state can be difficult for large circuits that contain many qubits. This difficulty arises from the large disparity between the magnitudes of different CP components in the intermediate state produced in the early iterations of the Grover's algorithm. To be specific, the intermediate state produced by the first iteration of the Grover's algorithm can be expressed as $\alpha|h\rangle + \beta|x^*\rangle$, where the coefficient β has a magnitude close to $1/\sqrt{N}$ or $1/\sqrt{2^n}$, whereas $\alpha \sim O(1)$. The amplitude of β decreases exponentially with respect to the number of qubits. For a large n, we found that the CP-ALS output were in the direction of $|h\rangle$ for most of the random initial guesses, and the amplitude of $|x^*\rangle$ were not effectively amplified. As a result, it is difficult for CP-ALS to numerically identify the x^* component in the intermediate state.

Table 9.5 shows the feasibility of using CP-ALS in the simulation of Grover's algorithm. We use Grover's circuits that encode one marked item to be searched ($|A| = 1$) as well as circuits that encode 20 marked items to be searched ($|A| = 20$). For all the experiments, we set the CP rank limit to 2. In each experiment, we use multiple random initial guesses in the first iteration of the Grover's algorithm to produce a low rank approximation. The number of initial guesses are listed in the row labelled by Num-ALS-init. The use of more initial guesses can yield a more accurate low rank approximation. In subsequent iterations of

Grover's algorithm, we use the CP-ALS approximation produced in the previous iterations as the initial guess. In exact arithmetic, this scheme guarantees that all intermediate tensors produced in the simulated Grover's algorithm lie within the subspace spanned by $|h\rangle$ and $|x^*\rangle$, and the amplitude of $|x^*\rangle$ is amplified incrementally.

Although we can use the metric defined in (9.33) to estimate the fidelity of the low rank approximation in the Grover's algorithm, the amplitude amplification feature of the Grover's algorithm allows us to assess the fidelity of low rank approximation by directly measuring amplitudes (coefficients) associated with the basis of the marked items in the final output state. To be specific, for a set of marked items A represented by $|A|$ computational basis of an n-qubit state, the fidelity of the approximation can be evaluated as

$$\sum_{x \in A} |\langle \psi | x \rangle|^2 \,, \tag{9.79}$$

where $|\psi\rangle$ is the simulation output. As the number of Grover's iterations approaches $\frac{\pi}{4}\sqrt{N}$, the amplitude sum in (9.79) should become close to 1.0.

As we can see from Table 9.5, when $|A| = 1$, the amplitude of the marked item becomes close to 1.0 when the number of qubits is less or equal to 14 and 3 ALS initial guesses are used. When the number of qubits reaches 16, 3 ALS initial guesses are not enough for an accurate simulation, and we need to try 10 different initial guesses in the CP-ALS algorithm to obtain an accurate rank-2 approximation to the output of Grover's circuit. When $|A| = 20$, the amplitude sum obtained at the end of all simulations are above 0.5. However, for a 10-qubit simulation, the amplitude sum is only slightly above 0.5. For a 14-qubit simulation, the amplitude sum is 0.6. These low amplitude sums indicate the difficulty of using CP-ALS to find an accurate low-rank approximation to intermediate states in some iterations of the Grover's algorithm. The success of CP-ALS depends on the random initial guesses used in the first iteration of the Grover's algorithm. For a 16 qubit simulation, we were able to obtain an amplitude sum that is close to 1.0. This is likely due to a good initial guess generated for CP-ALS.

The difficulty encountered in CP-ALS rank reduction can be easily overcome by the use of the DESM reduction technique aimed at identifying CP components that are scalar multiples of each other. This is the technique we discussed in Algorithm 9.3. Table 9.6 shows that by using this technique we can consistently simulate Grover's algorithm for single or multiple marked search items with high accuracy. We have simulated circuits with as many as 30 qubits. In all experiments, the amplitude sums of the marked items obtained at the end of the simulation are close to 1.0.

Number of qubits	8	10	12	14	16	16	8	10	12	14	16
Num-marked-item	1	1	1	1	1	1	20	20	20	20	20
Num-ALS-init	3	3	3	3	3	10	3	3	3	3	3
Amplitude of A (9.79)	1.	0.999	1.0	1.0	0.0	1.0	0.972	0.537	0.981	0.607	0.998

Table 9.5: Numerical experimental results for Grover's algorithm with CP-ALS. The rank limit (r) is set as 2 for all the experiments.

Number of qubits	10	15	20	25	30	10	15	20	25	30
Num-marked-item	1	1	1	1	1	20	20	20	20	20
Amplitude of A (9.79)	0.999	1.0	1.0	1.0	1.0	0.999	1.0	1.0	1.0	1.0

Table 9.6: Numerical experimental results for Grover's algorithm with DESM. The rank limit (r) is set as 2 for all the experiments.

9.10.4 Quantum Walks

In Section 9.8, we showed that intermediate states in a quantum walk can be low-rank when the starting point of the walk is a particular rank-1 state for some graphs (e.g. complete graphs with loops and complete bipartite graphs). We will show that in these cases, the quantum walk can be efficiently and accurately simulated using a low rank representation.

We measure the accuracy of the simulations by the estimated fidelity (9.33) and the amplitude (coefficient) of the marked item, x^*, which is defined as

$$\sum_{y \in V} |\langle \psi | x^*, y \rangle|^2 , \tag{9.80}$$

where $|\psi\rangle$ is the output state. High amplitude and high fidelity mean that the approximated algorithm is accurate. Similar to Grover's algorithm, whe the number of iterations approaches $\frac{\pi}{4}\sqrt{N}$, the amplitude sum in (9.80) should become large (greater than 0.5 for all test cases, and close to 1 for systems with a large number of qubits).

Table 9.7 and Table 9.8 show the experimental results for quantum walks on complete graphs with loops, using CP-ALS and DESM, respectively. As can be seen, we can accurately simulate the algorithm using low rank approximation by DESM for systems with different qubit counts. The rank of all intermediate tensors in these simulations are limited to 2. For CP-ALS, we find that for a large number of qubits, imposing a small rank limit (2) usually cannot guarantee accuracy in the rank-reduction procedure. For example, the amplitude of x^* for the 20-qubit system is close to 0, as is shown in Table 9.7. The reason is that CP-ALS tends to fail for intermediate states with a large disparity in the magnitudes of different CP components, similar to what is discussed in Section 9.10.3 for Grover's algorithm. To be specific, the intermediate state produced by the first iteration of the quantum walk algorithm can be expressed as $\alpha|h\rangle\,|h\rangle + \beta|x^*\rangle\,|h\rangle$, where the amplitude of β decreases exponentially with respect to the number of qubits. For a large n, the CP-ALS output were in the direction of $|h\rangle\,|h\rangle$ for most of the random initial guesses, and the amplitude of $|x^*\rangle$ were not effectively amplified. To achieve high accuracy, we need to slightly increase the rank limit. Similar to the behavior of the algorithm observed when the number of initial guesses for ALS is increased, increasing the rank limit of the intermediate tensor tends to improve the likelihood of finding better approximations in CP-ALS. For example, when we increase the rank limit to 5, the algorithm can be accurately simulated for the 20-qubits' system, and when we further increase the rank limit to 20, we can accurately simulate the 24-qubits' system.

Table 9.9 and Table 9.10 show the experimental results for quantum walks on complete bipartite graphs obtained from using CP-ALS and DESM rank reduction techniques, respectively. As discussed in Section 9.8.2, all intermediate states can be accurately represented by rank 4 tensors. As can be seen, we can accurately simulate these quantum walks with DESM for graphs of various sizes. We set the rank limit to 4 in these simulations. For CP-ALS, similar to the simulation of quantum walks on complete graphs with loops, we need to slightly increase the rank limit to achieve high accuracy. Without a sufficient increase in the allowed rank of the intermediate tensor, the CP-ALS procedure may produce a result that has a small or zero amplitude for x^*. As we can see in Table 9.9, the amplitude of x^* becomes 0.0 for the 20-qubit and 24-qubit runs. However, when we increase the rank limit to 40, the algorithm can be accurately simulated for a graphs with 2^{24} vertices.

Number of qubits	12	16	20	20	24	24
Rank limit r	2	2	2	5	5	20
Amplitude of x^* (9.80)	0.964	0.983	0.002	1.0	0.0	0.999

Table 9.7: Numerical experimental results for quantum walks on complete graphs with loops with CP-ALS. The number of initial guesses in CP-ALS is set as 3 for all the experiments.

Number of qubits	12	16	20	24	28	32
Amplitude of x^* (9.80)	0.964	0.983	0.998	0.999	1.0	1.0

Table 9.8: Numerical experimental results for quantum walks on complete graphs with loops with DESM. The rank limit (r) is set as 2 for all the experiments. All the experiments have 1.0 fidelity.

We also perform experiments to show that, for a general cyclic graph, a quantum walk based search algorithm is more difficult to simulate because the rank of the intermediate states increases rapidly and it is more difficult to reduce the rank of the intermediate states by using CP-ALS. We test on complete graphs (without loops) with one marked item x^*.

Number of qubits	8	12	16	20	20	24	24
Rank limit r	4	4	4	4	10	10	40
Amplitude of x^* (9.80)	0.781	0.897	0.942	0.0	0.988	0.0	0.998

Table 9.9: Numerical experimental results for quantum walks on complete bipartite graphs with CP-ALS. The number of initial guesses in CP-ALS is set as 3 for all the experiments.

Number of qubits	8	12	16	20	24	28	32	36
Amplitude of x^* (9.80)	0.781	0.897	0.942	0.988	0.998	1.0	1.0	1.0

Table 9.10: Numerical experimental results for quantum walks on complete bipartite graphs with DESM. The rank limit (r) is set as 4 for all the experiments. All the experiments have 1.0 fidelity.

The results are shown in Table 9.11. For all experiments, we use 3 different random CP-ALS initializations for each low-rank truncation routine. As can be seen in the table, as the number of qubits increases, the CP rank threshold necessarily needed to reach high simulation accuracy also increases exponentially. As a result, it becomes more difficult to simulate the quantum walk on larger graphs.

Although the low-rank simulation of this algorithm via CP-ALS is not accurate, CP decomposition requires lower memory cost compared to the naive state vector representation. As can be seen in Table 9.11, when the number of qubits is N, a CP rank of $2^{N/2+1}$ yields accurate results, and the memory cost is only $(2N) \cdot 2^{N/2+1}$.

Number of qubits	6	10	14
Rank	2^4	2^6	2^8
Fidelity estimation (9.33)	1.0	0.840	0.998
Amplitude of x^*	0.645	0.807	0.938

Table 9.11: Numerical experimental results for quantum walk on complete graphs with CP-ALS.

9.11 CONCLUSIONS

In this paper, we examined the possibility of using low-rank approximation via CP decomposition to simulate quantum algorithms on classical computers. The quantum algorithms we have considered include the quantum Fourier transform, phase estimation, Grover's algorithm and quantum walks.

For QFT, we have shown that all the intermediate states within the QFT quantum circuit and the output of the transform are rank-1 when the input is a standard (computational) basis. The same observation holds for the phase estimation algorithm, i.e., all the intermediate states in an phase estimation algorithm can be accurately approximated by a low-rank tensor. When the input to the QFT circuit is a general rank-1 tensor, the CP rank of the intermediate states can grow rapidly. Applying rank reduction in the simulation of the QFT can lead to loss of fidelity in the output.

For Grover's algorithms, we have shown that the CP ranks of all the intermediate states are bounded by $a + 1$, where a is the size of the marked set. Therefore, Grover's algorithm can, in principle, always be simulated efficiently by using low-rank CP decomposition when the size of the marked set is small.

For quantum walks, we have shown that the algorithm can be simulated efficiently on some graphs such as complete graphs with loops and complete bipartite graphs when the transition probability along edges of the graph is constant. We point out that it may be difficult to simulate a quantum walk defined on a more general graph, e.g., a general cyclic graph with non-uniform transition probabilities.

We presented two methods for performing rank reduction for intermediate tensors produced in the simulation of the quantum circuit. The CPD-ALS is a more general approach. However, it may suffer from numerical issues when the initial amplitudes associated with some of the terms in CP decomposition is significantly smaller than those associated with other terms. In this case, a method based on a direct elimination of scalar multiples is more effective.

Our numerical experimental results demonstrate that, by using CP decomposition and low rank representation/approximation, we can indeed simulate some quantum algorithms with a many-qubit input on a classical computer with high accuracy. Other quantum algorithms such as quantum walks on a more general graph with non-uniform transition probabilities are more difficult to simulate, because the CP rank of the intermediate tensors grows exponentially with respect to the system size (number of qubits), and low-rank approximations cannot maintain sufficient accuracy. This difficulty in fact demonstrates the real advantage or superiority of a quantum computer over a classical computer for solving certain classes of problems.

9.12 ADDITIONAL ANALYSIS FOR PHASE ESTIMATION

In Theorem 9.2 we will show that all the intermediate states on the first register of the phase estimation circuit can be approximated by a low-rank state whose CP rank is bounded by $\mathcal{O}(1/\epsilon)$, and the output state fidelity is at least $1 - \epsilon$. We look at the CP rank of the states on the first register rather than the overall state, because the rank of the overall state is highly dependent on both $|\psi\rangle$ and the oracle, hence is difficult to analyze.

