SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

On the Optimal Linear Contraction Order of Tree Tensor Networks, and Beyond

Mihail Stoian

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

On the Optimal Linear Contraction Order of Tree Tensor Networks, and Beyond

Über die optimale lineare Kontraktionsreihenfolge von Tree-Tensor-Networks und weitere Aspekte

Author:	Mihail Stoian
Supervisor:	Prof. Dr. Christian B. Mendl
Advisor:	Richard Milbradt, M.Sc.
Submission Date:	15.05.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.05.2023 Mihail Stoian

Acknowledgments

First of all, I would like to thank Prof. Dr. Christian B. Mendl and Richard Milbradt for the great lecture on tensor networks. Second, I would also like to thank Prof. Dr. Thomas Neumann and Bernhard Radke for the remarkable lecture on query optimization. The connections in this thesis would not have been possible without excellent teaching from both sides.

Finally, I would like to thank my parents who never stopped encouraging me to push my boundaries.

Abstract

Tensor networks are nowadays the backbone of classical simulations of quantum many-body systems and quantum circuits. This is because one can contract a corresponding tensor network, which also circumvents the memory problem that often arises in large simulations. While tensor contraction itself is a rather simple operation, the *order* in which the tensors of the network are contracted significantly impacts the time and memory performance. For this reason, one aims at finding a priori an optimal contraction order under a cost function that models the execution time of the whole network contraction.

However, there is a caveat: the problem of finding the optimal contraction is NP-hard. As a consequence, most works either improve over the exponential algorithm or fall back to greedy approaches. We argue that this is a defeatist position and show that we can indeed find optimal contraction orders for a more restrictive class of tensor networks. Namely, we prove that tree tensor networks accept optimal linear contraction orders. The result comes from a fascinating yet non-trivial link between database join ordering and the presented problem. To this end, we adapt a decades-old algorithm to the context of tensor networks.

Beyond the optimality results, we explore whether ad-hoc join ordering techniques aid in providing near-optimal contraction orders for general tensor networks. We empirically validate that our optimizers guarantee robustness for tree tensor networks.

This work extends and builds upon our preprint [1], providing a more in-depth exposition of the optimality result. Moreover, we extend its section on near-optimal optimizers, by providing a thorough study of their effectiveness for tensor networks.

Kurzfassung

Tensornetzwerke sind heute das Rückgrat klassischer Simulationen von Quanten-Vielteilchen systemen und Quantenschaltungen. Das liegt daran, dass man ein entsprechendes Tensornetz kontrahieren kann, wodurch auch das Speicherproblem umgangen wird, das bei großen Simulationen oft auftritt. Während die Tensorkontraktion selbst eine recht einfache Operation ist, hat die Reihenfolge, in der die Tensoren des Netzwerks kontrahiert werden, erhebliche Auswirkungen auf die Zeit- und Speicherleistung. Aus diesem Grund versucht man, a priori eine optimale Kontraktionsreihenfolge unter einer Kostenfunktion zu finden, die die Ausführungszeit der gesamten Netzwerkkontraktion modelliert.

Es gibt jedoch eine Einschränkung: Das Problem, die optimale Kontraktion zu finden, ist NP-hart. Infolgedessen verbessern die meisten Arbeiten entweder den exponentiellen Algorithmus oder greifen auf gierige Ansätze zurück. Wir argumentieren, dass dies eine defätistische Position ist und zeigen, dass wir tatsächlich optimale Kontraktionsreihenfolgen für eine restriktivere Klasse von Tensornetzen finden können. Wir beweisen nämlich, dass Baum-Tensornetzwerke optimale lineare Kontraktionsreihenfolgen akzeptieren. Das Ergebnis ergibt sich aus einer faszinierenden, aber nicht trivialen Verbindung zwischen der Optimierung der Join-Reihenfolge in Datenbanken und dem vorgestellten Problem. Zu diesem Zweck passen wir einen jahrzehntealten Algorithmus an den Kontext von Tensornetzen an.

Über die Optimalitätsergebnisse hinaus untersuchen wir, ob gängige Techniken zur Optimierung der Join-Reihenfolge dazu beitragen, nahezu optimale Kontraktionssequenzen für allgemeine Tensornetzwerke zu liefern. Wir validieren empirisch, dass unsere Optimierer Robustheit für Baum-Tensornetzwerke garantieren.

Diese Arbeit erweitert und baut auf unserem Preprint [1] auf, indem sie eine tiefer gehende Darstellung des Optimalitätsergebnisses liefert. Außerdem erweitern wir den Abschnitt über nahezu optimale Optimierer, indem wir eine gründliche Untersuchung ihrer Effektivität für Tensornetzwerke durchführen.

Contents

1 Introduction

Tensor networks are nowadays an active interdisciplinary field of research. This means not only that they benefit other fields [2, 3, 4, 5], but also that other fields contribute to their further development [6, 7]. Even more surprising is that such fields are even foreign to theoretical physics, the field in which tensor networks were originally developed. On the other hand, with their popularity comes a natural responsibility: in addition to their general applicability, tensor network methods are expected to be *efficient*.

In our context, this amounts to an efficient network contraction operation. It can be thought of as a step-wise transformation of a tensor network into a single tensor. A single step consists of contracting two tensors. This step is repeated until a single tensor remains, which is the result of the network contraction. Although the final result is invariant to the order in which the pairwise tensor contractions are performed, the overall performance is significantly affected. For this reason, one aims at finding a priori an optimal contraction order under a cost function that models the execution time of the whole network contraction.

However, there is a caveat: finding the optimal order is an NP-hard problem [8]. In other words, it is highly unlikely that we will find an efficient, i.e., polynomial-time, algorithm that solves the problem. Therefore, one must settle for a mixture of exponential algorithms for small-size instances, e.g., $n \leq 20$, and otherwise hope for good contraction orders. For this reason, previous research has focused on the latter part, trying to find better heuristics.

We argue that this is a defeatist position and show that we can indeed find optimal contraction orders for a more restrictive class of tensor networks. Instead of resorting to heuristics, we ask ourselves the following question:

> *Which classes of tensor networks accept optimal contraction orders?*

A first answer is represented by chain tensor networks. In fact, its solution is one of the introductory textbook examples for dynamic programming [9] (on which we provide more details in Sec. 4.1). The answer we give in this work considers a larger class, namely that of tree tensor networks, for which we show that they accept optimal *linear* contraction orders.

The result comes from a fascinating yet non-trivial link between database join ordering and our problem. In short, a database query optimizer aims at finding the order in which the joins are to be performed. Similar to our context, this order highly influences the performance of query execution. As a result, much research has been undertaken to develop optimal algorithms for restrictive input classes and otherwise near-optimal algorithms that behave well in practice. We will expand on this connection in the next chapter.

The thesis is structured as follows: Chapter 2 introduces general definitions used in tensor networks and query optimization. Afterward, in Chapter 3, we introduce the TensorIKKBZ algorithm and prove that it outputs optimal linear contraction orders for tree tensor networks. In Chapter 4, we study whether ad-hoc join ordering techniques aid in proving near-optimal contraction orders for general tensor networks. The algorithms are then evaluated in Chapter 5. Finally, we draw conclusions and discuss future work in Chapter 7.

2 Preliminaries

2.1 Tensor Networks

In this work, we regard tensors from a computational perspective, i.e., we treat a tensor as a multi-dimensional array and define it as an element of $\mathbb{C}^{n_1 \times \ldots \times n_d}$, where d is the rank of the tensor. A tensor can be visualized by means of a graphical notation, where the dimensions are pictured as legs. An example of a rank-five tensor in graphical notation is shown in Fig. 2.1.

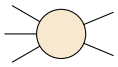

Figure 2.1: A rank-five tensor

A tensor network is defined as a set of n tensors, where we represent tensors as vertices and the legs along which the tensors are to be contracted as edges. Note that some tensors could have *open* legs, represented by edges with only one endpoint. In Fig. 2.2, we draw a tensor network of three tensors in graphical notation. Reading from left to right, both the first and last tensors have open legs. While the second and third tensors share only one common leg, the first and second tensors share two legs.

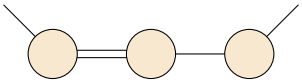

Figure 2.2: A tensor network of three tensors

2.1.1 Tensor and Network Contraction

Let $T^{[1]} \in \mathbb{C}^{p_1 \times \ldots \times p_a \times q_1 \times \ldots \times q_b}$ and $T^{[2]} \in \mathbb{C}^{q_1 \times \ldots \times q_b \times r_1 \times \ldots \times r_c}$ be two tensors. Contracting $T^{[1]}$ and $T^{[2]}$ results in another tensor $T^{[1,2]}$ which preserves only the *common* legs, i.e., $T^{[1,2]} \in \mathbb{C}^{p_1 \times \ldots \times p_a \times r_a \times \ldots \times r_c}$. This corresponds to a product over the common legs, i.e.,

$$T^{[1,2]}_{i_1,\ldots,i_a,k_1,\ldots,k_c} = \sum_{j_1,\ldots,j_b} T^{[1]}_{i_1,\ldots,i_a,j_1,\ldots,j_b} T^{[2]}_{j_1,\ldots,j_b,k_1,\ldots,k_c}. \tag{2.1}$$

To exemplify this, consider two matrices $A \in \mathbb{C}^{p \times q}$ and $B \in \mathbb{C}^{q \times r}$. Their contraction is the well-known matrix multiplication $C_{i,k} = \sum_j A_{i,j} B_{j,k}$ (cf. Eq. 2.1), where $C \in \mathbb{C}^{p \times r}$.

In applications, a contraction of the entire network is required. In Fig. 2.3, we visualize such a contraction of a tensor network of four tensors. In this example, the result is a scalar, as there are no open legs.

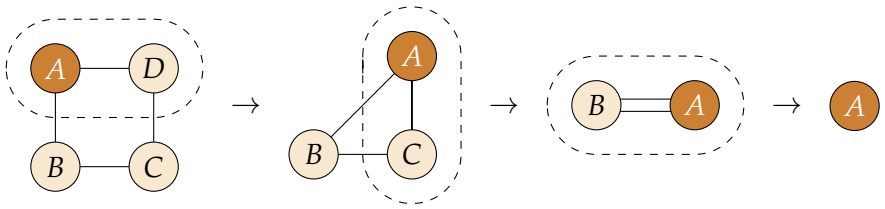

Figure 2.3: Linear contraction order of a tensor network of four tensors

2.1.2 Contraction Cost

While the output of the tensor network is invariant to the order in which the individual tensors are contracted, the execution time of the contraction is highly dependent on the chosen order. The cost of contracting $T^{[1]}$ with $T^{[2]}$ is equal to $c(T^{[1]}, T^{[2]}) = \prod_{i=1}^{a} p_i \prod_{j=1}^{b} q_j \prod_{k=1}^{c} r_k$, i.e., the product of the sizes of all legs involved in the operation. The reason for this is that the number of scalar multiplications in a contraction operation is indeed equal to $c(T^{[1]}, T^{[2]})$. Consequently, the contraction cost of an entire tensor network is defined as the sum of all contraction costs.

Note that the number of additions is not relevant, since they are executed significantly faster by the CPU[1]. Moreover, for simplicity, we ignore the constants involved in multiplying numbers in $\mathbb{C}$. Strictly speaking, one complex multiplication requires four real multiplications. However, the constants are not relevant to the minimization and can be neglected.

To exemplify the calculation, let us consider Fig. 2.3 again and calculate the network contraction cost. For simplicity, assume all legs are of size 2. In the first step, the cost is $2 \cdot 2 \cdot 2 = 8$, as there are three legs involved in the operation. Next, the contraction involves all legs, hence the cost equals again 8. Lastly, the cost is $2 \cdot 2$, as there are two legs between the remaining two tensors. Therefore, the contraction cost equals $8 + 8 + 4 = 20$.

2.1.3 Contraction Order

Complementary to the cost function used, the *structure* of the contraction order has a similar impact. As classified in [10], there are two types of contraction orders: *linear* and *general*. A linear contraction order can only contract a fixed tensor with the others, while a general one is allowed to contract the tensors in an arbitrary order. To exemplify their difference, let us consider the network contraction in Fig. 2.3 again. There, we contract the tensor A each time

[1]The validity of this statement has changed considerably in recent decades. Modern architectures reduce the speed difference between these two processes.

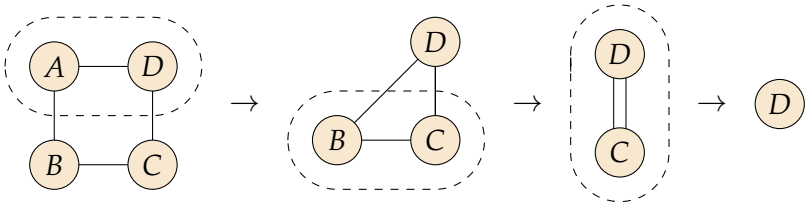

Figure 2.4: General contraction order of a tensor network of four tensors

with one of its neighbors. Thus, this is a *linear* contraction order. In contrast, Fig. 2.4 shows an example of a *general* contraction order for the same tensor network.

