OMEinsumContractionOrders: A Julia package for tensor network contraction order optimization

Jin-Guo Liu [1*], **Xuan-Zhao Gao** [2*], **and Richard Samuelson** [3*]

1 Hong Kong University of Science and Technology (Guangzhou) **2** **3** * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Statement of need

`OMEinsumContractionOrders` (One More Einsum Contraction Orders, or OMECO) is a Julia package(Bezanson et al., 2012) that implements state-of-the-art algorithms for optimizing tensor network contraction orders. This paper presents its key features, integration with the Julia ecosystem, and performance benchmarks.

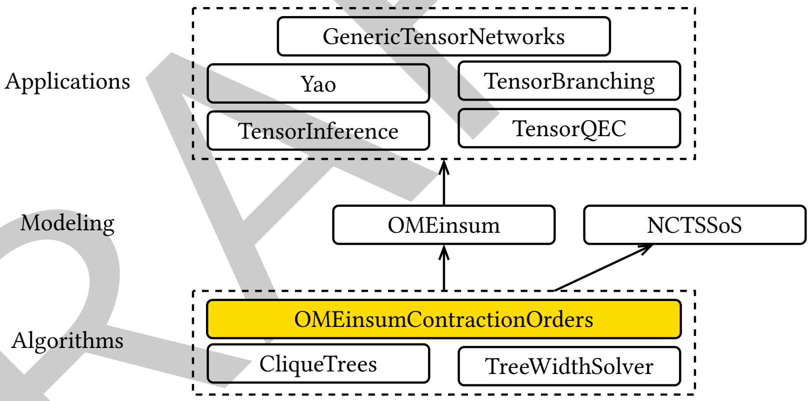

Figure 1: The ecosystem built on top of the `OMEinsumContractionOrders` and its dependencies. It is a key component of tensor network contractor `OMEinsum`. Applications include quantum simulator `Yao`, quantum decoder `TensorQEC`, probabilistic inference model `TensorInference`, combinatorial optimization `GenericTensorNetworks` and `TensorBranching`.

A *tensor network* is a mathematical framework that represents multilinear algebra operations as graphical structures, where tensors are nodes and shared indices are edges. This diagrammatic approach transforms complex high-dimensional contractions into visual networks that expose underlying computational structure.

The framework has remarkable universality across diverse domains: *einsum* notation (Harris et al., 2020) in numerical computing, *factor graphs* (Bishop & Nasrabadi, 2006) in probabilistic inference, *sum-product networks* in machine learning, and *junction trees* (Villescas et al., 2023) in graphical models. Tensor networks have enabled breakthroughs in quantum circuit simulation (Markov & Shi, 2008), quantum error correction (Piveteau et al., 2024), neural network compression (Qing et al., 2024), strongly correlated quantum materials (Haegeman et al., 2016), and combinatorial optimization problems (Liu et al., 2023). These are refleced in the ecosystem built on top of the `OMEinsumContractionOrders` package, as shown in Figure 1.

The computational cost of tensor network contraction depends critically on the *contraction order*—the sequence in which pairwise tensor multiplications are performed. This order can be represented as a binary tree where leaves correspond to input tensors and internal nodes represent intermediate results. The optimization objective balances multiple complexity measures through the cost function:

$$\mathcal{L} = w_\mathrm{t} \cdot \mathsf{tc} + w_\mathrm{s} \cdot \max(0, \mathsf{sc} - \mathsf{sc}_\mathrm{target}) + w_\mathrm{rw} \cdot \mathsf{rwc},$$

where w_t, w_s, and w_rw represent weights for time complexity (tc), space complexity (sc), and read-write complexity (rwc), respectively. Typically, the memory access costs outweigh the computational costs, hence $w_\mathrm{rw} > w_\mathrm{t}$. The penalty for space complexity is triggered only when $\mathsf{sc} > \mathsf{sc}_\mathrm{target}$, as memory does not impact performance if it fits into the available device capacity.