Theorem 9.2. *For the phase estimation circuit, if $|\psi\rangle$ is the eigenvector of U, then all the intermediate states on the first register before the QFT^{-1} operator can be represented by a rank-1 state. In addition, all the intermediate states in the QFT^{-1} operator as well as its output state can be approximated by a low-rank state with the CP rank bounded by $\mathcal{O}(1/\epsilon)$, and the fidelity of all the intermediate and output states on the first register are at least $1-\epsilon$.*

Proof. Under the assumption that $|\psi\rangle$ is the eigenvector of U, the output state of each controlled-U^{2^j} gate will have the same rank as the input. For example, the output of controlled-$U^{2^{n-1}}$ will be

$$(E_1|h\rangle)\otimes\cdots\otimes(I|\psi\rangle)+(E_2|h\rangle)\otimes\cdots\otimes(U^{2^{n-1}}|\psi\rangle)=\frac{1}{\sqrt{2}}(|0\rangle+e^{i2\pi 2^{n-1}\theta}|1\rangle)\otimes|h\rangle\otimes\cdots\otimes|\psi\rangle, \tag{9.81}$$

and the first register state is

$$\frac{1}{\sqrt{2}}(|0\rangle+e^{i2\pi 2^{n-1}\theta}|1\rangle)\otimes|h\rangle\otimes\cdots\otimes|h\rangle, \tag{9.82}$$

which also has rank 1. Other output states of controlled gates behave the same way, and all the intermediate states on the first register before QFT^{-1} all have rank 1.

Next we analyze the output state of QFT^{-1}. We can rewrite the expression for the state expressed in (9.39) as

$$\frac{1}{2^n}\sum_{x=0}^{2^n-1}\sum_{k=0}^{2^n-1}e^{-\frac{2\pi ik}{2^n}(x-2^n\theta)}|x\rangle=\frac{1}{2^n}\sum_{x=0}^{2^n-1}\sum_{k=0}^{2^n-1}e^{-\frac{2\pi ik}{2^n}(x-a)}e^{2\pi i\delta k}|x\rangle, \tag{9.83}$$

where a is the nearest integer to $2^n\theta$, $2^n\theta=a+2^n\delta$ and $0\le|2^n\delta|\le\frac{1}{2}$. Let $\alpha^{(t)}$ denote the amplitude of $|a-t \mod 2^n\rangle$ and $-2^{n-1}\le t<2^{n-1}$, as is shown in reference [270], $\alpha^{(t)}$ can be expressed as

$$\alpha^{(t)}=\frac{1}{2^n}\frac{1-e^{2\pi i2^n(\delta+\frac{t}{2^n})}}{1-e^{2\pi i(\delta+\frac{t}{2^n})}}, \tag{9.84}$$

and the probability of outputting states that are at least k-away from $|a\rangle$ is bounded by

$$\sum_{k\le|t|\le 2^{n-1}}|\alpha^{(t)}|^2<\frac{1}{2k-1}. \tag{9.85}$$

For $\frac{1}{2k-1}\le\epsilon$, we need $k=\mathcal{O}(1/\epsilon)$. Therefore, with high fidelity $(1-\epsilon)$, the output state of the first register can be approximated with a state that is in the space of $\{|x\rangle, a-k\le x\le a+k\}$. The approximated state can be written as the linear combination of $2k=\mathcal{O}(1/\epsilon)$ standard basis states, and the CP rank is bounded by $\mathcal{O}(1/\epsilon)$.

Let $|\psi_a\rangle$, $|\psi_t\rangle$ denote the accurate output state, and the approximated low-rank output state of QFT^{-1}, respectively. Let $|\phi_a\rangle$ denote one accurate intermediate state of QFT^{-1}. We can express $|\psi_a\rangle = U|\phi_a\rangle$ for some unitary U. The fidelity of $|\psi_t\rangle$ can be expressed as

$$|\langle\psi_t|\psi_a\rangle|^2 = |\langle\psi_t|U|\phi_a\rangle|^2 = |\langle\phi_a|\phi_t\rangle|^2 \geq 1-\epsilon, \tag{9.86}$$

where $|\psi_t\rangle = U|\phi_t\rangle$. Above equation means that all the intermediate states $|\phi_a\rangle$ can be approximated by $|\phi_t\rangle$, and the fidelity will also be at least $1-\epsilon$. Based on Theorem 9.1 and the discussion in Section 9.6.1, all the approximated intermediate states $|\phi_t\rangle$ will have rank $\mathcal{O}(1/\epsilon)$. Therefore, all the intermediate states can be accurately approximated by low-rank states, and the theorem is proved. Q.E.D.

9.13 ADDITIONAL ANALYSIS FOR GROVER'S ALGORITHM

The rank of $U^{(g)}|\psi\rangle$ is dependent on the implementation of $U^{(g)}$, which is usually regarded as a black box. For the simulation to be efficient, $U^{(g)}|\psi\rangle$ needs to be in the CP representation, and the rank needs to be low. We will show in Theorem 9.3 that the rank of the state $U^{(g)}|\psi\rangle$ will be at most $2(a+1)$ times the rank of $|\psi\rangle$, thus the rank is low consider that a is small.

Theorem 9.3. *Consider an state $|\psi\rangle$ in the CP representation, and the rank is R. Then there exists implementations for $U^{(g)}$ such that it consists of $a+1$ generalized controlled gates, and the output state $U^{(g)}|\psi\rangle$ is also in the CP representation and the rank is at most $2(a+1)R$.*

Proof. We assume that $U^{(d)}$ is implemented similar to what is shown in Fig. 9.3, and $U^{(o)}$ is implemented with $|A| = a$ generalized controlled gates. Each generalized controlled gate flips the sign for one state $|x\rangle$ representing an marked item. Let $CU^{(x)}$ denote the controlled gate for $|x\rangle = |x_1x_2\cdots x_n\rangle$, we have

$$CU^{(x)} = I\otimes\cdots\otimes I - 2E_{x_1}\otimes\cdots\otimes E_{x_n}, \tag{9.87}$$

and $U^{(o)}$ can be expressed as

$$U^{(g)} = CU^{(y^{(1)})}CU^{(y^{(2)})}\cdots CU^{(y^{(a)})}, \tag{9.88}$$

where the marked set $A = \{y^{(1)}, \ldots, y^{(a)}\}$. (9.88) can be rewritten as

$$U^{(o)} = I \otimes \cdots \otimes I - \sum_{i=1}^{a} 2E_{y_1^{(i)}} \otimes \cdots \otimes E_{y_n^{(i)}}, \tag{9.89}$$

which contains the summation of $a+1$ Kronecker products. (9.89) holds since for two different standard basis u and v,

$$(E_{u_1} \otimes \cdots \otimes E_{u_n})(E_{v_1} \otimes \cdots \otimes E_{v_n}) = O. \tag{9.90}$$

Therefore, applying $U^{(o)}$ can output a state in the CP representation with the rank being $(a+1)R$. Applying $U^{(d)}$ then will at most double the rank. Therefore, the output of $U^{(g)}$ is in the CP representation and the rank is at most $2(a+1)R$. Q.E.D.

In Theorem 9.4, we will show that for any marked set that is small in size, we can approximate the intermediate states by rank-2 states, and the simulation will still output one marked state will high probability with $\mathcal{O}(\sqrt{N})$ Grover's iterations.

Theorem 9.4. *When $|A| = a$ is a constant and $a \ll N$, one marked state will be found with high probability if all the intermediate states are approximated by a low-rank state with CP rank upper bounded by 2, with the operator $U^{(g)}$ applied $\mathcal{O}(\sqrt{N})$ times.*

Proof. Let $|a\rangle$ denote one of the states in the marked set. We consider the case when all the intermediate approximate states are in the space S' spanned by $|a\rangle$ and $|h\rangle$, whose CP ranks are at most 2. Note that the input state $|h\rangle$ is also in the span. We will show that after applying operator $U^{(g)}$ for $\lfloor \frac{\pi}{4}\sqrt{N} \rfloor$ times, the output is close to $|a\rangle$.

Consider an input state $|x\rangle = \alpha|a\rangle + \beta|h\rangle$, where a is one of the target states in A. Then

$$U^{(g)}|a\rangle = (2|h\rangle\langle h| - I)(U^{(o)})|a\rangle = |a\rangle - 2|h\rangle\langle h|a\rangle = |a\rangle - \frac{2}{\sqrt{N}}|h\rangle, \tag{9.91}$$

$$U^{(g)}|h\rangle = U^{(g)}\left(\sqrt{\frac{a}{N}}|A\rangle + \sqrt{\frac{b}{N}}|B\rangle\right) = (1 - \frac{4a}{N})|h\rangle + 2\sqrt{\frac{a}{N}}|A\rangle, \tag{9.92}$$

where $U^{(g)}|a\rangle$ is in S' while $U^{(g)}|h\rangle$ is not. The low-rank approximation is performed on $U^{(g)}|h\rangle$, resulting in the (unnormalized) state $(1 - \frac{4a}{N})|h\rangle + 2\sqrt{\frac{1}{N}}|a\rangle$. The step of applying $U^{(g)}|x\rangle$ and then perform the low-rank approximation can be written as a linear transformation G', where

$$G' = \begin{bmatrix} 1 & 2\sqrt{\frac{1}{N}} \\ -2\sqrt{\frac{1}{N}} & 1 - \frac{4a}{N} \end{bmatrix}. \tag{9.93}$$

The first row is applied on the $|a\rangle$ component and the second row on $|h\rangle$. Let $|b\rangle = \frac{\sqrt{N}|h\rangle - |a\rangle}{\sqrt{N-1}}$ denoting the superposition of all the states except $|a\rangle$. we can rewrite G' based on the basis $|a\rangle, |b\rangle$ as follows:

$$G' = \begin{bmatrix} 1 & \frac{1}{N} \\ 0 & \frac{N-1}{N} \end{bmatrix} \begin{bmatrix} 1 & 2\sqrt{\frac{1}{N}} \\ -2\sqrt{\frac{1}{N}} & 1 - \frac{4a}{N} \end{bmatrix} \begin{bmatrix} 1 & -\frac{1}{N-1} \\ 0 & \frac{N}{N-1} \end{bmatrix}, \tag{9.94}$$

and the first row is applied on the $|a\rangle$ component and the second row on $|b\rangle$. Above operator can be rewritten as

$$\begin{bmatrix} 1 - \frac{2}{N} & 2\frac{\sqrt{N-1}}{N} \\ -2\frac{\sqrt{N-1}}{N} & 1 - \frac{2}{N} \end{bmatrix} + \begin{bmatrix} 0 & \frac{4-4a}{N}\sqrt{\frac{1}{N-1}} \\ 0 & \frac{4-4a}{N} \end{bmatrix}. \tag{9.95}$$

Note that the first component is equal to the operator $U^{(g)}$ for the Grover's algorithm when the marked set only has $|a\rangle$, and the second component comes from both $|A| > 1$ and the low-rank approximation. When $a \ll N$, it can be observed that the second component can be negligible, and the approximated algorithm will output $|a\rangle$ with high probability.

Q.E.D.

Because the approximated operator G' in (9.94) is close to the $U^{(g)}$ for the system with the marked set size equal to 1, the number of iterations necessary to get a marked state with high probability increases from $\lfloor \frac{\pi}{4}\sqrt{\frac{N}{a}} \rfloor$ to $\lfloor \frac{\pi}{4}\sqrt{N} \rfloor$. For the system when a is unknown, $\lfloor \frac{\pi}{4}\sqrt{N} \rfloor$ iterations need to be performed for both.

Chapter 10: TENSOR RANK UPPER BOUNDS OF GRAPH STATES

In this Chapter, we provide improved upper bounds on the CP decomposition rank of quantum states that are defined on ring graphs with an odd number of vertices.

Entanglement is one of the defining properties of quantum systems [291] and has been recognized as a fundamental resource for quantum information processing [292], [293]. A pure state is considered to be entangled if it cannot be written in the form $|\psi\rangle = \bigotimes_{i=1}^{n} |\psi^{(i)}\rangle$. Similarly, a mixed state is entangled if it cannot be written as $\rho = \sum_k p_k \bigotimes_{i=1}^{n} \rho_k^{(i)}$.

Quantifying the amount of entanglement in a quantum state is not always straightforward. For pure bipartite systems, the Schmidt decomposition and resulting spectra fully characterize the entanglement properties and transformations under local operations and classical communication (LOCC) [292], [294], [295]. The Schmidt decomposition of a pure state takes the form $|\psi\rangle = \sum_i \sqrt{\mu_i}\, |u_i\rangle\, |v_i\rangle$, where $\mu = \{\mu_i\}$ are the Schmidt coefficients and $\{|u_i\rangle\}$ and $\{|v_i\rangle\}$ are sets of orthogonal states. Further, a variety of entanglement measures are known, i.e. functionals $E(\rho)$ that are non-increasing (on average) under LOCC and $E(\rho) = 0$ if ρ is a separable state [296]. Examples include the entanglement of formation [297], distillable entanglement [298], negativity [299], [300], geometric measure [301], and concurrence [302], [303]. However, the picture grows significantly more complicated when considering multipartite entanglement, as we discuss below.

We consider the amount and form of multipartite entanglement that arises in a class of quantum states known as graph states. These are of particular interest due to their application in measurement based quantum computing [304], [305], error correction codes [306], secret sharing [307], and stabilizer computation simulation [308]. Further, by studying the entanglement properties of graph states, we actually quantify the entanglement of the larger set of stabilizer states. This follows from the fact that every stabilizer state is equivalent under local unitaries to at least one graph state [309], [310]. As entanglement measures are invariant under local unitaries, one thus need only consider graph states to analyze all stabilizer states.

