



Electronic Communications of the EASST
Volume 86 (1) Year 2026

deRSE26 - Selected Contributions of the 6th Conference for Research Software Engineering in Germany (Part 1)

Edited by Bernd Flemisch, René Caspart, Bernadette Fritzsch, Florian Goth, Jean-Noël Grad, Florian Thiery

Algorithmic algorithm development with LLMs: A Case Study on LLM-Usage for Contraction Order Optimization in Tensor Networks

Fabian Hoppe, Melven Röhrig-Zöllner, Philipp Knechtges

DOI: 10.14279/eceasst.v86.1.2781

License: This article is licensed under a CC-BY 4.0 License.

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

Algorithmic algorithm development with LLMs: A Case Study on LLM-Usage for Contraction Order Optimization in Tensor Networks

Fabian Hoppe, Melven Röhrig-Zöllner, Philipp Knechtges

German Aerospace Center (DLR), Institute of Software Technology, High-Performance Computing, Cologne (Germany)

Abstract: We consider LLM-based algorithm development through a case study on contraction-order optimization for tensor networks with OpenEvolve. We pay particular attention to the choice of the LLM as well as design choices such as evaluation metric and test instances. Our results highlight both the promise of verifier-guided evolutionary coding agents for algorithm development/improvement and the continuing importance of evaluation, validation, and interpretation—and corresponding challenges—by the human scientist.

Keywords: Large Language Model, LLM, algorithm engineering, evolutionary coding agents, OpenEvolve, tensor networks, contraction order optimization

1 Introduction

As in many areas, LLM-based systems are increasingly used in science in recent times. In addition to their widespread use for coding assistance and auxiliary tasks such as literature research, summarisation, etc., the potential of AI for core scientific tasks is increasingly being taken into account. End-to-end "AI scientist" systems [LLLa26, YLLa25, GWDa25, MYCa25] aim to automate larger parts of the research cycle, including idea generation, coding and experimentation, and manuscript preparation—up to now, however, their capabilities appear to be limited, as critical assessments show [BKB25]. In this paper we take a step back and consider LLM usage for a core scientific task, namely algorithm development and/or improvement, but without end-to-end, i.e., idea-to-paper, automation. We put particular emphasis on the design decisions and assumptions that the human scientist must provide to the respective AI system.

Regarding contributions to coding assistance tailored to science, we mention exemplarily the SciCode benchmark [TGZa24] and agentic approaches to scientific computing such as Re4 [CZH26] and JutulGPT [LMSa26]. During the last two years, several approaches for LLM-based algorithm engineering, often also overlapping with performance engineering, searching for heuristics, or even broader applicability, have been presented. In the broader context we mention, e.g., "Evolution of Heuristics" (EoH, [LTYa24]), [YWCa24], RedAHD [TRHC25], [ZWZ26], and [ŠMQa25]. More closely to the topic investigated in the present paper are FunSearch [RBNa24], AlphaEvolve [NVEa25], OpenEvolve, GigaEvo [KGBa25], and CodeEvolve [AFCa25], as well as ShinkaEvolve [LIC25], DeepEvolve [LZCa25], LLM4AD [LZX+24], ThetaEvolve [WSZa25], Escher-Loop [LGWa26], and AlgoTuner [PA+25]. Roughly speaking, EoH, ReEvo, RedAHD, and G-LNS mainly address automated heuristic design for combinato-

rial optimization, whereas FunSearch and the AlphaEvolve/OpenEvolve line search directly over executable program variants under automated evaluation; LLM4AD, in turn, is best viewed as a unifying platform rather than one specific search strategy. For systematic evaluation of such systems, benchmarks like ALE-Bench [IHIa] or the AlgoTune benchmark [PA+25] have been introduced; ALE-Bench focuses on long-horizon score-based optimization tasks, whereas AlgoTune evaluates speedups on expert-provided numerical programs under correctness and runtime constraints. Often, however, an ad-hoc selection of problems from various areas is used. Applications include, but are not limited to, e.g., mathematical discovery [GGTa25], systems performance research [CLPa25], automated discovery of language model scaling laws [LYFa26], finding new acquisition functions in Bayesian optimization [AKSa24], and performance optimization [NGWI25].

At the same time, first challenges with the use of LLM-based systems in scientific software development are emerging. As large quantities of software can increasingly be produced automatically without major effort, authorship and appreciation may increasingly be shifting towards high-level design, verification by humans and the associated assumption of responsibility, as well as transparency and sustainability; the JOSS governance documents[1] can be interpreted as an indication for this shift. Furthermore, there are concerns regarding stability [RBSa25], reproducibility [SISa25], digital sovereignty (especially for sensitive areas of research), and impacts on the social, organizational, and career aspects of research software. In fact, even our concept of what software engineering actually is, could be changing, e.g., by extending the object of engineering from executable code to "semi-executable artifacts" [FLFa26] such as prompts, workflows, evaluation harnesses, controls, and organizational routines. Also, there are similar, albeit more philosophical, considerations on a possible change in the self-image of mathematical-scientific work due to AI [KT26]. Closer to the present scope is recent work on human-AI-interaction for mathematical discovery with AlphaEvolve [BCNa26]: it is argued that this process is not a one-off question-answer interaction, but rather an iterative process of "intentmaking"—defining and sharpening the goal, setting up and updating the experiment—and "sensemaking"—the interpretation, validation and debugging of the results. Consequently, it is suggested that such kind of system should be viewed and used as collaborative scientific instruments rather than as question-answer assistants.

