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** University of Florida * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Johan Larsson

Reviewers:

- @meandmytram
- @epatters

Submitted: 01 December 2025
Published: unpublished

Summary

`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. OMECO is designed to search for near-optimal contraction orders for exact tensor network contraction, and provides a comprehensive suite of optimization algorithms for tensor network contraction orders, including greedy heuristics, simulated annealing, and tree width solvers. In this paper, we present the key features of OMECO, its integration with the Julia ecosystem, and performance benchmarks.

Statement of need

A *tensor network* is a mathematical structure that represents multilinear transformations as hypergraphs. Arrays—called *tensors*—correspond to nodes, and shared indices correspond to hyperedges. To *contract* a tensor network is to evaluate the transformation on a collection of tensors by performing a sequence of pairwise bilinear operations. The computational cost—both running time and memory usage—depends critically on the order in which these operations are performed. A specific choice of ordering is called a *contraction order*, and the problem of finding an efficient ordering is called *contraction order optimization*.

Notably, this optimization task is analogous to compilation: a scheduler compiles a tensor network specification into an executable plan that can be evaluated efficiently. It is important to distinguish the *optimization time* (how long the scheduler runs) from the *contraction complexity* (the time and memory required to execute the resulting plan). An effective scheduler must balance these competing objectives—spending sufficient time to find a high-quality contraction order while remaining computationally tractable itself.

The tensor network 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. Applications span 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 (J.-G. Liu et al., 2023).

A contraction 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 $w_{\text{rw}} > w_{\text{t}}$. The space complexity penalty activates only when $\text{sc} > \text{sc}_{\text{target}}$, 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). However, the problem exhibits fixed-parameter tractability: for tensor networks with bounded tree-width, optimal contraction orders can be found—and the resulting contractions executed—in polynomial time. This connection to tree decomposition motivates several of OMECO's optimization strategies, which leverage graph-theoretic techniques to exploit this structure.

Algorithms for finding near-optimal contraction orders have been developed and achieve impressive scalability (Gray & Kourtis, 2021; Roa-Villescas et al., 2024), handling tensor networks with over 10^3 tensors. While the Python package `cotengra` (Gray & Kourtis, 2021) has been widely adopted in the community, achieving optimal performance across diverse problem instances—particularly when balancing solution quality against optimization time constraints—remains an open challenge.

OMECO addresses this challenge through a unified and extensible framework that integrates multiple complementary optimization strategies, including greedy heuristics, simulated annealing, and tree-width-based solvers. This comprehensive approach enables more systematic exploration of the optimization time-solution quality trade-off space. OMECO has been integrated into the `OMEinsum` package and powers several downstream applications: `Yao` (Luo et al., 2020) for quantum circuit simulation, `GenericTensorNetworks` (J.-G. Liu et al., 2023) and `TensorBranching` 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). These applications are reflected in the ecosystem built around OMECO, as illustrated in Figure 1.

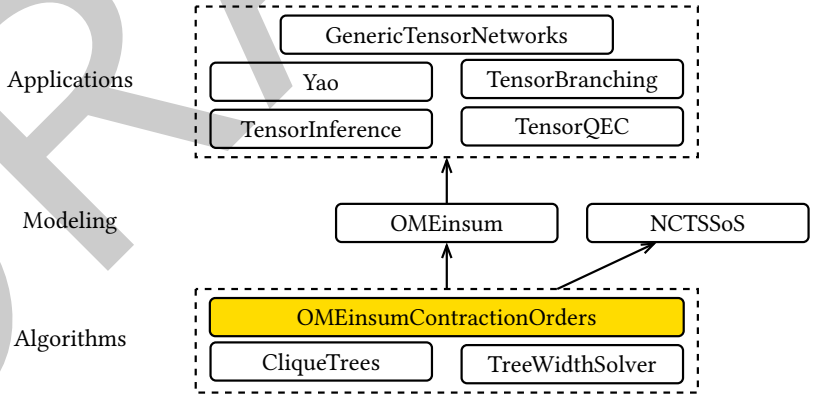

Figure 1: The ecosystem built around `OMEinsumContractionOrders` and its dependencies. OMECO serves as a core component of the tensor network contractor `OMEinsum`, which powers applications including `Yao` (quantum simulation), `TensorQEC` (quantum error correction), `TensorInference` (probabilistic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