We consider ring states of $2n+1$ qubits and sharpen the bound on tensor rank from $2^n \leq \text{rank}(|R_{2n+1}\rangle) \leq 2^{n+1}$ to $2^n \leq \text{rank}(|R_{2n+1}\rangle) \leq 3 \cdot 2^{n-1}$. The lower bound has been further improved from 2^n to 2^n+1 in [311]. While this may seem like incremental progress, we stress that computing the tensor rank is a very challenging problem, and any progress in this direction is noteworthy. Indeed, the analysis we employ goes beyond the bipartite bounding techniques of previous approaches. This work thus contributes to the steadily growing research on the tensor rank of multipartite entangled states [312]–[320]. Operationally, the improved bounds help better characterize the amount of entanglement needed to generate ring states using LOCC.

10.1 BACKGROUNDS

10.1.1 Schmidt Measure and Tensor Decomposition

Any N-party pure state $|\psi\rangle \in \mathcal{H}^{(1)} \otimes \cdots \otimes \mathcal{H}^{(N)}$ can be represented as

$$|\psi\rangle = \sum_{i=1}^{R} \mu_i |\psi_i^{(1)}\rangle \otimes \cdots \otimes |\psi_i^{(N)}\rangle , \tag{10.1}$$

where each $|\psi_i^{(j)}\rangle \in \mathcal{H}^{(j)}$. When $|\psi\rangle$ is viewed as an N-dimensional tensor, Eq. (10.1) is also known as a canonical polyadic (CP) tensor decomposition [7], [13]. The CP rank $r = \text{rank}(|\psi\rangle)$ of a tensor is defined as the smallest R such that (10.1) can be satisfied. The CP rank is also known as the tensor rank, and we will use both types of terminology throughout this paper. In general, finding the CP rank of a tensor is NP-hard [15].

Different from the matrix case, for tensors the best rank-R approximation may not exist. And there exists tensors that can be approximated arbitrarily well by rank-R tensors where $R < \text{rank}(|\psi\rangle)$. In this case, *border rank* [321], [322] is defined as the minimum number of rank-one tensors that are sufficient to approximate the given tensor with arbitrarily small error.

The tensor rank is a *bona fide* entanglement measure [312] that is particularly useful studying state transformations under stochastic local operations and classical communication (SLOCC). These are transformations such that $|\psi\rangle \xrightarrow{SLOCC} |\phi\rangle$ with some non-zero probability (and is thus a generalization of LOCC). It is known that if $|\psi\rangle \xrightarrow{SLOCC} |\phi\rangle$, then $\text{rank}\,|\psi\rangle \geq \text{rank}\,|\phi\rangle$ [313]. Note that we can characterize SLOCC equivalence (i.e. $|\psi\rangle \xrightarrow{SLOCC} |\phi\rangle$ and $|\phi\rangle \xrightarrow{SLOCC} |\psi\rangle$) via invertible operators:

$$|\psi\rangle = A_1 \otimes A_2 \otimes \ldots \otimes A_n |\phi\rangle , \tag{10.2}$$

implying that $\text{rank}(|\psi\rangle) = \text{rank}(|\phi\rangle)$. Lastly, we note that the tensor rank relates to entanglement cost. In particular, a generalized d-dimensional GHZ state (or any equivalent state) can be converted to an arbitrary state $|\psi\rangle$ iff $d \geq \text{rank}(|\psi\rangle)$ via SLOCC [314]. This provides an operational meaning to the tensor rank in terms of the entanglement resources needed to build $|\psi\rangle$ using GHZ states in the distributed setting.

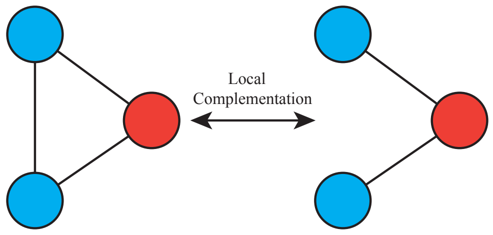


Figure 10.1: Example of local complementation. Here the rule is applied to the red vertex, adding or removing the edge connecting the two other vertices.

10.1.2 Graph States

Graph states are quantum states corresponding to some graph $G = (V, E)$, where V is the vertex set and E is the edge set with corresponding adjacency matrix Γ [304], [323]. There are two equivalent ways to think of graph states. The first is operational in the sense that it provides a formula for preparing the state given a graph:

$$|G\rangle = \prod_{(a,b)\in E} U^{(a,b)} |+\rangle^{\otimes |V|}, \tag{10.3}$$

where

$$U^{(a,b)} = |0\rangle \langle 0|^{(a)} \otimes \mathbb{I}^{(b)} + |1\rangle \langle 1|^{(a)} \otimes \sigma_z^{(b)} \tag{10.4}$$

is a controlled Z operation on qubits a and b, and

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{10.5}$$

forms the Hadamard basis. Thus, given a graph, the graph state initialize $|V|$ qubits in the state $|+\rangle^{\otimes |V|}$ and, for each edge, apply a controlled Z between the corresponding qubits.

Graph states can be equivalently thought of as stabilizer states [92]. Here the stabilizers are $S_a = \sigma_x^{(a)} \prod_{b\in N_a} \sigma_z^{(b)}$, where N_a is the neighborhood of vertex a. As there are $|V|$ qubits and stabilizers, $|G\rangle$ is the unique state stabilized by all S_a.

Also note that a basis for $\mathcal{H} = \bigotimes_{i=1}^{n} \mathcal{H}_2^{(i)}$ can be constructed given a graph G [305]:

$$|G_{\mathbf{s}}\rangle = \sigma_z^{\mathbf{s}} |G\rangle = \prod_{(a,b)\in E} U^{(a,b)} \bigotimes_{i=1}^{n} (\sigma_z^{(i)})^{s_i} |+\rangle^{(i)}. \tag{10.6}$$

It is clear that there are 2^n such orthogonal states and thus they form a basis. Further, one can think of $\mathbf{s}$ as flipping the eigenvalues of stabilizers S_a from $+1$ to -1. Going forward, we will denote these as graph basis states. We will later use a result from [92] that the partial

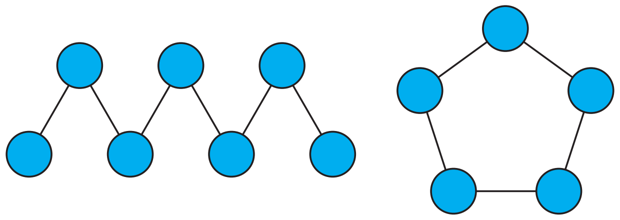


Figure 10.2: Line and ring graphs. The left graph is a line (one dimensional cluster state) on 7 qubits, which we denote by $|L_7\rangle$. The right graph is an odd ring on 5 qubits, which we denote by $\mathbb{R}5$.

trace of a graph state can be expressed in the graph basis:

$$\underset{A}{\mathbf{Tr}}[|G\rangle\langle G|] = \frac{1}{2^{|A|}} \sum_{\mathbf{z}\in\mathbb{F}_2^{|A|}} U(\mathbf{z})\,|G-A\rangle\langle G-A|\,U(\mathbf{z})^{\dagger}, \tag{10.7}$$

where $U(\mathbf{z}) = \prod_{a\in A}(\prod_{b\in N_a}\sigma_z^{(b)})^{z_a}$, and $|G-A\rangle$ denotes the state corresponding to deleting all vertices in A from G. Further analysis of this state leads to the useful structural fact.

Lemma 10.1 ([92]). *The states $U(\mathbf{z})\,|G-A\rangle$ satisfy the orthogonality condition*

$$\langle G-A|\,U^{\dagger}(\mathbf{z}')U(\mathbf{z})\,|G-A\rangle = \begin{cases} 0 \text{ if } \mathbf{z}-\mathbf{z}' \in \ker(\Gamma_{A\overline{A}}) \\ 1 \text{ if } \mathbf{z}-\mathbf{z}' \notin \ker(\Gamma_{A\overline{A}}) \end{cases}, \tag{10.8}$$

where $\Gamma_{A\overline{A}}$ is the submatrix of Γ_G restricted to edges from A to $\overline{A}$. Hence, $\rho^{(\overline{A})} = \mathbf{Tr}_A\,|G\rangle\langle G|$ is maximally mixed over a subspace of dimension 2^d, where $d = \operatorname{rank}(\Gamma_{A\overline{A}})$.

10.1.3 Existing Tensor Rank Bounds for Graph States

Here we briefly review existing results on graph state CP rank/Schmidt measure. From [92] we have that

$$\operatorname{rank}(|\psi\rangle) \geq 2^{(\operatorname{rank}\Gamma_{A\overline{A}})/2}, \tag{10.9}$$

where $\Gamma_{A\overline{A}}$ is the subset of the adjacency matrix restricted to edges from $\overline{A}$ to A. The authors also give a general case upper bound

$$\operatorname{rank}(|\psi\rangle) \leq 2^{\tau(G)}, \tag{10.10}$$

where $\tau(G)$ is the size of the smallest vertex cover of G.

While these bounds may not be tight, it is often possible to use complementation rules to find locally equivalent graphs for which these bounds improve. It is known that the full orbit of any graph state under local clifford operations can be found via local complementations [92], [309]. That is, for some vertex $a \in V$, complement the subgraph given by the neighborhood N_a (Fig. 10.1). These rules have been used to classify all graph states of up to 8 qubits [92], [324], [325]. Further, classes of two-colorable graphs corresponding to states of maximal schmidt measure are known [326]. However, odd rings, corresponding to non two-colorable graphs, lead to loose bounds.

Line states (Fig. 10.2), also known as one-dimensional cluster states, are those with one-dimensional nearest neighbor connections. We will write $|L_n\rangle$ to denote a line state on n qubits. We will find line states to be useful in proving an upper bound on the rank of ring states. An explicit construction of a minimal CP decomposition of line states is given in the appendix.

Lemma 10.2.

$$\text{rank}(|L_n\rangle) = 2^{\lfloor \frac{n}{2} \rfloor}. \tag{10.11}$$

Proof. This readily follows from the mentioned graph theoretic tools. See [92] for details. Q.E.D.

For any even ring $|R_{2n}\rangle$, it's known that the lower bound equals the upper bound, thus the CP rank is 2^n [92]. For any odd ring $|R_{2n+1}\rangle$, it is known that $2^n \leq \text{rank}(|R_{2n+1}\rangle) \leq 2^{n+1}$, coming from the rank of the adjacency matrix and minimal vertex cover. Any tightening of these bounds will therefore require a new type of analysis not based on the latter graph-theoretic concepts.

10.2 THE TENSOR RANK UPPER BOUND OF RING STATES

In this section we provide a CP rank upper bound of $3 \cdot 2^{n-1}$ for odd ring graph states $|R_{2n+1}\rangle$. Throughout the proof, we let

$$P_0 = |0\rangle \langle 0|, P_1 = |1\rangle \langle 1|, \tag{10.12}$$

and let $U^{(a,b)}$ denote a controlled Z operation where the ath qubit is the controlling qubit and the bth qubit is the controlled qubit. We have

$$U^{(a,b)} = P_0^{(a)} \otimes I^{(b)} + P_1^{(a)} \otimes \sigma_z^{(b)} = I^{(a)} \otimes \sigma_z^{(b)} + 2P_0^{(a)} \otimes P_1^{(b)}. \tag{10.13}$$

Below we show the main statement.

Theorem 10.1 (CP rank upper bound for odd ring graph states)**.** *The CP rank of any odd ring state $|R_{2n+1}\rangle$ is upper bounded by*

$$\text{rank}(|R_{2n+1}\rangle) \leq 3 \cdot 2^{n-1}. \tag{10.14}$$

Proof. For the case with $n = 1$, we can easily verify that

$$|R_3\rangle = |++-\rangle + \frac{1}{\sqrt{2}}|001\rangle - \frac{1}{\sqrt{2}}|110\rangle, \tag{10.15}$$

thus satisfying the upper bound. Below we show the cases with $n \geq 2$.

Based on (10.13) and the fact that

$$|R_{2n+1}\rangle = U^{(1,2n+1)}|L_{2n+1}\rangle, \tag{10.16}$$

we have

$$|R_{2n+1}\rangle = U^{(1,2n+1)}|L_{2n+1}\rangle = I^{(1)} \otimes \sigma_z^{(2n+1)}|L_{2n+1}\rangle + 2P_0^{(1)} \otimes P_1^{(2n+1)}|L_{2n+1}\rangle. \tag{10.17}$$

The CP rank of the term $I^{(1)} \otimes \sigma_z^{(2n+1)}|L_{2n+1}\rangle$ is 2^n since the CP rank of $|L_{2n+1}\rangle$ is 2^n. Define the state

$$|\phi_{2n+1}\rangle = P_0^{(1)}|L_{2n+1}\rangle. \tag{10.18}$$

Based on Lemma 10.3 and Lemma 10.4 below, the CP rank of the term $P_1^{(2n+1)}|\phi_{2n+1}\rangle$ for all integers $n \geq 2$ is upper-bounded by 2^{n-1}, thus proving the statement. Q.E.D.

