Improved Cut Strategy for Tensor Network Contraction Orders

Christoph Staudt
Friedrich Schiller University Jena, Germany

Mark Blacher
Friedrich Schiller University Jena, Germany

Julien Klaus
Friedrich Schiller University Jena, Germany

Farin Lippmann
Friedrich Schiller University Jena, Germany

Joachim Giesen
Friedrich Schiller University Jena, Germany

Abstract

In the field of quantum computing, simulating quantum systems on classical computers is crucial. Tensor networks are fundamental in simulating quantum systems. A tensor network is a collection of tensors, that need to be contracted into a result tensor. Tensor contraction is a generalization of matrix multiplication to higher order tensors. The contractions can be performed in different orders, and the order has a significant impact on the number of floating point operations (flops) needed to get the result tensor. It is known that finding an optimal contraction order is NP-hard. The current state-of-the-art approach for finding efficient contraction orders is to combinine graph partitioning with a greedy strategy. Although heavily used in practice, the current approach ignores so-called free indices, chooses node weights without regarding previous computations, and requires numerous hyperparameters that need to be tuned at runtime. In this paper, we address these shortcomings by developing a novel graph cut strategy. The proposed modifications yield contraction orders that significantly reduce the number of flops in the tensor contractions compared to the current state of the art. Moreover, by removing the need for hyperparameter tuning at runtime, our approach converges to an efficient solution faster, which reduces the required optimization time by at least an order of magnitude.

2012 ACM Subject Classification Theory of computation → Algorithm design techniques; Mathematics of computing → Solvers; Applied computing → Physics

Keywords and phrases tensor network, contraction order, graph partitioniong, quantum simulation

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.27

Supplementary Material *Software (Source Code)*:
`https://github.com/ti2-group/hybrid_contraction_tree_optimizer` [25]
archived at `swh:1:dir:72b3334932a79d590af0d303cb339bd8fc93abe6`

Funding Supported by the Carl Zeiss Stiftung within the project "Interactive Inference".

1 Introduction

Quantum computing is a rapidly evolving field that holds a lot of promise, especially for simulating quantum systems. However, building reliably working quantum computers is still a major challenge, because of to the fragile nature of quantum states, which can easily be disturbed by environmental factors [3]. Therefore, simulating quantum systems on classical computers is still a viable alternative. It is, however, a computationally intensive task, as the size of the state space grows exponentially with the number of qubits.

Tensor networks are a fundamental data structure used in quantum simulations [16], including the simulation of quantum circuits that are designed to run on quantum computers, but can also be simulated on classical computers. A tensor network is a graph, where the nodes are tensors and the edges represent indices shared among the tensors. Evaluating a tensor network means contracting the tensors over the shared indices, one index after the other. We provide a formal definition of tensor networks and tensor network contractions in Section 2. The efficiency of evaluating a tensor network heavily depends on the order of the indices that is used for the contractions. Finding a good, or even optimal, contraction order is known as the *tensor network contraction order problem*. Finding an optimal order is NP-hard [12]. Therefore, heuristics are used in practice for computing good contraction orders. There are two main classes of contraction order heuristics, namely greedy [24] and graph-structure-based heuristics, such as tree decompositions, graph partitioning, or community detection [8, 24]. In practice, the best results are reported when combining graph-structure and greedy algorithms [8]. Graph-structure-based algorithms are used to hierarchically decompose the problem into smaller and balanced subproblems and subproblems below a certain size are optimized by a greedy approach [8, 10]. Here, we present a new hybrid of graph-structure and greedy heuristics that addresses three main shortcomings of the state of the art: (1) Relating balanced partitions to balanced contraction trees, (2) disregarding free indices in the cut size, and (3) a weak correlation between partition size and contraction cost.

Experimental results show that our hybrid approach outperforms the state of the art in terms of contraction order quality and optimization time. Compared to the state of the art, our hybrid approach reduces the number of floating point operations (flops) in the tensor contraction on the Google Sycamore circuits [2] by at least a third, in a fraction of the time.

Furthermore, we present an anytime algorithm that does not rely on hard-to-tune hyperparameters. Both aspects are practically relevant as it happens in practice that more time is spent on optimizing contraction orders, including hyperparameter tuning, than on actually contracting the tensor network. An anytime algorithm for contraction order optimization that tracks the required number of flops can stop when it no longer pays off in terms of the overall running time. This includes optimizing the contraction order and actually contracting the tensor network. Experimental results show that our hybrid algorithm exhibits a better contraction order quality to optimization time ratio than state-of-the-art algorithms for contraction order optimization.

2 Tensor Network Contractions

Tensor networks are given by tensors with a shared set of indices. For $n \in \mathbb{N}^+$, the set of tensor indices is given as $[n] = \{1, \ldots, n\}$, the length of the i-th index is given by $k_i \in \mathbb{N}^+$, and the set of positions is the Cartesian product $K = \prod_{i=1}^{n} [k_i]$. For an index set $I \subseteq [n]$, a *tensor* over the real numbers is a mapping

$$T : K_{|_I} \to \mathbb{R}$$

that maps positions to real numbers. Here, $K_{|_I}$ is the projection of K onto the indices in I. The size $|I|$ of the index set I is called the *order* of T. If $I = \emptyset$, then $K_{|_\emptyset} = \{()\}$ contains only the empty tuple and the corresponding order-0 tensor is a scalar. A *tensor network* is a set $\{T_1, \ldots, T_m\}$ of tensors that, for all $i \in [n]$, satisfy

$$J_i := \big|\{j \in [m] \mid i \in I_j\}\big| \leq 2,$$

that is, any index is shared by at most two tensors. Indices of axes that belong to only one tensor are called *free indices*. The structure of a tensor network can be encoded in a weighted graph with a vertex for every tensor and an edge for every shared index. The edge weights

are the lengths of the corresponding axes. We illustrate the graphical representation on the simple example of a matrix chain multiplication that we will use throughout this article. Here, free indices are depicted by dangling edges. Let $M_1 \in \mathbb{R}^{2\times 8}$ and $M_2, M_3, M_4 \in \mathbb{R}^{8\times 8}$ be four matrices, where M_i and M_j share an index if $|i-j| = 1$. The corresponding tensor network is shown in Figure 1.

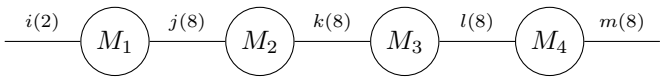


Figure 1 Matrix chain tensor network. The edges correspond to tensor axes and are labeled by indices. Their weights (in brackets) are the lengths of the axes. Dangling edges belong to free indices.

Let $L = \{i \in [n] \,|\, |J_i| = 2\} =: \{i_1 < \ldots < i_\ell\}$ be the set of *contraction indices*. The *tensor network contraction problem* asks to compute

$$\sum_{\alpha \in K_{|L}} \prod_{j=1}^{m} T_j(\alpha_{|I_j}),$$

where $\alpha_{|I}$ is the projection of α onto the indices in I. The tensor network contraction problem can also be written in the form

$$\sum_{\alpha_1 \in [k_{i_1}]} \cdots \sum_{\alpha_\ell \in [k_{i_\ell}]} \prod_{j=1}^{m} T_j(\alpha_{|I_j}),$$

where $\alpha = (\alpha_1, \ldots, \alpha_\ell)$. The interpretation of the latter form is that the edges of the tensor network are contracted one after the other in the order given by the order on $[\ell]$. Changing the contraction order by a permutation π on $[\ell]$ does not change the value. This can be significantly more efficient with a good choice for π, because, by the distributive law, products and sums can be interleaved.