The aim of this publication is not to tell a "success story" for one or more specific problems. Instead, we want to use a manageable, but non-trivial and realistic example to analyse the factors that contribute to success or failure. In other words, given a concrete algorithmic problem and a concrete LLM-based system for algorithm search, we consider, as an example, the design of the experiment to be provided by the human scientist and analyse its effects as systematically as possible. Specifically, we use OpenEvolve to determine a contraction sequence with low FLOP count for a special class of tensor networks. As will be explained in more detail in Section 2.2 below, the problem of determining a contraction order with as few FLOPs as possible for a given tensor network is difficult to solve, but easy to verify: for a given contraction order, the number of FLOPs can be calculated without the actual execution of the contraction and at the same time one can check whether the given order is permissible. In fact, it is sufficient

[1] see, e.g., https://joss.readthedocs.io/en/latest/submitting.html, https://blog.joss.theoj.org/2026/01/preparing-joss-for-a-generative-ai-future [Accessed 05/2026]

to know only the shape of all tensors involved, whereas "real" tensors available in memory are not necessary. This suggests a strong conceptual fit to what OpenEvolve can do: evolving code on the basis of a reliable verification/scoring routine—for more details see Section 2.1 below. The fact that during the evaluation only the candidate program (i.e. contraction order determination) but not the contraction itself needs to be evaluated on the respective test examples seems to suggest an unusually cheap setting for computing scores. Moreover, by avoiding actual performance measurements that would be hardware dependent and noisy, the overall setting becomes more controllable. This enables us, in particular, to carry out the experiments in Python; nevertheless, the overall approach applies in principle to other programming languages supported by the underlying LLM and situations that require actual performance measurements, too.

Related but distinct work has recently used LLMs for tensor-network *structure* search rather than contraction ordering [ZLSa24, IZLa25]. In the broader context of quantum computing, GigaEvo has recently been applied to algorithm improvement for quantum compilation [FKSa26]. To the best of our knowledge, however, there are no publications that apply LLM-based algorithm search specifically to the problem of tensor network contraction ordering yet. Even though our test problem seems to be very well suited for OpenEvolve and unusually benign, the situation is by no means trivial, as we will experimentally show in the following sections. We examine the influence of the (open) LLMs used, the sample data used during optimization and the metric used for optimization, and then analyse the "best" result found in detail. In view of the large number of different approaches and the extremely rapid development of the entire environment, the results presented here, which are limited to one specific approach, OpenEvolve, are necessarily limited, of course. Nevertheless, we believe that our detailed observations and considerations on the influencing factors in automatic algorithm search are relevant and can be generalised to similar problems at least in the broadest sense. Moreover, given the rapid progress in this field, the snapshot presented here may one day be of "historical" interest.

Structure of the Paper. In Section 2 we provide the reader with the necessary background information on OpenEvolve and the problem of tensor network contraction. The overall experimental setup is described at the beginning of Section 3, followed by the main results of different experiments and an in-depth analysis of the "best" result. Concluding considerations and lessons learned are stated in Section 4 and additional technical details and results are provided in the Appendix.

2 Background

2.1 OpenEvolve

OpenEvolve[2] can be understood as an *evolutionary coding agent*. It is an open-source Python-based framework inspired by DeepMind's closed-source AlphaEvolve [NVEa25], which itself builds on ideas from the earlier FunSearch approach [RBNa24]. While FunSearch searches in the space of implementations of a designated function inside a fixed scaffold, AlphaEvolve/OpenEvolve generalise this idea to direct edits of larger code artefacts. In its current implemen-

[2] https://github.com/algorithmicsuperintelligence/openevolve [Accessed 05/2026]. We use version 0.2.25.

tation, OpenEvolve combines LLM-generated code mutations with island-based evolutionary search and a MAP-Elites-style quality-diversity archive. The user needs to provide (i) an initial program and (ii) an automated evaluator; OpenEvolve then repeatedly asks one or more LLMs to propose edits, executes the resulting candidates, scores them with the evaluator, and retains or recombines promising variants through evolutionary search. Hence, this approach is suitable for problems with a reliable automatic verifier and a measurable objective. For algorithmic and implementation details we refer the reader to the repository, and only note the practically relevant detail that OpenEvolve allows the explicit propagation of random seeds across major components to improve reproducibility.

Among the applications this approach let us highlight just a few: The original AlphaEvolve white paper reports improvements for circle packing problems and a new multiplication algorithm for 4×4 matrices, but also more practical applications such as data-center scheduling [NVEa25]; further applications, including production level ones, have recently been highlighted in a blog post[3]. OpenEvolve has so far been used for, e.g., combinatorial bijection discovery [BHJa25] and systems-oriented AI-driven research [CLPa25], multi-agent coordination [KSNa26], password guessing [MZKa26], technology mapping [FLXa26], and improved estimates for certain mathematical constants [BNL26].