Below we present Lemma 10.3 and Lemma 10.4, which upper-bound the CP rank of $P_0^{(2n+1)}|\phi_{2n+1}\rangle$ and $P_1^{(2n+1)}|\phi_{2n+1}\rangle$ for all integers $n \geq 2$. In our analysis below, we define a generalized controlled gate

$$CZZ^{(i,j,k)} := U^{(i,j)}U^{(i,k)} = P_0^{(i)} \otimes I^{(j)} \otimes I^{(k)} + P_1^{(i)} \otimes \sigma_z^{(j)} \otimes \sigma_z^{(k)}, \tag{10.19}$$

whose CP rank is also 2. The line state $|L_{2n+1}\rangle$ can be expressed as

$$|L_{2n+1}\rangle = \prod_{i=1}^{n} U^{(2i,2i-1)}U^{(2i,2i+1)}|+\rangle^{(1,\ldots,2n+1)} = \prod_{i=1}^{n} CZZ^{(2i,2i-1,2i+1)}|+\rangle^{(1,\ldots,2n+1)}. \tag{10.20}$$

Lemma 10.3. *When $n = 2$, the ranks of both $P_0^{(2n+1)} |\phi_{2n+1}\rangle$ and $P_1^{(2n+1)} |\phi_{2n+1}\rangle$ with $|\phi_{2n+1}\rangle$ defined in (10.18) are bounded by 2.*

Proof. For $n = 2$,

$$\begin{aligned}
&|L_{2n+1}\rangle = |L_5\rangle \\
&= CZZ^{(4,3,5)} CZZ^{(2,1,3)} |+\rangle^{(1,\ldots,5)} \\
&= (I \otimes P_0 \otimes I \otimes P_0 \otimes I + I \otimes P_0 \otimes \sigma_z \otimes P_1 \otimes \sigma_z) |+\rangle^{(1,\ldots,5)} \\
&+ (\sigma_z \otimes P_1 \otimes \sigma_z \otimes P_0 \otimes I + \sigma_z \otimes P_1 \otimes I \otimes P_1 \otimes \sigma_z) |+\rangle^{(1,\ldots,5)} \\
&= \frac{1}{2} |{+0+0+}\rangle + \frac{1}{2} |{+0-1-}\rangle + \frac{1}{2} |{-1-0+}\rangle + \frac{1}{2} |{-1+1-}\rangle ,
\end{aligned} \tag{10.21}$$

thus we have

$$\begin{aligned}
&|\phi_5\rangle = P_0^{(1)} |L_5\rangle \\
&= \frac{1}{2\sqrt{2}} |0\rangle \Big(|0+0+\rangle + |0-1-\rangle + |1-0+\rangle + |1+1-\rangle \Big) \\
&= \underbrace{\frac{1}{4} |0\rangle \Big(|0+0\rangle + |0-1\rangle + |1-0\rangle + |1+1\rangle \Big) |0\rangle}_{P_0^{(5)}|\phi_5\rangle} \\
&+ \underbrace{\frac{1}{4} |0\rangle \Big(|0+0\rangle - |0-1\rangle + |1-0\rangle - |1+1\rangle \Big) |1\rangle}_{P_1^{(5)}|\phi_5\rangle} .
\end{aligned} \tag{10.22}$$

Above expressions for $P_0^{(5)} |\phi_5\rangle$ and $P_1^{(5)} |\phi_5\rangle$ can be rewritten as follows,

$$P_0^{(5)} |\phi_5\rangle = \frac{1}{2\sqrt{2}} |0\rangle \Big(|{+0+}\rangle + |{-1-}\rangle \Big) |0\rangle , \quad P_1^{(5)} |\phi_5\rangle = \frac{1}{2\sqrt{2}} |0\rangle \Big(|{+0-}\rangle + |{-1+}\rangle \Big) |1\rangle , \tag{10.23}$$

thus the CP ranks are bounded by 2. Q.E.D.

Lemma 10.4. *When $n \geq 2$, the CP ranks of both states $P_0^{(2n+1)} |\phi_{2n+1}\rangle$ and $P_1^{(2n+1)} |\phi_{2n+1}\rangle$ are bounded by 2^{n-1}.*

Proof. We argue by induction on n. Assume that the ranks of both $P_0^{(2n+1)} |\phi_{2n+1}\rangle$ and $P_1^{(2n+1)} |\phi_{2n+1}\rangle$ are bounded by 2^{n-1}. We will show that the CP ranks of both states $P_0^{(2n+3)} |\phi_{2n+3}\rangle$ and $P_1^{(2n+3)} |\phi_{2n+3}\rangle$ are bounded by 2^n.

$|\phi_{2n+3}\rangle$ can be rewritten as follows,

$$\begin{aligned}
&|\phi_{2n+3}\rangle = P_0^{(1)}\,|L_{2n+3}\rangle \\
&= P_0^{(1)} CZZ^{(2n+2,2n+1,2n+3)}\,|L_{2n+1}\rangle\,|{+}{+}\rangle = CZZ^{(2n+2,2n+1,2n+3)} P_0^{(1)}\,|L_{2n+1}\rangle\,|{+}{+}\rangle \\
&= CZZ^{(2n+2,2n+1,2n+3)} P_0^{(2n+1)}\,|\phi_{2n+1}\rangle\,|{+}{+}\rangle + CZZ^{(2n+2,2n+1,2n+3)} P_1^{(2n+1)}\,|\phi_{2n+1}\rangle\,|{+}{+}\rangle \\
&= \frac{1}{\sqrt{2}} P_0^{(2n+1)}\,|\phi_{2n+1}\rangle \Big(|0{+}\rangle + |1{-}\rangle \Big) + \frac{1}{\sqrt{2}} P_1^{(2n+1)}\,|\phi_{2n+1}\rangle \Big(|0{+}\rangle - |1{-}\rangle \Big).
\end{aligned} \tag{10.24}$$

Note that the third equality comes from the commutativity of $CZZ^{(2n+2,2n+1,2n+3)}, P_0^{(1)}$. Based on the transformation

$$|0{+}\rangle + |1{-}\rangle = |{+}0\rangle + |{-}1\rangle\,, \quad |0{+}\rangle - |1{-}\rangle = |{+}1\rangle + |{-}0\rangle\,, \tag{10.25}$$

(10.24) can be rewritten as

$$\begin{aligned}
&|\phi_{2n+3}\rangle = \frac{1}{\sqrt{2}} P_0^{(2n+1)}\,|\phi_{2n+1}\rangle \Big(|{+}0\rangle + |{-}1\rangle \Big) + \frac{1}{\sqrt{2}} P_1^{(2n+1)}\,|\phi_{2n+1}\rangle \Big(|{+}1\rangle + |{-}0\rangle \Big) \\
&= \underbrace{\frac{1}{\sqrt{2}} \Big(P_0^{(2n+1)}\,|\phi_{2n+1}\rangle\,|{+}\rangle + P_1^{(2n+1)}\,|\phi_{2n+1}\rangle\,|{-}\rangle \Big)\,|0\rangle}_{P_0^{(2n+3)}|\phi_{2n+3}\rangle} \\
&+ \underbrace{\frac{1}{\sqrt{2}} \Big(P_0^{(2n+1)}\,|\phi_{2n+1}\rangle\,|{-}\rangle + P_1^{(2n+1)}\,|\phi_{2n+1}\rangle\,|{+}\rangle \Big)\,|1\rangle}_{P_1^{(2n+3)}|\phi_{2n+3}\rangle}.
\end{aligned} \tag{10.26}$$

It can be easily seen that the CP ranks of both states $P_0^{(2n+3)}\,|\phi_{2n+3}\rangle$ and $P_1^{(2n+3)}\,|\phi_{2n+3}\rangle$ are bounded by 2^n. Since the rank upper bounds for the base case ($n = 2$) has been shown in Lemma 10.3, the lemma is proved. Q.E.D.

Part V

CONCLUSION

Chapter 11: CONCLUSION AND FUTURE WORK

In this thesis, we present a set of computationally efficient numerical algorithms and computer systems designed for tensor decompositions and problems involving tensor networks. We introduce a range of efficient algorithms tailored for various scenarios in tensor decompositions. These scenarios include dense and sparse input tensors, as well as low-rank and high-rank decompositions. Additionally, we introduce new methods for approximate tensor network contractions, incorporating novel contributions in contraction path selection, cost-efficient low-rank approximation algorithms, and flexible environment utilization. Our novel systems automate the algorithmic advancements for both tensor decompositions and tensor network contractions. To summarize, our contributions significantly accelerate tensor computations in diverse fields such as data science, quantum chemistry, computational physics, and quantum computing.

Throughout the preceding Chapters, numerous open problems have been identified and discussed. Here, we further emphasize several potential future directions that need further exploration and investigation.

1. Sketching with sparse tensor network embeddings. Chapter 6 proposes a general framework for sketching tensor network data with Gaussian tensor network embeddings. However, each Gaussian random tensor within the embedding is dense, which may lead to inefficiency when sketching sparse tensors or tensor networks. As a potential future direction, it is worth exploring the generalization of the analysis presented in Chapter 6 to include other sparse embeddings. By doing so, the algorithm can be extended to efficiently sketch sparse tensors and tensor networks.

2. Leveraging sketching techniques for approximate tensor network contractions. As is described in Chapters 7 and 8, the low-rank tensor network approximation is the bottleneck for complexity. One future direction is to explore the possibility to use the tensor network sketching technique to accelerate low-rank approximation. In addition, it is of interest to compare its performance with the current canonicalization-based algorithm and the density matrix algorithm.

3. Deriving efficient contraction paths for CATN-GO without the uniform rank assumption. In Section 7.8, we present an algorithm that uses dynamic programming as well as the cut analysis to derive the contraction path that minimizes the computational cost of the CATN-GO algorithm. However, this algorithm has a high complexity of $O(n^3m^3)$, where n is the number of vertices and m is the number of edges in the graph. It is worth

investigating whether the complexity can be reduced, and whether the contraction path derived without the uniform rank assumption can further accelerate the CATN-GO algorithm.

4. Selecting efficient partial contraction paths for Partitioned Contract based on the algorithms in CATN-GO. The Partitioned Contract algorithm in Chapter 8 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 whether the strategies proposed in CATN-GO in Chapter 7 can also be used to find efficient partial contraction paths for Partitioned Contract.

REFERENCES

[1] F. Cong, Q.-H. Lin, L.-D. Kuang, X.-F. Gong, P. Astikainen, and T. Ristaniemi, "Tensor decomposition of EEG signals: A brief review," *Journal of neuroscience methods*, vol. 248, pp. 59–69, 2015.

[2] J. Pearl, "Bayesian networks: A model of self-activated memory for evidential reasoning," in *Proceedings of the 7th conference of the Cognitive Science Society, University of California, Irvine, CA, USA*, 1985, pp. 15–17.

[3] J. C. Bridgeman and C. T. Chubb, "Hand-waving and interpretive dance: An introductory course on tensor networks," *Journal of Physics A: Mathematical and Theoretical*, vol. 50, no. 22, p. 223 001, 2017.

[4] R. Orús, "A practical introduction to tensor networks: Matrix product states and projected entangled pair states," *Annals of Physics*, vol. 349, pp. 117–158, 2014.

[5] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[6] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[7] R. A. Harshman, "Foundations of the PARAFAC procedure: Models and conditions for an explanatory multimodal factor analysis," 1970.

[8] H. A. Kiers, "Towards a standardized notation and terminology in multiway analysis," *Journal of Chemometrics: A Journal of the Chemometrics Society*, vol. 14, no. 3, pp. 105–122, 2000.

[9] J. Mocks, "Topographic components model for event-related potentials and some biophysical considerations," *IEEE transactions on biomedical engineering*, vol. 35, no. 6, pp. 482–484, 1988.

[10] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[11] D. Perez-Garcia, F. Verstraete, M. Wolf, and J. Cirac, "Matrix product state representations," *Quantum Information & Computation*, vol. 7, no. 5, pp. 401–430, 2007.

[12] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[13] F. L. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *Studies in Applied Mathematics*, vol. 6, no. 1-4, pp. 164–189, 1927.

[14] J. B. Kruskal, "Three-way arrays: Rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics," *Linear algebra and its applications*, vol. 18, no. 2, pp. 95–138, 1977.

[15] J. Håstad, "Tensor rank is NP-complete," in *International Colloquium on Automata, Languages, and Programming*, Springer, 1989, pp. 451–460.

[16] A. Cichocki, "Tensor networks for big data analytics and large-scale optimization problems," *arXiv preprint arXiv:1407.3124*, 2014.

[17] E. Stoudenmire and D. J. Schwab, "Supervised learning with tensor networks," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[18] J. Reyes and E. Stoudenmire, "Multi-scale tensor network architecture for machine learning," *Machine Learning: Science and Technology*, vol. 2, no. 3, p. 035 036, 2021.

[19] J. Li, Y. Sun, J. Su, T. Suzuki, and F. Huang, "Understanding generalization in deep learning via tensor methods," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2020, pp. 504–515.

[20] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582,

[21] Y. Panagakis, J. Kossaifi, G. G. Chrysos, *et al.*, "Tensor methods in computer vision and deep learning," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 863–890, 2021.

[22] E. Acar and B. Yener, "Unsupervised multiway data analysis: A literature survey," *IEEE transactions on knowledge and data engineering*, vol. 21, no. 1, pp. 6–20, 2008.