In this case, the two contractions between A and D, and B and C, respectively, are performed separately. The final contraction is then carried out between the resulting tensors. Note that the naming convention is not relevant, that is, we could rename the tensor representing the contraction between A and D by A as well.

Due to their flexibility, general orders allow for an exponentially larger search space, which makes them desirable in practical applications as they can provide better contraction costs. However, for the same reason, they are also more difficult to optimize.

2.1.4 Contraction Tree

Complementary to contraction orders, *contraction trees* offer a more intuitive visualization of the order in which the contractions are performed. Formally, a contraction tree is a binary tree the leaves of which are the tensors of the network. Its internal nodes stand for the contractions between the tensors represented by the left and right children, respectively.

We depict in Fig. 2.5 the contraction trees associated with the contraction orders in Fig. 2.3, represented by contraction tree (1), and Fig. 2.4, represented by contraction tree (2), respectively. For convenience, we refer to a contraction tree associated with a linear contraction order as a *linear* contraction tree; similarly, we refer to a *general* contraction tree.

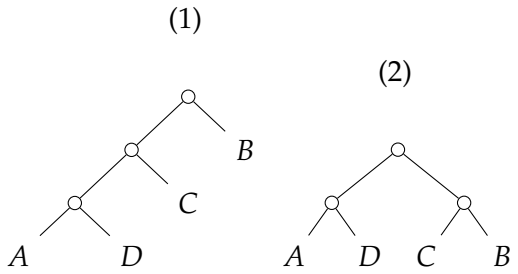

Figure 2.5: Contraction trees: (1) linear, (2) general

2.2 Query Optimization

As pointed out in the introduction, the new results are only possible due to a surprising connection to database query optimization. Database systems provide a mechanism for storing tables and answering queries through a programming language called SQL (structured query language). The underlying mathematical foundation is relational algebra which defines operators on the tables [11].

2.2.1 Query Graph

For simplicity, we will assume that tables have only one column, i.e., can be represented as a set of numbers. The relational operator we are interested in is the binary operator JOIN, which takes two tables A and B as input and outputs another table C containing the values that A and B have in common. A query can contain multiple joins and can therefore be represented graphically as a *query graph*, where the vertices represent the relations, their labels the sizes (cardinalities) of the tables, and the edges the joins between them. In Fig. 2.6, we illustrate a query graph with five relations. The edge weights correspond to join selectivities, and their definition will be introduced later in Sec. 2.3.

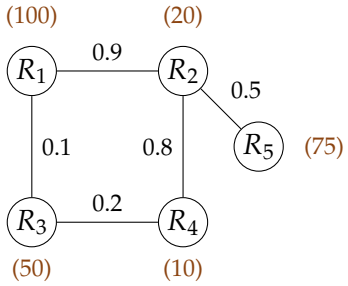

Figure 2.6: A query graph of five relations

2.2.2 Join Ordering

The result of the query consists of a single table that represents the join between all the tables present in the query. Joining the tables in the order specified by the user (by means of a SQL query) can lead to slow execution times. Therefore, the goal of query optimization and especially its subfield, join ordering, is to find or approximate the optimal execution order of the joins when an optimal solution is infeasible.

From a complexity hardness perspective, the feasibility of finding the optimal order highly depends on the shape of the query graphs. As a result, query graphs are grouped in several classes, according to their hardness. Most notably, in this thesis we will mostly deal with chain and tree queries, that is, query graphs having chain and tree shapes, respectively.

2.3 Tensor Contraction as Relational Join

The reader can already notice the similarities between our problem and the one present in join ordering. Indeed, the problems are almost identical: we can interpret a tensor contraction as a relational join. As we will see in the next chapters, the cost function used in query optimization is much more powerful, allowing us to "upgrade" decades-old database algorithms for join ordering to optimize tensor contractions. At the end of this section, we underpin the main differences which are to be considered when adapting join ordering algorithms to tensor networks.

Note that this connection between these two problems has already been highlighted in Dudek et al. [7], where the notion of tensor networks has been associated with factor graphs as well. However, their work does not exploit any database algorithms but relies on graph decompositions, specifically tree decompositions, which are solved by heuristics solvers.

2.3.1 Commonalities

Intuitively, the query optimizer, the database system component responsible for optimizing the query, aims at a join order which minimizes the size of the *intermediate* tables created to obtain the final result. To estimate these sizes, each join has an associated *join selectivity* f, defined as $f = \frac{|R \bowtie S|}{|R||S|}$. For instance, in Fig. 2.6, the selectivity between R_1 and R_2 is 0.9. When there is no common attribute between R and S, the join becomes a *cross-product*. This is similar to the outer tensor product, i.e., there is no common leg to contract the two tensors over. Ideally, an optimizer should aim for finding solutions that could contain outer products, as this allows for even more flexibility. We discuss the particularities of considering outer products during optimization in Chapter 6, where we revisit well-known hardness results from query optimization.

The equivalent of a contraction tree (Sec. 2.1.4) in query optimization is a *join tree*. A join tree is a binary tree with join operators as inner nodes and relations as leaf nodes. The established classes of join trees are left-deep, zigzag, and bushy, where the first two are summarized as linear trees [12]. We illustrate a left-deep tree and a bushy tree, respectively, in Fig. 2.7, as these are the only types we will need throughout this thesis. As the figure already shows, they correspond to *linear* and *general* contraction trees, respectively, so we will only present the results in terms of *linear* and *general*.

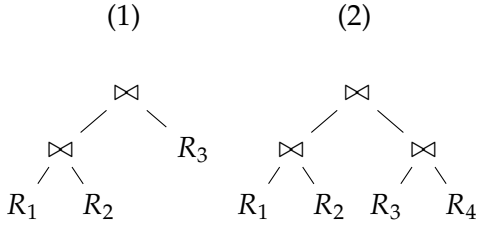

Figure 2.7: Join trees: (1) left-deep (*linear*), (2) bushy (*general*)

2.3.2 Discrepancies

Despite all similarities between the two problems, there are several differences which, at first sight, impede results in join ordering, notably optimality ones, to be applicable to tensor networks:

1) **Graphical Representation.** One prominent difference lies in the graphical representation of a tensor network and a query. In particular, the edge weights in a tensor network represent the leg sizes of the tensors, while in a query graph, they represent the join selectivities that are not directly part of the relations themselves, as is the case in tensor networks.

2) **Contraction Cost.** Orthogonal to the previous issue, there is another notable issue, which, in particular, has to be considered in implementation. Namely, the contraction cost of a set S of tensors depends on the choice of its subsets S_1 and S_2, where $S = S_1 \dot\cup S_2$. This is because the contraction cost depends on the open legs of both S_1 and S_2, which might differ for a different choice S_1' and S_2', with $S = S_1' \dot\cup S_2'$. In contrast, in query optimization, the join cost does not depend on this choice.

3) **Open Legs.** Another particularity of tensor networks is the presence of open legs. Graphically, they are edges with only one end-point and persist in the tensor network. There is no corresponding definition in query optimization.

4) **Hyperedges.** While rare in the context of query optimization, they have become a tool for simplifying the query graph [13]. However, they are only a special type of hyperedge. In particular, they are defined as an unordered pair (A, B), where $A, B \subseteq V$, the vertex set. On the other hand, tensor networks allow arbitrary hyperedges.

In Chapter 3, we address the aforementioned issues by regarding the contraction cost of two tensors from the perspective of query optimization.

2.4 Cost Function

The cost function to be minimized represents another commonality of both problems. Even so, we dedicate a separate section to it, as we need its formal description for later reference.

The most commonly used cost function in tensor networks is defined as the total number of scalar multiplications during all contraction steps, which we (informally) introduced in Sec. 2.1.2. Similarly, in query optimization, the most common cost function is C_{out}, which sums the sizes of all intermediate join results. Since both cost functions are the same, we will refer to them collectively as $\mathcal{C}$. Formally, $\mathcal{C}$ is defined as

$$\mathcal{C}(T) = \begin{cases} 0, & \text{if } T \text{ is a single tensor} \\ \mathcal{C}(T^{[a]}) + \mathcal{C}(T^{[b]}) + c(T^{[a]}, T^{[b]}), & \text{if } T = T^{[a]}T^{[b]}. \end{cases} \tag{2.2}$$

Having introduced the necessary concepts, we are now ready to state the main result of this thesis, namely an algorithm that computes the optimal linear contraction order of a tree tensor network, i.e., a tree-shaped tensor network.

3 Algorithm

First introduced in [14] by Ibaraki and Kameda, the polynomial-time algorithm IKKBZ only operates on *tree* queries and returns the optimal linear join tree without cross-products, under the assumption that the cost function has the adjacent sequence interchange (ASI) property. Translated in the context of tensor networks, this represents the optimal linear order for tree tensor networks, where we disallow outer products. Consequently, if we can show that the cost function of tensor contraction satisfies the ASI property, then we can *directly* employ IKKBZ for tensor networks. Hence, in this chapter, we provide the proof that this is the case, along with the full-fledged algorithm, called TensorIKKBZ, which finds the optimal *linear* contraction order of *tree* tensor networks.

3.1 Precedence Graph

The main idea of the original algorithm stems from the observation that we work solely with linear solutions. In a linear solution, we always start with a tensor and then sequentially contract neighboring tensors. Moreover, as we work on tree-shaped networks, this naturally imposes a *precedence* relation on the tensors of the network. To this end, we can consider each tensor as the first element of the solution and thus root the tree in it. The subsequent to-be-contracted tensors must adhere to the precedence relation, i.e., a child node must not be contracted before its parent. An example of a precedence graph of a tree tensor network is depicted in Fig. 3.1. In that example, the precedence graph enforces that $T^{[4]} \rightarrow T^{[3]}$ and $T^{[4]} \rightarrow T^{[2]}$, i.e., $T^{[4]}$ should be contracted before $T^{[3]}$ and $T^{[2]}$, respectively. In the same manner, we have $T^{[2]} \rightarrow T^{[1]}$ and $T^{[2]} \rightarrow T^{[5]}$.

To achieve optimality, the algorithm roots the tree network in each relation, solves the ordering problem for the obtained rooted tree, and picks the solution which leads to the minimum cost.

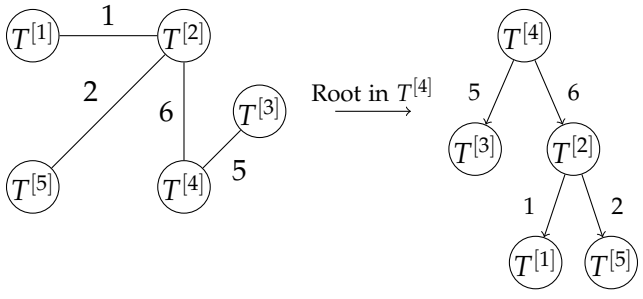

Figure 3.1: Precedence graph of $T^{[4]}$ (right) is obtained by rooting the network (left) in $T^{[4]}$

3.2 ASI Property

Another prerequisite for the optimality of the algorithm is the adjacent sequence interchange (ASI) property of the cost function. Fortunately, our cost function $\mathcal{C}$ has this property, as we shall prove in this section.

3.2.1 Motivation

Let us first motivate the benefit of having such a property by means of an example. Consider the tree tensor network and one of its precedence graphs (that of $T^{[1]}$) in Fig. 3.2, where $T^{[1]} \in \mathbb{C}^{p\times q}$, $T^{[2]} \in \mathbb{C}^{p\times r}$, $T^{[3]} \in \mathbb{C}^{r}$, and $T^{[4]} \in \mathbb{C}^{q}$. In the following, we denote by $T^{[a,b]} := T^{[a]}T^{[b]}$ the contraction of two tensors $T^{[a]}$ and $T^{[b]}$.