In our matrix chain multiplication example, the contraction indices are $L = \{j, k, i\}$ with $k_j = k_k = k_l = 8$, while the free indices are i and m with $k_i = 2, k_m = 8$. The tensor network contraction problem for the matrix chain is to compute

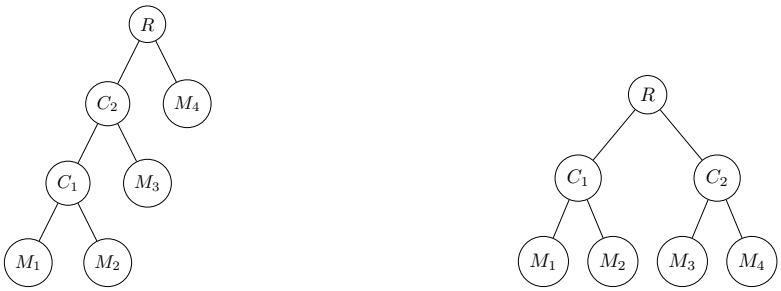
$$R(i,m) = M_1 M_2 M_3 M_4 = \sum_{j=1}^{8} \sum_{k=1}^{8} \sum_{l=1}^{8} M_1(i,j) \cdot M_2(j,k) \cdot M_3(k,l) \cdot M_4(l,m).$$

By the associativity of matrix multiplication, we have different contraction orders. For instance, $\big((M_1 M_2) M_3\big) M_4$ corresponds to the order jkl, and

$$(M_1 M_2)(M_3 M_4) = \sum_{k=1}^{8} \left[\sum_{j=1}^{8} M_1(i,j) \cdot M_2(j,k) \cdot \sum_{l=1}^{8} M_3(k,l) \cdot M_4(l,m) \right],$$

where products and sums have been interleaved by the distributive law, corresponds to the orders jlk and ljk. The contraction orders differ in the number of multiplications (flops), which are 384 for the contraction order jkl and 768 for the contraction orders jlk and ljk. Every contraction order can be encoded in a contraction tree, where the tree is uniquely defined by the order, but the tree does not uniquely determine the order. However, all contraction orders of the same tree require the same number of flops. Therefore, it is sufficient

to find a good contraction tree and not a specific contraction order. The contraction trees for $\big((M_1M_2)M_3\big)M_4$ and $(M_1M_2)(M_3M_4)$ with intermediate tensors, labeled by $C1$ and $C2$, are shown in Figure 2.



(a) Imbalanced contraction tree for the order jkl. **(b)** Balanced contraction tree for jlk and ljk.

Figure 2 Comparison of contraction trees for the matrix chain problem from Figure 1, using different contraction orders.

3 Related Work

Finding an optimal contraction order, that is, a contraction order that minimizes the number of flops, is known to be NP-hard [12]. Therefore, randomized search heuristics [22] and, especially, greedy heuristics [8, 24] are in widespread use. Greedy heuristics score the edges of the tensor network, for instance by the size difference of a contracted tensor and the two tensors that have been contracted. For the contraction ordering, an edge with minimal score is selected, and the scores of the remaining edges have to be updated, because the contraction along the selected edge can affect the scores of the remaining edges. Greedy heuristics build a contraction tree from the bottom up.

Alternatively, contraction trees can also be built top down. Since every subtree of a contraction tree corresponds to a subnetwork of the original tensor network, any contraction tree corresponds to a hierarchical partitioning of the tensor network. Graph partitioning was first used by Kourtis et al. [11] for computing contraction orders. It has become a standard technique for challenging quantum supremacy circuits [9, 17, 18] and is an integral part of cotengra, the state-of-the-art implemention for computing contraction orders [8]. It was integrated into a recently presented quantum circuit simulator targeting the current state of quantum computing, also known as the noisy intermediate-scale quantum (NISQ) era [26]. For a subdivison, cotengra aims at minimizing the cost of the cut, measured by the sum of the edge weights in the cut. The subdivision scheme shares the disadvantage of optimizing locally and not considering contraction costs down the line with the greedy approach. To counteract expensive contractions down the line, cotengra balances the cuts. For computing a balanced, low cost hierarchical partitioning of a tensor network, cotengra builds on the Karlsruhe (hyper-)graph partitioner (kahypar) [23] that constitutes the state of the art. However, cotengra still faces some challenges. Most notably, balanced contraction trees can be suboptimal. A first example can be seen in Figure 2, where the number of flops for the balanced contraction tree is significantly larger than for the unbalanced tree. The problem can be mitigated by increasing the number of partitions and the imbalance parameter of the graph partitioner. Therefore, given a user defined time limit, cotengra searches over these hyperparameter settings. Moreover, kahypar is randomized and also needs several runs. Together, this results in a fairly large variance in the quality of cotengra's solutions. The issues become evident on a larger example [15], which we also discuss in our experiments in Subsection 5.5.

Motivated by the special case of matrix chain multiplication, where the contraction order problem can be solved by dynamic programming (DP) in polynomial time [4], Ibrahim et al. first bring the tensors of the tensor network into an optimized linear order and then use the DP approach for matrix chains on the linear order [10]. We included this DP approach in our experimental evaluation. Dynamic programming has also been used by Pfeifer et al. [20] to exhaustively search the space of all contraction orders to find an optimal contraction order. The time and space requirements for this DP approach have been improved by Liang et al. [13]. Still, computing an optimal contraction order is only feasible for small tensor networks.

On the theoretical side, for tensor networks where all indices have the same length, Markov and Shi [14] have shown that the most expensive contraction requires exponentially many flops only in the treewidth of the line graph of the tensor network. Furthermore, given a tree decomposition of the line graph, they have devised an algorithm for computing a contraction order that requires exponentially many flops in the width of the given tree decomposition. The cost of the most expensive contraction can be a reasonable proxy for the overall magnitude of the number of flops if there are not many large contractions. In general, however, it is only a rough approximation for the number of flops [5]. Moreover, the analysis and algorithm apply only to networks without free indices and where all indices have the same length.

Another line of research is to learn good contraction orders. Meirom et al. [15] present a reinforcement Learning (RL) approach combined with graph neural networks (GNNs) for computing contraction orders. While the reported results compare well to the state of the art established by cotengra, their performance depends on the training data. At the moment it is difficult to assess the out-of-sample performance of the RL approach in terms of time and quality. Nevertheless, we have included it in our experiments in Section 5.

4 Algorithmic Improvements

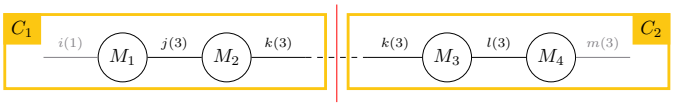
Top-down algorithms like cotengra that build on cost optimized balanced cuts of the tensor network face three main challenges:

1. **Relating balanced partitions to balanced contraction trees.** For many tensor networks the optimal contraction tree is very imbalanced. Current approaches can only generate imbalanced contraction trees, if the partitions are imbalanced. However, highly imbalanced partitions lead to a more greedy-like approach that can produce high contraction costs down the line.
2. **Accounting for free indices in the cut size.** The cut size should relate to the contraction cost between the partitions. Currently, however, free indices, which also contribute to the contraction cost, are discarded before the problem is passed to the graph partitioner. cotengra can address this problem by putting all nodes with a free index into a single partition. However, this constraint limits the possible cuts.
3. **Accounting for a node's contribution to the contraction cost of a subnetwork.** The size of the partitions should relate to contraction cost of the corresponding subnetworks. Current node weighting schemes, however, do not reflect the contributions of individual nodes to the contraction costs very well.

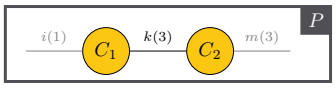
Moreover, by addressing the three challenges, many of the hyperparameters used by cotengra can be eliminated. In the following, we present our solutions to the three challenges.

4.1 Arbitrarily balanced contraction trees from balanced partitions

To address the first challenge, that is, relating balanced partitions to balanced contraction trees, we propose a new problem formulation that allows for arbitrarily balanced contraction trees from balanced partitions. Our algorithmic changes can be illustrated on the matrix chain multiplication example from Figure 1. The edge weights passed to the partitioner are the binary logarithms of the index lengths, because the contraction cost between two tensors is proportional to the product of their index lengths, which is equal to the sum of the edge weights, that is, the cut size, after applying the binary-logarithm transformation. Here, we color the free indices gray, because in the standard approach, the graph partitioner receives the network without them. A balanced cut leads to the child and parent networks that are depicted in Figure 3a and Figure 3b, respectively.



(a) The two child networks after the cut.



(b) Parent network, combining the two partitions.

Figure 3 The standard approach creates a parent network, which has a child node for each partition. The children can be partitioned further. The edge weights are logarithms of the index lengths. Gray edges represent free indices that are not passed to the graph partitioning algorithm.

Since the initial partition is balanced and there is only one way to contract the parent network, this inevitably leads to the more expensive (balanced) contraction tree in Figure 2b. The problem can be circumvented by picking one of the partitions and declaring it as the parent partition. We will explain later how the parent is picked. Instead of creating a new parent network with two nodes, one for each child, we just combine the nodes of the parent partition with a single child node that represents the result of the other partition as shown in Figure 4. The parent network P can still lead to the balanced tree in Figure 2b if needed by contracting the index l next. However, it can also lead to the imbalanced tree in Figure 2a by contracting the index k next. Thereby we have decoupled balancing the partitions from balancing the contraction tree. This allows us to always run the graph partitioner with a small imbalance parameter, typically resulting in smaller total contraction costs.