[23] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *Journal of Machine Learning Research*, vol. 15, pp. 2773–2832, 2014.

[24] E. G. Hohenstein, R. M. Parrish, and T. J. Martinez, "Tensor hypercontraction density fitting. I. quartic scaling second-and third-order Møller-Plesset perturbation theory," *The Journal of chemical physics*, vol. 137, no. 4, p. 044 103, 2012.

[25] F. Hummel, T. Tsatsoulis, and A. Grüneis, "Low rank factorization of the Coulomb integrals for periodic coupled cluster theory," *The Journal of chemical physics*, vol. 146, no. 12, p. 124 105, 2017.

[26] E. G. Hohenstein, R. M. Parrish, C. D. Sherrill, and T. J. Martinez, "Communication: Tensor hypercontraction. III. least-squares tensor hypercontraction for the determination of correlated wavefunctions," *J Chem Phys.*, vol. 137, no. 22, p. 221 101, 2012.

[27] N. J. Mayhall, "Using higher-order singular value decomposition to define weakly coupled and strongly correlated clusters: The n-body tucker approximation," *Journal of Chemical Theory and Computation*, vol. 13, no. 10, pp. 4818–4828, 2017.

[28] C. J. Hillar and L.-H. Lim, "Most tensor problems are NP-hard," *Journal of the ACM (JACM)*, vol. 60, no. 6, pp. 1–39, 2013.

[29] C. A. Andersson and R. Bro, "Improving the speed of multi-way algorithms: Part I. Tucker3," *Chemometrics and intelligent laboratory systems*, vol. 42, no. 1-2, pp. 93–103, 1998.

[30] L. De Lathauwer, B. De Moor, and J. Vandewalle, "On the best rank-1 and rank-$(r_1, r_2, \ldots, r_n)$ approximation of higher-order tensors," *SIAM journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1324–1342, 2000.

[31] S. Holtz, T. Rohwedder, and R. Schneider, "The alternating linear scheme for tensor optimization in the tensor train format," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, A683–A713, 2012.

[32] G. Vidal, "Efficient classical simulation of slightly entangled quantum computations," *Physical review letters*, vol. 91, no. 14, p. 147 902, 2003.

[33] 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*, vol. 57, no. 2, pp. 143–224, 2008.

[34] S. R. White, "Density matrix formulation for quantum renormalization groups," *Physical review letters*, vol. 69, no. 19, p. 2863, 1992.

[35] F. Verstraete and J. I. Cirac, "Renormalization algorithms for quantum-many body systems in two and higher dimensions," *arXiv preprint cond-mat/0407066*, 2004.

[36] Y.-Y. Shi, L.-M. Duan, and G. Vidal, "Classical simulation of quantum many-body systems with a tree tensor network," *Physical review A*, vol. 74, no. 2, p. 022 320, 2006.

[37] U. Schollwöck, "The density-matrix renormalization group," *Reviews of modern physics*, vol. 77, no. 1, p. 259, 2005.

[38] L. Ma and C. Yang, "Low rank approximation in simulations of quantum algorithms," *Journal of Computational Science*, p. 101 561, 2022.

[39] Y. Zhou, E. M. Stoudenmire, and X. Waintal, "What limits the simulation of quantum computers?" *Physical Review X*, vol. 10, no. 4, p. 041 038, 2020.

[40] Y. Pang, T. Hao, A. Dugad, Y. Zhou, and E. Solomonik, "Efficient 2D tensor network simulation of quantum systems," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2020, pp. 1–14.

[41] F. Pan, P. Zhou, S. Li, and P. Zhang, "Contracting arbitrary tensor networks: General approximate algorithm and applications in graphical models and quantum circuit simulations," *Physical Review Letters*, vol. 125, no. 6, p. 060 503, 2020.

[42] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Advances in neural information processing systems*, 2015, pp. 442–450.

[43] N. Pham and R. Pagh, "Fast and scalable polynomial kernels via explicit feature maps," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 239–247.

[44] T. D. Ahle, M. Kapralov, J. B. Knudsen, *et al.*, "Oblivious sketching of high-degree polynomial kernels," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2020, pp. 141–160.

[45] D. Woodruff and A. Zandieh, "Leverage score sampling for tensor product matrices in input sparsity time," in *International Conference on Machine Learning*, PMLR, 2022, pp. 23 933–23 964.

[46] M. Meister, T. Sarlos, and D. Woodruff, "Tight dimensionality reduction for sketching low degree polynomial kernels," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[47] S. Paeckel, T. Köhler, A. Swoboda, S. R. Manmana, U. Schollwöck, and C. Hubig, "Time-evolution methods for matrix-product states," *Annals of Physics*, vol. 411, p. 167 998, 2019.

[48] I. P. McCulloch, "From density-matrix renormalization group to matrix product states," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2007, no. 10, P10014, 2007.

[49] L. De Lathauwer, B. De Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1253–1278, 2000.

[50] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen, "A new truncation strategy for the higher-order singular value decomposition," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, A1027–A1052, 2012.

[51] W. Hackbusch, *Tensor spaces and numerical tensor calculus*. Springer Science & Business Media, 2012, vol. 42.

[52] M. Levin and C. P. Nave, "Tensor renormalization group approach to two-dimensional classical lattice models," *Physical review letters*, vol. 99, no. 12, p. 120 601, 2007.

[53] S. Kourtis, C. Chamon, E. Mucciolo, and A. Ruckenstein, "Fast counting with tensor networks," *SciPost Physics*, vol. 7, no. 5, p. 060, 2019.

[54] C. Damm, M. Holzer, and P. McKenzie, "The complexity of tensor calculus," *computational complexity*, vol. 11, no. 1-2, pp. 54–89, 2002.

[55] B. O'Gorman, "Parameterization of tensor network contraction," in *14th Conference on the Theory of Quantum Computation, Communication and Cryptography*, 2019.

[56] J. D. Biamonte, J. Morton, and J. Turner, "Tensor network contractions for # SAT," *Journal of Statistical Physics*, vol. 160, no. 5, pp. 1389–1404, 2015.

[57] R. Orús, "Tensor networks for complex quantum systems," *Nature Reviews Physics*, vol. 1, no. 9, pp. 538–550, 2019.

[58] O. Kaya and Y. Robert, "Computing dense tensor decompositions with optimal dimension trees," *Algorithmica*, vol. 81, no. 5, pp. 2092–2121, 2019.

[59] A.-H. Phan, P. Tichavskỳ, and A. Cichocki, "Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations," *IEEE Transactions on Signal Processing*, vol. 61, no. 19, pp. 4834–4846, 2013.

[60] D. P. Woodruff, "Sketching as a tool for numerical linear algebra," *Theoretical Computer Science*, vol. 10, no. 1-2, pp. 1–157, 2014.

[61] V. Strassen *et al.*, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[62] G. Ballard, K. Hayashi, and K. Ramakrishnan, "Parallel nonnegative CP decomposition of dense tensors," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, IEEE, 2018, pp. 22–31.

[63] K. Hayashi, G. Ballard, J. Jiang, and M. Tobia, "Shared memory parallelization of MTTKRP for dense tensors," *arXiv preprint arXiv:1708.08976*, 2017.

[64] A. P. Liavas, G. Kostoulas, G. Lourakis, K. Huang, and N. D. Sidiropoulos, "Nesterov-based parallel algorithm for large-scale nonnegative tensor factorization," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 5895–5899.

[65] I. Nisa, J. Li, A. Sukumaran-Rajam, R. Vuduc, and P. Sadayappan, "Load-balanced sparse MTTKRP on GPUs," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2019, pp. 123–133.

[66] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2015, pp. 61–70.

[67] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, "Model-driven sparse CP decomposition for higher-order tensors," in *2017 IEEE international parallel and distributed processing symposium (IPDPS)*, IEEE, 2017, pp. 1048–1057.

[68] S. Smith and G. Karypis, "A medium-grained algorithm for sparse tensor factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2016, pp. 902–911.

[69] O. Kaya and B. Uçar, "Parallel candecomp/parafac decomposition of sparse tensors using dimension trees," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C99–C130, 2018.

[70] G. Ballard, N. Knight, and K. Rouse, "Communication lower bounds for matricized tensor times Khatri-Rao product," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2018, pp. 557–567.

[71] G. Ballard and K. Rouse, "General memory-independent lower bound for MTTKRP," in *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 2020, pp. 1–11.

[72] L. S. Blackford, J. Choi, A. Cleary, *et al.*, *ScaLAPACK users' guide*. SIAM, 1997.

[73] S. Eswar, K. Hayashi, G. Ballard, R. Kannan, M. A. Matheson, and H. Park, "Planc: Parallel low-rank approximation with nonnegativity constraints," *ACM Transactions on Mathematical Software (TOMS)*, vol. 47, no. 3, pp. 1–37, 2021.

[74] T. Sarlos, "Improved approximation algorithms for large matrices via random projections," in *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, IEEE, 2006, pp. 143–152.

[75] M. Che, Y. Wei, and H. Yan, "Randomized algorithms for the low multilinear rank approximations of tensors," *Journal of Computational and Applied Mathematics*, p. 113 380, 2021.

[76] M. Che and Y. Wei, "Randomized algorithms for the approximations of Tucker and the tensor train decompositions," *Advances in Computational Mathematics*, vol. 45, no. 1, pp. 395–428, 2019.

[77] G. Zhou, A. Cichocki, and S. Xie, "Decomposition of big tensors with low multilinear rank," *arXiv preprint arXiv:1412.1885*, 2014.

[78] Y. Sun, Y. Guo, J. A. Tropp, and M. Udell, "Tensor random projection for low memory dimension reduction," in *NeurIPS Workshop on Relational Representation Learning*, 2018.

[79] O. A. Malik and S. Becker, "Low-rank tucker decomposition of large tensors using Tensorsketch," *Advances in neural information processing systems*, vol. 31, pp. 10 096–10 106, 2018.

[80] C. Battaglino, G. Ballard, and T. G. Kolda, "A practical randomized CP tensor decomposition," *SIAM Journal on Matrix Analysis and Applications*, vol. 39, no. 2, pp. 876–901, 2018.

[81] R. Jin, T. G. Kolda, and R. Ward, "Faster Johnson-Lindenstrauss transforms via Kronecker products," *Information and Inference: A Journal of the IMA*, vol. 10, no. 4, pp. 1533–1562, 2021.

[82] B. W. Larsen and T. G. Kolda, "Practical leverage-based sampling for low-rank tensor decomposition," *SIAM Journal on Matrix Analysis and Applications*, vol. 43, no. 3, pp. 1488–1517, 2022.

[83] M. Lubasch, J. I. Cirac, and M.-C. Banuls, "Unifying projected entangled pair state contractions," *New Journal of Physics*, vol. 16, no. 3, p. 033 014, 2014.

[84] M. Lubasch, J. I. Cirac, and M.-C. Banuls, "Algorithms for finite projected entangled pair states," *Physical Review B*, vol. 90, no. 6, p. 064 425, 2014.

[85] A. Jermyn, "Automatic contraction of unstructured tensor networks," *SciPost Physics*, vol. 8, no. 1, p. 005, 2020.

[86] C. T. Chubb, "General tensor network decoding of 2D Pauli codes," *arXiv preprint arXiv:2101.04125*, 2021.

[87] J. Gray and G. K. Chan, "Hyper-optimized compressed contraction of tensor networks with arbitrary geometry," *arXiv preprint arXiv:2206.07044*, 2022.

[88] O. A. Malik, "More efficient sampling for tensor decomposition with worst-case guarantees," in *International Conference on Machine Learning*, PMLR, 2022, pp. 14 887–14 917.

[89] H. Al Daas, G. Ballard, P. Cazeaux, *et al.*, "Randomized algorithms for rounding in the tensor-train format," *SIAM Journal on Scientific Computing*, vol. 45, no. 1, A74–A95, 2023.

[90] T. contributors, "Density matrix algorithm - tensornetwork.org," 2021.

[91] M. Fishman, S. White, and E. Stoudenmire, "The ITensor software library for tensor network calculations," *SciPost Physics Codebases*, p. 004, 2022.

[92] M. Hein, J. Eisert, and H. J. Briegel, "Multiparty entanglement in graph states," *Physical Review A*, vol. 69, no. 6, p. 062 311, 2004.

[93] L. Ma and E. Solomonik, "Accelerating alternating least squares for tensor decomposition by pairwise perturbation," *Numerical Linear Algebra with Applications*, vol. 29, no. 4, e2431, 2022.

[94] L. Ma and E. Solomonik, "Efficient parallel CP decomposition with pairwise perturbation and multi-sweep dimension tree," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2021, pp. 412–421.

[95] A. Hurwitz, "Über die Composition der quadratischen Formen von belibig vielen Variablen," *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, vol. 1898, pp. 309–316, 1898.

[96] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.

[97] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, *et al.*, "On optimizing distributed Tucker decomposition for dense tensors," in *2017 IEEE Int Parallel Distrib Process Symp. (IPDPS)*, IEEE, 2017, pp. 1038–1047.

[98] O. Kaya, "High performance parallel algorithms for tensor decompositions," *Université de Lyon*, 2017.

[99] N. Vannieuwenhoven, K. Meerbergen, and R. Vandebril, "Computing the gradient in optimization algorithms for the CP decomposition in constant memory through tensor blocking," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. C415–C438, 2015.

[100] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3176–3190, 2014.