State of the field

Several software packages support contraction order optimization. The Python package `cotengra` (Gray & Kourtis, 2021) provides a hyper-optimization framework that combines greedy methods, simulated annealing, and graph partitioning to find high-quality contraction orders. However, integrating Python packages into Julia workflows introduces challenges in dependency version control and environment reproducibility. The `opt_einsum` package

(Smith & Gray, 2018) provides contraction optimization for einsum-style expressions using dynamic programming and greedy-based heuristics, while `quimb` (Gray, 2018) offers tensor network capabilities using `cotengra` as its optimization backend. In the Julia ecosystem, `TensorOperations.jl` (Devos et al., 2023) implements exact contraction order optimization, but is limited to tensor networks without hyperedges (where each index appears in at most two tensors) and lacks scalability to large networks. `ITensors.jl` (Fishman et al., 2022) primarily focuses on physics applications such as MPS and DMRG algorithms, rather than general-purpose contraction order optimization.

Software Design

OMECO provides the first open-source implementation of the TreeSA algorithm (Kalachev et al., 2021), a powerful simulated annealing approach that operates directly on contraction tree structures. Beyond TreeSA, OMECO implements a comprehensive suite of optimization algorithms from the literature, including nested dissection methods for hypergraphs, bipartition-based approaches, and exact tree decomposition solvers. While the Python package `cotengra` (Gray & Kourtis, 2021) exists in this space, we chose to build OMECO as a native Julia implementation to provide these capabilities to the Julia scientific computing ecosystem without language barriers, enabling new applications in quantum computing and combinatorial optimization that require tight integration with Julia's tensor network infrastructure.

OMECO's architecture emphasizes modularity and extensibility. The core abstraction separates the contraction order representation (as trees or paths) from the optimization algorithms, enabling users to compose different strategies and implement custom optimizers. The `optimize_code` function provides a unified interface across eight different optimization methods, from fast greedy heuristics to high-quality tree decomposition solvers. Integration with graph partitioning libraries (`KaHyPar`, `Metis`) and tree decomposition packages (`CliqueTrees` (Samuelson & Fairbanks, 2025)) demonstrates OMECO's interoperability within the Julia ecosystem. The native Julia implementation enables tight coupling between optimization and execution phases in `OMEinsum`, supporting diverse numeric types through multiple dispatch. This design has proven successful: OMECO serves as the optimization backend for multiple packages across quantum simulation, probabilistic inference, and combinatorial optimization domains.

Research Impact Statement

OMECO has demonstrated significant research impact through both technical performance and ecosystem adoption. Benchmarking against the widely-used `cotengra` package on standard tensor network problems—including the Sycamore quantum circuit and problems from quantum simulation, probabilistic inference, and combinatorial optimization—shows that OMECO optimizers consistently dominate the Pareto front for the trade-off between optimization time and solution quality. The `TreeSA` algorithm achieves lower time and space complexity than existing methods when given sufficient optimization time, while `HyperND` provides superior balance for time-constrained scenarios.

Beyond performance benchmarks, OMECO has achieved substantial ecosystem adoption within the Julia scientific computing community. It powers the tensor contraction backend for `Yao` (a quantum computing framework with over 1000 GitHub stars), `GenericTensorNetworks` and `TensorBranching` (combinatorial optimization libraries used for solving constraint satisfaction and graph problems), `TensorInference` (probabilistic inference on graphical models), and `TensorQEC` (quantum error correction code analysis). These packages collectively support research across quantum computing, statistical physics, and discrete optimization, with impactful publications (Ebadi et al., 2022; Gao et al., 2024; J.-G. Liu et al., 2023; Roa-Villescas et al., 2024).

The software exhibits strong community-readiness signals: comprehensive documentation with examples, extensive test coverage (>90%), active maintenance with regular releases, MIT open-source license, and well-defined contribution processes. The modular architecture has enabled external contributors to implement new optimization algorithms and extend functionality to emerging application domains.