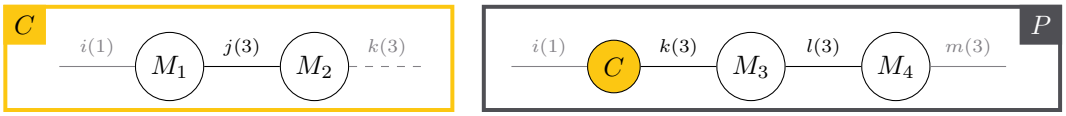


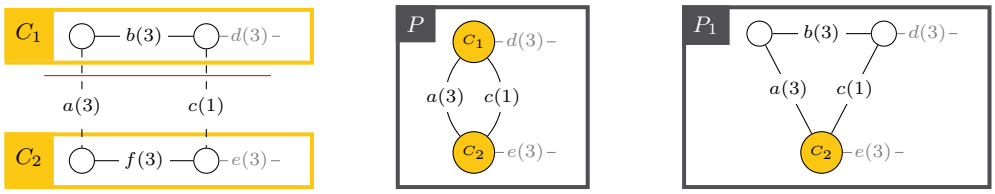
Figure 4 The child network C (on the left) is contracted first and the resulting tensor is added as node C (yellow) to the new parent network P (on the right). Note that the index k is a free index of the child network C.

If the tensor network has free indices, the parent network will be chosen as described in the next section. Otherwise, we compute a contraction order for the two possible parent and child network combinations using a greedy algorithm and pick the one with the lowest total cost.

4.2 Accounting for free indices in the cut size

The second challenge is to account for free indices when computing the cut size, because, without the free indices, the cut size does not accurately represent the contraction cost. It is important to consider free indices even if the original tensor network had none, because every cut introduces free indices into the child networks.

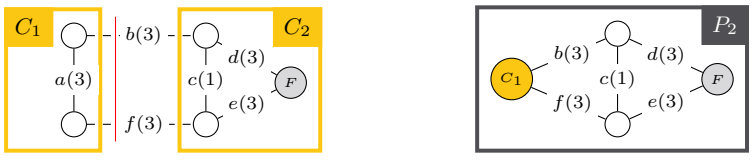
We illustrate the problem on a square tensor network with two free indices that is shown in Figure 5. The minimal balanced cut contains only the edges a and c, which suggests a contraction cost of $2 \cdot 2^3 = 16$ for the parent network. However, the child networks C_1 and C_2 have free indices that contribute to the overall contraction cost. Contracting the parent network and contracting each child network all have the same cost, namely $2 \cdot 2^3 \cdot 2^3 \cdot 2^3 = 1\,024$ flops, leading to total contraction costs of 3 072 flops.



(a) Smallest balanced cut. **(b)** Standard contraction. **(c)** Integrated child node.

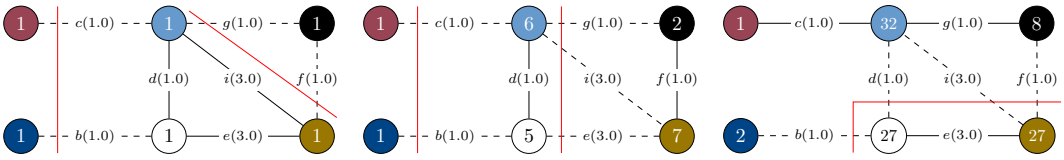
Figure 5 The standard approach disregards free indices in the cut size.

As shown in Figure 5c, the approach of turning one child network into a parent network and integrating the other child network as a node into the parent network, which we have proposed in Subsection 4.1 to counteract problems with balanced cuts, does not resolve the problem. However, the idea to add additional nodes to the network can also be used to account for indices in the cut size. Here, we add a new node F, called free node, to the network and connect it to the existing nodes by the free indices. As can be seen in Figure 6a, a minimal balanced cut in our example network now contains the edges b and f, because all other cuts contain at least three edges or produce a partition with only one node. The new node F represents the result tensor of the contraction and thus becomes the root of the contraction tree. Therefore, as depicted in Figure 6b, the child network C_2, which contains the node F, becomes the parent network P_2. The child network C_1 is added as a node to P_2. The minimal balanced cut now accounts for all edges shared between the two partitions but also for the free indices of the child network. The total contraction cost is the sum of the contraction cost of the child network C_1 and the contraction cost of the parent network P_2. Contracting C_1 requires $2^3 \cdot 2^3 \cdot 2^3 = 512$ flops, and the two remaining contractions in P_2 require 1 024 flops each. Thus, the total contraction cost is 2 560 flops, which is about 30% less than the contraction cost for the standard approach.



(a) Adding the free node F to C_2. **(b)** Parent with integrated child node.

Figure 6 The new approach considers free indices in the cut size.



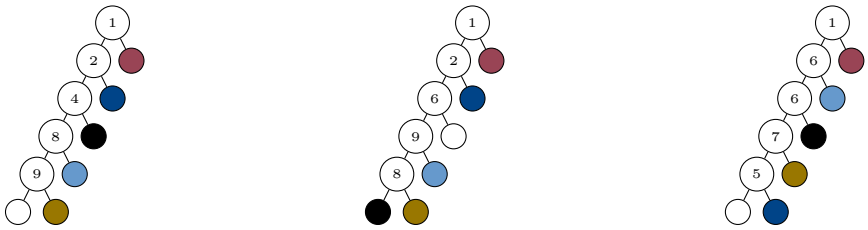
(a) Unit weighted nodes. **(b)** Log-size weighted nodes. **(c)** Cost weighted nodes.

Figure 7 Different node weight functions for the same tensor network give rise to different minimal balanced cuts (depicted by dashed edges).

4.3 Contraction cost based node weights

The third challenge is to find good estimates for the contraction costs for the partitions. We address this challenge by assigning weights to the nodes of the network, so that the weight of a partition reflects its expected contraction costs.

At the moment, there are two standard node weight functions, namely assigning unit weight to all nodes or to assign the binary logarithm of the node size. We refer to the first weight function as *unit weight* and the second as *log-size weight*. Before we introduce our own weight function, we discuss problems with the two standard weight functions. Figure 7a,b shows an example graph with weighted nodes. In the unit weighted case, the smallest balanced cut contains the edges b, c, f, and g, which separates the lower triangle from the other nodes. To use this cut to find a good contraction order, we need to decide which partition becomes the parent and which becomes the child subnetwork. We can then use the opt_einsum greedy algorithm to compute contraction orders on the subnetworks. Regardless which partition is chosen as parent, the total cost is 790 flops. A corresponding contraction tree is shown in Figure 8a. In the log-size weighted case, the smallest balanced cut contains the edges b, c, e, i, and g. Here, the best result is obtained by choosing the two nodes in the middle of the network as the parent subnetwork. However, with 838 flops the cost is even higher than in the unit weighted case. A corresponding contraction tree is shown in Figure 8b. Both weight functions miss that the top middle tensor is rather expensive to contract and needs to be separated from the tensor with indices b, d and e (white) and the tensor with indices e and f (dark yellow). To address this problem, we define a new weight function.



(a) Full/unit weighted. **(b)** Log-size weighted. **(c)** Cost weighted.

Figure 8 Contraction trees for the example graph using different node weight functions. The numbers represent the binary logarithms of the contraction costs.

The core idea is to improve on the log-size weight function. For a given contraction tree, the log-size does not necessarily reflect the actual contraction costs that should be attributed to the tensor. For any node of the contraction tree, we can compute the cost of contracting the two associated and already contracted subnetworks. We want to distribute this cost among the tensors in the subnetworks. Intuitively, tensors that contribute many indices to

the node should be assigned more of the cost. This leads to the following idea for a weight function: Given a contraction tree G and a node T of the tensor network, the nodes of G correspond to subnetworks of the tensor network. The set of subnetworks that contains the tensor corresponding to T forms a path from a leaf to the root of G. The contraction costs along this path should be partially attributed to T depending on how many indices T shares with the nodes on the path. To keep the weight function simple, we assign the contraction cost times the number of contributed indices to T. Formally, let A be a node on the path, let shared-indices(T, A) be the number of indices shared between T and A, and cost(A, G) be the contraction cost at node A in G. The weight assigned to T at A would be $\log_2(\text{cost}(A, G)) \cdot \text{shared-indices}(T, A)$. To mitigate the effect of long paths, we only assign the maximum weight along the path, that is,

$$\text{cost-weight}(T, G) = \max_{A \in \text{path}(T,G)} \Big[\log_2(\text{cost}(A, G)) \cdot \text{shared-indices}(T, A)\Big].$$

Coming back to our example, the greedy contraction order gives rise to the contraction tree in Figure 8a. Using this contraction tree results in the cost weight function that is shown in the weighted network in Figure 7c. The smallest balanced cut in this weighted network contains the edges b, d, i, and f. Here, the best result is obtained by choosing the two lower right nodes of the network as the parent subnetwork. The resulting total contraction costs, computed as before, are only 290 flops. A corresponding contraction tree is shown in Figure 8c.