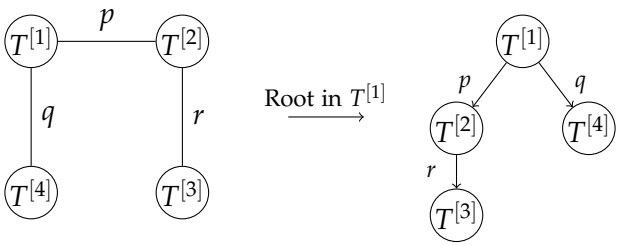

Figure 3.2: Precedence graph of $T^{[1]}$, where $T^{[1]} \in \mathbb{C}^{p\times q}$, $T^{[2]} \in \mathbb{C}^{p\times r}$, $T^{[3]} \in \mathbb{C}^{r}$, and $T^{[4]} \in \mathbb{C}^{q}$

In the following, we want to (manually) unearth a way to decide which of the following linear contraction orders is better: $T^{[1]}T^{[2]}T^{[4]}T^{[3]}$ vs. $T^{[1]}T^{[4]}T^{[2]}T^{[3]}$. Let us first calculate their costs using Eq. 2.2 to recursively decompose the cost until we arrive at contractions between individual tensors, as follows:

$$
\begin{aligned}
\mathcal{C}(T^{[1]}T^{[2]}T^{[4]}T^{[3]}) &= \mathcal{C}(T^{[1]}T^{[2]}T^{[4]}) + \mathcal{C}(T^{[3]}) + c(T^{[1,2,4]}, T^{[3]}) \\
&= \mathcal{C}(T^{[1]}T^{[2]}T^{[4]}) + 0 + c(T^{[1,2,4]}, T^{[3]}) \\
&= \mathcal{C}(T^{[1]}T^{[2]}) + \mathcal{C}(T^{[4]}) + c(T^{[1,2]}, T^{[4]}) + c(T^{[1,2,4]}, T^{[3]}) \\
&= \mathcal{C}(T^{[1]}T^{[2]}) + 0 + c(T^{[1,2]}, T^{[4]}) + c(T^{[1,2,4]}, T^{[3]}) \\
&= \mathcal{C}(T^{[1]}) + \mathcal{C}(T^{[2]}) + c(T^{[1]}, T^{[2]}) + c(T^{[1,2]}, T^{[4]}) + c(T^{[1,2,4]}, T^{[3]}) \\
&= 0 + 0 + c(T^{[1]}, T^{[2]}) + c(T^{[1,2]}, T^{[4]}) + c(T^{[1,2,4]}, T^{[3]}) \\
&= c(T^{[1]}, T^{[2]}) + c(T^{[1,2]}, T^{[4]}) + c(T^{[1,2,4]}, T^{[3]}) \\
&= pqr + qr + r
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}(T^{[1]}T^{[4]}T^{[2]}T^{[3]}) &= \mathcal{C}(T^{[1]}T^{[4]}T^{[2]}) + \mathcal{C}(T^{[3]}) + c(T^{[1,4,2]}, T^{[3]}) \\
&= \mathcal{C}(T^{[1]}T^{[4]}T^{[2]}) + 0 + c(T^{[1,4,2]}, T^{[3]}) \\
&= \mathcal{C}(T^{[1]}T^{[4]}) + \mathcal{C}(T^{[2]}) + c(T^{[1,4]}, T^{[2]}) + c(T^{[1,4,2]}, T^{[3]}) \\
&= \mathcal{C}(T^{[1]}T^{[4]}) + 0 + c(T^{[1,4]}, T^{[2]}) + c(T^{[1,4,2]}, T^{[3]}) \\
&= \mathcal{C}(T^{[1]}) + \mathcal{C}(T^{[4]}) + c(T^{[1]}, T^{[4]}) + c(T^{[1,4]}, T^{[2]}) + c(T^{[1,4,2]}, T^{[3]}) \\
&= 0 + 0 + c(T^{[1]}, T^{[4]}) + c(T^{[1,4]}, T^{[2]}) + c(T^{[1,4,2]}, T^{[3]}) \\
&= c(T^{[1]}, T^{[4]}) + c(T^{[1,4]}, T^{[2]}) + c(T^{[1,4,2]}, T^{[3]}) \\
&= pq + pr + r
\end{aligned}
$$

When comparing both costs, i.e.,

$$
\begin{aligned}
& pqr + qr + r \leq pq + pr + r \\
\iff \quad & pqr + qr \leq pq + pr \\
\iff \quad & pr(q-1) \leq q(p-r),
\end{aligned}
$$

we observe an interesting pattern: the original inequality reduces to a simpler one which only considers the tensors $T^{[2]}$ and $T^{[4]}$. To see this, assume $q > 1$ and $p > r$, such that the inequality sign is maintained. Then the inequality reduces to $\frac{pr}{p-r} \leq \frac{q}{q-1}$. Indeed, p and r are the sizes of the legs of $T^{[2]}$, while q is the only leg of $T^{[4]}$. Hence, a *local* decision is enough to decide whether the order $T^{[1]}T^{[2]}T^{[4]}T^{[3]}$ is better than $T^{[1]}T^{[4]}T^{[2]}T^{[3]}$. That is, we only need the information about the tensors to be interchanged in order to make a global decision.

3.2.2 Definition

The pattern illustrated by the previous example is not arbitrary. It is the core idea behind the ASI property [15], employed by the IKKBZ algorithm:

Definition 1 (ASI Property)**.** *Let A and B be two sequences and U and V two non-empty sequences. We say a cost function C has the adjacent sequence interchange (ASI) property, iff there exists function r such that the following holds*

$$C(AUVB) \leq C(AVUB) \iff r(U) \leq r(V)$$

if $AUVB$ and $AVUB$ satisfy the precedence constraints imposed by a given precedence graph.

Thus, a cost function C with ASI property allows us to use *local* comparisons based on the rank function, i.e., $r(U) \leq r(V)$, to infer comparisons between *global* costs, i.e., $C(AUVB) \leq C(AVUB)$. The IKKBZ algorithm, therefore, makes extensive use of rank comparisons to find optimal orders. Before we prove that $\mathcal{C}$ has the ASI property, we first need to simplify the definition of $\mathcal{C}$, as we deal with *linear* orders and *tree-shaped* tensor networks only.

3.2.3 Simplifications

In the following, we reinterpret the contraction cost of two tensors. Namely, we attribute each tensor $T \in \mathbb{C}^{n_1 \times \ldots \times n_d}$ a *size*, which amounts to the product of its leg sizes, i.e., $|T| = \prod_{i=1}^{d} n_i$. We can then rewrite $c(T^{[1]}, T^{[2]})$ as

$$\begin{aligned} c(T^{[1]}, T^{[2]}) &= \prod_{i=1}^{a} p_i \prod_{j=1}^{b} q_j \prod_{k=1}^{c} r_k \\ &= \frac{\prod_{i=1}^{a} p_i \prod_{j=1}^{b} q_j \prod_{j=1}^{b} q_j \prod_{k=1}^{c} r_k}{\prod_{j=1}^{b} q_j} \\ &= \frac{|T^{[1]}||T^{[2]}|}{\prod_{j=1}^{b} q_j}, \end{aligned} \tag{3.1}$$

where $T^{[1]} \in \mathbb{C}^{p_1 \times \ldots \times p_a \times q_1 \times \ldots \times q_b}$ and $T^{[2]} \in \mathbb{C}^{q_1 \times \ldots \times q_b \times r_1 \times \ldots \times r_c}$. While this transformation is trivial, it is one of the key ingredients in the process of adapting the IKKBZ algorithm to tensor networks. To see why, note that this expression resembles the equation of join selectivity f defined for query optimization (Sec. 2.2). Namely, we can now regard $\frac{1}{\prod_{j=1}^{b} q_j}$, i.e., the inverse product of the sizes of the common legs, as a join selectivity.

The definition of the ASI property (Def. 1) works on sequences. A sequence S represents a permutation π of tensors which specifies in which order the tensors shall be contracted. Since we deal with tree tensor networks and enforce that no outer products emerge, each contraction takes place over a *single* leg. As such, we can rewrite the definition of $\mathcal{C}$ for sequences as follows, where the empty sequence is marked with $\boldsymbol{\epsilon}$:

$$\mathcal{C}(S) = \begin{cases} 0, & S = \boldsymbol{\epsilon} \vee S = T^{[\pi_1]} \\ |T^{[\pi_i]}|, & S = T^{[\pi_i]},\ i \in [n] \setminus \{\pi_1\} \\ \mathcal{C}(S_1) + \frac{\|T^{[S_1]}\|}{|e_{S_1,S_2}|}\mathcal{C}(S_2), & S = S_1 S_2, \end{cases} \tag{3.2}$$

where $T^{[S]}$ is the tensor spanning the tensors of sequence S, $\|T^{[S]}\|$ is the size of $T^{[S]}$ *without* the (single) leg to its parent in the precedence graph, and $|e_{S_1,S_2}|$ is the size of the *single* leg between $T^{[S_1]}$ and $T^{[S_2]}$. Note that in a precedence graph, this edge will always lead to the first tensor occurring in S_2. Therefore, when it is clear from the context, we will write e_{S_2} instead of e_{S_1,S_2}. With this notation, we can rewrite $\|T^{[S]}\|$ as $\frac{|T^{[S]}|}{|e_S|}$ (hence the natural notation as norm).

This reformulation of $\mathcal{C}$ now allows us to easily prove that $\mathcal{C}$ satisfies the ASI property. Before outlining the proof in the next section, we consider an identity for the norm in the spirit of the reinterpretation introduced in Eq. 3.1. Namely, for the tensor $T^{[1,2]}$, i.e. the contraction between $T^{[1]}$ and $T^{[2]}$, it holds that

$$\|T^{[1,2]}\| = \frac{\|T^{[1]}\|\|T^{[2]}\|}{|e_{S_1,S_2}|}. \tag{3.3}$$

This identity follows from the fact that the common leg represented by the edge e_{S_1,S_2} does not matter for either the size or the norm of $T^{[1,2]}$. We draw the attention of the reader familiar with query optimization to the fact that this identity does not hold for join cardinalities.

3.2.4 Proof

Lemma 1. *$\mathcal{C}$ satisfies the ASI property.*

Proof. We recursively expand $\mathcal{C}(AUVB)$ and $\mathcal{C}(AVUB)$ using Eq. 3.2:

$$\begin{aligned}
& \mathcal{C}(AUVB) \leq \mathcal{C}(AVUB) \\
\overset{(3.2)}{\iff} \quad & \mathcal{C}(AUV) + \frac{\|T^{[AUV]}\|}{|e_{AUV,B}|}\mathcal{C}(B) \leq \mathcal{C}(AVU) + \frac{\|T^{[AVU]}\|}{|e_{AVU,B}|}\mathcal{C}(B)
\end{aligned}$$

As both $T^{[AUV]}$ and $T^{[AVU]}$ represent the same tensor and both $e_{AUV,B}$ and $e_{AVU,B}$ refer to the same leg, we have

$$\begin{aligned}
\iff \quad & \mathcal{C}(AUV) \leq \mathcal{C}(AVU) \\
\overset{(3.2)}{\iff} \quad & \mathcal{C}(AU) + \frac{\|T^{[AU]}\|}{|e_{AU,V}|}\mathcal{C}(V) \leq \mathcal{C}(AV) + \frac{\|T^{[AV]}\|}{|e_{AV,U}|}\mathcal{C}(U) \\
\overset{(3.2)}{\iff} \quad & \mathcal{C}(A) + \frac{\|T^{[A]}\|}{|e_{A,U}|}\mathcal{C}(U) + \frac{\|T^{[AU]}\|}{|e_{AU,V}|}\mathcal{C}(V) \leq \mathcal{C}(A) + \frac{\|T^{[A]}\|}{|e_{A,V}|}\mathcal{C}(V) + \frac{\|T^{[AV]}\|}{|e_{AV,U}|}\mathcal{C}(U) \\
\iff \quad & \frac{\|T^{[A]}\|}{|e_{A,U}|}\mathcal{C}(U) + \frac{\|T^{[AU]}\|}{|e_{AU,V}|}\mathcal{C}(V) \leq \frac{\|T^{[A]}\|}{|e_{A,V}|}\mathcal{C}(V) + \frac{\|T^{[AV]}\|}{|e_{AV,U}|}\mathcal{C}(U)
\end{aligned}$$

Note that $\|T^{[AU]}\| = \frac{\|T^{[A]}\|\|T^{[U]}\|}{|e_{AU}|}$, due to Eq. 3.3. Analogous for $\|T^{[AV]}\|$. Hence, we obtain

$$\begin{aligned}
\iff \quad & \frac{\|T^{[A]}\|}{|e_{A,U}|}\mathcal{C}(U) + \frac{\|T^{[A]}\|\|T^{[U]}\|}{|e_{AU,V}||e_{A,U}|}\mathcal{C}(V) \leq \frac{\|T^{[A]}\|}{|e_{A,V}|}\mathcal{C}(V) + \frac{\|T^{[A]}\|\|T^{[V]}\|}{|e_{AV,U}||e_{A,V}|}\mathcal{C}(U) \\
\iff \quad & \frac{\mathcal{C}(U)}{|e_{A,U}|} + \frac{\|T^{[U]}\|}{|e_{AU,V}||e_{A,U}|}\mathcal{C}(V) \leq \frac{\mathcal{C}(V)}{|e_{A,V}|} + \frac{\|T^{[V]}\|}{|e_{AV,U}||e_{A,V}|}\mathcal{C}(U)
\end{aligned}$$

