The Journal of Open Source Software

OMEinsumContractionOrders: A Julia package for tensor network contraction order optimization

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

1 Hong Kong University of Science and Technology (Guangzhou) **2** Center of Computational Mathematics, Flatiron Institute **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Published: unpublished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.

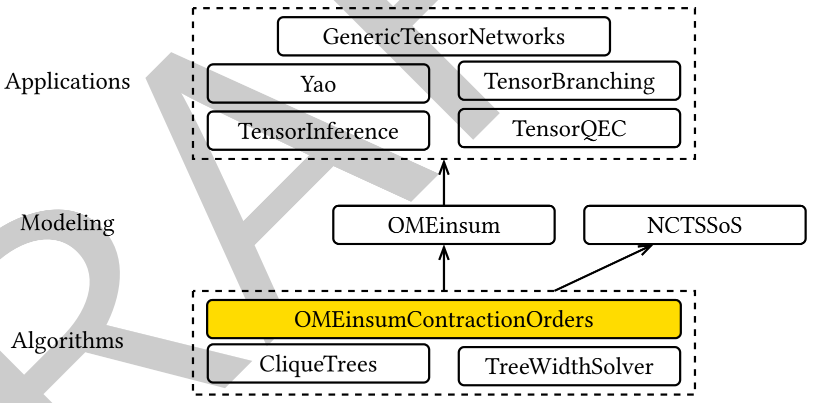

Figure 1: The ecosystem built around `OMEinsumContractionOrders` and its dependencies. OMECO serves as a core component of the tensor network contractor `OMEinsum`, which powers applications including `Yao` (quantum simulation), `TensorQEC` (quantum error correction), `TensorInference` (probabilistic inference), `GenericTensorNetworks` and `TensorBranching` (combinatorial optimization).

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 applications are reflected in the ecosystem built around OMECO, as illustrated 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:

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

where w_{t}, w_{s}, and w_{rw} represent weights for time complexity (tc), space complexity (sc), and read-write complexity (rwc), respectively. In practice, memory access costs typically dominate computational costs, motivating $w_{\mathrm{rw}} > w_{\mathrm{t}}$. The space complexity penalty activates only when $\mathrm{sc} > \mathrm{sc}_{\mathrm{target}}$, allowing unconstrained optimization when memory fits within 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` operate on the tensor network's line graph and utilize the `CliqueTrees` and `TreeWidthSolver` packages, as illustrated in Figure 1. Additionally, the `PathSA` optimizer implements path decomposition by constraining contraction orders to path graphs, serving as a variant of `TreeSA`.

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.

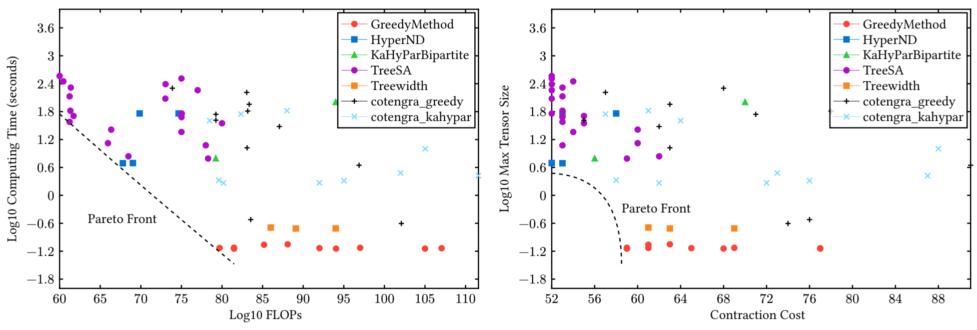

Figure 2: Time complexity (left) and space complexity (right) benchmark results for contraction order optimization on the Sycamore quantum circuit tensor network (Intel Xeon Gold 6226R CPU @ 2.90GHz, single-threaded). The x-axis shows contraction cost, y-axis shows optimization time. Each point represents a different optimizer configuration tested with varying parameters. `TreeSA` and `HyperND` achieve the lowest contraction costs, while `GreedyMethod` offers the fastest optimization time.

[JG: TODO: Please also cite CliqueTree paper.]

Optimizers prefixed with `cotengra_` are from the Python package cotengra (Gray & Kourtis, 2021); all others are OMECO implementations. For both optimization objectives (minimizing time and space complexity), OMECO optimizers dominate the Pareto front. Given sufficient optimization time, `TreeSA` consistently achieves the lowest time and space complexity. `GreedyMethod` provides the fastest optimization but yields suboptimal contraction orders, while `HyperND` offers a favorable balance between optimization time and solution quality.

OMECO has been integrated into the `OMEinsum` package and powers several downstream applications: `Yao` (Luo et al., 2020) for quantum circuit simulation, `GenericTensorNetworks` (Liu et al., 2023) and `TensorBranching` (TODO: add citation) for combinatorial optimization, `TensorInference` (Roa-Villescas & Liu, 2023) for probabilistic inference, and `TensorQEC` for quantum error correction. This infrastructure is expected to benefit other applications requiring tree or path decomposition, such as 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 specifying the contraction tree with three fields: `args` (child nodes), `tensorindex` (input tensor index for leaf nodes), and `eins` (einsum notation for the node). The time complexity $\approx 2^{17.2}$ is 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, returning a `SlicedEinsum` object with two fields: `slicing` (sliced indices) and `eins` (a `NestedEinsum` object). 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), Remove?]

[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 the Open Source Promotion Plan (OSPP 2023). We thank Feng Pan for insightful discussions and code contributions on the slicing technique.

References

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