[101] G. Ballard, N. Knight, and K. Rouse, "Communication lower bounds for matricized tensor times Khatri-Rao product," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2018, pp. 557–567.

[102] I. V. Oseledets and E. E. Tyrtyshnikov, "Breaking the curse of dimensionality, or how to use SVD in many dimensions," *SIAM Journal on Scientific Computing*, vol. 31, no. 5, pp. 3744–3759, 2009.

[103] J. Choi, X. Liu, and V. Chakaravarthy, "High-performance dense Tucker decomposition on GPU clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, IEEE Press, 2018, p. 42.

[104] O. Kaya and B. Uçar, "High performance parallel algorithms for the Tucker decomposition of sparse tensors," in *2016 45th International Conference on Parallel Processing (ICPP)*, IEEE, 2016, pp. 103–112.

[105] L.-H. Lim, "Singular values and eigenvalues of tensors: A variational approach," in *1st IEEE Int Workshop on Comput Adv in Multi-Sensor Adaptive Process.*, IEEE, 2005, pp. 129–132.

[106] S. Friedland, V. Mehrmann, R. Pajarola, and S. K. Suter, "On best rank one approximation of tensors," *Numer Linear Algebra Appl.*, vol. 20, no. 6, pp. 942–955, 2013.

[107] P. Springer, T. Su, and P. Bientinesi, "HPTT: A high-performance tensor transposition C++ library," in *ACM SIGPLAN Int Workshop Libr Lang Compil Array Program.*, ACM, 2017, pp. 56–62.

[108] T. G. Kolda and B. W. Bader, "Matlab tensor toolbox," Sandia National Laboratories, Tech. Rep., 2006.

[109] M. Rajih, P. Comon, and R. A. Harshman, "Enhanced line search: A novel method to accelerate PARAFAC," *SIAM J Matrix Anal Appl.*, vol. 30, no. 3, pp. 1128–1147, 2008.

[110] Q. Sun, T. C. Berkelbach, N. S. Blunt, *et al.*, "PySCF: The Python-based simulations of chemistry framework," *Wiley Interdiscip Rev Comput Mol Sci.*, vol. 8, no. 1, e1340, 2018.

[111] S. A. Nene, S. K. Nayar, H. Murase, *et al.*, "Columbia object image library (coil-100)," *Citeseer*, 1996.

[112] S. M. Nascimento, K. Amano, and D. H. Foster, "Spatial distributions of local illumination color in natural scenes," *Vision research*, vol. 120, pp. 39–44, 2016.

[113] B. N. Khoromskij and V. Khoromskaia, "Multigrid accelerated tensor approximation of function related multidimensional arrays," *SIAM J Sci Comput.*, vol. 31, no. 4, pp. 3002–3026, 2009.

[114] J. D. Carroll, S. Pruzansky, and J. B. Kruskal, "Candelinc: A general approach to multidimensional analysis of many-way arrays with linear constraints on parameters," *Psychometrika*, vol. 45, no. 1, pp. 3–24, 1980.

[115] B. Khoromskij and V. Khoromskaia, "Low rank tucker-type tensor approximation to classical potentials," *Open Math.*, vol. 5, no. 3, pp. 523–550, 2007.

[116] M. J. Mohlenkamp, "The dynamics of swamps in the canonical tensor approximation problem," *SIAM J Appl Dyn Syst.*, vol. 18, no. 3, pp. 1293–1333, 2019.

[117] W. Austin, G. Ballard, and T. G. Kolda, "Parallel tensor compression for large-scale scientific data," in *2016 IEEE Int Parallel Distrib Process Symp. (IPDPS)*, IEEE, 2016, pp. 912–922.

[118] B. C. Mitchell and D. S. Burdick, "Slowly converging PARAFAC sequences: Swamps and two-factor degeneracies," *Journal of Chemometrics*, vol. 8, no. 2, pp. 155–168, 1994.

[119] C. Navasca, L. De Lathauwer, and S. Kindermann, "Swamp reducing technique for tensor decomposition," in *2008 16th Eur Signal Process Conf.*, IEEE, 2008, pp. 1–5.

[120] N. Li, S. Kindermann, and C. Navasca, "Some convergence results on the regularized alternating least-squares method for tensor decomposition," *Linear Algebra Appl.*, vol. 438, no. 2, pp. 796–812, 2013.

[121] D. Mitchell, N. Ye, and H. De Sterck, "Nesterov acceleration of alternating least squares for canonical tensor decomposition: Momentum step size selection and restart mechanisms," *Numer Linear Algebra Appl.*, vol. 27, no. 4, e2297, 2020.

[122] J. Radon, "Lineare Scharen orthogonaler Matrizen," in *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, Springer, vol. 1, 1922, pp. 1–14.

[123] J. F. Adams, "Vector fields on spheres," *Ann Math.*, pp. 603–632, 1962.

[124] J. F. Adams, P. D. Lax, and R. S. Phillips, "On matrices whose real linear combinations are nonsingular," *Proc Am Math Soc.*, vol. 16, no. 2, pp. 318–322, 1965.

[125] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[126] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[127] E. Solomonik, G. Ballard, J. Demmel, and T. Hoefler, "A communication-avoiding parallel algorithm for the symmetric eigenvalue problem," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, 2017, pp. 111–121.

[128] P. Jørgensen and J. Simons, *Geometrical derivatives of energy surfaces and molecular properties*. Springer Science & Business Media, 2012, vol. 166.

[129] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[130] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.1.55, 2018. [Online]. Available: http://github.com/google/jax.

[131] M. Abadi, P. Barham, J. Chen, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[132] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 47–62.

[133] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, "Tensorly: Tensor learning in Python," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 925–930, 2019.

[134] C. Roberts, A. Milsted, M. Ganahl, *et al.*, "Tensornetwork: A library for physics and machine learning," *arXiv preprint arXiv:1905.01330*, 2019.

[135] J. Gray, "Quimb: A Python package for quantum information and many-body calculations," *Journal of Open Source Software*, vol. 3, no. 29, p. 819, 2018.

[136] F. Verstraete, J. J. Garcia-Ripoll, and J. I. Cirac, "Matrix product density operators: Simulation of finite-temperature and dissipative systems," *Physical review letters*, vol. 93, no. 20, p. 207 204, 2004.

[137] P. Tichavskỳ, A. H. Phan, and A. Cichocki, "A further improvement of a fast damped Gauss-Newton algorithm for CANDECOMP-PARAFAC tensor decomposition," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, IEEE, 2013, pp. 5964–5968.

[138] N. Singh, L. Ma, H. Yang, and E. Solomonik, "Comparison of accuracy and scalability of Gauss-Newton and alternating least squares for CANDECOMC/PARAFAC decomposition," *SIAM Journal on Scientific Computing*, vol. 43, no. 4, pp. C290–C311, 2021.

[139] E. Acar, D. M. Dunlavy, and T. G. Kolda, "A scalable optimization approach for fitting canonical tensor decompositions," *Journal of Chemometrics*, vol. 25, no. 2, pp. 67–86, 2011.

[140] N. Singh, Z. Zhang, X. Wu, N. Zhang, S. Zhang, and E. Solomonik, "Distributed-memory tensor completion for generalized loss functions in python using new sparse tensor kernels," *Journal of Parallel and Distributed Computing*, vol. 169, pp. 269–285, 2022.

[141] S. Hirata, "Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories," *The Journal of Physical Chemistry A*, vol. 107, no. 46, pp. 9887–9897, 2003.

[142] S. Tokui, R. Okuta, T. Akiba, *et al.*, "Chainer: A deep learning framework for accelerating the research cycle," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2002–2011.

[143] S. F. Walter and L. Lehmann, "Algorithmic differentiation in Python with AlgoPy," *Journal of Computational Science*, vol. 4, no. 5, pp. 334–344, 2013.

[144] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Autograd: Effortless gradients in NumPy."

[145] B. van Merrienboer, D. Moldovan, and A. Wiltschko, "Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming," in *Advances in Neural Information Processing Systems*, 2018, pp. 6256–6265.

[146] Y. Jia, E. Shelhamer, J. Donahue, *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[147] S. Rajbhandari, A. Nikam, P.-W. Lai, K. Stock, S. Krishnamoorthy, and P. Sadayappan, "A communication-optimal framework for contracting distributed tensors," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2014, pp. 375–386.

[148] R. A. Kendall, E. Aprà, D. E. Bernholdt, *et al.*, "High performance computational chemistry: An overview of NWChem a distributed parallel application," *Computer Physics Communications*, vol. 128, no. 1-2, pp. 260–283, 2000.

[149] D. Kats and F. R. Manby, "Sparse tensor framework for implementation of general local correlation methods," *The Journal of Chemical Physics*, vol. 138, no. 14, 2013.

[150] J. Choi, X. Liu, S. Smith, and T. Simon, "Blocking optimization techniques for sparse tensor computation," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2018, pp. 568–577.

[151] J. Li, J. Sun, and R. Vuduc, "HiCOO: Hierarchical storage of sparse tensors," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2018, pp. 238–252.

[152] K. Z. Ibrahim, S. W. Williams, E. Epifanovsky, and A. I. Krylov, "Analysis and tuning of libtensor framework on multicore architectures," in *2014 21st International Conference on High Performance Computing (HiPC)*, IEEE, 2014, pp. 1–10.

[153] R. Senanayake, F. Kjolstad, C. Hong, S. Kamil, and S. Amarasinghe, "A unified iteration space transformation framework for sparse and dense tensor algebra," *arXiv preprint arXiv:2001.00532*, 2019.

[154] E. Solomonik and J. Demmel, "Fast bilinear algorithms for symmetric tensor contractions," *Computational Methods in Applied Mathematics*, 2020.

[155] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[156] C. Peng, J. A. Calvin, F. Pavosevic, J. Zhang, and E. F. Valeev, "Massively parallel implementation of explicitly correlated coupled-cluster singles and doubles using TiledArray framework," *The Journal of Physical Chemistry A*, vol. 120, no. 51, pp. 10 231–10 244, 2016.

[157] S. Manzer, E. Epifanovsky, A. I. Krylov, and M. Head-Gordon, "A general sparse tensor framework for electronic structure theory," *Journal of chemical theory and computation*, vol. 13, no. 3, pp. 1108–1116, 2017.

[158] S. Smith and G. Karypis, “Tensor-matrix products with a compressed sparse tensor,” in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 2015, pp. 1–7.

[159] X. Team *et al.*, *XLA-TensorFlow compiled. post in the Google developers blog*, 2017.

[160] T. Chen, T. Moreau, Z. Jiang, *et al.*, “TVM: An automated End-to-End optimizing compiler for deep learning,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[161] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers, principles, techniques,” *Addison wesley*, vol. 7, no. 8, p. 9,

[162] A. Hartono, Q. Lu, X. Gao, *et al.*, “Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations,” in *International Conference on Computational Science*, Springer, 2006, pp. 267–275.

[163] D. Smith and J. Gray, “Opt_einsum - a Python package for optimizing contraction order for einsum-like expressions,” *Journal of Open Source Software*, vol. 3, no. 26, p. 753, 2018.

[164] A. A. Auer, G. Baumgartner, D. E. Bernholdt, *et al.*, “Automatic code generation for many-body electronic structure methods: The tensor contraction engine,” *Molecular Physics*, vol. 104, no. 2, pp. 211–228, 2006.

[165] A. Hartono, Q. Lu, T. Henretty, *et al.*, “Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry,” *The Journal of Physical Chemistry A*, vol. 113, no. 45, pp. 12 715–12 723, 2009.

[166] A. Hartono, A. Sibiryakov, M. Nooijen, *et al.*, “Automated operation minimization of tensor contraction expressions in electronic structure calculations,” in *International Conference on Computational Science*, Springer, 2005, pp. 155–164.

[167] L. Chi-Chung, P. Sadayappan, and R. Wenger, “On optimizing a class of multi-dimensional loops with reduction for parallel execution,” *Parallel Processing Letters*, vol. 7, no. 02, pp. 157–168, 1997.

[168] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: A survey,” *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 5595–5637, 2017.

[169] A. Meurer, C. P. Smith, M. Paprocki, *et al.*, “SymPy: Symbolic computing in Python,” *PeerJ Computer Science*, vol. 3, e103, 2017.

[170] L. Sorber, M. Van Barel, and L. De Lathauwer, "Optimization-based algorithms for tensor decompositions: canonical polyadic decomposition, decomposition in rank-$(L_r, L_r, 1)$ terms, and a new generalization," *SIAM Journal on Optimization*, vol. 23, no. 2, pp. 695–720, 2013.

[171] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[172] L. Ma and E. Solomonik, "Fast and accurate randomized algorithms for low-rank tensor decompositions," in *Advances in Neural Information Processing Systems*, 2021.

[173] M. Pilanci and M. J. Wainwright, "Iterative Hessian sketch: Fast and accurate solution approximation for constrained least-squares," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1842–1879, 2016.

[174] R. Pagh, "Compressed matrix multiplication," *ACM Transactions on Computation Theory (TOCT)*, vol. 5, no. 3, pp. 1–17, 2013.

[175] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*, Springer, 2002, pp. 693–703.

[176] P. Drineas, M. Magdon-Ismail, M. W. Mahoney, and D. P. Woodruff, "Fast approximation of matrix coherence and statistical leverage," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 3475–3506, 2012.

[177] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM review*, vol. 53, no. 2, pp. 217–288, 2011.