Next, since AUV and AVU must both satisfy the constraints of the precedence graph due to Def. 1, there is no edge between U and V. This is because the underlying precedence graph is a rooted tree, so we cannot have a directed edge between U and V *and* between V and U at the same time, otherwise, a cycle would form. This implies that the edge $e_{AU,V}$ is simply the edge $e_{A,V}$. Analogously for $e_{AV,U}$.

$$
\begin{aligned}
\iff \quad & \frac{\mathcal{C}(U)}{|e_{A,U}|} + \frac{\|T^{[U]}\|}{|e_{A,V}||e_{A,U}|}\mathcal{C}(V) \leq \frac{\mathcal{C}(V)}{|e_{A,V}|} + \frac{\|T^{[V]}\|}{|e_{A,U}||e_{A,V}|}\mathcal{C}(U) \\
\iff \quad & \frac{\mathcal{C}(U)}{|e_{A,U}|}(1 - \frac{\|T^{[V]}\|}{|e_{A,V}|}) \leq \frac{\mathcal{C}(V)}{|e_{A,V}|}(1 - \frac{\|T^{[U]}\|}{|e_{A,U}|}) \\
\iff \quad & \frac{\mathcal{C}(U)}{|e_{A,U}|}(\frac{|e_{A,V}|}{|e_{A,V}|} - \frac{\|T^{[V]}\|}{|e_{A,V}|}) \leq \frac{\mathcal{C}(V)}{|e_{A,V}|}(\frac{|e_{A,U}|}{|e_{A,U}|} - \frac{\|T^{[U]}\|}{|e_{A,U}|}) \\
\iff \quad & \mathcal{C}(U)(|e_{A,V}| - \|T^{[V]}\|) \leq \mathcal{C}(V)(|e_{A,U}| - \|T^{[U]}\|).
\end{aligned}
$$

First, observe that the inequality reduces to one where solely functions of U and V are involved. Hence, the ASI property is satisfied. As for the rank function, we can define $r(S) := \frac{\mathcal{C}(S)}{|e_S| - \|T^{[S]}\|}$, where e_S is the *unique* edge between S (strictly speaking, the first tensor occurring in S) and its parent in the precedence graph. However, this is not allowed because the sign of the inequality is reversed when the denominator is negative. Therefore, we write $r(S)$ as a *symbolic* fraction, letting $n(S) := \mathcal{C}(S)$ and $d(S) := |e_S| - \|T^{[S]}\|$ be its numerator and denominator, respectively. Thus, we evaluate $r(U) \leq r(V)$ as $n(U) \cdot d(V) \leq n(V) \cdot d(U)$. □

3.3 TensorIKKBZ

In this section, we adapt the IKKBZ algorithm to tensor networks using the terms introduced earlier. In the sequel, we refer to the adapted algorithm as TensorIKKBZ.

The algorithm treats each tensor tensor $T^{[i]}$ separately, by executing two consecutive phases:

First Phase: Build the precedence graph of $T^{[i]}$, as described in Sec. 3.1.

Second Phase: Compute the optimal permutation that reflects the order in which the tensors should be contracted. The process of transforming the precedence graph into a permutation is called *linearization* and can be regarded as a recursive method: once the subtrees of the children are linearized, we can obtain the linearization of the whole tree by interleaving (merging) the tensors according to their ranks. The invariant is that the ranks of the tensors must always be in increasing order.

After performing the linearization of a subtree rooted in $T^{[u]}$, it is possible that the rank of $T^{[u]}$ is *greater* than the rank of the first tensor in the linearization, hence violating the above invariant. To this end, the notion of contradictory or conflicting sequences comes into play: whenever the precedence graph requires $A \to B$, but $r(A) \geq r(B)$, A and B must be fused into a new single node. This node represents a *compound tensor* that comprises all tensors in A and B in that order. This operation, called *normalization*, is applied after interleaving the linearizations and preserves the invariant that the ranks in the linearization are in increasing

order. Thus, the only contradictory sequences we might obtain are between the root of a subtree and the first (compound) tensors in the result of the merge step. In that case, several normalization steps are applied, until no contradictory sequence is due.

To see why normalization steps are required for optimality, consider the following Lemma (for reference, see [15, Theorem 2]).

Lemma 2. *Let $\{A, B\}$ be a compound tensor. If $A \to B$ and $r(A) \geq r(B)$, then we find an optimal sequence in which B directly follows A.*

Proof. Assume, by contradiction, that B does not directly follow A. Since $A \to B$, any optimal sequence must be of the form $UAVBW$, with $V \neq \epsilon$.

Case 1. *$r(A) \geq r(V)$. We can exchange V and A without increasing the costs.*

Case 2. *$r(A) \leq r(V)$. Since $r(B) \leq r(A)$, due to transitivity we have $r(B) \leq r(V)$. Thus, we can exchange B and V without increasing the costs.* □

At the end, the reverse operation, called denormalization, is performed on the linearization of the precedence graph, where each compound tensor is replaced by the sequence of tensors it contains. The obtained sequence of tensors represents exactly the optimal permutation for the given precedence graph.

Finally, after applying both phases for each tensor $T^{[i]}$, the linearization with the lowest cost is the optimal solution.

3.3.1 Pseudocode

In Alg. 1, we outline the pseudocode of the TensorIKKBZ algorithm.

Algorithm 1 TensorIKKBZ

```
1:  Input: Tree tensor network 𝒯 = (V, E, c)
2:  Output: Optimal linear contraction order ω
3:  ω ← ϵ
4:  for each T^[i] in V do
5:      Build precedence graph 𝒫_T^[i]
6:      while 𝒫_T^[i] is not linearized do
7:          Select T^[u] whose children are linearized
8:          Interleave their linearizations by rank
9:          Normalize the obtained linearization (Alg. 2)
10:     end while
11:     if ω = ϵ or 𝒞(𝒫_T^[i]) < 𝒞(ω) then
12:         ω ← 𝒫_T^[i]
13:     end if
14: end for
15: return ω
```

The algorithm takes as input the tree tensor network $\mathcal{T}$ and outputs the optimal linear contraction order ω. For each $T^{[i]}$, we build the precedence graph $\mathcal{P}_{T^{[i]}}$ (line 5) and linearize it (lines 6-10). After the precedence graph has been linearized, we calculate its cost and compare it to the best obtained so far (line 11). Finally, we return the optimal order.

The linearization operation resembles the classical problem of merging n sorted lists. The sorted lists are the linearizations of the children c_i of node u, the node corresponding to tensor $T^{[u]}$ in line 7. For an efficient implementation, we employ a min-heap data structure [16].

Subsequently, the normalization operation requires solving contradictory sequences after a linearization. Note that this is the only operation in which we need to update ranks. In Alg. 2, we provide the pseudocode for this operation.

Algorithm 2 Normalization

1: **Input:** Current tensor T and the linearization $\mathcal{L}$ of its subtree
2: **Output:** Normalized linearization $\mathcal{L}$
3: $i \leftarrow 1$
4: **while** $r(T) > r(\mathcal{L}_i)$ **do**
5: Update numerator of $r(T)$: $n(T) \leftarrow n(T) + \frac{\|T\|}{|e_{\mathcal{L}_i}|} n(\mathcal{L}_i)$ (cf. Eq. 3.2)
6: Update denominator of $r(T)$: $d(T) \leftarrow |e_T| - \frac{\|T_{\mathcal{L}_i}\|}{|e_{\mathcal{L}_i}|} \|T\|$ (cf. Eq. 3.3)
7: Fuse T with $\mathcal{L}_i$ as a compound tensor
8: $i \leftarrow i + 1$
9: **end while**

The algorithm receives a tensor T, which will always be tensor $T^{[u]}$ from Alg. 1, along with the linearization $\mathcal{L}$ of its subtree, and performs multiple normalization steps until there is no contradictory sequence left. At each iteration, the rank $r(T)$ must be updated, which is represented by the symbolic fraction $\frac{n(T)}{d(T)}$, as introduced in the proof of Lemma 1. The new numerator $n(T)$ will represent the cost $\mathcal{C}(T)$ after performing the contraction with tensor $\mathcal{L}_i$. To this end, we use Eq. 3.2 to calculate the new cost (line 5). As regards the new denominator $d(T)$, we need to calculate the norm of the newly created tensor, which can be done via Eq. 3.3 (line 6). Finally, we create the compound tensor which comprises both T and $\mathcal{L}_i$ and replaces T in the precedence graph (line 7).

3.3.2 Example

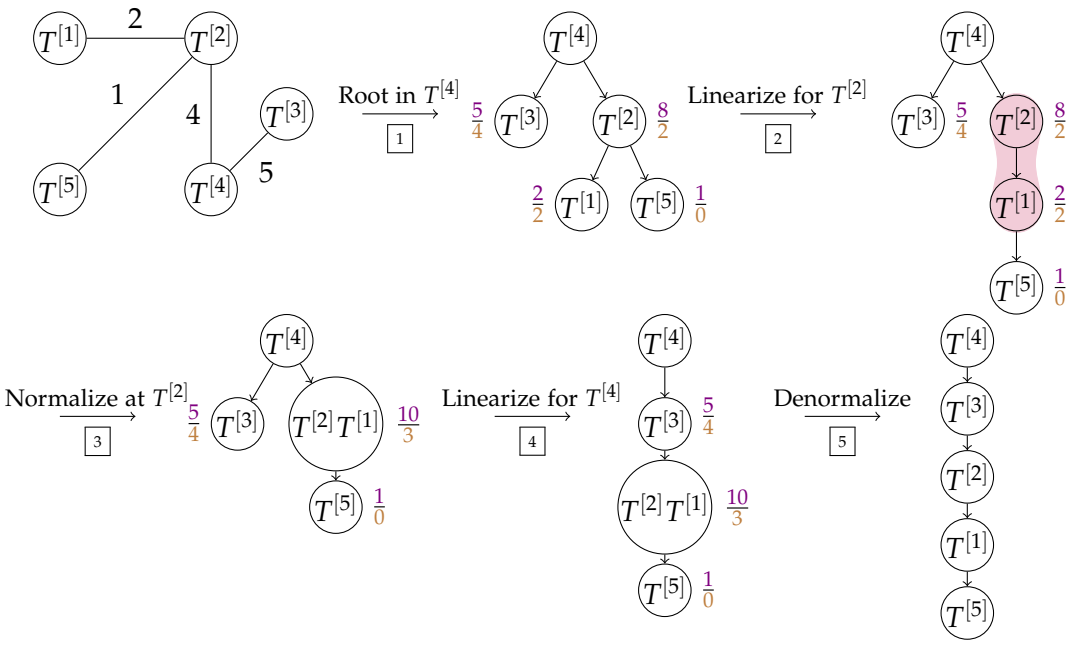

Figure 3.3: Execution of the TensorIKKBZ algorithm for the precedence graph of $T^{[4]}$. Leg sizes are specified in the original tree tensor network, while the ranks are shown as *symbolic* fractions in the upcoming steps, e.g., $r(T^{[3]}) = \frac{5}{4}$. Conflict sequences are highlighted when a normalization is due. In the last step, we obtain the optimal linear contraction order for the precedence graph of $T^{[4]}$.

An execution of the algorithm can be visualized in Fig. 3.3. We are considering only the precedence graph of $T^{[4]}$, but the algorithm would have to consider all precedence graphs. In the first phase, the algorithm builds the precedence graph of $T^{[4]}$, which is equivalent to rooting the tree tensor network in $T^{[4]}$. The subtree of $T^{[2]}$ is not yet linearized, thus the algorithm merges the linearizations of $T^{[1]}$ and $T^{[5]}$, which in this case are the nodes themselves, by observing the rank monotonicity (as pointed out earlier, we evaluate the inequality based on the symbolic fractions representing the ranks). After the merge step, a normalization step is required (step 3), since $T^{[2]} \to T^{[1]}$, but $r(T^{[2]}) \geq r(T^{[1]})$. This creates the compound tensor $T^{[2]}T^{[1]}$ which comprises both $T^{[2]}$ and $T^{[1]}$ in that order. Its rank is calculated as described in Alg. 2.

Afterward, we are able to linearize $T^{[4]}$ and, in the final step, denormalize it. In our case, the compound tensor $T^{[2]}T^{[1]}$ is decomposed into individual tensors, as in step 5. Finally, we obtain the optimal linear contraction order for the precedence graph of $T^{[4]}$. That is, $T^{[4]}$ should be contracted first with $T^{[3]}$, then with $T^{[2]}$, followed by $T^{[1]}$ and $T^{[5]}$.

3.3.3 Time Complexity

The first phase, constructing the precedence graph, can be done by depth-first search in linear time, starting from the root and routing the edges to the unexplored nodes of the tree.