Being based on population-based verifier-guided search, OpenEvolve differs from other agentic approaches in the field. Compared to budgeted iterative edit-run-select agents such as AlgoTuner [PA+25], it does not primarily follow a single incumbent solution but maintains a population of competing candidate programs and lets external task metrics drive selection. Relative to prompt-evolution approaches such as Promptbreeder or GEPA, it mutates and selects *program source code* rather than prompts or agent policies [FBMa24, ATSa26]. Different from multi-agent scientific assistants such as AI co-scientist [GWDa25], which aim to cover larger parts of the scientific cycle, OpenEvolve performs closed-loop optimization in a program space defined by the user-provided verifier/evaluator. Among the evolutionary coding frameworks in the field, OpenEvolve is by construction closest to AlphaEvolve, whereas CodeEvolve [AFCa25] and GigaEvo [KGBa25] put more emphasis on the orchestration of the evolutionary search, LLM4AD [LZX+24] provides a broader experimental platform for LLM-based algorithm development, ShinkaEvolve [LIC25] focuses strongly on sample-efficient evolution via parent/LLM selection and novelty rejection-sampling, and ThetaEvolve [WSZa25] extends the approach towards in-context and reinforcement learning at test time. Escher-Loop [LGWa26] extends OpenEvolve towards a self-improving evolution system. Finally, [LZCa25] combines OpenEvolve with Deep Research in order to incorporate knowledge beyond the LLMs internal knowledge.

2.2 Tensor Network Contraction

For the purpose of this paper, it suffices to consider a tensor as a multidimensional array with entries indexed by a finite set of *indices* (sometimes also called "modes" or "legs"). For example, a rank-3 tensor A with indices (i, j, k) has entries A_{ijk}, whereas a rank-0 tensor is just a number. A *tensor network* (TN) is a collection of tensors together with an identification of certain indices between tensors, indicating that those indices are to be *summed over*, a procedure referred to as

[3] https://deepmind.google/blog/alphaevolve-impact/ [Accessed 05/2026]

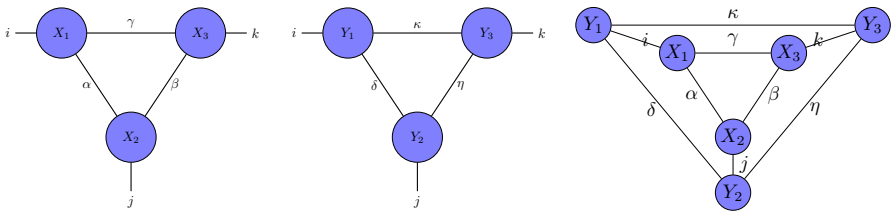


Figure 1: Examples for tensor networks

"contraction". TNs are usually described by graphs (or hypergraphs): vertices represent tensors, and edges represent shared indices. Contracting all shared indices yields either a scalar (if no indices remain) or a tensor (if indices, so-called "free indices", remain). For an introduction we refer the interested reader to, e.g., [Orú14, BB17, BK25].

As an example let us consider two three-dimensional tensors X and Y with free indices i, j, k, both represented as so-called periodic Matrix Product States (MPS, [VPC04]); cf. the two graphs on the left in Figure 1. X is represented as contraction of X_1, X_2, X_3 over the indices α, β, γ, and similarly for Y. Contracting now X and Y along the shared indices i, j, k, which is basically a generalised scalar product between X and Y, results in the tensor network on the right-hand side of Figure 1.

$$\begin{aligned}
&\sum_{i,j,k} X[i,j,k]Y[i,j,k] \\
&= \sum_{i,j,k} \left(\sum_{\alpha,\beta,\gamma} X_1[\gamma,i,\alpha] X_2[\alpha,j,\beta] X_3[\beta,k,\gamma] \right) \left(\sum_{\delta,\eta,\kappa} Y_1[\kappa,i,\delta] Y_2[\delta,j,\eta] Y_3[\eta,k,\kappa] \right) \\
&= \sum_{i,j,k,\alpha,\beta,\gamma,\delta,\eta,\kappa} X_1[\gamma,i,\alpha] X_2[\alpha,j,\beta] X_3[\beta,k,\gamma] Y_1[\kappa,i,\delta] Y_2[\delta,j,\eta] Y_3[\eta,k,\kappa]
\end{aligned}$$

Contracting two tensors is a direct generalization of matrix multiplication and can be realized numerically by repeated reshaping and matrix multiplication (GEMM-like kernel) operations. Here, the order in which the indices are contracted is crucial, as computational complexity and also memory requirements for intermediate results can vary dramatically with respect to the chosen order [Orú14, O'G]. Apparently this is also the case for our example in Figure 1 (cf. also the formula). As for matrix multiplication, the number of FLOPs for contracting a TN given a specified contraction order can be computed without performing the actual computation. Finding optimal contraction orders, however, is in general NP-hard [DFGa18, XZLa23] and thus computationally intractable except for very small instances. Consequently, contraction orders for TNs are often determined using heuristics or meta-heuristics, or by dedicated algorithms for restricted families of networks. We restrict ourselves to a very concise overview over some common approaches:

- Greedy and random-greedy heuristics are fast and simple, and can be surprisingly effective on many instances [SG18, GK21]
- Optimization-based: simulated annealing and genetic algorithms [SJ20]

- (Hyper-)graph- and tree-based methods [MS08, DFGa18, GD04, Str17, Tam, SHGa23, SBK+, ILHa22]
- Hybrid frameworks and search over hyperparameters [GK21, SBK+]
- AI-driven approaches such as reinforcement learning [MMMa, LZ]
- approaches tailored to very specific structures, e.g., tree tensor networks or tensor-train-type scalar products [SMM24, TDPa26].