Features and benchmarks

The major feature of OMECO is contraction order optimization. OMECO provides several algorithms with complementary performance characteristics that can be simply called by the `optimize_code` function:

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 (Samuelson & Fairbanks, 2025)

The algorithms `HyperND`, `Treewidth`, and `ExactTreewidth` are tree-width based solvers that operate on graphs. They first convert tensor networks to their line graph representation (Markov & Shi, 2008) and then find an optimized tree decomposition of the line graph using the `CliqueTrees` and `TreeWidthSolver` packages, as illustrated in Figure 1. Additionally, the `PathSA` optimizer optimizes path decomposition instead of tree decomposition. It is a variant of `TreeSA` by constraining contraction orders to path graphs, which is useful for applications requiring a linear contraction order.

These methods balance optimization time against solution quality. Figure 2 displays benchmark results for the tensor network of the Sycamore quantum circuit (Arute et al., 2019; Pan & Zhang, 2021), which is widely used as a benchmark for quantum supremacy and is believed to have an optimal space complexity of 52. The Pareto front highlights the optimal trade-off between optimization time and solution quality.

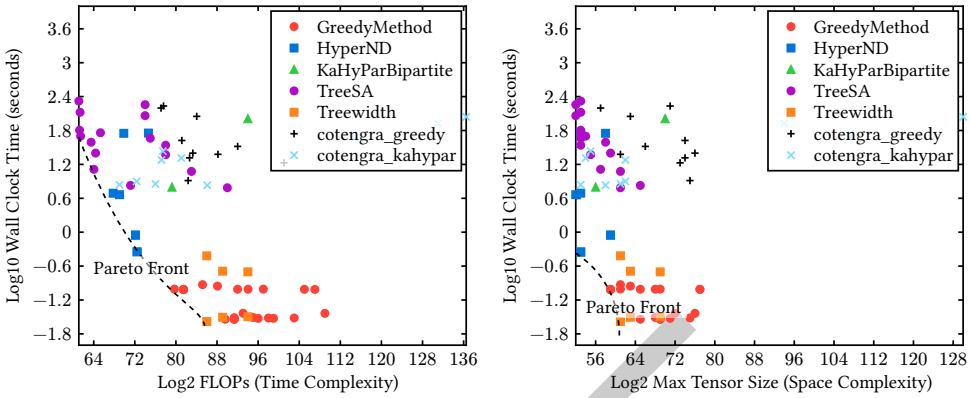

Figure 2: Time complexity (left) and space complexity (right) benchmark results for contraction order optimization on the Sycamore quantum circuit tensor network (Intel Xeon Gold 6226R CPU @ 2.90GHz, single-threaded). The x-axis shows contraction cost, y-axis shows optimization time. Each point represents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

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` and `Treewidth` (backed by minimum fill (MF) (Ng & Peyton, 2014), multiple minimum degree (MMD) (J. W. Liu, 1985), and approximate minimum fill (AMF) (Rothberg & Eisenstat, 1998)) provide the fastest optimization but yield suboptimal contraction orders, while `HyperND` offers a favorable balance between optimization time and solution quality.

More real-world examples demonstrating applications to quantum circuit simulation, combinatorial optimization, and probabilistic inference are available in the OMEinsumContractionOrdersBe repository. We find that optimizer performance is highly problem-dependent, with no single algorithm dominating across all metrics and graph topologies.

Another key feature of OMECO is index slicing, a technique that trades time complexity for reduced space complexity by explicitly looping over a subset of tensor indices. OMECO provides the `slice_code` interface for this purpose, currently supporting the `TreeSASlicer` algorithm, which implements dynamic slicing based on the `TreeSA` optimizer. Figure 3 demonstrates this capability using the Sycamore quantum circuit, where slicing reduces the space complexity from 2^{52} to 2^{31}.