In the second phase, while normalization is a cheap operation, requiring only linear time, linearization is the most expensive one. Interleaving m linearizations of size s each requires $\mathcal{O}(ms \log m)$ operations. This can be done with a min-heap data structure of size m. The worst-case time complexity of each iteration is $\mathcal{O}(n \log n)$ [16], so $\mathcal{O}(n^2 \log n)$ in total, since the algorithm repeats the two phases for each of the n tensors.

Note that in the second of the two papers introducing IKKBZ, a more efficient algorithm was proposed. It reduces the redundancy in the original algorithm, by observing that the linearizations of two neighboring nodes are similar. The optimized algorithm takes $\mathcal{O}(n^2)$-time. The same optimization can be applied to our case. Therefore, we conclude that TensorIKKBZ can be implemented in $\mathcal{O}(n^2)$-time. We refer the reader to the author's Bachelor thesis [17] for details on an efficient implementation of the IKKBZ algorithm.

3.3.4 Discussion

Earlier, Sec. 2.3.2 put forth several issues which impede optimality results from being directly applicable to tensor networks. The key to solving the first two issues resided in the interpretation of the contraction cost from the perspective of query optimization (Eq. 3.1). Surprisingly, the third issue, that of open legs, is also solved by this interpretation. This is because open legs solely contribute to the size of the tensors, i.e., $T^{[1]}$ and $T^{[2]}$, respectively, from Eq. 3.1.

However, the issue of hyperedges remains open. While there is an algorithm in the spirit of IKKBZ for hyperedges [18], it does not guarantee optimality and was only designed for hyperedges present in query optimization (which differ from those present in tensor networks). In Chapter 7, we leave open as future work whether we can extend TensorIKKBZ to hyperedges as defined in the context of tensor networks.

4 Beyond Optimality

In the last chapter, we have mainly dealt with optimal algorithms. To this end, we identified the setting under which we can provide optimal contraction orders. This was the case for *tree* tensor networks and *linear* contraction orders. While this result can already provide practical benefits, in this chapter we aim to close the gap to *general* tensor networks and contraction orders. Inspired by recent research in query optimization, we extend the previous algorithm to find near-optimal contraction trees, given an initial *fixed* permutation of the tensors. Recently, Ibrahim et al. [19] have proposed the same algorithm, however, they resort to heuristics to find the initial permutation. In contrast, we have already presented a way to order tensors, namely by means of the TensorIKKBZ algorithm. Therefore, we can use its linear contraction orders as the initial permutation.

The Chapter is structured as follows: Section 4.1 presents the aforementioned extension. Section 4.2 outlines a last-mile optimizer, already known in the context of join ordering, and newly proposed for tensor networks as well. Additionally, in Section 4.3, we lift the restriction of tree tensor networks and adapt the algorithms to work on general tensor networks by means of an ad-hoc join ordering technique.

Remark that the contribution of this Chapter is to introduce various query optimization techniques to the tensor network community. This will naturally uncover results that appear redundant as they have been discovered in both fields.

4.1 Linearized Dynamic Programming

Recent research in join ordering has attempted to bridge the gap between optimizers for small-size problem instances and those for large-size instances. That is, while there is an optimal algorithm for small-size instances, for medium- and large-size instances only a greedy algorithm is applied, which gives poor results. This setting is also applicable in the context of tensor networks. The goal is to develop an algorithm for medium-sized instances that is competitive with the greedy algorithm.

Textbooks introduce dynamic programming (DP) with the classical problem of optimal matrix parenthesization and present a cubic-time solution. Matrices are only a special case of tensors. Consequently, the parenthesization problem is, in fact, the optimization problem on *chain* tensor networks. To apply the same algorithm, we need to find a way to transform the tensor network into a chain, i.e., to find an order of its tensors. We can then naturally apply the cubic-time algorithm to optimize the parenthesization, which corresponds to the optimal contraction tree with the given order. The beauty of this technique is that it is intended to be general-purpose.

This novel optimizer was first explored in the adaptive optimization framework of Neumann et al. [20] in the context of query optimization. The novelty is to use the IKKBZ algorithm as a seed for the initial tensor orders. The motivation behind this approach is that IKKBZ already provides the optimal linear order (especially for trees). Hence, the final result will *always* be better than or equal to the linear solution. This method is called linearized dynamic programming (LinDP) and has been shown to yield near-optimal results. Recently, Ibrahim et al. [19] employed the same cubic-time algorithm in the context of tensor networks. However, the initial orders used in their work are still greedy and thus cannot provide any theoretical guarantees. In the following, we adapt LinDP for tensor networks. In particular, we show how TensorIKKBZ can be used as a seed for LinDP to provide near-optimal general contraction orders for tree tensor networks.

4.1.1 Pseudocode

In Alg. 3, we outline the pseudocode of LinDP [20], adapted to the context of tensor networks. In principle, one can only run the cubic-time algorithm on the optimal linear order provided by TensorIKKBZ. Note, however, that TensorIKKBZ considers *each* tensor as the root of the precedence graph. As a consequence, we obtain n many such linearizations. We can take advantage of this fact and consider each linearization as an initial order. This gives the algorithm more flexibility to find better general contraction orders.

Overview. More precisely, the algorithm aims to find the optimal contraction tree given an initial permutation π of the tensors. To this end, it exploits Bellman's optimality condition [21]. Intuitively, the condition says that if a problem P has an optimal solution $S := S_1 S_2$, then both S_1 and S_2 are optimal solutions for P_1 and P_2, respectively, with $P = P_1 P_2$. In our context, this means that every contraction subtree of the original contraction tree must be an optimal contraction tree for the tensors it contains.

Algorithm. Once the precedence graph $\mathcal{P}_{T^{[i]}}$ of $T^{[i]}$ has been linearized, the key idea is to compute $\mathcal{C}(\cdot)$ for all intervals $[i, j]$, $i \leq j$, using Eq. 2.2. These values are stored in the dynamic programming table `dp` of size $n \times n$. For the base case in Eq. 2.2, we initialize $\texttt{dp}[i, i]$ to 0, $\forall i \in [n]$ (line 6). Subsequently, we fix an interval size s and iterate all intervals of that size. Then, by employing Bellman's optimality principle, we have to guess where the problem $[i, j]$ splits. This is represented by the index k, for each of which we compute the cost (cf. Eq. 2.2) and compare with the current minimum obtained for $[i, j]$ (lines 11-12). If the current k improves the cost, we store it in the table `opt` (of size $n \times n$) which will be used to reconstruct the contraction tree of optimal cost. Finally, after the cubic-time algorithm has finished, we update the contraction tree τ in case the cost $\texttt{dp}[1, n]$, which represents the optimal cost of the interval $[1, n]$ for the given linearization, is better. If so, we reconstruct the new contraction tree using the table `opt` (line 20).

Time Complexity. The fact that we have to consider all TensorIKKBZ linearizations makes LinDP an expensive algorithm. Namely, its time complexity is $\mathcal{O}(n^4)$, since we have to run the cubic-time algorithm for each tensor. This problem can be partially solved by parallelizing the main for-loop (line 4). This guarantees acceptable optimization times for medium-sized tensor networks. We will use this variant in the benchmarks of Chapter 5.

Algorithm 3 LinDP [20]

Input: Tensor network $\mathcal{T} = (V, E, c)$
Output: General contraction tree τ
$\tau \leftarrow \boldsymbol{\epsilon}$
for each $T^{[i]}$ in V **do**
    $\pi \leftarrow$ Linearization of precedence graph $\mathcal{P}_{T^{[i]}}$ (cf. TensorIKKBZ)
    $\texttt{dp}[i,\, i] = 0, \forall i \in [n]$
    for each $s \in [2, \ldots, n]$ **do**
        for each $i \in [1, \ldots, n - s + 1]$ **do**
            $j \leftarrow i + s - 1$
            for each $k \in [i, j[$ **do**
                $c' \leftarrow \texttt{dp}[i,\, k] + \texttt{dp}[k + 1,\, j] + c(T^{[\pi_i, \ldots, \pi_k]}, T^{[\pi_{k+1}, \ldots, \pi_j]})$
                if $c' < \texttt{dp}[i,\, j]$ **then**
                    $\texttt{dp}[i,\, j] = c'$
                    $\texttt{opt}[i,\, j] = k$
                end if
            end for
        end for
    end for
    if $\tau = \boldsymbol{\epsilon}$ or $\texttt{dp}[1,\, n] < \mathcal{C}(\tau)$ **then**
        $\tau \leftarrow$ Reconstruct contraction tree from `opt`
    end if
end for
return τ

4.2 Iterative Dynamic Programming

In this Section, we illustrate another ad-hoc join ordering technique. The goal of Iterative Dynamic Programming (IDP) [22] is to refine the solution found by an arbitrary optimizer. In its original setting, it was used in join ordering to optimize the join tree found by a heuristic optimizer. In our context, this translates to optimizing the contraction tree found by an arbitrary optimizer. The key idea is to find an expensive subtree of the contraction tree, optimize it via an exact algorithm, contract the subtree, and repeat this process until the contraction tree contains only one node.

It was recently used in Neumann et al. [20], where it was employed for large-size join ordering instances. In the context of tensor networks, the same idea was introduced in Huang et al. [23] and in the implementation of `cotengra` [24]. Thus, this is another algorithm that has been developed independently in both areas. In the following, we outline the general idea of the algorithm, while in Sec. 5.2.2, we evaluate it using the implementation of [24].

4.2.1 Pseudocode

As its name suggests, the algorithm iteratively optimizes subproblems using exact dynamic programming. In Alg. 4, we outline the pseudocode of IDP. It receives as input the tensor network, a contraction tree τ thereof, which has been output by any optimizer, and a threshold k, which specifies the subtree size one can optimize via an exact dynamic programming optimizer. Finally, IDP will return a refined contraction tree with (possibly) better cost.

Algorithm 4 IDP [22]

1: **Input:** Tensor network $\mathcal{T} = (V, E, c)$, contraction tree τ, threshold k
2: **Output:** Refined contraction tree τ
3: $\tau' \leftarrow \tau$
4: **while** $|\tau'| > 1$ **do**
5: $\quad \sigma \leftarrow$ Expensive subtree of τ
6: $\quad \sigma' = \texttt{ExactDynamicProgramming}(\sigma)$
7: $\quad$ In τ, replace σ with σ', i.e., $\tau[\sigma] = \sigma'$
8: $\quad$ In τ', consider σ' as *base* tensor
9: **end while**
10: **return** τ

Algorithm. At each iteration, it considers the current contraction tree τ' (initially set to τ) and searches for a sub-tree σ whose size is below the threshold k and which is prone to be non-optimal, that is, expensive. Deciding this is as difficult as the original problem, however, such subtrees tend to create an imbalance in the contraction tree, e.g., one of the children has a significantly higher cost than its sibling. Note that the meaning of "expensive" subtree (line 5) is intentionally left open. In experiments, we use the contraction cost $\mathcal{C}$ as an indicator of how expensive a subtree is. This is indeed the default choice in `cotengra`.

Once the subtree σ is fixed, we can optimize it by employing exact dynamic programming on the subnetwork induced by the tensors spanned by σ. This will return a new contraction tree σ' with $\mathcal{C}(\sigma') \leq \mathcal{C}(\sigma)$ (line 6), which will substitute σ in the original contraction tree, τ (line 7). To allow further optimization, we replace σ' in τ' with a single tensor (line 8). In particular, this reduces the size of τ' by $k - 1$. This refinement is repeated until τ' contains only one tensor, i.e., its root.

Time Complexity. Assuming that exact dynamic programming takes $O(3^k)$ time, where k is the number of tensors in the network (this is the time complexity of an optimizer for general contraction orders which does not consider outer products, as analyzed in Moerkotte et al. [25]), the time complexity of IDP is $O((n/k)3^k)$. This leads to a trade-off between the number of tensors that can be exactly optimized for each iteration, which implies a higher cost reduction, and the optimization time of IDP. In the experiments, we use the default value of `cotengra` for k, namely $k = 8$.

4.3 General Tensor Networks

While the last two optimizers, LinDP and IDP, are applicable to general tensor networks (by general we mean that we are not restricted to trees), we have not explicitly described how one can leverage the TensorIKKBZ algorithm to work on any tensor network. This is especially necessary for LinDP since it requires computing the initial permutations via the TensorIKKBZ algorithm.

Remember that the TensorIKKBZ requires that the input is a tree tensor network. As such, we need a mechanism to transform the original tensor network into a tree tensor network. A simple solution is to consider a *spanning tree* of the network and use it as input. Indeed, this is a common join ordering technique in query optimization, introduced in the second IKKBZ paper as a way to deal with arbitrary query graphs [26]. While this technique works well in practice in query optimization, the results of experiments in Sec. 5.2 show us that this is not the case for tensor networks. In the following, we underline why this argument is not directly applicable.