As we will use cotengra[4], a Python library implementing one of these approaches, [GK21], as baseline in our subsequent experiments, we provide some more details; cotengra is aimed at contraction planning for tensor networks and focuses on the "planning" layer, i.e., finding a high-quality contraction tree/path, that then can be executed using standard tensor primitives. In this context, cotengra provides data structures for tensor networks and routines for computing FLOPs, maximal intermediate sizes, etc. for a given contraction order/tree. On an algorithmic level, multiple heuristics (greedy, partitioning-based, local reconfiguration, slicing) are combined with randomized search/hyper-optimization.

3 Experiments and Results

Obviously, the experiments presented in this Section are only a subset of those one could perform in theory. Our selection may be justified as follows: First, we performed those experiments that were of highest interest to us, namely those influencing variables by means of which the human defines the AI system's room for manoeuvre and defines the goal: the choice of test data and the choice of the metric for evaluation. As there is a large amount of literature on prompting, cf., e.g., [JWSa26, Sect. 5.7], we do not investigate different system messages. Second, among the hyper-parameters of the agentic system itself, we decided to focus on the choice of the LLM as this potentially raises some very delicate questions; in comparison, we consider the analysis of the remaining hyper-parameters (temperature, top_p, etc. for the LLM, and population size, number of islands, sampling temperature, etc. for the evolutionary algorithm) to be less interesting—although clearly relevant for performance—and also as too expensive in terms of compute due to the much larger overall parameter space. Third, we address the difficult issue of reproducibility as much as possible within a reasonable budget of compute. Finally, when addressing the quality of the "best" outcome, we take a pragmatic approach and use a limited number of non-specialized baselines that are available in a single library; our point here is not the actual outcome itself and how good it is (in the sense of an algorithmic contribution), but rather which influencing factors and pitfalls need to be considered on the way there and which questions arise from this. Already focussing on optimizing FLOP counts alone is a very strong design decision, here largely made for the sake of simplicity. For the practical feasibility of a contraction sequence the maximum size of the intermediate is also decisive, and even the runtime is in reality dependent on other influencing variables (e.g., memory speed) that go beyond the pure FLOP count.

[4] https://github.com/jcmgray/cotengra, https://cotengra.readthedocs.io/en/latest/ [Accessed 05/2026]. We use version 0.7.5 for our experiments.

To keep the presentation brief and clear, we limit ourselves to the essentials in the following; the interested reader is referred to the Appendix for further information and details. We generate three different sets of tensor networks referred to as "small", "middle" and "large" in the following; all networks are of type "scalar product of two periodic MPS" but differ in length and dimensions. Reference contraction orders for all networks are obtained with cotengra; we note that the options referred to as *"cotengra cheap"* in the following are described as "cheap to run [...] but will still yield much better results than simple algorithms" in the documentation[5]; therefore, we decided to use this as our main baseline. Consequently, the feature dimensions that guide evolution are designed as follows: Given a tensor network x, denote by $o_A(x)$ the contraction order for x obtained by algorithm A and by $o_{\text{ref}}(x)$ the contraction order by the baseline "cotengra cheap". Let $\text{FLOPs}(o_A(x),x)$ and $\text{FLOPs}(o_{\text{ref}}(x),x)$ denote the corresponding number of FLOPs. Then we define the log10_speedup as

$$\text{log10_speedup}(x,o_A) = \log_{10}\text{FLOPs}(o_{\text{ref}}(x),x) - \log_{10}\text{FLOPs}(o_A(x),x), \tag{1}$$

and correspondingly the average ("avg_log10_speedup"), median ("median_log10_speedup"), maximum ("max_log10_speedup"), minimum ("min_log10_speedup") over all tensor networks x contained in the respective test set $\mathscr{T}$. Similarly, we define

$$\begin{aligned}\text{log10_total_flops_reduction} := \log_{10} \sum_{x\in\mathscr{T}} \text{FLOPs}\big(o_{\text{ref}}(x),x\big) \\ - \log_{10} \sum_{x\in\mathscr{T}} \text{FLOPs}\big(o_A(x),x\big).\end{aligned} \tag{2}$$

Metrics defined upon (1) aggregate the individual, i.e., per-network, improvement in the "order of magnitude" of FLOPs over the entire test set, whereas (2) measures the cumulative improvement in the order of magnitude of FLOPs required for the entire set of TNs. To obtain the "combined_score" metric that actually guides evolution, we define

$$\text{combined_score} := \begin{cases} 2^f & \text{if the evaluation is successful on the entire} \\ & \text{test set } \mathscr{T} \text{ within the timeout,} \\ 0.0 & \text{otherwise.} \end{cases} \tag{3}$$

where f is the respective feature, i.e., always avg_log10_speedup except for Experiment 3. The choice of (3) is heuristic and almost purely ad-hoc and goes back to a few trial-and-error experiments with a small number of iterations only in which we got the impression that this choice is likely advantageous compared to using the metrics directly. In OpenEvolve, each evaluation is assigned a timeout (here: 600 seconds); this means that programs which are not successfully evaluated within this time—e.g., because they freeze due to an error or simply take too long—are terminated and treated as faulty, i.e., assigned combined_score=0.0; cf. (3). This mechanism does not only make the overall process more robust against faulty programmes, but also, in particular, rules out theoretically correct but impractically expensive candidates—in our case, e.g., finding a FLOP-optimal contraction sequence by brute-forcing all possible permutations.