[178] R. Bro and C. A. Andersson, "Improving the speed of multiway algorithms: Part II: Compression," *Chemometrics and intelligent laboratory systems*, vol. 42, no. 1-2, pp. 105–113, 1998.

[179] I. T. Jolliffe, "Discarding variables in a principal component analysis. I: Artificial data," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 21, no. 2, pp. 160–173, 1972.

[180] D. Papailiopoulos, A. Kyrillidis, and C. Boutsidis, "Provable deterministic leverage score sampling," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 997–1006.

[181] S. Ahmadi-Asl, A. Cichocki, A. H. Phan, I. Oseledets, S. Abukhovich, and T. Tanaka, "Randomized algorithms for computation of Tucker decomposition and Higher Order SVD (HOSVD)," *arXiv preprint arXiv:2001.07124*, 2020.

[182] Y. Sun, Y. Guo, C. Luo, J. Tropp, and M. Udell, "Low-rank tucker approximation of a tensor from streaming data," *SIAM Journal on Mathematics of Data Science*, vol. 2, no. 4, pp. 1123–1150, 2020.

[183] R. Minster, A. K. Saibaba, and M. E. Kilmer, "Randomized algorithms for low-rank tensor decompositions in the Tucker format," *SIAM Journal on Mathematics of Data Science*, vol. 2, no. 1, pp. 189–215, 2020.

[184] M. Gu and S. C. Eisenstat, "Efficient algorithms for computing a strong rank-revealing QR factorization," *SIAM Journal on Scientific Computing*, vol. 17, no. 4, pp. 848–869, 1996.

[185] S. Oh, N. Park, S. Lee, and U. Kang, "Scalable Tucker factorization for sparse tensors-algorithms and discoveries," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, IEEE, 2018, pp. 1120–1131.

[186] H. Li, Z. Li, K. Li, J. S. Rellermeyer, L. Chen, and K. Li, "SGD_tucker: A novel stochastic optimization strategy for parallel sparse tucker decomposition," *arXiv preprint arXiv:2012.03550*, 2020.

[187] K. S. Aggour, A. Gittens, and B. Yener, "Adaptive sketching for fast and convergent canonical polyadic decomposition," in *International Conference on Machine Learning. PMLR*, 2020.

[188] Z. Song, D. P. Woodruff, and P. Zhong, "Relative error tensor low rank approximation," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2019, pp. 2772–2789.

[189] D. Cheng, R. Peng, Y. Liu, and I. Perros, "SPALS: Fast alternating least squares via implicit leverage scores sampling," *Advances in neural information processing systems*, vol. 29, pp. 721–729, 2016.

[190] N. B. Erichson, K. Manohar, S. L. Brunton, and J. N. Kutz, "Randomized CP tensor decomposition," *Machine Learning: Science and Technology*, vol. 1, no. 2, p. 025 012, 2020.

[191] D. Nion and L. De Lathauwer, "An enhanced line search scheme for complex-valued tensor decompositions. Application in DS-CDMA," *Signal Processing*, vol. 88, no. 3, pp. 749–755, 2008.

[192] P. Paatero, "A weighted non-negative least squares algorithm for three-way PARAFAC factor analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 38, no. 2, pp. 223–242, 1997.

[193] A.-H. Phan, P. Tichavsky, and A. Cichocki, "Low complexity damped Gauss-Newton algorithms for CANDECOMP/PARAFAC," *SIAM Journal on Matrix Analysis and Applications*, vol. 34, no. 1, pp. 126–147, 2013.

[194] T. E. Oliphant, "A guide to NumPy," *Trelgol Publishing USA*, 2006.

[195] D. Kressner, B. Vandereycken, and R. Voorhaar, "Streaming tensor train approximation," *arXiv preprint arXiv:2208.02600*, 2022.

[196] M. W. Mahoney *et al.*, "Randomized algorithms for matrices and data," *Foundations and Trends® in Machine Learning*, vol. 3, no. 2, pp. 123–224, 2011.

[197] E. J. Candès and B. Recht, "Exact matrix completion via convex optimization," *Foundations of Computational mathematics*, vol. 9, no. 6, p. 717, 2009.

[198] C. Boutsidis and D. Woodruff, "Communication-optimal distributed principal component analysis in the column-partition model," *ArXiv*, vol. abs/1504.06729, 2015.

[199] S. Wang, "A practical guide to randomized matrix computations with matlab implementations," *arXiv preprint arXiv:1505.07570*, 2015.

[200] P. Drineas, M. W. Mahoney, S. Muthukrishnan, and T. Sarlós, "Faster least squares approximation," *Numerische mathematik*, vol. 117, no. 2, pp. 219–249, 2011.

[201] L. Mirsky, "Symmetric gauge functions and unitarily invariant norms," *The quarterly journal of mathematics*, vol. 11, no. 1, pp. 50–59, 1960.

[202] H. Avron, K. L. Clarkson, and D. P. Woodruff, "Sharper bounds for regularized data fitting," *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, 2017.

[203] H. Diao, Z. Song, W. Sun, and D. Woodruff, "Sketching for Kronecker product regression and p-splines," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2018, pp. 1299–1308.

[204] L. Ma and E. Solomonik, "Cost-efficient gaussian tensor network embeddings for tensor-structured inputs," *Advances in Neural Information Processing Systems*, vol. 35, pp. 38 980–38 993, 2022.

[205] B. Rakhshan and G. Rabusseau, "Tensorized random projections," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2020, pp. 3306–3316.

[206] A. V. Mahankali, D. P. Woodruff, and Z. Zhang, "Low rank approximation for general tensor networks," *arXiv preprint arXiv:2207.07417*, 2022.

[207] N. Ailon and B. Chazelle, "Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform," in *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, 2006, pp. 557–563.

[208] K. Chen and R. Jin, "Tensor-structured sketching for constrained least squares," *SIAM Journal on Matrix Analysis and Applications*, vol. 42, no. 4, pp. 1703–1731, 2021.

[209] M. Pilanci and M. J. Wainwright, "Randomized sketches of convex programs with sharp guarantees," *IEEE Transactions on Information Theory*, vol. 61, no. 9, pp. 5096–5115, 2015.

[210] K. Batselier, W. Yu, L. Daniel, and N. Wong, "Computing low-rank approximations of large-scale matrices with the tensor network randomized SVD," *SIAM Journal on Matrix Analysis and Applications*, vol. 39, no. 3, pp. 1221–1244, 2018.

[211] L. Ma, J. Ye, and E. Solomonik, "AutoHOOT: Automatic high-order optimization for tensors," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 125–137.

[212] D. Kane, R. Meka, and J. Nelson, "Almost optimal explicit Johnson-Lindenstrauss families," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, Springer, 2011, pp. 628–639.

[213] D. M. Kane and J. Nelson, "Sparser Johnson-Lindenstrauss transforms," *Journal of the ACM (JACM)*, vol. 61, no. 1, pp. 1–23, 2014.

[214] U. Schollwöck, "The density-matrix renormalization group in the age of matrix product states," *Annals of physics*, vol. 326, no. 1, pp. 96–192, 2011.

[215] S. V. Dolgov, B. N. Khoromskij, and I. V. Oseledets, "Fast solution of parabolic problems in the tensor train/quantized tensor train format with initial application to the Fokker–Planck equation," *SIAM Journal on Scientific Computing*, vol. 34, no. 6, A3016–A3038, 2012.

[216] L. Richter, L. Sallandt, and N. Nüsken, "Solving high-dimensional parabolic PDEs using the tensor train format," in *International Conference on Machine Learning*, PMLR, 2021, pp. 8998–9009.

[217] S. Klus, P. Gelß, S. Peitz, and C. Schütte, "Tensor-based dynamic mode decomposition," *Nonlinearity*, vol. 31, no. 7, p. 3359, 2018.

[218] S. Klus, P. Koltai, and C. Schütte, "On the numerical approximation of the Perron-Frobenius and Koopman operator," *arXiv preprint arXiv:1512.05997*, 2015.

[219] G. Beylkin, J. Garcke, and M. J. Mohlenkamp, "Multivariate regression and machine learning with sums of separable functions," *SIAM Journal on Scientific Computing*, vol. 31, no. 3, pp. 1840–1857, 2009.

[220] A. Obukhov, M. Rakhuba, A. Liniger, *et al.*, "Spectral tensor train parameterization of deep learning layers," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2021, pp. 3547–3555.

[221] L. G. Valiant, "The complexity of computing the permanent," *Theoretical computer science*, vol. 8, no. 2, pp. 189–201, 1979.

[222] P. Erdös, A. W. Goodman, and L. Pósa, "The representation of a graph by set intersections," *Canadian Journal of Mathematics*, vol. 18, pp. 106–112, 1966.

[223] E. Mäkinen, "On circular layouts," *International Journal of Computer Mathematics*, vol. 24, no. 1, pp. 29–37, 1988.

[224] F. Bernhart and P. C. Kainen, "The book thickness of a graph," *Journal of Combinatorial Theory, Series B*, vol. 27, no. 3, pp. 320–331, 1979.

[225] A. Riskin, "The circular k-partite crossing number of $K_{m,n}$," *arXiv preprint*, 2006.

[226] C. Ibrahim, D. Lykov, Z. He, Y. Alexeev, and I. Safro, "Constructing optimal contraction trees for tensor network quantum circuit simulation," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2022, pp. 1–8.

[227] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita, "Compressing graphs and indexes with recursive graph bisection," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1535–1544.

[228] 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*, IEEE Computer Society, 1989, pp. 604–609.

[229] F. Shahrokhi, O. Sỳkora, L. A. Székely, and I. Vrt'o, "Book embeddings and crossing numbers," in *Graph-Theoretic Concepts in Computer Science: 20th International Workshop, WG'94 Herrsching, Germany, June 16–18, 1994 Proceedings 20*, Springer, 1995, pp. 256–268.

[230] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[231] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva, "Maximum flow and minimum-cost flow in almost-linear time," in *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2022, pp. 612–623.

[232] G. Dantzig and D. R. Fulkerson, "On the max flow min cut theorem of networks," *Linear inequalities and related systems*, vol. 38, pp. 225–231, 2003.

[233] N. Nakatani and G. K.-L. Chan, "Efficient tree tensor network states (TTNS) for quantum chemistry: Generalizations of the density matrix renormalization group algorithm," *The Journal of chemical physics*, vol. 138, no. 13, p. 134 113, 2013.

[234] V. Murg, F. Verstraete, R. Schneider, P. R. Nagy, and O. Legeza, "Tree tensor network state with variable tensor order: An efficient multireference method for strongly correlated systems," *Journal of Chemical Theory and Computation*, vol. 11, no. 3, pp. 1027–1036, 2015.

[235] T. Felser, S. Notarnicola, and S. Montangero, "Efficient tensor network ansatz for high-dimensional quantum many-body problems," *Physical Review Letters*, vol. 126, no. 17, p. 170 603, 2021.

[236] S. Sahu and B. Swingle, "Efficient tensor network simulation of quantum many-body physics on sparse graphs," *arXiv preprint arXiv:2206.04701*, 2022.

[237] R. Alkabetz and I. Arad, "Tensor networks contraction and the belief propagation algorithm," *Physical Review Research*, vol. 3, no. 2, p. 023 073, 2021.

[238] C. Li, J. Zeng, Z. Tao, and Q. Zhao, "Permutation search of tensor network structures via local sampling," in *International Conference on Machine Learning*, PMLR, 2022, pp. 13 106–13 124.

[239] S. Schlag, T. Heuer, L. Gottesbüren, Y. Akhremtsev, C. Schulz, and P. Sanders, "High-quality hypergraph partitioning," *ACM Journal of Experimental Algorithmics*, vol. 27, pp. 1–39, 2023.

[240] G. Karypis, "METIS: Unstructured graph partitioning and sparse matrix ordering system," *Technical report*, 1997.

[241] S. W. Hruska, "On tree congestion of graphs," *Discrete mathematics*, vol. 308, no. 10, pp. 1801–1809, 2008.

[242] D. Bienstock, "On embedding graphs in trees," *Journal of Combinatorial Theory, Series B*, vol. 49, no. 1, pp. 103–136, 1990.

[243] A. Matsubayashi, "Separator-based graph embedding into multidimensional grids with small edge-congestion," *Discrete Applied Mathematics*, vol. 185, pp. 119–137, 2015.

[244] 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*, vol. 213, no. 1-3, pp. 13–19, 2000.

[245] P. Manuel, I. Rajasingh, B. Rajan, and H. Mercy, "Exact wirelength of hypercubes on a grid," *Discrete Applied Mathematics*, vol. 157, no. 7, pp. 1486–1495, 2009.

[246] Y. Zhang and E. Solomonik, "On stability of tensor networks and canonical forms," *arXiv preprint arXiv:2001.01191*, 2020.

[247] E. Stoudenmire and S. R. White, "Minimally entangled typical thermal state algorithms," *New Journal of Physics*, vol. 12, no. 5, p. 055 026, 2010.

[248] D. M. Thilikos, M. Serna, and H. L. Bodlaender, "Cutwidth I: A linear time fixed parameter algorithm," *Journal of Algorithms*, vol. 56, no. 1, pp. 1–24, 2005.

[249] L. H. Harper, "Optimal assignments of numbers to vertices," *Journal of the Society for Industrial and Applied Mathematics*, vol. 12, no. 1, pp. 131–135, 1964.