Finding the optimal contraction order—even when minimizing only time complexity—is NP-complete (Markov & Shi, 2008). This optimization problem has a deep mathematical connection to *tree decomposition* (Markov & Shi, 2008) of the tensor network's line graph, where finding the optimal order corresponds to finding a weighted minimal-width tree decomposition. The logarithmic time complexity of the bottleneck contraction step equals the largest bag size in the tree decomposition, while the logarithmic space complexity equals the largest separator size (vertices shared between adjacent bags).

Despite this computational hardness, near-optimal solutions suffice for most practical applications and can be obtained efficiently through heuristic methods. Modern optimization algorithms have achieved remarkable scalability, handling tensor networks with over 10^4 tensors (Gray & Kourtis, 2021; Roa-Villescas et al., 2024).

OMECO implements several optimization algorithms with complementary performance characteristics:

Optimizer	Description
`GreedyMethod`	Fast greedy heuristic with modest solution quality
`TreeSA`	Reliable simulated annealing optimizer (Kalachev et al., 2021) with high-quality solutions
`PathSA`	Simulated annealing optimizer for path decomposition
`HyperND`	Nested dissection algorithm for hypergraphs, requires `KaHyPar` or `Metis`
`KaHyParBipartite`	Graph bipartition method for large tensor networks (Gray & Kourtis, 2021), requires `KaHyPar`
`SABipartite`	Simulated annealing bipartition method, pure Julia implementation
`ExactTreewidth`	Exact algorithm with exponential runtime (Bouchitté & Todinca, 2001), based on `TreeWidthSolver`
`Treewidth`	Clique tree elimination methods from `CliqueTrees` package

The algorithms `HyperND`, `Treewidth`, and `ExactTreewidth` are applied to the tensor network's line graph and utilize the `CliqueTrees` and `TreeWidthSolver` packages, as illustrated in Figure 1. We also implement the `PathSA` optimizer for path decomposition, which is a variant of the `TreeSA` optimizer by constraining the contraction order to be a path graph.

These methods balance optimization time against solution quality. Figure 2 displays benchmark results for the Sycamore quantum supremacy circuit, highlighting the Pareto front where contraction order quality is balanced with optimization runtime.

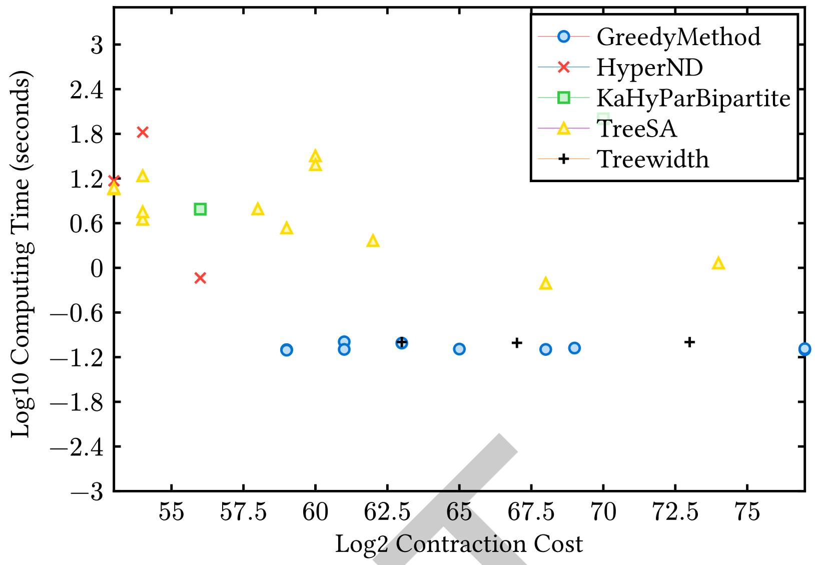

Figure 2: Benchmark results on finding the optimal contraction order for the tensor network mapped from the Sycamore quantum circuit. The device is an Apple M2 CPU. The x-axis shows contraction cost, the y-axis shows optimization time. Each point represents a different optimizer configuration. `TreeSA` and `HyperND` are the best in terms of the contraction cost, while `GreedyMethod` is the fastest in terms of the optimization time.