[5] https://cotengra.readthedocs.io/en/latest/basics.html#economical-optimizer [Accessed 05/2026]

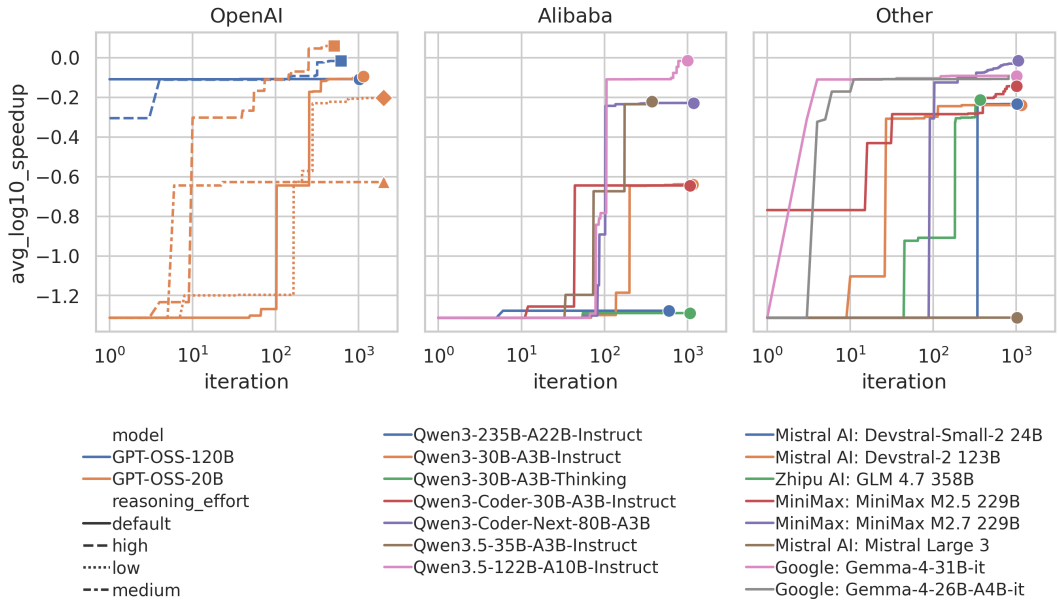


Figure 2: Experiment 1 — Evolution of the average reduction of $\log_{10}$ FLOPs for various LLMs on the "full small" TNs set.

Influence of different LLMs (Experiment 1)

In this first experiment we compare the capabilities of different open-weight LLMs on behalf of the set of small tensor networks and evolution w.r.t. avg_log10_speedup. It can be seen that the models of OpenAI performed best, being the only ones that reach an improvment over the baseline, i.e., avg_log10_speedup > 0 with GPT-OSS-20B with "high" reasoning effort. Surprisingly the smaller 20B-model was able to outperform the larger 120B-model, and even much larger competing models. Among the remaining models, only Qwen3.5-122B-A10B, MiniMax M2.5/M2.7, and the Gemma-4 models, i.e., considerably newer and/or larger models, were able to reach comparable final scores. Some models, e.g., Qwen3 235B or Mistral Large 3, were not able to generate a (correct) program that improved significantly beyond the initial program (avg_log10_speedup≈ -1.31). As GPT-OSS-20B performed best in this initial experiment and is extremely cheap to host, we perform the following more extensive experiments with this model only. Without wanting to judge the performance of individual models specifically, we can summarise this experiment as follows: In this single-run experiment, the choice of the model had a large influence on the quality of the result.

(Non-)Reproducibility and Stochasticity (Experiment 2)

Reproducibility is an obvious issue when dealing with LLMs as their text generation is inherently stochastic. To assess whether the outcome of the evolutionary procedure is at least reproducible in a statistical sense, we run the same 3 experiments ("small" TNs and GPT-OSS-20B with "low", "medium" and "high" reasoning effort) 20 times each. The distributions of best achieved