5 Experiments

We will first describe the datasets and baselines that we use in our experiments, then we quickly go over implementation details of our algorithm. Afterwards, we discuss the impact of our algorithmic modifications on the contraction cost for the two largest networks in the dataset. Finally, we present comparative experimental results on all networks. Our code, including a python implementation of the algorithm and the experiments, is available at `https://github.com/ti2-group/hybrid_contraction_tree_optimizer`.

The experiments were run on a computer with two Intel Xeon 6140 processors with 18 Cores, 2.3 Ghz clock speed and 192 GB RAM.

5.1 Datasets and baselines

Unfortunately, we were not able to run the code from Meirom et al. [15] for lack of a suitable GPU cluster that is needed for training, and for missing pre-trained weights for inference. Nevertheless, since Meirom et al. published all networks that they have used in their experiments, we benchmark us and all other baselines against them. For their approach, we report the numbers as given in their paper. We clarified with the authors that they used the opt_einsum reporting for flops. In contrast to our description so far, opt_einsum counts additions and multiplications leading to twice as many flops compared to counting only multiplications. To be consistent, we report the flops counted by opt_einsum in all experiments and all algorithms.

Meirom et al. [15] have used the following three types of tensor networks in their experiments:

1. **Synthetic random 3-regular networks**, which were generated with the Python package opt_einsum [24]. The number of tensors ranges from 25 to 225 and for each network size there are 100 different networks. We excluded networks with only 25 tensors

from our benchmarks, because they can be solved by an optimal dynamic programming algorithm in less than a second. The number of indices per tensor was sampled i.i.d. from $\{2, 3, 4, 5, 6\}$.

2. The **Sycamore circuits** from the Google quantum supremacy experiment [2]. Meirom et al. have used the circuits with 53 qubits and $m = 10, 12, 14$ and 20 cycles. These networks have 162 to 379 tensors after simplification with the Python package quimb [6].
3. A single **Max-cut circuit** with 1 688 qubits and depth 21 resulting in 5 908 tensors with four free indices. Meirom et al. modeled the network based on previous work [19]. Since the network is much larger than the others used in their experiments, they only optimized the last 100, 1 000, and 2 000 contraction steps. To that end, they computed a contraction order using the opt_einsum greedy heuristic and then contracted the network up to the step they wanted to optimize. In their published data, the file for the last 2 000 steps was missing, so we generated one ourselves and made sure that our implementation matches theirs for the last 100 and 1 000 steps. For the last 100 steps, the overall cost is dominated by the previous steps, so we will not report the results for that. The costs for contracting up to the last 1 000 and 2 000 steps are 1.61e+12 and 4.49e+08, respectively. In order to always compare the costs for the entire network, we added these preprocessing flops to the results.

Besides Meirom et al.'s reinforcement learning based approach, named RL-TNCO from now on, we benchmark the following baseline algorithms that also have been used in [15]:

- The **greedy and kahypar algorithms** from the Python package cotengra [8]. We will refer to them as CTG greedy and CTG kahypar in the following. Because the greedy algorithm now utilizes an updated Rust implementation and because faster implementations allow for more trials, the results have improved compared to the earlier benchmarks [15]. cotengra offers three hyperparameter optimizers: *random*, *nevergrad* and *optuna* [21, 1]. For the greedy algorithm, mostly the *random* hyperparameter optimizer works best. It is fast and thus allows for more trials. Only on large networks with long run times, *optuna* can provide some benefit. For the kahypar algorithm, *optuna* generally seems to give superior results compared to *nevergrad* and *random*. Additionally, cotengra offers various methods to improve a given path, which are commonly used in practice. However, since this is possible for paths produced by any algorithm, we excluded this feature to simplify our comparison.
- The **miminmal linear arrangement and dynamic programming** approach, called MLA-DP [10]. We integrated the Julia implementation of MLA-DP into cotengra. Unfortunately, MLA-DP's hyperparameters are not documented. Changing the hyperparameters renders the algorithm unstable. Therefore, we only vary the seed randomly. Moreover, MLA-DP does neither support networks with more than one connected component nor networks with free indices. Since the random regular networks include disconnected networks and the max-cut network has free indices we cannot benchmark MLA-DP on these networks.

5.2 Implementation details

We first describe small further improvements of the algorithm and then detail the hyperparameters that are used by our algorithm and its dependencies.

- To decide which part of the network should be partitioned next, the algorithm uses a priority queue. The priority queue is sorted by the contraction costs of the subnetworks, which are calculated by a greedy algorithm. While the priority queue is not empty,

the subnetwork with smallest contraction cost is removed from the priority queue and partitioned into a parent and a child network. Partial contraction orders are computed for both networks, again by the greedy algorithm. The partition is rejected if the sum of the costs of the two partial contraction orders is larger than the cost of the old contraction order. Otherwise, both subnetworks are added to the priority queue with the cost of the respective partial contraction order.
- Since networks commonly have more than one minimal balanced cut of the same size, we repeat each partitioning step several times and choose the partition with the lowest total amount of flops. The number of repetitions is a hyperparameter that we describe below.
- As described in Subsection 4.3, the weights that we assign to the nodes depend on a contraction tree. To make the weights less dependent on a single contraction tree and guiding the partitioning towards a more balanced cut, we average the weights over the contraction trees that result from the repetitions that we have described in the item above.
- To turn our algorithm into an anytime algorithm, we use the total costs of the greedy contraction trees to update our current cost estimate for the whole network.
- Since both cotengrust [7] and kahypar also support (tensor) hypernetworks, our algorithm does so as well.

Our algorithm is using the following hyperparameters and default choices for them:
- *Partition imbalance parameter*: We configure kahypar to partition the networks into two parts with an imbalance parameter of 0.05.
- *Choice of greedy algorithm*: We use the greedy algorithm implemented in cotengrust. We run the greedy algorithm 256 times with random choices for its hyperparameters.
- *Node weight function*: We use only cost based node weights, as they give superior results in our ablation study in Subsection 5.3.
- *Number of repetitions of the partitioner on a given (sub) network*: 10.
- *Base case*: If a subnetwork has less than 15 nodes we compute an optimal contraction tree by the optimal dynamic programing algorithm by Pfeifer et al. [20] that is implemented in cotengrust.
- *Stopping criterion*: The algorithm stops if the cost of the next subnetwork scheduled for partitioning, and therefore the cost for each remaining subnetwork, is less than 0.001% of the current contraction cost.

The exact versions of the used softwares in our implementation can be found in the code repository[1].

5.3 Ablation study

Figure 9 shows how our algorithmic ideas reduce the number of flops required to contract the max-cut and Sycamore network for $m = 20$, where m is the number of cycles. On the x-axis we compare how different mappings of the tensor network to the graph partitioning problem influence the number of flops. Since the max-cut network has output indices adding a free node proves to be effective. It reduces the required number of flops by a fifth. Without the free node, all the other algorithmic modifications are not effective. This is because, without the free node, as shown in Figure 10, the algorithm cannot find a cut that improves the greedy baseline. Using the cost weighted nodes improves the result slightly.

[1] https://github.com/ti2-group/hybrid_contraction_tree_optimizer/blob/main/requirements.txt.

The results on the Sycamore network are very different. Here, every algorithmic modification improves the result. However, the overall improvement is smaller. Again, the improvements are a result of more successful cuts as can be seen in Figure 10. While the Sycamore network does not have free indices, new free indices are introduced into the child network of each cut. Therefore, the free node proves to be effective in this setting as well. Additionally, weighting the nodes based on the contraction tree results in a clear improvement.