[JG: TODO: We need a benchmark with CoTengra optimizer, maybe just benchmark two algorithms: TreeSA and HyperND (Richard fill in), please also cite CliqueTree paper, and the benchmark result could be stored in json format, I can handle the visualization to make the plots consistent.]

OMECO has been integrated into the `OMEinsum` package, and is used in `Yao`(Luo et al., 2020) for quantum circuit simulation, `GenericTensorNetworks`(Liu et al., 2023) and `TensorBranching` (TODO: add citation) for solving combinatorial optimization problems, `TensorInference` (Roa-Villescas & Liu, 2023) for exact probabilistic inference, and `TensorQEC` (TODO: add citation) for quantum error correction. We expect this infrastructure to be useful for other applications that needs tree decomposition or path decomposition as well, e.g. the polynomial optimization (Magron & Wang, 2021).

Usage Example

OMECO provides two main functions: `optimize_code` for finding optimal contraction orders, and `slice_code` for trading time complexity for reduced space complexity through the slicing technique.

To demonstrate basic usage, we generate a random 3-regular graph with 100 vertices using the `Graphs` package, associating each vertex with a binary variable and each edge with a 2×2 tensor.

```
julia> using Graphs: random_regular_graph, edges, vertices

julia> using OMEinsumContractionOrders: EinCode, uniquelabels, contraction_complexity, o

julia> function demo_network(n::Int)
           g = random_regular_graph(n, 3)
           code = EinCode([[e.src, e.dst] for e in edges(g)], Int[])
           sizes = Dict(i=>2 for i in uniquelabels(code))
```

```
                tensors = [randn([sizes[index] for index in ix]...) for ix in code.ixs]
                return code, tensors, sizes
            end
demo_network (generic function with 1 method)

julia> code, tensors, sizes = demo_network(100);
```

The tensor network topology is represented by an `EinCode` object with two fields: `ixs` (a vector of index vectors for each input tensor) and `iy` (output indices). This structure defines a hypergraph with potentially open edges. Combining this hypergraph with tensor sizes determines the contraction complexity.

```
julia> contraction_complexity(code, sizes)
Time complexity: 2^100.0
Space complexity: 2^0.0
Read-write complexity: 2^9.231221180711184
```

The return type contains three fields (`tc`, `sc`, `rwc`) for time, space, and read-write complexity. Without optimization, the time complexity is 2^{100}, equivalent to brute-force enumeration.

We now use the `TreeSA` optimizer to find an improved contraction order.

```
julia> optcode = optimize_code(code, sizes, TreeSA(; score=ScoreFunction(tc_weight=1.0,

julia> cc = contraction_complexity(optcode, sizes)
Time complexity: 2^17.241796993093228
Space complexity: 2^13.0
Read-write complexity: 2^16.360864226366807
```

The `optimize_code` function takes three arguments: the `EinCode` object, tensor size dictionary, and optimizer configuration. It returns a `NestedEinsum` object with time complexity $\approx 2^{17.2}$, dramatically improved from the original 2^{100}. This result aligns with theory, as the treewidth of a 3-regular graph is approximately upper bounded by $1/6$ of the number of vertices (Fomin & Høie, 2006). The `score` keyword argument configures the cost function weights; here we set the read-write weight to $10\times$ the time weight, reflecting the higher cost of memory access.