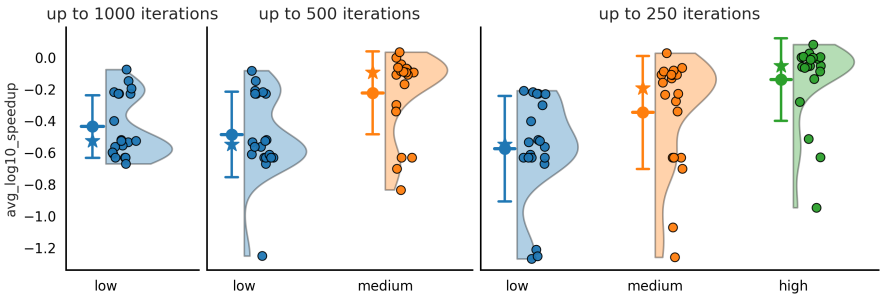


Figure 3: Experiment 2 — Distribution of average reduction of $\log_{10}$ FLOPs on the "small" TNs over 20 runs with different random seeds after 1000, 500, and 250 iterations, respectively. The LLM is GPT-OSS-20B with different levels of reasoning effort ("low" to "high"). The bar left to the violin plot indicates mean (circle), standard deviation, and median (star).

	metric measured				
metric optimized	min	avg	median	max	total
min	-0.17	+0.05	+0.05	+0.36	+0.07
avg	-0.44	+0.06	+0.06	+0.42	+0.07
median	-0.76	-0.01	-0.01	+0.27	-0.01
max	-1.06	-0.16	-0.12	+0.40	-0.24
total	-0.47	+0.00	+0.01	+0.32	+0.02

Table 1: Experiment 3 — Final measured metrics ("min", "max", "avg", "median" as introduced below (1) and "total" as defined in (2)); larger is better.

metrics in Figure 3 seem to become tighter for an increasing number of iterations, and a trend of higher scores for higher reasoning effort seems to be on the horizon. Notably, the improvement over the baseline observed in Experiment 1 could be reproduced in some cases. We are tempted to suggest the following interpretation of our observations: Individual evolution runs tend to be highly stochastic, while certain commonalities seem to emerge across an ensemble of different runs.

Influence of different Metrics (Experiment 3) and Test Cases (Experiment 4)

In order to assess the influence of the metric chosen to define the combined score, we run experiments on the set of "small" TNs for each possible choice while tracking all available metrics. The results of Experiment 3 are summarized in Table 1. As key observation we may regard that even when evaluated on a fixed set of test examples and without measurement errors, metrics can differ in terms of how well they can be optimized by OpenEvolve, and to which extent their optimization simultaneously results in an improvement of other, also potentially useful, metrics. This should in particularly be seen in the light of the fact that a number of rather strong conceptual assumptions have already been made before with regard to the definition of the metrics, e.g., the restriction to pure FLOP counts. In Experiment 4, we fix the metric to be optimized again to "avg_log10_speedup" and investigate the influence of the set of test examples by optimizing

	optimization on		
evaluation on	small	middle	large
small	+0.05 ($\pm$0.09)	-0.02 ($\pm$0.09)	-0.24 ($\pm$0.13)
middle	+1.05 ($\pm$1.21)	+1.61 ($\pm$1.07)	+0.20 ($\pm$0.89)
large	$\star$	$\star$	-0.23 ($\pm$0.05)

Table 2: Experiment 4 — Final average reduction of $\log_{10}$ FLOPs (larger is better, $\pm$ indicates standard deviations); "$\star$" indicates that the evaluation did not terminate in time.

over "small", "middle" and "large" TNs, respectively, while evaluating also on the other sets. As this setup is similar to the split of training and test/validation data in deep learning [GBC16] one may expect typical in-distribution vs out-of-distribution phenomena [YZLL24]; our experimental results summarized in Table 2 appear to roughly support this hypothesis. In any way, the selection of test examples for the evaluation of the LLM-generated programs seems to strongly influence the outcome of evolution. Altogether, Experiments 3 and 4 suggest that the following two questions are crucial for success of the evolutionary algorithm: How to define and measure "improvement"?—in other words: Which metric to optimize and how to evaluate this metric, i.e., on behalf of which examples etc.?

Detailed Examination of the "best" Outcome

Finally, we examine the best outcome of Experiment 1 (by GPT-OSS-20B, reasoning level "high") in more detail. In order to further assess the quality of the final code, we compare it to stronger baselines and also take the run time of the contraction order determination itself into account, both on "small", "middle" and "large" TNs. The fact that already the run time of the OpenEvolve-generated algorithm (not of the actual contraction) is only feasible for "small" and "middle" TNs (cf. also Experiment 4) underpins the relevance of this consideration for practical purposes.

Figure 4 shows that our final program clearly outperforms the baseline "cotengra cheap" on "middle" TNs (average reduction of FLOPs by a factor of $\sim 10^{1.67} \approx 47$) while the effect is smaller, but still positive on "small" TNs (average reduction of FLOPs by a factor of $\sim 10^{0.06} \approx 1.15$). For "large" TNs, however, we could not obtain results as the run time of the algorithm in the final program was too high. This highlights another important fact: the combination of test examples (in our case: "small" TNs) and the timeout specified for the evaluation also formulates an implicit constraint on the run time of the program to be optimized. Obviously, choosing "large" TNs as test examples would have resulted in a different final program as the present final program would not have been evaluated successfully in that setting. In order to take into account the much higher computing time on "large" TNs and more expensive baselines, we only perform the corresponding analysis on 100 TNs of the respective TN quantities; see the second row in Figure 4. Compared to the stronger baselines "cotengra+cmaes" and "cotengra+optuna", there is still a clear improvement on "middle" TNs (average reduction of FLOPs by factors of ≈ 6.6 and ≈ 5.6), while we cannot clearly interpret the results on "small" and "large" TNs. In terms of run times (Figure 4, bottom), the program generated by OpenEvolve is slightly faster than contraction order determination by the baseline "cotengra cheap", while on "large" TNs the