Query graphs tend to be acyclic. Tensor networks, on the other hand, usually have a lattice structure. When a spanning tree is selected from a graph, it naturally loses several edges between nodes that are not directly connected in the spanning tree, i.e., non-tree edges. Consequently, TensorIKKBZ is not aware of the non-tree edges and therefore overlooks many contractions. This issue worsens with the number of edges in the graph.

Despite this limitation, we also evaluate this technique in Sec. 5.2.2. In the experiments, we select the *maximum* spanning tree. The reason for this heuristic is as follows. In the second IKKBZ paper [26], the *minimum* spanning tree was used. Given the reformulation we introduced in Eq. 3.1, namely that we can regard the inverse product of the sizes of the common legs as a join selectivity, the choice of the maximum spanning tree is a natural adaptation of the original heuristic to the context of tensor networks.

5 Evaluations

In this Chapter, we evaluate the algorithms presented so far. The main goals are twofold: first, we emphasize the advantage of having an optimal algorithm as a starting point (even one for a more restrictive class of tensor networks). Second, we explore the benefits of ad-hoc join ordering techniques for tensor networks. Therefore, the Chapter is structured as follows: Section 5.1 contains evaluations for tree tensor networks, while Section 5.2 considers general tensor networks, in particular those representing quantum circuits.

Both TensorIKKBZ and LinDP are implemented in C++ and wrapped with Python. In addition, DPccp [25], the standard exact optimizer for join ordering, is also included. All other algorithms are available directly in `opt_einsum` [27] and `cotengra` [24]. To distinguish them, we prefix their names with `oe` and `ctg`, respectively.

The experiments were run on an Ubuntu server with an Intel Ivy Bridge 2.20 GHz CPU with 20 cores and 256 GB of main memory. In the experiments, we let the algorithms fully exploit the potential of parallelization. Namely, for `opt_einsum` algorithms, we set `parallel` to `multiprocessing.cpu_count()`, which is 40. Similarly, for `cotengra` we set `parallel=True`.

5.1 Tree Tensor Networks

We benchmark two classes of tree tensor networks: FTPS (Fork Tensor-Product States) and TTN (Tree Tensor Network, also known as Hierarchical Tucker). For each tensor network class, we provide a separate benchmark for the case where open legs are present and not, respectively. In each case, for the inter-tensor legs, we seed a leg size and randomly assign powers of that size to each leg. For open legs, we set a constant leg size of 2. Note that for tensors with more than one open leg, we bundle them into a single one, by multiplying their sizes.

As optimizers we choose TensorIKKBZ to emphasize that linear contraction orders, although not sufficient, provide robustness, LinDP, to underscore the benefits of building on optimal algorithms, DPccp and the dynamic programming algorithm of `opt_einsum`, `oe.DynamicProgramming`, to show the lower bound of contraction costs for small instances, and the random-greedy optimizer from `opt_einsum`, `oe.RandomGreedy`, which improves on the naïve greedy algorithm by repeatedly sampling greedy paths and selecting the best one. To ensure fairness, we let `oe.RandomGreedy` sample n times, since both TensorIKKBZ and LinDP generate n orders themselves.

5.1.1 FTPS

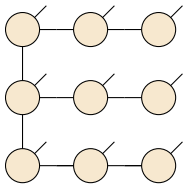

Figure 5.1: FTPS: Graphical notation

The recently introduced Fork Tensor-Product State (FTPS) has been used as a solver for multi-orbital dynamical mean-field theory [28]. Its graphical notation is shown in Fig. 5.1, where a FTPS with nine tensors is drawn. In Fig. 5.2 and Fig. 5.3 we plot the contraction costs and optimization times, respectively. We observe that TensorIKKBZ and LinDP provide better contraction orders for the instances without open legs, which is due to the optimality of the linear contraction orders that TensorIKKBZ outputs. Since LinDP is an inherently expensive algorithm, it is faster than the random-greedy optimizer only up to 150 tensors. This threshold is similar to the threshold in Neumann et al. [20], which is 100. Note that since we enable parallelization, we can optimize larger instances.

For instances with open legs, the contraction cost is much higher (even though we set the size of all open legs to 2). This is because, by definition, the open legs are maintained during network contraction. In this case, LinDP is on par with `oe.RandomGreedy`. However, in terms of optimization time, LinDP is faster up to 175 tensors.

5.1.2 TTN

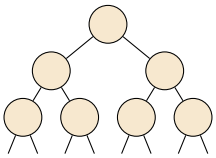

Figure 5.4: TTN: Graphical notation

TTN is a popular class of tensor networks that has the form of a complete binary tree. For example, it has recently been used in generative modeling [29]. Its graphical notation is shown in Fig. 5.4, where a TTN with seven tensors is drawn. Note that a TNN has open legs only at the leaf tensors. In Fig. 5.5 and Fig. 5.6 we plot the contraction costs and the optimization times, respectively. First, we observe that the linear contraction orders provided by TensorIKKBZ are not sufficient. Despite this, LinDP manages to outperform the greedy optimizer on all TTN instances. For the instances with open legs, we see the same scenario as for FTPS, namely that the contraction costs are of the same order of magnitude.