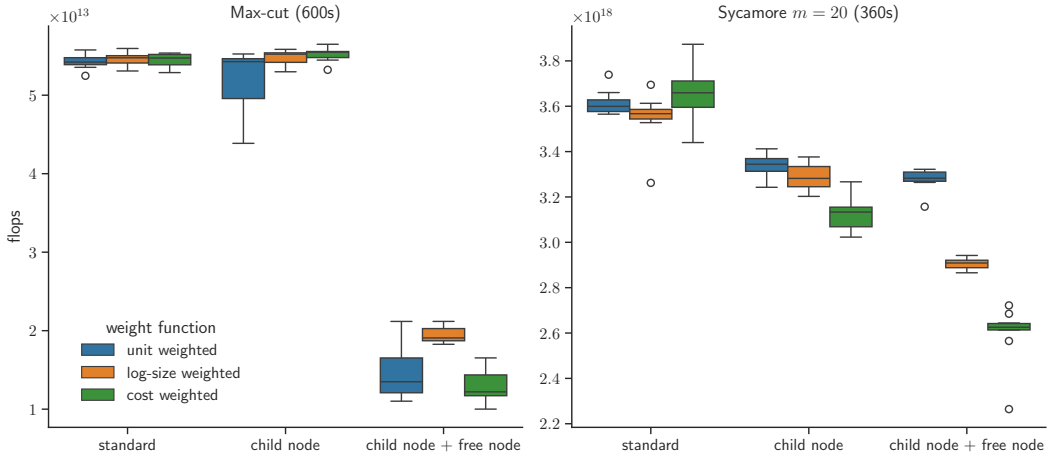


Figure 9 Comparison of the number of flops for the standard partitioning approach, the integrated child node, and the integrated child and free node modifications. The colors encode how the nodes are weighted. Each hyperparameter combination is executed ten times.

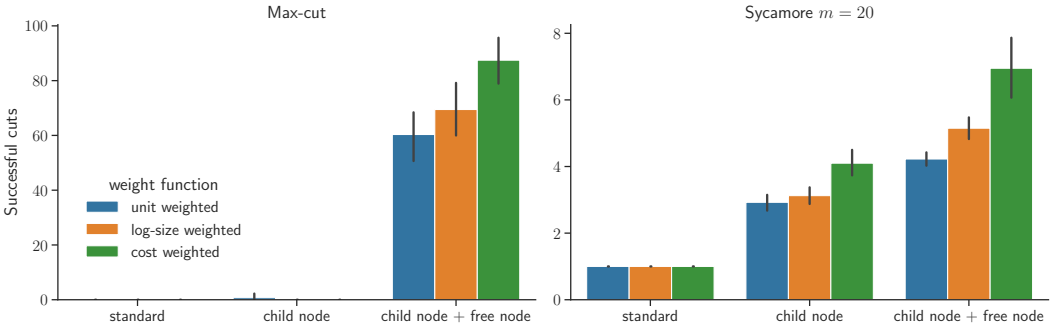


Figure 10 Comparison of the number of cuts for the different algorithmic modifications that result in a reduction of contraction costs compared to the cost estimation by the greedy algorithm.

5.4 Contraction order quality, run time, and variance

Here, we present comparative results on the random regular networks and the Sycamore circuits. We compare the contraction order quality, the runtimes, and the variance of the results. Results for the max-cut network will be presented in the next subsection.

5.4.1 Synthetic random regular networks

RL-TNCO was only benchmarked on networks up to size 100. They trained a network for each size on randomly generated graphs of the same size and then evaluated them on the benchmark set. For each problem they sampled 50 and 1000 paths and reported the results for both separately. For all methods the experiment was run three times per network and then averaged over all networks of the same size [15]. The median was first calculated for each network and then over all networks of the same size. All results are shown in Table 1 and Table 2. On small networks with only 50 tensors the reinforcement learning approach outperforms the other methods. However, on larger networks it falls behind, at 100 nodes it performs worse than all other algorithms. Our algorithm achieves the best mean and median cost on all networks with more than 50 tensors. On the larger networks our algorithm finds up to 40 times better paths on average for a runtime of 300s. For shorter runtimes the difference becomes even larger. In general our algorithm improves only slightly when the runtime is increased from one minute to 5 minutes, whereas cotengra improves significantly. This might indicate that cotengra would need more time to find good paths. However, as we will see in the next section, even running CTG kahypar for hours may not result in much better paths.

Table 1 Number of flops on the synthetic networks with 50 to 125 nodes. The run times in parenthesis belong to RL-TNCO.

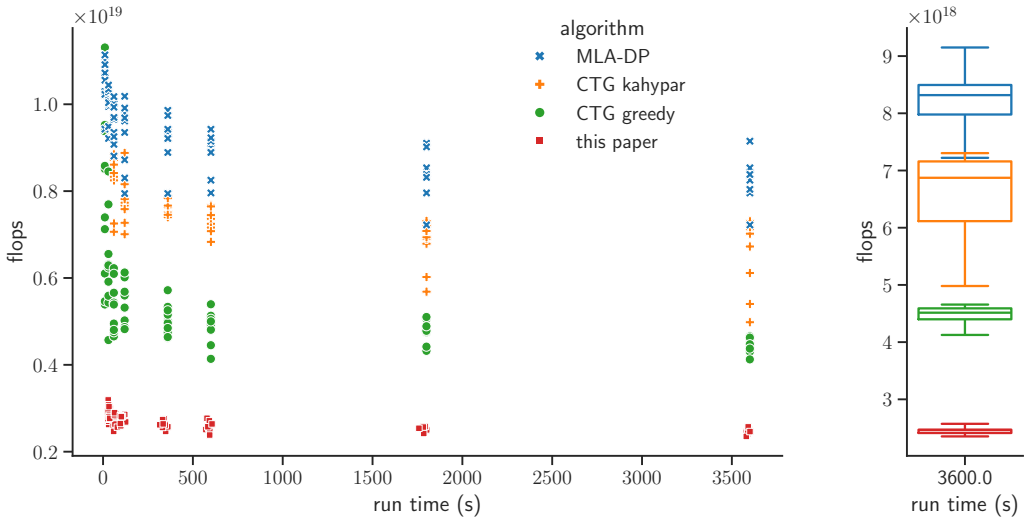
	Mean ± std				Median ± MAD			
n	50	75	100	125	50	75	100	125
Scale	10^7	10^{10}	10^{12}	10^{13}	10^7	10^{10}	10^{12}	10^{13}
run time	5s (4.2s)	5s (6.8s)	10s (9.6s)	60s	5s (4.2s)	5s (6.8s)	10s (9.6s)	60s
CTG greedy	10.2 ± 41.7	3.08 ± 9.11	5.08 ± 15.2	50.2 ± 114	3.46 ± 2.80	0.50 ± 0.46	0.92 ± 0.81	10.7 ± 9.62
CTG kahypar	8.29 ± 22.9	8.62 ± 30.3	9.89 ± 41.2	18.2 ± 50.4	3.43 ± 2.68	0.54 ± 0.50	0.62 ± 0.56	2.53 ± 2.35
RL-TNCO	**3.93 ± 4.54**	1.68 ± 4.62	9.18 ± 21.16	N/A	**2.05 ± 4.54**	0.24 ± 4.62	2.37 ± 21.16	N/A
this paper	5.67 ± 13.1	**0.82 ± 2.02**	**0.53 ± 1.28**	**2.91 ± 6.36**	2.85 ± 2.29	**0.17 ± 0.15**	**0.14 ± 0.11**	**0.79 ± 0.70**
run time	60s (74.6s)	120 (124.2s)	120s (164s)	300s	60s (74.6s)	120s (124.2s)	120s (164s)	300s
CTG greedy	8.02 ± 30.4	1.85 ± 5.38	2.74 ± 9.38	29.8 ± 64.4	3.00 ± 2.49	0.30 ± 0.28	0.49 ± 0.42	6.56 ± 5.74
CTG kahypar	5.55 ± 13.8	0.80 ± 2.21	1.03 ± 2.91	8.44 ± 20.6	2.63 ± 2.16	0.17 ± 0.15	0.21 ± 0.18	1.75 ± 1.59
RL-TNCO	**3.22 ± 3.53**	1.23 ± 2.75	5.55 ± 11.09	N/A	**1.87 ± 1.57**	0.21 ± 0.14	1.86 ± 1.69	N/A
this paper	5.25 ± 12.7	**0.60 ± 1.39**	**0.40 ± 1.05**	**2.42 ± 5.19**	2.69 ± 2.09	**0.15 ± 0.13**	**0.11 ± 0.09**	**0.63 ± 0.55**

Table 2 Number of flops on the synthetic networks with 150 to 225 nodes.