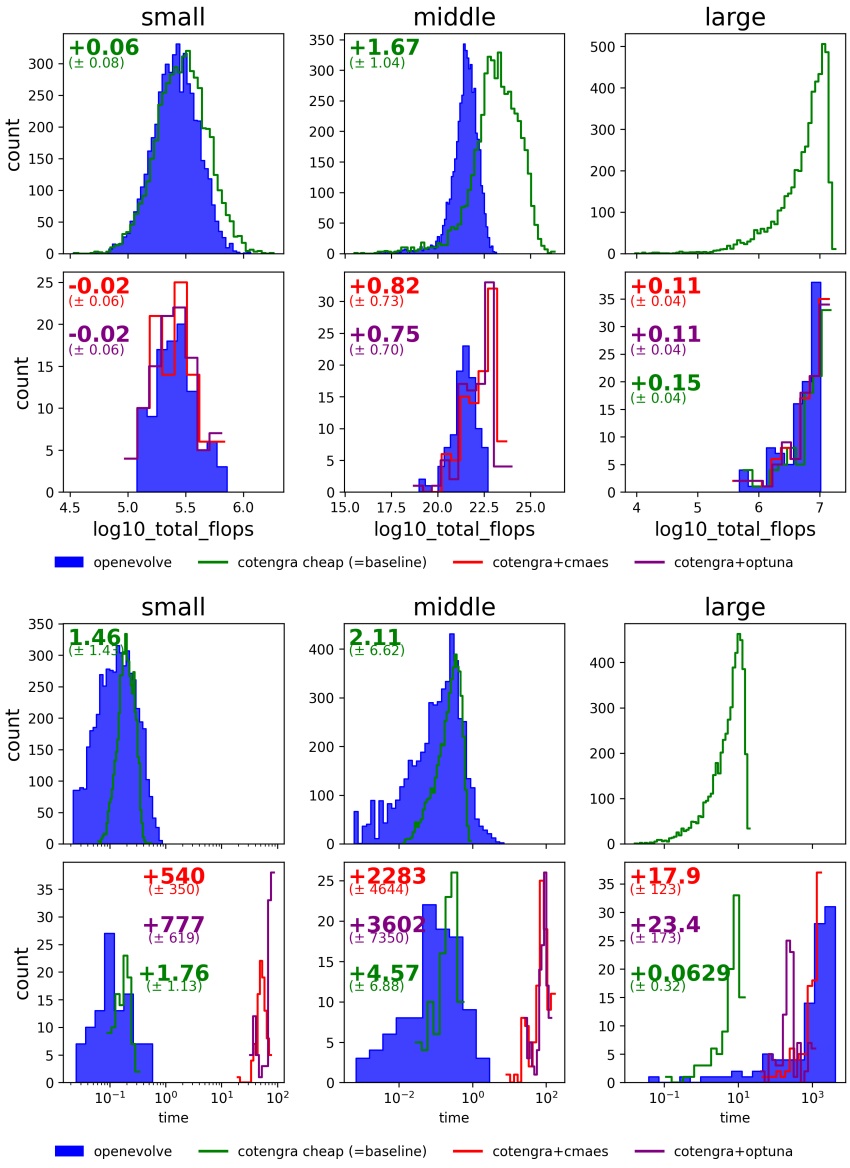


Figure 4: Experiment 1 — Distribution of $\log_{10}$ FLOPs (top) and distribution of runtimes for computing the contraction order (bottom) for the final code of the best run (blue) vs different baselines ("cotengra cheap", "cotengra+cmaes", and "cotengra+optuna" in green, red, and purple, respectively) on different TNs sets (columns). First and third rows use the full TNs sets, second and fourth row use only the last 100 of the TNs from the respective sets. The colored numbers give the average improvement of $\log_{10}$ FLOPs (higher is better, > 0 means improvement) or the average reduction factor of runtime (higher is better, > 1 means improvement) vs the respective baseline on the respective set of TNs; "$\pm$" indicates standard deviations.

situation is completely opposite and our program exhibits run times exceeding reasonable limits for experimentation. On the subsets of 100 TNs, respectively, it turns out that the generated algorithm is clearly faster than the stronger baselines. However, our experiments do not take into account the potential of parallelization for these search-based baselines that might heavily reduce the observed gap. On "large" TNs, finally, the adverse run time behaviour of the algorithm becomes apparent when it is clearly outperformed (in terms of run time) by the baselines while having a comparable quality of the outcome (in terms of FLOPs of the resulting paths, cf. Figure 4 (top)). Finally, we recall that the actual speed-up in real-world applications will not depend solely on the number of FLOPs. Moreover, both the resource utilization of the contraction itself and the determination of the contraction order must be taken into account to determine a true benefit for a particular use-case. Both aspects go beyond the scope of this paper. Nevertheless, our results indicate in our view that careful (experimental) validation is just as important as the well-considered design of the harness for evolution.