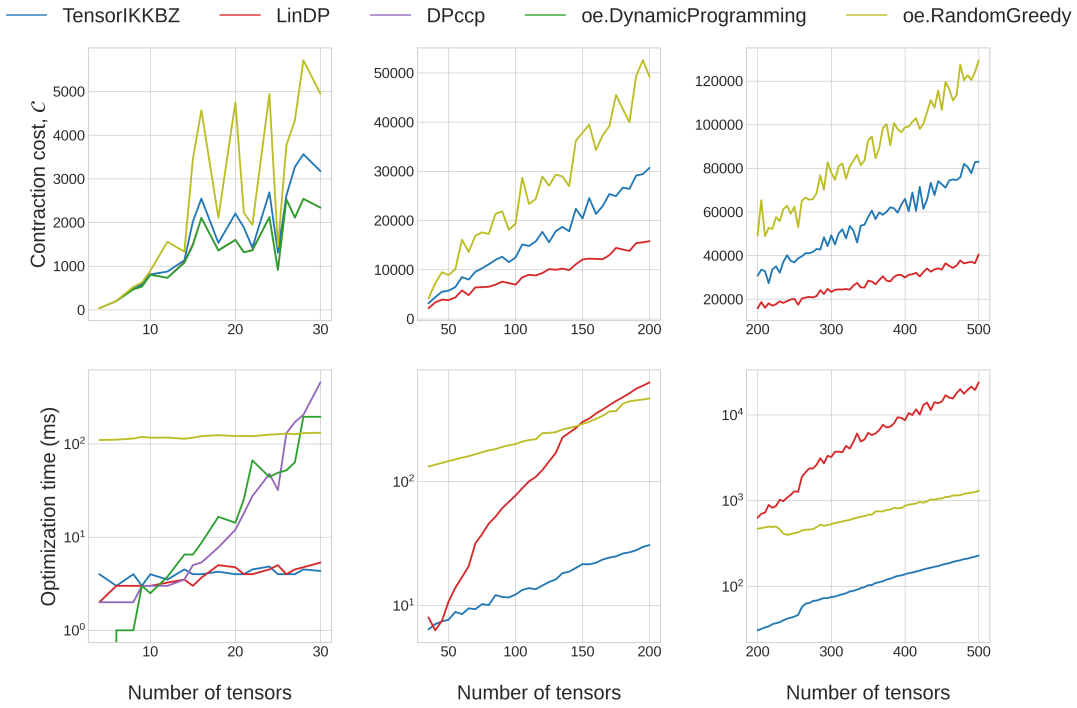

Figure 5.2: FTPS: Experiments without open legs

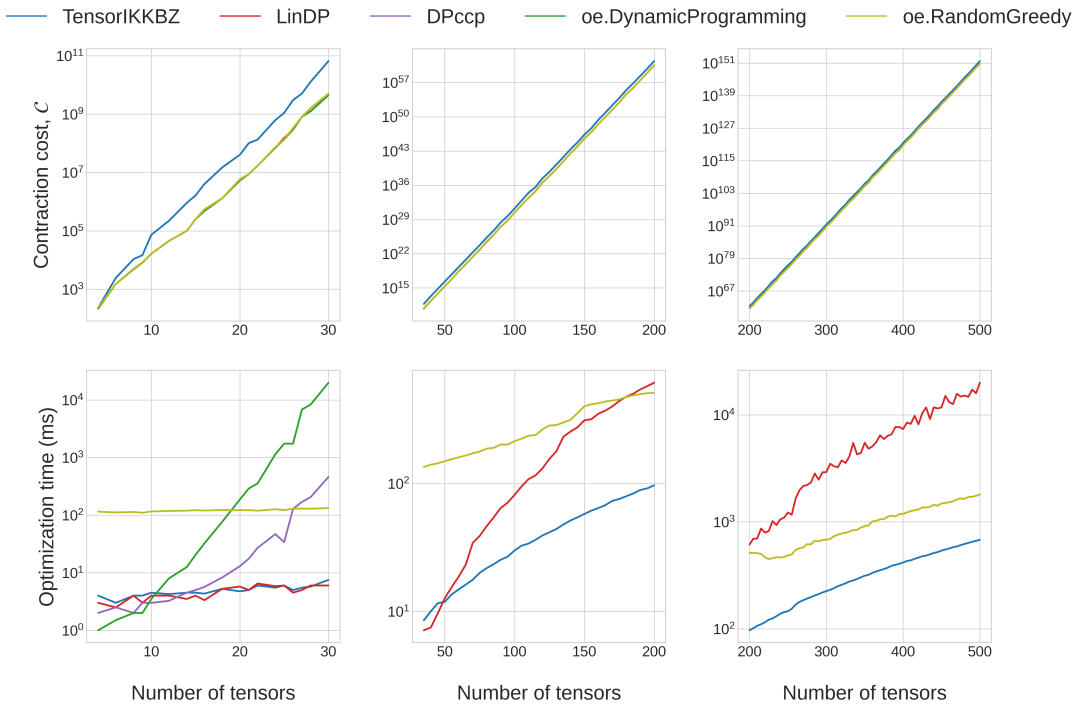

Figure 5.3: FTPS: Experiments with open legs

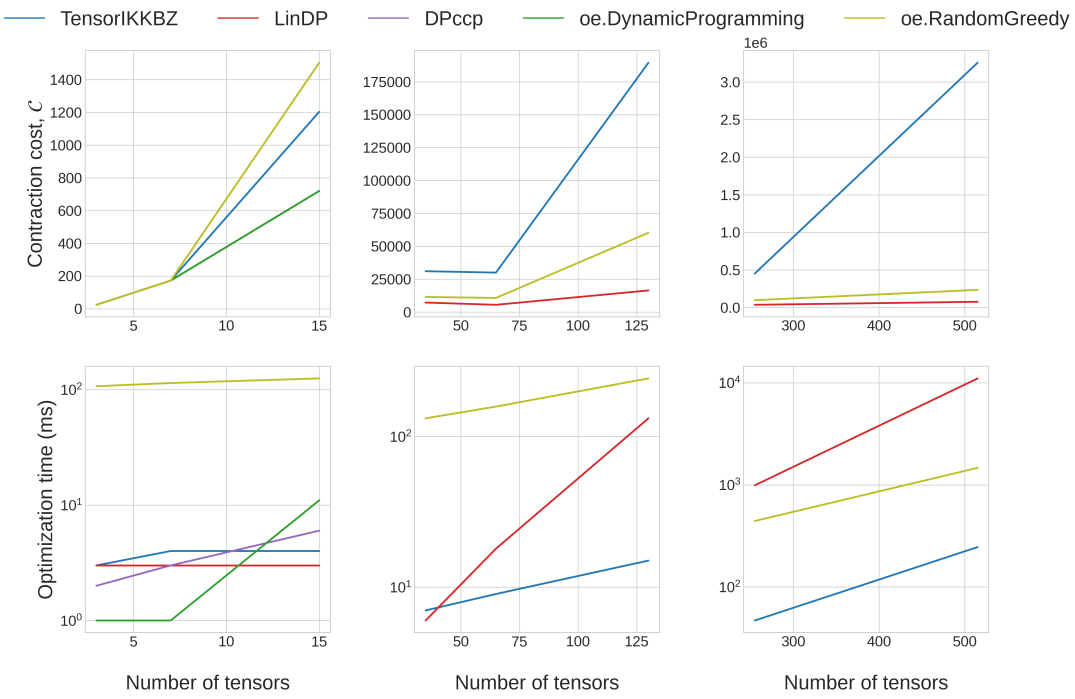

Figure 5.5: TTN: Experiments without open legs

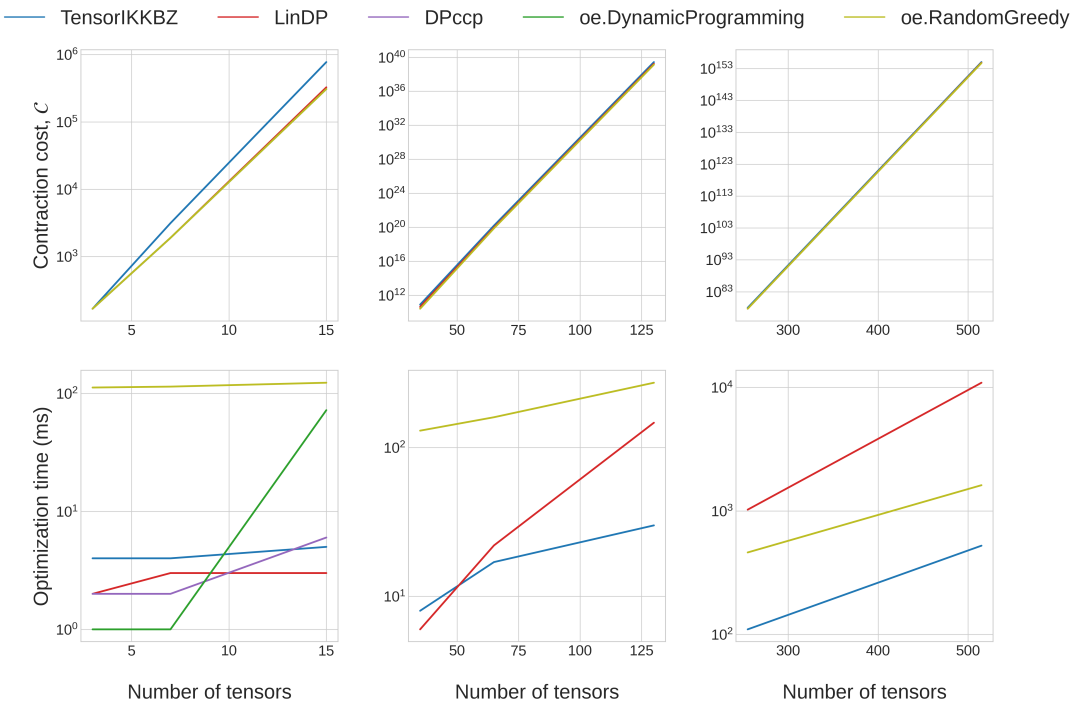

Figure 5.6: TTN: Experiments with open legs

5.2 General Tensor Networks

5.2.1 PEPS

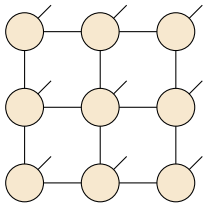

Figure 5.7: PEPS: Graphical notation

PEPS is a generalization of the Matrix-Product State (MPS) to a 2D lattice. Its graphical notation is shown in Fig. 5.7, where a PEPS with nine tensors is drawn. Note that, since TensorIKKBZ and LinDP are required to operate on tree tensor networks, we need to extract a spanning tree of the underlying tensor network. As discussed in Sec. 4.3, we select the maximum spanning tree. In Fig. 5.8 and Fig. 5.9, we plot the contraction costs and optimization times, respectively. Notably, TensorIKKBZ and LinDP do not provide near-optimal contraction orders. This is due to the fact that TensorIKKBZ is forced to work on a spanning tree of the original graph. This naturally excludes an important number of contraction orders.

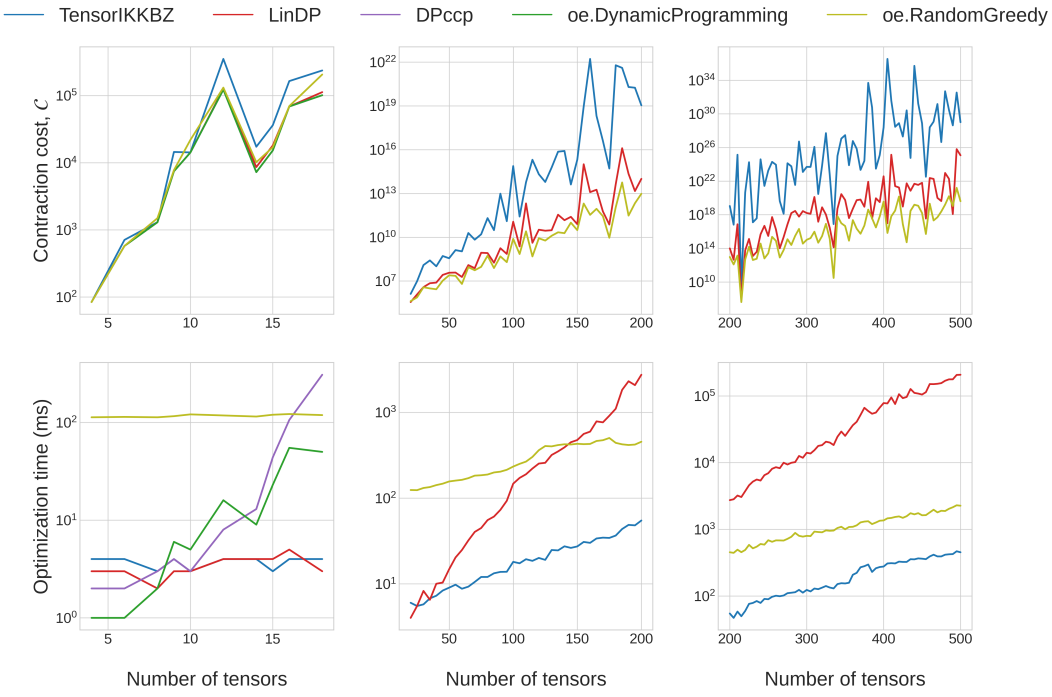

Figure 5.8: PEPS: Experiments without open legs

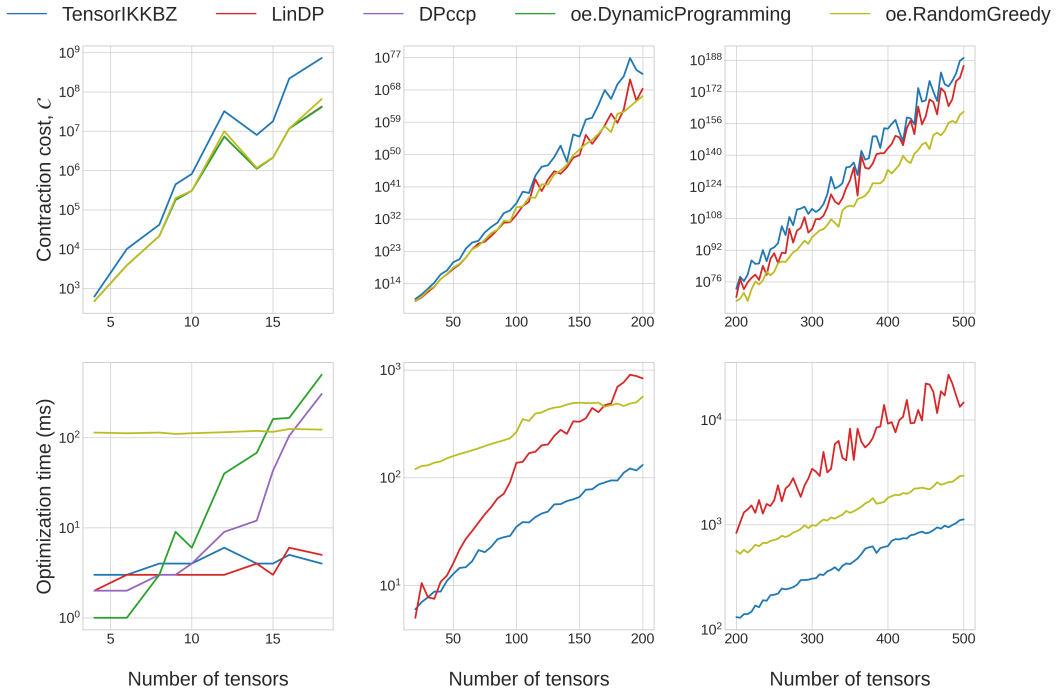

Figure 5.9: PEPS: Experiments with open legs

5.2.2 Sycamore

The Sycamore architecture is the one used and executed in [30]. As shown in [31], one can represent the quantum circuit as a tensor network. In the paper introducing `cotengra` [24], Gray et al. developed a novel tensor network optimizer built on top of the hypergraph partitioner `KaHypar` [32]. As discussed in Sec. 4.2, we enable subtree reconfiguration, and suffix the optimizers where we have enabled it with "+". For both `ctg.Hyper-Greedy` and `ctg.Hyper-Par`, we use the default setting `max_repeats=128`. In Fig. 5.10 we plot the contraction costs and optimization times.

We remark TensorIKKBZ and LinDP do not achieve the contraction cost of `ctg.Hyper-Par`. This is because the underlying tensor network representing the circuit is not a tree tensor network. The contraction cost obtained by `ctg.Hyper-Par` for 20 cycles, namely $\log \mathcal{C} \approx 18$, is close to optimal given the current results. We are aware of two other works which achieve the same order of magnitude, namely: Huang et al. [23] report 6.66×10^{18}, while Pan et al. [31] report 4.51×10^{18}.

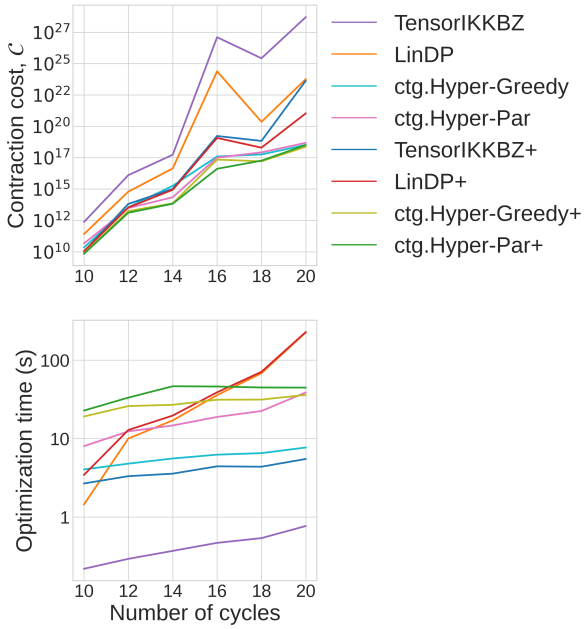

Figure 5.10: Sycamore circuit: contraction cost and optimization time. We suffix with + the optimizers for which we enabled subtree reconfiguration in `cotengra`.

6 Related Work

This Chapter summarizes related work in the fields of tensor network contraction ordering and database join ordering by placing them side by side. In addition, open problems and hardness results in both areas are described.

6.1 Algorithms

Join ordering has a longer history than tensor network contraction ordering. This is not surprising, since relational database systems began to appear in the 1970s, bringing with them the need for efficient optimizers. Apart from the standard dynamic programs introduced by Selinger et al. [33], there are many heuristics, a notable one being GOO [34], which is the same algorithm in tensor network folklore. Kossmann et al. [22] proposed Iterative Dynamic Programming, which makes local improvements to the greedy order. We have described it in Sec. 4.2.

Later, in the tensor network community, a prominent work that emerged is that of Pfeifer et al. [35], which introduces a graph-theoretic dynamic program. The algorithm works in a top-down manner, while in [25], Moerkotte et al. present a bottom-up approach to join ordering, namely DPccp (which is also used in our benchmarks). Recently, Neumann et al. [20] presented the adaptive optimization framework, along with the novel optimizer LinDP, which we also studied for tensor networks in this thesis. Lately, Ibrahim et al. [19] proposed the same idea, namely running the cubic-time dynamic programming algorithm on a permutation of the tensors. Unlike LinDP, they rely on heuristics to find the initial permutation, which does not come with any guarantee for the final contraction order. The work closest to ours is that of Xu et al. [36], which provides a polynomial-time algorithm for minimizing the *time* complexity, in terms of $\mathcal{O}$-notation, of a tree tensor network contraction. Note that this cost function is only a proxy for the original one, $\mathcal{C}$. On the contrary, our work optimizes $\mathcal{C}$ directly.

Newly, learned optimizers have emerged. In the context of join ordering, the first work is that of Marcus et al. [37]. The work of Meirom et al. [38] also uses reinforcement learning and proposes a learned optimizer for tensor networks.

Several of these algorithms have been implemented in general-purpose frameworks, such as `opt_einsum` [27] and `cotengra` [24].

While this section is only a summary of work done in both areas, it is also a motivation for future work to consider the problem in a holistic way, by appropriating techniques developed in both areas.