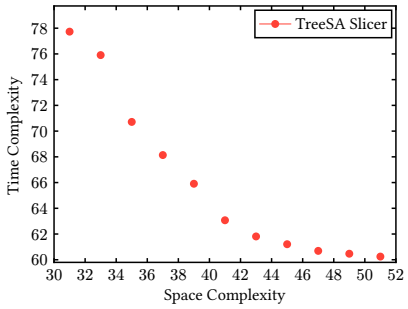

Figure 3: Trade-off between time complexity and target space complexity using `TreeSASlicer` on the Sycamore quantum circuit. The original network has a space complexity of 2^{52}.

The numerical experiments show that moderate slicing increases time complexity only slightly, while aggressive slicing can induce significant overhead. There is a critical transition point around 42 where the time complexity begins to increase significantly.

Acknowledgements

We thank the Julia community and all contributors to the `OMEinsum` and `OMEinsumContractionOrders` packages. We are grateful to Feng Pan for his effort in improving the slicing algorithm, and Xiwei Pan for valuable writing suggestions that improved this manuscript.

AI Usage Disclosure

Generative AI tools were used to assist with paper writing and documentation. The software implementation, algorithmic design, benchmarking, and core technical contributions were developed without AI assistance. All AI-assisted content has been reviewed and validated by the authors for technical accuracy and scholarly integrity.

References

Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J. C., Barends, R., Biswas, R., Boixo, S., Brandao, F. G., Buell, D. A., & others. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, *574*(7779), 505–510. https://www.nature.com/articles/s41586-019-1666-5

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. https://doi.org/10.1137/s0097539799359683

Devos, L., Damme, M. V., Haegeman, J., & contributors. (2023). *TensorOperations.jl* (Version v4.0.7). https://doi.org/10.5281/zenodo.3245496

Ebadi, S., Keesling, A., Cain, M., Wang, T. T., Levine, H., Bluvstein, D., Semeghini, G., Omran, A., Liu, J.-G., Samajdar, R., & others. (2022). Quantum optimization of maximum independent set using rydberg atom arrays. *Science*, *376*(6598), 1209–1215. https://doi.org/10.1126/science.abo6587

Fishman, M., White, S. R., & Stoudenmire, E. M. (2022). The ITensor Software Library for Tensor Network Calculations. *SciPost Phys. Codebases*, 4. https://doi.org/10.21468/SciPostPhysCodeb.4

Gao, X., Kalinowski, M., Chou, C.-N., Lukin, M. D., Barak, B., & Choi, S. (2024). Limitations of linear cross-entropy as a measure for quantum advantage. *PRX Quantum*, *5*(1), 010334. https://doi.org/10.1103/PRXQuantum.5.010334

Gray, J. (2018). Quimb: A python package for quantum information and many-body calculations. *Journal of Open Source Software*, *3*(29), 819. https://doi.org/10.21105/joss.00819

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. W. (1985). Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software (TOMS)*, *11*(2), 141–153. https://doi.org/10.1145/214392.214398

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. https://doi.org/10.1137/22m1501787

Luo, X.-Z., Liu, J.-G., Zhang, P., & Wang, L. (2020). Yao. Jl: Extensible, efficient framework for quantum algorithm design. *Quantum*, *4*, 341. https://doi.org/10.22331/q-2020-10-11-341

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

Ng, E. G., & Peyton, B. W. (2014). Fast implementation of the minimum local fill ordering heuristic. *CSC14: The Sixth SIAM Workshop on Combinatorial Scientific Computing*, 4. https://doi.org/10.1137/070680680

Pan, F., & Zhang, P. (2021). *Simulating the sycamore quantum supremacy circuits*. https://arxiv.org/abs/2103.03074

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. https://doi.org/10.21105/joss.05700

Rothberg, E., & Eisenstat, S. C. (1998). Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, *19*(3), 682–695. https://doi.org/10.1137/s0895479896302692

Samuelson, R., & Fairbanks, J. (2025). *CliqueTrees.jl: A julia library for computing tree decompositions and chordal completions of graphs*. https://github.com/AlgebraicJulia/CliqueTrees.jl

Smith, D. G. A., & Gray, J. (2018). Opt_einsum - a python package for optimizing contraction

order for einsum-like expressions. *Journal of Open Source Software*, *3*(26), 753. https://doi.org/10.21105/joss.00753

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