	Mean ± std				Median ± MAD			
n	150	175	200	225	150	175	200	225
Scale	10^{16}	10^{18}	10^{20}	10^{22}	10^{16}	10^{18}	10^{20}	10^{22}
run time	60s							
CTG greedy	33.8 ± 154	33.6 ± 95.4	340 ± 2337	617 ± 2353	4.27 ± 4.02	4.99 ± 4.80	11.1 ± 10.7	21.8 ± 20.3
CTG kahypar	10.6 ± 49.6	3.41 ± 9.78	18.1 ± 67.6	321 ± 1985	0.56 ± 0.52	0.34 ± 0.32	0.88 ± 0.79	1.53 ± 1.49
this paper	**0.43 ± 0.95**	**0.12 ± 0.22**	**0.36 ± 1.20**	**0.23 ± 0.83**	**0.08 ± 0.07**	**0.03 ± 0.03**	**0.03 ± 0.02**	**0.02 ± 0.02**
run time	300s							
CTG greedy	14.6 ± 56.3	18.1 ± 74.1	228 ± 1794	213 ± 698	2.51 ± 2.35	2.44 ± 2.28	5.13 ± 4.88	8.61 ± 8.24
CTG kahypar	1.43 ± 3.44	0.69 ± 1.45	2.58 ± 10.2	4.71 ± 18.8	0.26 ± 0.24	0.11 ± 0.10	0.20 ± 0.18	0.23 ± 0.21
this paper	**0.34 ± 0.84**	**0.08 ± 0.14**	**0.21 ± 0.70**	**0.11 ± 0.38**	**0.06 ± 0.06**	**0.02 ± 0.02**	**0.02 ± 0.01**	**0.01 ± 0.01**

5.4.2 Sycamore networks

To compare against RL-TNCO, we first present results on the Sycamore networks for a optimization time of 1 hour. We report the results after 3 hours for RL-TNCO, since they did not report their results after 1h for the smaller networks. As shown in Table 3 we outperform all other approaches on all networks. This even holds when our algorithm runs for only one minute. We present extended statistics in Table 4 in the appendix.



(a) Flops vs. time. **(b)** Variance (1h).

Figure 11 Comparison of flops over time and variance after 1h for the Sycamore network with $m = 20$.

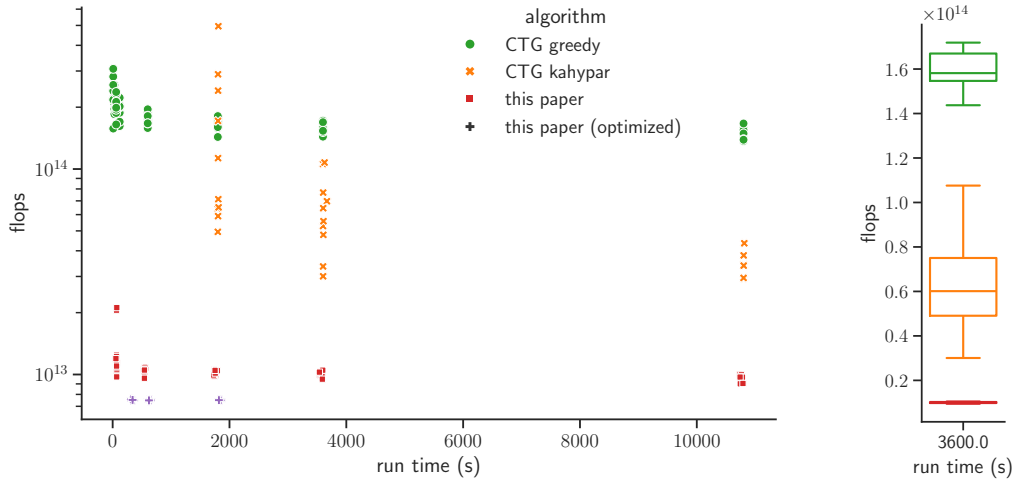
Besides contraction order quality measured in flops, practical relevant performance measures are quality over run time and variance of the solution quality. Therefore, we present the flops over time and the variance after 1h for the $m = 20$ network in Figure 11. We run each algorithm 10 times for 10, 30, 60, 120, 360, 600, 1800 and 3600 seconds. Our algorithm reliably finds good contraction trees after 360s, while the other algorithms slowly improve towards the 1h time limit. However, as can be seen in Figure 11b, even after 1h the quality of their results, especially of CTG kahypar varies greatly.

Table 3 Median number of flops on the Sycamore networks over 10 independent runs. The parameter m represents the cycles of the Sycamore circuits and controls their size and complexity.

	$m = 10$	$m = 12$	$m = 14$	$m = 20$
Scale	10^{10}	10^{12}	10^{14}	10^{18}
MLA-DP 1h	2.45	10.7	1.94	8.32
CTG greedy 1h	2.34	8.61	2.21	4.51
CTG kahypar 1h	2.20	10.9	2.06	6.87
RL-TNCO 3h	5.44	7.4	2.63	3.5
this paper 60s	1.78	6.05	1.61	2.79
this paper 1h	**1.52**	**5.76**	**1.37**	**2.46**

5.5 Hyperparameter optimization and max-cut network

So far, all reported experimental results for our algorithm were obtained with the default hyperparameter settings. Since we observed that the hyperparameter optimizer optuna for CTG greedy is able to find better contraction orders than CTG greedy with randomly chosen hyperparameters on the large max-cut network, we also examined the effect of optimizing the imbalance hyperparameter on our algorithm. Results are shown in Figure 12 and in the appendix in Table 5. On the max-cut network, both cotengra algorithms struggle to find a good contraction order. CTG kahypar finds a reasonably good contraction order, but only after 6h. The same holds for RL-TNCO, which finds a good contraction order after nearly 6h. Even though the network is large, the contraction costs are rather low. On most modern computers it can be contracted within minutes. For the long optimzation times to pay off, one needs to contract the network fairly often. Our algorithm with default hyperparameter settings finds a good contraction order after only 10 minutes. Optimizing the imbalance parameter improves the results by about 25%, leading to the best contraction order for this network.



(a) Flops vs. time.

(b) Variance (1h).

Figure 12 Comparison of flops over time and variance after 1h for the last 2000 steps of the max-cut network.

6 Conclusions

We have introduced a novel cut strategy for finding efficient tensor network contraction orders. The strategy consists of three algorithmic modifications over the state-of-the-art approach based on top-down balanced graph partitioning and bottom-up greedy algorithms.

Our first modification addresses a mismatch between the partition balance objective and imbalanced optimal contraction trees. For many tensor networks the optimal contraction tree is imbalanced, which is difficult to achieve with a balanced cut objective in the partitioner. In our modification, we designate one partition as the child and the other as the parent network. The child network is integrated as a node into the parent network, which facilitates the generation of both unbalanced and balanced contraction trees. Moreover, the modification reduces the number of hyperparameters.

Our second modification addresses that free indices were not accounted for in the cut size objective. Accounting for free indices is important, because they contribute to the contraction cost. In our modification, we include an additional node that represents the free indices in each partitioning step. The additional node ensures that the contraction costs are more accurately reflected in the cut objective.

Our third modification addresses the problem of estimating the contraction costs of partitions. A partitioner can use the estimates as weights when computing a small balanced cut. In our modification we address this challenge by assigning weights to the nodes of the network, so that the weight of a partition reflects its expected contraction costs. To iteratively update the weights we leverage the greedy approach for computing contraction orders on the partitions. The iterative refinement ensures that the weighted network graph more accurately reflects the optimization objective.

Experimental results demonstrate that all three algorithmic modifications can improve the current state of the art in finding good contraction orders. Overall, the proposed algorithm computes more efficient contraction orders in significantly less time. Even though our algorithm is randomized, just like the baseline algorithms, it exhibits a reduced variance in the quality of the computed solutions. Moreover, the algorithm serves as an anytime algorithm that can stop when it becomes obvious that further optimizations no longer pay off.

References

1 Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis, editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 2623–2631. ACM, 2019. doi:10.1145/3292500.3330701.

2 Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, and et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, October 2019. Number: 7779 Publisher: Nature Publishing Group. doi:10.1038/s41586-019-1666-5.