6.2 Unifying Framework

Our work is not the first to highlight the similarity of the two problems. The first work we know of is that of Khamis et al. [39], which includes several problems from database theory, tensor networks, and probabilistic graphical models under the same umbrella. Later, Dudek et al. [7] reiterate the similarity between these specific instances of the problem by providing solvers that use graph decompositions, specifically tree decompositions.

6.3 Open Problems

Apart from common algorithms, both fields also share the same open problems. A notable open problem, which is related to our work, is that of general contraction orders of tree tensor networks [36]. In the same manner, in query optimization, finding the optimal bushy join tree for tree query graphs is still unsolved [12].

Since join ordering is an older problem, there are hardness results for several instances of the problems. In particular, Cluet et al. [40] proved that the problem is NP-hard if we consider cross products, even for star graphs, which are in particular trees. This underlines the fact that considering cross-products is not feasible. Later, Scheufele et al. [41] showed that the problem is already NP-hard if we only consider cross-products on query graphs, i.e., if the join selectivities are all 1.

Note that we do not claim that the following hardness results hold in the context of tensor networks. A proof is needed to guarantee that they hold in this setting as well. This is left as future work.

7 Conclusion & Future Work

We proved that tree tensor networks accept linear contraction orders. To this end, we have provided an algorithm called TensorIKKBZ, which is an adaptation of a decades-old algorithm from database join ordering. Its requirement is that the cost function satisfies the adjacent sequence interchange (ASI) property. We proved that the cost function used in the context of tensor networks does indeed satisfy this property, and modified the mathematical foundations of the algorithm to work in this setting.

Beyond the optimality result, we have extended TensorIKKBZ by employing a recent optimizer in join ordering, called LinDP, which given an initial permutation of the tensors, outputs the optimal general contraction order. Its advantage is that while it is not guaranteed to be optimal, it provides robustness by building on the contraction orders provided by the TensorIKKBZ algorithm. We have further adapted these algorithms to work on general tensor networks by using an ad-joc join ordering technique, namely that of selecting a spanning tree of the underlying graph as input to the optimizers. However, tensor networks tend to be highly cyclic. In contrast, the graphs encountered in query optimization are mainly acyclic, which is why this technique is not effective for tensor networks. We have empirically shown that TensorIKKBZ and LinDP provide robust contraction orders for tree tensor networks compared to greedy optimizers.

Finally, we have summarized related work in both fields, highlighting open problems and hardness results common to both fields. Our work thus facilitates the integration of results from one field into the other.

As future work, we see the following directions:

1) *Hyperedges.* Extend TensorIKKBZ to hyperedges (fourth point in Sec. 2.3.2). The work in Radke et al. [18] is not directly applicable because the type of hyperedges in query optimization is different from that in tensor networks.

2) *Spanning tree.* Is there a way to choose a spanning tree for TensorIKKBZ so that the final contraction cost is close to that of the original tensor network?

3) *Hardness results.* Do the hardness results in query optimization presented in Sec. 6.3 also hold for tensor networks?

4) *Open problem.* Is the problem of general contraction orders of tree tensor networks NP-hard? This is currently an open problem in both areas.

List of Figures

Bibliography

[1] M. Stoian. *On the Optimal Linear Contraction Order for Tree Tensor Networks*. 2023. arXiv: 2209.12332 [quant-ph].

[2] A. Cichocki, N. Lee, I. Oseledets, A.-H. Phan, Q. Zhao, and D. P. Mandic. "Tensor Networks for Dimensionality Reduction and Large-scale Optimization: Part 1 Low-Rank Tensor Decompositions". In: *Foundations and Trends® in Machine Learning* 9.4-5 (2016), pp. 249–429. ISSN: 1935-8237. DOI: 10.1561/2200000059. URL: http://dx.doi.org/10.1561/2200000059.

[3] A. Cichocki, A.-H. Phan, Q. Zhao, N. Lee, I. Oseledets, M. Sugiyama, and D. P. Mandic. "Tensor Networks for Dimensionality Reduction and Large-scale Optimization: Part 2 Applications and Future Perspectives". In: *Foundations and Trends® in Machine Learning* 9.6 (2017), pp. 431–673. ISSN: 1935-8237. DOI: 10.1561/2200000067. URL: http://dx.doi.org/10.1561/2200000067.

[4] I. L. Markov and Y. Shi. "Simulating Quantum Computation by Contracting Tensor Networks". In: *SIAM Journal on Computing* 38.3 (2008), pp. 963–981. DOI: 10.1137/050644756. eprint: https://doi.org/10.1137/050644756. URL: https://doi.org/10.1137/050644756.

[5] E. M. Stoudenmire and D. J. Schwab. "Supervised Learning with Tensor Networks". In: *NIPS*. 2016.

[6] M. Abo Khamis, H. Q. Ngo, and A. Rudra. "FAQ: questions asked frequently". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2016, pp. 13–28.

[7] J. M. Dudek, L. Dueñas-Osorio, and M. Y. Vardi. *Efficient Contraction of Large Tensor Networks for Weighted Model Counting through Graph Decompositions*. 2020. arXiv: 1908.04381 [cs.DS].

[8] C.-C. Lam, P. Sadayappan, and R. Wenger. "On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution". In: *Parallel Process. Lett.* 7 (1997), pp. 157–168.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[10] B. O'Gorman. "Parameterization of Tensor Network Contraction". en. In: (2019). DOI: 10.4230/LIPICS.TQC.2019.10. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10402/.

[11] E. F. Codd. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[12] G. Moerkotte. "Building Query Compilers (Under Construction) (expected time to completion: 5 years)". In: 2006.

[13] T. Neumann. "Query Simplification: Graceful Degradation for Join-Order Optimization". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 403–414. ISBN: 9781605585512. DOI: 10.1145/1559845.1559889. URL: https://doi.org/10.1145/1559845.1559889.

[14] T. Ibaraki and T. Kameda. "On the Optimal Nesting Order for Computing N-Relational Joins". In: *ACM Trans. Database Syst.* 9.3 (Sept. 1984), pp. 482–502. ISSN: 0362-5915. DOI: 10.1145/1270.1498. URL: https://doi.org/10.1145/1270.1498.

[15] C. Monma and J. Sidney. "Sequencing with Series-Parallel Precedence Constraints". In: *Math. Oper. Res.* 4 (1979), pp. 215–224.

[16] D. Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley, 1973, pp. 391–392.

[17] M. Stoian. "An Efficient Implementation of Polynomial-Time Join Ordering". Bachelor's Thesis. Technical University of Munich, 2021.

[18] B. Radke and T. Neumann. "LinDP++: Generalizing Linearized DP to Crossproducts and Non-Inner Joins". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*. Ed. by T. Grust, F. Naumann, A. Böhm, W. Lehner, T. Härder, E. Rahm, A. Heuer, M. Klettke, and H. Meyer. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 57–76. DOI: 10.18420/btw2019-05. URL: https://doi.org/10.18420/btw2019-05.

[19] C. Ibrahim, D. Lykov, Z. He, Y. Alexeev, and I. Safro. *Constructing Optimal Contraction Trees for Tensor Network Quantum Circuit Simulation*. 2022. DOI: 10.48550/ARXIV.2209.02895. URL: https://arxiv.org/abs/2209.02895.

[20] T. Neumann and B. Radke. "Adaptive Optimization of Very Large Join Queries". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 677–692. ISBN: 9781450347037. DOI: 10.1145/3183713.3183733. URL: https://doi.org/10.1145/3183713.3183733.

[21] R. Bellman. "On the Theory of Dynamic Programming". In: *Proceedings of the National Academy of Sciences of the United States of America* 38.8 (1952), pp. 716–719. ISSN: 00278424. URL: http://www.jstor.org/stable/88493 (visited on 05/08/2023).

[22] D. Kossmann and K. Stocker. "Iterative Dynamic Programming: A New Class of Query Optimization Algorithms". In: *ACM Trans. Database Syst.* 25.1 (Mar. 2000), pp. 43–82. ISSN: 0362-5915. DOI: 10.1145/352958.352982. URL: https://doi.org/10.1145/352958.352982.

[23] C. Huang, F. Zhang, M. Newman, J. Cai, X. Gao, Z. Tian, J. Wu, H. Xu, H. Yu, B. Yuan, M. Szegedy, Y. Shi, and J. Chen. *Classical Simulation of Quantum Supremacy Circuits*. 2020. arXiv: 2005.06787 [quant-ph].

[24] J. Gray and S. Kourtis. "Hyper-optimized tensor network contraction". In: *Quantum* 5 (Mar. 2021), p. 410. DOI: 10.22331/q-2021-03-15-410. URL: https://doi.org/10.22331%2Fq-2021-03-15-410.

[25] G. Moerkotte and T. Neumann. "Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products". In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 930–941.

[26] R. Krishnamurthy, H. Boral, and C. Zaniolo. "Optimization of Nonrecursive Queries". In: *Proceedings of the 12th International Conference on Very Large Data Bases*. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 128–137. ISBN: 0934613184.

[27] D. G. a. Smith and J. Gray. "opt_einsum - A Python package for optimizing contraction order for einsum-like expressions". In: *Journal of Open Source Software* 3.26 (2018), p. 753. DOI: 10.21105/joss.00753. URL: https://doi.org/10.21105/joss.00753.

[28] D. Bauernfeind, M. Zingl, R. Triebl, M. Aichhorn, and H. G. Evertz. "Fork Tensor-Product States: Efficient Multiorbital Real-Time DMFT Solver". In: *Physical Review X* 7.3 (July 2017). DOI: 10.1103/physrevx.7.031013. URL: https://doi.org/10.1103%2Fphysrevx.7.031013.

[29] S. Cheng, L. Wang, T. Xiang, and P. Zhang. "Tree tensor networks for generative modeling". In: *Physical Review B* 99.15 (Apr. 2019). DOI: 10.1103/physrevb.99.155131. URL: https://doi.org/10.1103%2Fphysrevb.99.155131.

[30] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (Oct. 2019), pp. 505–510. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1666-5. URL: https://doi.org/10.1038/s41586-019-1666-5.

[31] F. Pan and P. Zhang. *Simulating the Sycamore quantum supremacy circuits*. 2021. arXiv: 2103.03074 [quant-ph].

[32] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. "k-way Hypergraph Partitioning via *n*-Level Recursive Bisection". In: *18th Workshop on Algorithm Engineering and Experiments, (ALENEX 2016)*. 2016, pp. 53–67.

[33] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. "Access Path Selection in a Relational Database Management System". In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. SIGMOD '79. Boston, Massachusetts: Association for Computing Machinery, 1979, pp. 23–34. ISBN: 089791001X. DOI: 10.1145/582095.582099. URL: https://doi.org/10.1145/582095.582099.

[34] L. Fegaras. "A New Heuristic for Optimizing Large Queries". In: *Proceedings of the 9th International Conference on Database and Expert Systems Applications*. DEXA '98. Berlin, Heidelberg: Springer-Verlag, 1998, pp. 726–735. ISBN: 3540649506.

[35] R. N. C. Pfeifer, J. Haegeman, and F. Verstraete. "Faster identification of optimal contraction sequences for tensor networks". In: *Phys. Rev. E* 90 (3 Sept. 2014), p. 033315. DOI: 10.1103/PhysRevE.90.033315. URL: https://link.aps.org/doi/10.1103/PhysRevE.90.033315.

[36] J. Xu, L. Liang, L. Deng, C. Wen, Y. Xie, and G. Li. "Towards a polynomial algorithm for optimal contraction sequence of tensor networks from trees". In: *Phys. Rev. E* 100 (4 Oct. 2019), p. 043309. DOI: 10.1103/PhysRevE.100.043309. URL: https://link.aps.org/doi/10.1103/PhysRevE.100.043309.

[37] R. Marcus and O. Papaemmanouil. "Deep Reinforcement Learning for Join Order Enumeration". In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, June 2018. DOI: 10.1145/3211954.3211957. URL: https://doi.org/10.1145%2F3211954.3211957.

[38] E. A. Meirom, H. Maron, S. Mannor, and G. Chechik. *Optimizing Tensor Network Contraction Using Reinforcement Learning*. 2022. arXiv: 2204.09052 [quant-ph].

[39] M. A. Khamis, H. Q. Ngo, and A. Rudra. *FAQ: Questions Asked Frequently*. 2017. arXiv: 1504.04044 [cs.DB].

[40] S. Cluet and G. Moerkotte. "On the complexity of generating optimal left-deep processing trees with cross products". In: *Database Theory — ICDT '95*. Ed. by G. Gottlob and M. Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 54–67. ISBN: 978-3-540-49136-1.

[41] W. Scheufele and G. Moerkotte. "On the Complexity of Generating Optimal Plans with Cross Products (Extended Abstract)". In: *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '97. Tucson, Arizona, USA: Association for Computing Machinery, 1997, pp. 238–248. ISBN: 0897919106. DOI: 10.1145/263661.263687. URL: https://doi.org/10.1145/263661.263687.