Space complexity can be further reduced using `slice_code`, which implements the slicing technique to trade time for space.

```
julia> sliced_code = slice_code(optcode, sizes, TreeSASlicer(score=ScoreFunction(sc_targ

julia> sliced_code.slicing
3-element Vector{Int64}:
 14
 76
 60

julia> contraction_complexity(sliced_code, sizes)
Time complexity: 2^17.800899899920303
Space complexity: 2^10.0
Read-write complexity: 2^17.199595668955244
```

The `slice_code` function takes the `NestedEinsum` object, tensor sizes, and slicing strategy. Using `TreeSASlicer`, we reduce space complexity by 2^3 (from 2^{13} to 2^{10}) with only a modest increase in time complexity. The resulting `SlicedEinsum` object maintains the same interface as `NestedEinsum` for contraction evaluation.

```
julia> @assert sliced_code(tensors...) ≈ optcode(tensors...)
```

[JG: TODO: Mention the API to convert between contraction graph and treewidth. (Xuan-Zhao fill in)]

[JG: TODO: Show a plot about using slicing to reduce the space complexity (based on the above example). (Xuan-Zhao fill in)]

Real-world examples demonstrating applications to quantum circuit simulation, combinatorial optimization, and probabilistic inference are available in the OMEinsumContractionOrders-Benchmark repository. Optimizer performance is highly problem-dependent, with no single algorithm dominating across all metrics and graph topologies.

Acknowledgments

This work was partially funded by Google Summer of Code 2024, and open source promotion plan (OSPP 2023). We thank Feng Pan for insightful discussions and code contributions on the slicing technique.

References

Supporting

[JG: we will clean up this part in the future]

Definition (Tree decomposition and treewidth): A *tree decomposition* of a (hyper)graph $G = (V, E)$ is a tree $T = (B, F)$ where each node $B_i \in B$ contains a subset of vertices in V (called a "bag"), satisfying:

1. Every vertex $v \in V$ appears in at least one bag.
2. For each (hyper)edge $e \in E$, there exists a bag containing all vertices in e.
3. For each vertex $v \in V$, the bags containing v form a connected subtree of T.

The *width* of a tree decomposition is the size of its largest bag minus one. The *treewidth* of a graph is the minimum width among all possible tree decompositions.

The line graph of a tensor network is a graph where vertices represent indices and edges represent tensors sharing those indices. The relationship between a tensor network's contraction order and the tree decomposition of its line graph can be understood through several key correspondences:

- Each leg (index) in the tensor network becomes a vertex in the line graph, while each tensor becomes a hyperedge connecting multiple vertices.
- The tree decomposition's first two requirements ensure that all tensors are accounted for in the contraction sequence - each tensor must appear in at least one bag, with each bag representing a contraction step.
- The third requirement of the tree decomposition maps to an important constraint in tensor contraction: an index can only be eliminated after considering all tensors connected to it.
- For tensor networks with varying index dimensions, we can extend this relationship to weighted tree decompositions, where vertex weights correspond to the logarithm of the index dimensions.

The figure below illustrates these concepts with (a) a tensor network containing four tensors T_1, T_2, T_3 and T_4 and eight indices labeled A through H, (b) its corresponding line graph, and (c) a tree decomposition of that line graph.

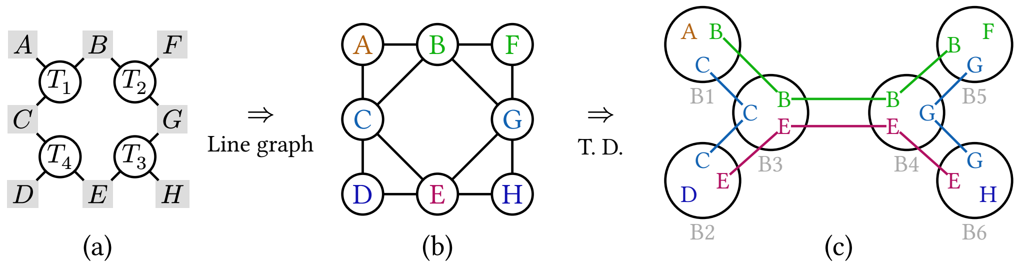

Figure 3: (a) A tensor network. (b) A line graph for the tensor network. Labels are connected if and only if they appear in the same tensor. (c) A tree decomposition (T. D.) of the line graph.