3 Adrian Cho. The biggest flipping challenge in quantum computing. *Science*, 2020. doi:10.1126/science.abd7332.

4 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

5 Eugene F. Dumitrescu, Allison L. Fisher, Timothy D. Goodrich, Travis S. Humble, Blair D. Sullivan, and Andrew L. Wright. Benchmarking treewidth as a practical component of tensor network simulations. *PLOS ONE*, 13(12):1–19, December 2018. doi:10.1371/journal.pone.0207827.

6 Johnnie Gray. quimb: A python package for quantum information and many-body calculations. *J. Open Source Softw.*, 3(29):819, 2018. doi:10.21105/joss.00819.

7 Johnnie Gray. jcmgray/cotengrust, February 2024. original-date: 2023-08-31T18:57:15Z. URL: https://github.com/jcmgray/cotengrust.

8 Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, 2021. doi:10.22331/q-2021-03-15-410.

9 Cupjin Huang, Fang Zhang, Michael Newman, Xiaotong Ni, Dawei Ding, Junjie Cai, Xun Gao, Tenghui Wang, Feng Wu, Gengyan Zhang, Hsiang-Sheng Ku, Zhengxiong Tian, Junyin Wu, Haihong Xu, Huanjun Yu, Bo Yuan, Mario Szegedy, Yaoyun Shi, Hui-Hai Zhao, Chunqing Deng, and Jianxin Chen. Efficient parallelization of tensor network contraction for simulating quantum computation. *Nat. Comput. Sci.*, 1(9):578–587, 2021. doi:10.1038/s43588-021-00119-7.

10 Cameron Ibrahim, Danylo Lykov, Zichang He, Yuri Alexeev, and Ilya Safro. Constructing optimal contraction trees for tensor network quantum circuit simulation. In *IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022*, pages 1–8. IEEE, 2022. doi:10.1109/HPEC55821.2022.9926353.

11 Stefanos Kourtis, Claudio Chamon, Eduardo R. Mucciolo, and Andrei E. Ruckenstein. Fast counting with tensor networks. *SciPost Physics*, 7(5):060, November 2019. arXiv:1805.00475 [cond-mat, physics:physics]. doi:10.48550/arXiv.1805.00475.

12 Chi-Chung Lam, P. Sadayappan, and Rephael Wenger. On optimizing a class of multi-dimensional loops with reductions for parallel execution. *Parallel Process. Lett.*, 7(2):157–168, 1997. doi:10.1142/S0129626497000176.

13 Ling Liang, Jianyu Xu, Lei Deng, Mingyu Yan, Xing Hu, Zheng Zhang, Guoqi Li, and Yuan Xie. Fast search of the optimal contraction sequence in tensor networks. *IEEE J. Sel. Top. Signal Process.*, 15(3):574–586, January 2021. doi:10.1109/JSTSP.2021.3051231.

14 Igor L. Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM J. Comput.*, 38(3):963–981, 2008. doi:10.1137/050644756.

15 Eli A. Meirom, Haggai Maron, Shie Mannor, and Gal Chechik. Optimizing tensor network contraction using reinforcement learning. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 15278–15292. PMLR, 2022. URL: https://proceedings.mlr.press/v162/meirom22a.html.

16 Román Orús. Tensor networks for complex quantum systems. *Nature Reviews Physics*, 1(9):538–550, September 2019. doi:10.1038/s42254-019-0086-7.

17 Feng Pan, Keyang Chen, and Pan Zhang. Solving the sampling problem of the sycamore quantum circuits. *Phys. Rev. Lett.*, 129:090502, August 2022. doi:10.1103/PhysRevLett.129.090502.

18 Feng Pan and Pan Zhang. Simulating the Sycamore quantum supremacy circuits, March 2021. URL: https://arxiv.org/abs/2103.03074v1.

19 Taylor L. Patti, Jean Kossaifi, Anima Anandkumar, and Susanne F. Yelin. Variational quantum optimization with multibasis encodings. *Phys. Rev. Res.*, 4:033142, August 2022. doi:10.1103/PhysRevResearch.4.033142.

20 Robert N. C. Pfeifer, Jutho Haegeman, and Frank Verstraete. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E*, 90(3):033315, September 2014. arXiv:1304.6112 [cond-mat, physics:physics, physics:quant-ph]. doi:10.48550/arXiv.1304.6112.

21 J. Rapin and O. Teytaud. Nevergrad - A gradient-free optimization platform. https://GitHub.com/FacebookResearch/Nevergrad, 2018.

22 Frank Schindler and Adam S. Jermyn. Algorithms for tensor network contraction ordering. *Mach. Learn. Sci. Technol.*, 1(3):35001, July 2020. Publisher: IOP Publishing. doi:10.1088/2632-2153/ab94c5.

23 Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM J. Exp. Algorithmics*, 27:1.9:1–1.9:39, February 2022. doi:10.1145/3529090.

24 Daniel G. A. Smith and Johnnie Gray. opt_einsum - A python package for optimizing contraction order for einsum-like expressions. *J. Open Source Softw.*, 3(26):753, 2018. doi:10.21105/joss.00753.

25 Christoph Staudt. Hybrid Contraction Tree Optimizer. Software, swhId: `swh:1:dir:72b3334932a79d590af0d303cb339bd8fc93abe6`, (visited on 13/05/2024). URL: `https://github.com/ti2-group/hybrid_contraction_tree_optimizer`.

26 Shi-Xin Zhang, Jonathan Allcock, Zhou-Quan Wan, Shuo Liu, Jiace Sun, Hao Yu, Xing-Han Yang, Jiezhong Qiu, Zhaofeng Ye, Yu-Qin Chen, Chee-Kong Lee, Yicong Zheng, Shao-Kai Jian, Hong Yao, Chang-Yu Hsieh, and Shengyu Zhang. Tensorcircuit: a quantum software framework for the NISQ era. *Quantum*, 7:912, February 2023. Publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften. `doi:10.22331/q-2023-02-02-912`.

A Comparison of hyperoptimizers

In the following Figure 13 we compare different hyperoptimizers on the max-cut and Sycamore ($m = 20$) networks. While randomly choosing hyperparameters for the CTG greedy algorithm outperforms the other hyperoptimizers on the Sycamore network, optimizing the hyperparameters with optuna pays off for the Max-Cut network. Optimizing the hyperparameters for CTG kahypar always leads to better contraction orders. Overall optuna is the better choice in this case as well. Note that on the Max-Cut network we only report a runtime of 6h since, smaller run times lead to too much variance in the results.

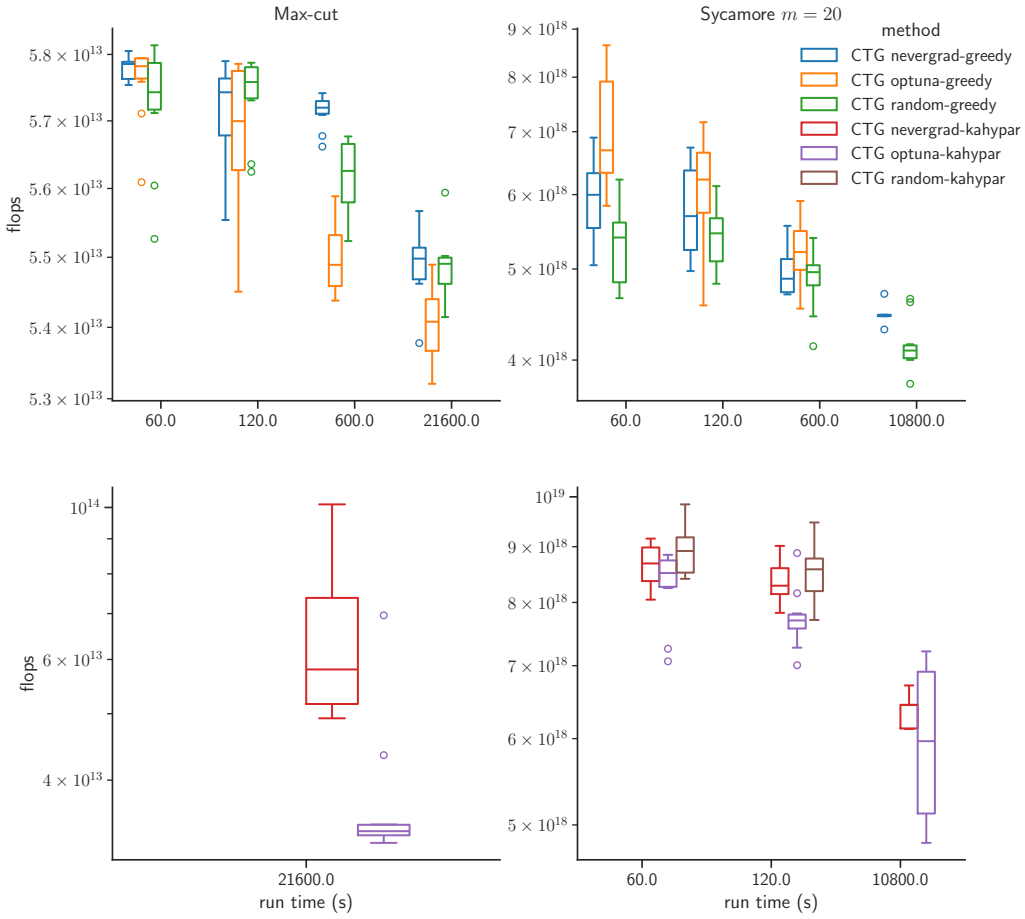


Figure 13 Comparison of hyperoptimizers on the max-cut and Sycamore $m = 20$ networks.