[250] J. Diaz, J. Petit, and M. Serna, "A survey of graph layout problems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 313–356, 2002.

[251] G. Even, J. S. Naor, S. Rao, and B. Schieber, "Divide-and-conquer approximation algorithms via spreading metrics," *Journal of the ACM (JACM)*, vol. 47, no. 4, pp. 585–616, 2000.

[252] S. Rao and A. W. Richa, "New approximation techniques for some linear ordering problems," *SIAM Journal on Computing*, vol. 34, no. 2, pp. 388–404, 2005.

[253] U. Feige and J. R. Lee, "An improved approximation ratio for the minimum linear arrangement problem," *Information Processing Letters*, vol. 101, no. 1, pp. 26–29, 2007.

[254] M. Charikar, M. T. Hajiaghayi, H. Karloff, and S. Rao, "ℓ_2^2 Spreading metrics for vertex ordering problems," *Algorithmica*, vol. 56, pp. 577–604, 2010.

[255] N. R. Devanur, S. A. Khot, R. Saket, and N. 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*, 2006, pp. 537–546.

[256] H. D. Simon and S.-H. Teng, "How good is recursive bisection?" *SIAM Journal on Scientific Computing*, vol. 18, no. 5, pp. 1436–1445, 1997.

[257] V. V. Vazirani, *Approximation algorithms.* Springer, 2001, vol. 1.

[258] T. Leighton and S. Rao, "An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms," Massachusetts Inst Of Tech Cambridge Microsystems Research Center, Tech. Rep., 1989.

[259] S. Arora, S. Rao, and U. Vazirani, "Expander flows, geometric embeddings and graph partitioning," *Journal of the ACM (JACM)*, vol. 56, no. 2, pp. 1–37, 2009.

[260] D. Bauernfeind, M. Zingl, R. Triebl, M. Aichhorn, and H. G. Evertz, "Fork tensor-product states: Efficient multiorbital real-time dmft solver," *Physical Review X*, vol. 7, no. 3, p. 031 013, 2017.

[261] N. Chepiga and S. R. White, "Comb tensor networks," *Physical Review B*, vol. 99, no. 23, p. 235 426, 2019.

[262] J. Gray and S. Kourtis, "Hyper-optimized tensor network contraction," *Quantum*, vol. 5, p. 410, 2021.

[263] J.-G. Liu, X. Gao, M. Cain, M. D. Lukin, and S.-T. Wang, "Computing solution space properties of combinatorial optimization problems via generic tensor networks," *SIAM Journal on Scientific Computing*, vol. 45, no. 3, A1239–A1270, 2023.

[264] H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang, "Differentiable programming tensor networks," *Physical Review X*, vol. 9, no. 3, p. 031 041, 2019.

[265] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," *SIAM Journal on Computing*, vol. 38, no. 3, pp. 963–981, 2008.

[266] R. A. Harshman, "Determination and proof of minimum uniqueness conditions for PARAFAC1," *UCLA working papers in phonetics*, vol. 22, no. 111-117, p. 3, 1972.

[267] C. Guo, Y. Liu, M. Xiong, *et al.*, "General-purpose quantum circuit simulator with projected entangled-pair states and the quantum supremacy frontier," *Physical review letters*, vol. 123, no. 19, p. 190 501, 2019.

[268] C. Chamon and E. R. Mucciolo, "Virtual parallel computing and a search algorithm using matrix product states," *Physical review letters*, vol. 109, no. 3, p. 030 503, 2012.

[269] D. Coppersmith, "An approximate Fourier Transform useful in quantum factoring," *arXiv preprint quant-ph/0201067*, 2002.

[270] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, "Quantum algorithms revisited," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1969, pp. 339–354, 1998.

[271] L. K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[272] G. Brassard, M. Mosca, and A. Tapp, “Quantum amplitude amplification and estimation,” 2002.

[273] M. Szegedy, “Quantum speed-up of Markov chain based algorithms,” in *45th Annual IEEE symposium on foundations of computer science*, IEEE, 2004, pp. 32–41.

[274] A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman, “Exponential algorithmic speedup by a quantum walk,” in *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, 2003, pp. 59–68.

[275] D. E. Deutsch, “Quantum computational networks,” *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 425, no. 1868, pp. 73–90, 1989.

[276] F. Arute, K. Arya, R. Babbush, *et al.*, “Quantum supremacy using a programmable superconducting processor,” *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[277] M. Kjaergaard, M. E. Schwartz, J. Braumüller, *et al.*, “Superconducting qubits: Current state of play,” *Annual Review of Condensed Matter Physics*, vol. 11, pp. 369–395, 2020.

[278] K. Woolfe, “Matrix product operator simulations of quantum algorithms,” University of Melbourne School of Physics Melbourne Australia, Tech. Rep., 2015.

[279] D. Camps, R. Van Beeumen, and C. Yang, “Quantum Fourier transform revisited,” *arXiv preprint arXiv:2003.03011*, 2020.

[280] T. E. O’Brien, B. Tarasinski, and B. M. Terhal, “Quantum phase estimation of multiple eigenvalues for small-scale (noisy) experiments,” *New Journal of Physics*, vol. 21, no. 2, p. 023 022, 2019.

[281] M. Santha, “Quantum walk based search algorithms,” in *International Conference on Theory and Applications of Models of Computation*, Springer, 2008, pp. 31–46.

[282] N. Nahimovs and A. Rivošs, “A note on the optimality of the Grover’s algorithm,”

[283] T. D. Mackay, S. D. Bartlett, L. T. Stephenson, and B. C. Sanders, “Quantum walks in higher dimensions,” *Journal of Physics A: Mathematical and General*, vol. 35, no. 12, p. 2745, 2002.

[284] S. E. Venegas-Andraca, “Quantum walks: A comprehensive review,” *Quantum Information Processing*, vol. 11, no. 5, pp. 1015–1106, 2012.

[285] N. Shenvi, J. Kempe, and K. B. Whaley, "Quantum random-walk search algorithm," *Physical Review A*, vol. 67, no. 5, p. 052 307, 2003.

[286] G. D. Paparo and M. Martin-Delgado, "Google in a quantum network," *Scientific reports*, vol. 2, p. 444, 2012.

[287] R. A. Santos, "Szegedy's quantum walk with queries," *Quantum Information Processing*, vol. 15, no. 11, pp. 4461–4475, 2016.

[288] B. Douglas and J. Wang, "Efficient quantum circuit implementation of quantum walks," *Physical Review A*, vol. 79, no. 5, p. 052 335, 2009.

[289] T. Loke and J. Wang, "Efficient quantum circuits for Szegedy quantum walks," *Annals of Physics*, vol. 382, pp. 64–84, 2017.

[290] R. Okuta, Y. Unno, D. Nishino, S. Hido, and Crissman, "CuPy: A NumPy-compatible library for NVIDIA GPU calculations," *31st confernce on neural information processing systems*, 2017.

[291] A. Einstein, B. Podolsky, and N. Rosen, "Can quantum-mechanical description of physical reality be considered complete?" *Physical review*, vol. 47, no. 10, p. 777, 1935.

[292] R. Horodecki, P. Horodecki, M. Horodecki, and K. Horodecki, "Quantum entanglement," *Reviews of modern physics*, vol. 81, no. 2, p. 865, 2009.

[293] A. Ekert and R. Jozsa, "Quantum algorithms: Entanglement-enhanced information processing," *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 356, no. 1743, pp. 1769–1782, 1998.

[294] A. Ekert and P. L. Knight, "Entangled quantum systems and the Schmidt decomposition," *American Journal of Physics*, vol. 63, no. 5, pp. 415–423, 1995.

[295] M. Walter, D. Gross, and J. Eisert, "Multipartite entanglement," *Quantum Information: From Foundations to Quantum Technology Applications*, pp. 293–330, 2016.

[296] G. Vidal, "Entanglement monotones," *Journal of Modern Optics*, vol. 47, no. 2-3, pp. 355–376, 2000.

[297] C. H. Bennett, D. P. DiVincenzo, J. A. Smolin, and W. K. Wootters, "Mixed-state entanglement and quantum error correction," *Physical Review A*, vol. 54, no. 5, p. 3824, 1996.

[298] E. M. Rains, "Rigorous treatment of distillable entanglement," *Physical Review A*, vol. 60, no. 1, p. 173, 1999.

[299] G. Vidal and R. F. Werner, "Computable measure of entanglement," *Physical Review A*, vol. 65, no. 3, p. 032 314, 2002.

[300] S. Lee, D. P. Chi, S. D. Oh, and J. Kim, "Convex-roof extended negativity as an entanglement measure for bipartite quantum systems," *Physical Review A*, vol. 68, no. 6, p. 062 304, 2003.

[301] T.-C. Wei and P. M. Goldbart, "Geometric measure of entanglement and applications to bipartite and multipartite quantum states," *Physical Review A*, vol. 68, no. 4, p. 042 307, 2003.

[302] W. K. Wootters, "Entanglement of formation and concurrence.," *Quantum Inf. Comput.*, vol. 1, no. 1, pp. 27–44, 2001.

[303] F. Mintert, M. Kuś, and A. Buchleitner, "Concurrence of mixed multipartite quantum states," *Physical review letters*, vol. 95, no. 26, p. 260 502, 2005.

[304] R. Raussendorf and H. J. Briegel, "A one-way quantum computer," *Physical review letters*, vol. 86, no. 22, p. 5188, 2001.

[305] M. Nest and H.-J. Briegel, "Entanglement in graph states and its applications," *arXiv preprint quant-ph/0602096*, 2006.

[306] D. Schlingemann and R. F. Werner, "Quantum error-correcting codes associated with graphs," *Physical Review A*, vol. 65, no. 1, p. 012 308, 2001.

[307] D. Markham and B. C. Sanders, "Graph states for quantum secret sharing," *Physical Review A*, vol. 78, no. 4, p. 042 309, 2008.

[308] S. Anders and H. J. Briegel, "Fast simulation of stabilizer circuits using a graph-state representation," *Physical Review A*, vol. 73, no. 2, p. 022 334, 2006.

[309] M. Van den Nest, J. Dehaene, and B. De Moor, "Graphical description of the action of local clifford transformations on graph states," *Physical Review A*, vol. 69, no. 2, p. 022 316, 2004.

[310] D. Schlingemann, "Stabilizer codes can be realized as graph codes," *arXiv preprint quant-ph/0111080*, 2001.

[311] L. Schatzki, L. Ma, E. Solomonik, and E. Chitambar, "Tensor rank and other multipartite entanglement measures of graph states," *arXiv preprint arXiv:2209.06320*, 2022.

[312] J. Eisert and H. J. Briegel, "Schmidt measure as a tool for quantifying multiparticle entanglement," *Physical Review A*, vol. 64, no. 2, p. 022 306, 2001.

[313] E. Chitambar, R. Duan, and Y. Shi, "Tripartite entanglement transformations and tensor rank," *Physical review letters*, vol. 101, no. 14, p. 140 502, 2008.

[314] L. Chen, E. Chitambar, R. Duan, Z. Ji, and A. Winter, "Tensor rank and stochastic entanglement catalysis for multipartite pure states," *Physical review letters*, vol. 105, no. 20, p. 200 501, 2010.

[315] N. Yu, C. Guo, and R. Duan, "Obtaining a W state from a Greenberger-Horne-Zeilinger state via stochastic local operations and classical communication with a rate approaching unity," *Physical review letters*, vol. 112, no. 16, p. 160 401, 2014.

[316] P. Vrana and M. Christandl, "Asymptotic entanglement transformation between W and GHZ states," *Journal of Mathematical Physics*, vol. 56, no. 2, 2015.

[317] P. Vrana and M. Christandl, "Entanglement distillation from Greenberger–Horne–Zeilinger shares," *Communications in Mathematical Physics*, vol. 352, pp. 621–627, 2017.

[318] L. Chen and S. Friedland, "The tensor rank of tensor product of two three-qubit W states is eight," *Linear Algebra and Its Applications*, vol. 543, pp. 1–16, 2018.

[319] M. Christandl and J. Zuiddam, "Tensor surgery and tensor rank," *computational complexity*, vol. 28, pp. 27–56, 2019.

[320] W. Bruzda, S. Friedland, and K. Życzkowski, "Tensor rank and entanglement of pure quantum states," *arXiv preprint arXiv:1912.06854*, 2019.

[321] D. Bini, "The role of tensor rank in the complexity analysis of bilinear forms," *Presentation at ICIAM07, Zürich, Switzerland*, 2007.

[322] D. Bini *et al.*, "$O(n^{2.7799})$ Complexity for $n \times n$ approximate matrix multiplication," 1979.

[323] H. J. Briegel and R. Raussendorf, "Persistent entanglement in arrays of interacting particles," *Physical Review Letters*, vol. 86, no. 5, p. 910, 2001.

[324] J. C. Adcock, S. Morley-Short, A. Dahlberg, and J. W. Silverstone, "Mapping graph state orbits under local complementation," *Quantum*, vol. 4, p. 305, 2020.

[325] A. Cabello, A. J. López-Tarrida, P. Moreno, and J. R. Portillo, "Entanglement in eight-qubit graph states," *Physics Letters A*, vol. 373, no. 26, pp. 2219–2225, 2009.

[326] S. Severini, "Two-colorable graph states with maximal schmidt measure," *Physics Letters A*, vol. 356, no. 2, pp. 99–103, 2006.