The tree decomposition in Figure 3(c) consists of 6 bags, each containing at most 3 indices, indicating that the treewidth of the tensor network is 2. The tensors T_1, T_2, T_3 and T_4 are contained in bags B_1, B_5, B_6 and B_2 respectively. Following the tree structure, we perform the contraction from the leaves. First, we contract bags B_1 and B_2 into B_3, yielding an intermediate tensor $I_{14} = T_1 * T_4$ (where "$*$" denotes tensor contraction) with indices B and E. Next, we contract bags B_5 and B_6 into B_4, producing another intermediate tensor $I_{23} = T_2 * T_3$ also with indices B and E. Finally, contracting B_3 and B_4 yields the desired scalar result.

Bezanson, J., Karpinski, S., Shah, V. B., & Edelman, A. (2012). Julia: A fast dynamic language for technical computing. *arXiv:1209.5145 [Cs]*. https://doi.org/10.48550/arXiv.1209.5145

Bishop, C. M., & Nasrabadi, N. M. (2006). *Pattern recognition and machine learning* (Vol. 4). Springer.

Bouchitté, V., & Todinca, I. (2001). Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM Journal on Computing*, *31*(1), 212–232.

Fomin, F. V., & Høie, K. (2006). Pathwidth of cubic graphs and exact algorithms. *Information Processing Letters*, *97*(5), 191–196. https://doi.org/10.1016/j.ipl.2005.10.012

Gray, J., & Kourtis, S. (2021). Hyper-optimized tensor network contraction. *Quantum*, *5*, 410. https://doi.org/10.22331/q-2021-03-15-410

Haegeman, J., Lubich, C., Oseledets, I., Vandereycken, B., & Verstraete, F. (2016). Unifying time evolution and optimization with matrix product states. *Physical Review B*. https://doi.org/10.1103/PhysRevB.94.165116

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Kalachev, G., Panteleev, P., & Yung, M.-H. (2021). *Recursive multi-tensor contraction for XEB verification of quantum circuits*. https://arxiv.org/abs/2108.05665

Liu, J.-G., Gao, X., Cain, M., Lukin, M. D., & Wang, S.-T. (2023). Computing solution space properties of combinatorial optimization problems via generic tensor networks. *SIAM Journal on Scientific Computing*, *45*(3), A1239–A1270.

Luo, X.-Z., Liu, J.-G., Zhang, P., & Wang, L. (2020). Yao. Jl: Extensible, efficient framework for quantum algorithm design. *Quantum*, *4*, 341.

Magron, V., & Wang, J. (2021). TSSOS: A julia library to exploit sparsity for large-scale polynomial optimization. *arXiv:2103.00915*.

Markov, I. L., & Shi, Y. (2008). Simulating Quantum Computation by Contracting Tensor

Networks. *SIAM Journal on Computing*, *38*(3), 963–981. https://doi.org/10.1137/050644756

Piveteau, C., Chubb, C. T., & Renes, J. M. (2024). Tensor-Network Decoding Beyond 2D. *PRX Quantum*, *5*(4), 040303. https://doi.org/10.1103/PRXQuantum.5.040303

Qing, Y., Li, K., Zhou, P.-F., & Ran, S.-J. (2024). *Compressing neural network by tensor network with exponentially fewer variational parameters* (No. arXiv:2305.06058). arXiv. https://doi.org/10.48550/arXiv.2305.06058

Roa-Villescas, M., Gao, X., Stuijk, S., Corporaal, H., & Liu, J.-G. (2024). Probabilistic Inference in the Era of Tensor Networks and Differential Programming. *Physical Review Research*, *6*(3), 033261. https://doi.org/10.1103/PhysRevResearch.6.033261

Roa-Villescas, M., & Liu, J.-G. (2023). TensorInference: A julia package for tensor-based probabilistic inference. *Journal of Open Source Software*, *8*(90), 5700.

Villescas, M. R., Liu, J.-G., Wijnings, P. W. A., Stuijk, S., & Corporaal, H. (2023). Scaling Probabilistic Inference Through Message Contraction Optimization. *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*, 123–130. https://doi.org/10.1109/CSCE60160.2023.00025