B Extended result tables

Table 4 Number of flops for the sycamore networks aggregated over 10 independent runs.

	Mean ± std				Median ± mad			
$m =$	10	12	14	20	10	12	14	20
Scale	10^{10}	10^{12}	10^{14}	10^{18}	10^{10}	10^{12}	10^{14}	10^{18}
MLA-DP 1min	2.80 ± 0.13	12.0 ± 0.82	2.15 ± 0.14	9.59 ± 0.45	2.76 ± 0.09	12.3 ± 0.33	2.11 ± 0.05	9.67 ± 0.33
CTG greedy 1min	3.24 ± 0.31	13.4 ± 2.43	2.80 ± 0.45	5.34 ± 0.55	3.30 ± 0.16	13.5 ± 0.99	2.71 ± 0.28	5.40 ± 0.52
CTG kahypar 1min	2.47 ± 0.43	31.3 ± 9.67	4.09 ± 1.63	8.31 ± 0.64	2.46 ± 0.36	31.7 ± 10.3	3.61 ± 0.85	8.51 ± 0.25
this paper 1min	1.79 ± 0.10	6.05 ± 0.15	1.59 ± 0.06	2.76 ± 0.16	1.78 ± 0.05	6.05 ± 0.03	1.61 ± 0.04	2.79 ± 0.09
MLA-DP 10min	2.59 ± 0.08	10.9 ± 0.46	2.00 ± 0.05	8.85 ± 0.52	2.60 ± 0.03	10.7 ± 0.31	2.00 ± 0.04	8.95 ± 0.38
CTG greedy 10min	2.78 ± 0.26	9.67 ± 1.45	2.41 ± 0.18	4.87 ± 0.36	2.80 ± 0.21	9.41 ± 0.90	2.39 ± 0.12	4.96 ± 0.16
CTG kahypar 10min	2.34 ± 0.29	16.0 ± 2.87	2.19 ± 0.15	7.34 ± 0.26	2.34 ± 0.27	16.0 ± 1.12	2.14 ± 0.06	7.40 ± 0.20
this paper 10min	1.60 ± 0.09	5.79 ± 0.16	1.42 ± 0.04	2.57 ± 0.11	1.58 ± 0.06	5.76 ± 0.14	1.43 ± 0.02	2.56 ± 0.06
MLA-DP 1h	2.47 ± 0.04	10.7 ± 0.36	1.93 ± 0.03	8.24 ± 0.50	2.45 ± 0.00	10.7 ± 0.36	1.94 ± 0.03	8.32 ± 0.25
CTG greedy 1h	2.35 ± 0.23	8.58 ± 0.87	2.20 ± 0.07	4.48 ± 0.17	2.34 ± 0.16	8.61 ± 0.63	2.21 ± 0.04	4.51 ± 0.09
CTG kahypar 1h	2.25 ± 0.33	11.1 ± 1.16	2.14 ± 0.29	6.51 ± 0.82	2.20 ± 0.27	10.9 ± 0.26	2.06 ± 0.09	6.87 ± 0.37
RL-TNCO 1h	N/A	N/A	N/A	N/A	N/A	N/A	N/A	6.81 ± N/A
this paper 1h	**1.50 ± 0.04**	**5.73 ± 0.13**	**1.34 ± 0.08**	2.45 ± 0.07	**1.52 ± 0.01**	**5.76 ± 0.08**	**1.37 ± 0.03**	2.46 ± 0.03
CTG greedy 3h	2.24 ± 0.21	7.92 ± 0.49	2.18 ± 0.05	4.15 ± 0.27	2.24 ± 0.15	7.77 ± 0.30	2.20 ± 0.04	4.09 ± 0.07
CTG kahypar 3h	2.21 ± 0.29	11.6 ± 2.20	2.05 ± 0.14	6.00 ± 1.02	2.15 ± 0.25	11.0 ± 1.00	2.01 ± 0.05	5.97 ± 0.93
RL-TNCO 3h	N/A	N/A	N/A	N/A	5.44 ± N/A	7.4 ± N/A	2.63 ± N/A	3.5 ± N/A
this paper 3h	N/A	N/A	N/A	**2.41 ± 0.04**	N/A	N/A	N/A	**2.40 ± 0.03**

Table 5 Number of flops for the max cut networks aggregated over 10 independent runs.

	Mean ± std			Median ± mad		
Last steps	1000	2000	all	1000	2000	all
Scale	10^{12}	10^{12}	10^{12}	10^{12}	10^{12}	10^{12}
CTG greedy 10min	133 ± 4.06	176 ± 11.2	55.0 ± 0.53	133 ± 2.16	175 ± 7.37	54.9 ± 0.42
CTG kahypar 10min	72399 ± 1.92e+05	1.22e+06 ± 2.48e+06	6.49e+66 ± 2.05e+67	9087 ± 7480	42552 ± 31574	4.09e+11 ± 4.09e+11
this paper 10min	15.4 ± 0.25	10.3 ± 0.37	15.0 ± 4.55	15.4 ± 0.20	10.4 ± 0.25	14.1 ± 2.65
this paper (optimized) 10min	**10.4 ± 0.00**	7.49 ± 0.03	9.04 ± 1.22	**10.4 ± 0.00**	**7.49 ± 0.01**	8.88 ± 0.68
CTG greedy 30min	130 ± 6.92	168 ± 11.6	54.9 ± 0.55	133 ± 3.85	170 ± 8.41	54.9 ± 0.36
CTG kahypar 30min	796 ± 1549	162 ± 144	4347 ± 7686	81.2 ± 20.0	92.2 ± 37.9	486 ± 430
this paper 30min	15.0 ± 0.36	10.2 ± 0.27	11.5 ± 0.46	15.0 ± 0.18	10.2 ± 0.25	11.3 ± 0.25
this paper (optimized) 30min	**10.4 ± 0.00**	**7.49 ± 0.01**	**8.07 ± 0.31**	**10.4 ± 0.00**	**7.49 ± 0.01**	**8.00 ± 0.09**
CTG greedy 1h	125 ± 7.60	160 ± 8.85	54.7 ± 0.33	129 ± 2.87	158 ± 7.49	54.8 ± 0.24
CTG kahypar 1h	68.4 ± 21.7	64.4 ± 26.6	283 ± 526	56.7 ± 2.11	60.1 ± 14.5	82.3 ± 38.2
RL-TNCO 1.5h	N/A	N/A	N/A	19.05 ± N/A	N/A	N/A
this paper 1h	14.9 ± 0.22	10.0 ± 0.32	11.4 ± 0.48	15.0 ± 0.16	10.1 ± 0.23	11.2 ± 0.23
CTG greedy 3h	123 ± 3.48	149 ± 9.58	54.4 ± 0.54	123 ± 1.39	148 ± 8.17	54.7 ± 0.31
CTG kahypar 3h	42.5 ± 7.53	33.1 ± 4.74	46.1 ± 14.4	41.5 ± 5.49	31.7 ± 2.25	41.2 ± 7.72
this paper 3h	N/A	9.61 ± 0.34	10.8 ± 0.41	N/A	9.71 ± 0.14	10.8 ± 0.25
CTG greedy 6h	N/A	N/A	54.1 ± 0.54	N/A	N/A	54.1 ± 0.42
CTG kahypar 6h	N/A	N/A	38.1 ± 11.5	N/A	N/A	33.7 ± 0.71
RL-TNCO 5.5	N/A	N/A	N/A	N/A	9.29 ± N/A	N/A