In contrast to, e.g., the training of a neural network, the result of evolution in OpenEvolve is typically not a black-box, but a human-readable code, so it can be subject to a theoretical validation (in addition to an experimental, practical validation) as well. In the following, we therefore examine the overall best code in detail. Before doing so, let us briefly mention in this context that autonomously AI-generated code—and, as we believe, algorithms, too—can be difficult to understand as it might exhibit properties known for making code comprehension challenging: It is not self-written and the context might not be fully known; thus, building a mental model is more difficult and understanding might have to happen bottom-up instead of top-down; cf., e.g., [Fei22, SS15, AAGa24, GRSa25, NRT]. In the context of research software, this is a particularly serious issue, as comprehensibility/understandability should not be a "nice to have" but an important quality requirement; cf. the emphasis on understandability, documentation, reuse, evolvability, and adaptability as demands for sustainable research software in [FGGa25]. Nevertheless, it should also be noted that these difficulties naturally go hand in hand with the greatest (at least hoped-for) advantage of AI-based algorithm engineering: the users are hoping for ideas or approaches, e.g., from other specialist areas, that are not yet known to them or the discovery of genuinely new procedures. For an analysis of code complexity of the codes generated in Experiments 1 and 2, we refer the interested reader to Appendix A.5.

OpenEvolve did not change the docstring of `find_edge_path` but turned the original 4 lines of code (including 1 line of comments, doc string excluded) into 164 lines of code (including 21 lines of comments, doc string excluded). Thus, the comments are rather thin on the ground (cf. also Appendix A.5), and, in addition, they do not contain any references to the literature at all. Consequently, an assessment of novelty of the algorithmic core had to be done from scratch.

It turns out that the overall skeleton of `find_edge_path` is well known. In particular, tensor-network contraction planning can be formulated as a *graph elimination* (variable-elimination) problem: eliminating a vertex induces *fill-in* edges among its neighbours, and the resulting induced graph controls intermediate tensor growth and cost. In the tensor-network setting, optimal contraction complexity has been related to the treewidth of the network's *line graph*, a viewpoint that can be expressed directly in terms of elimination orderings and fill-in [MS08, DFGa18]. The routine follows this paradigm by building an *index interaction graph* whose vertices are index labels and where two indices are adjacent if and only if they co-occur in a tensor (equivalently, each tensor induces a clique on its incident indices); this is essentially the line-graph/interaction-

graph representation well-known from treewidth-based formulations [MS08, DFGa18]. It then computes an elimination ordering using a greedy *min-fill*-style rule and explicit fill-in updates [RTL76], followed by a lightweight local refinement that repeatedly swaps adjacent positions in the ordering (a simple local-search strategy for improving triangulations/orderings) [Kjæ90]. In contrast, many modern tensor-network planners, e.g. [GK21], optimize in the space of contraction trees/hypergraph contractions using randomized and local reconfiguration strategies rather than operating purely on an elimination ordering.

The nonstandard aspect lies in the *particular surrogate weighting* of prospective fill edges: these are penalized by $\min(\log\dim a, \log\dim b)$ (with $\dim(\cdot)$ the *index* dimensions), whereas standard *weighted* min-fill variants typically weight fill edges by multiplicative measures of variable cardinalities (equivalently, additive in log-space) [Kjæ90, Dec99]. We were not able to find prior tensor-network contraction heuristics that employ this "minimum-of-logs" fill weighting specifically, and, additionally, we are not aware of a theoretical justification of this. Consequently, if the results shown in Figure 4 would not have indicated that the algorithm indeed yields reasonable results, one could regard this choice as an error.

4 Conclusion

During this case study we were not able to discover a significantly new algorithm. Instead, we obtained known approaches with minor modifications. However, the clear limitations of this work should be taken into account: we performed our experiments with a single framework for LLM-based algorithm evolution only—and, in fact, one that is not a very autonomous agentic system and that exclusively relies on the internal knowledge of the LLM. In addition, we restricted ourselves to a single, very specific and rather easy to handle, though interesting, example task, too. Moreover, we did not make use of commercial, closed-source frontier LLMs (which could potentially have delivered better results), and the number of experiments was limited to a feasible[6] amount of compute budget. Finally, both LLM capabilities and surrounding frameworks are evolving extremely fast; so one might consider the setup of the shown experiments as outdated quite soon.

Nevertheless, we believe that some higher-level lessons can be learned. We regard OpenEvolve and similar or more advanced systems as an exciting, highly promising approach to scaling algorithm search and optimization through (more or less) brute-force trial-and-verify-guided search. A major advantage—and at the same time a major challenge in subsequently understanding the generated code—is the enormous cross-domain knowledge base of LLMs. The design of the space within which the LLM is given freedom of design, so to speak, remains crucial: the human scientist must condense the overarching algorithm optimization goal into a suitable evaluation function; even in a comparatively simple case such as the example discussed here, there are numerous design decisions that can have a significant influence on the final result, as we have exemplarily seen in Experiments 3 and 4. It should be noted that building such an evaluation function might even be more complex for different problem types where evaluation might require compilation and/or actual run time- and/or memory measurements, possibly on certain hardware configurations. In addition, experimental validation of the outcome of the evolution

[6] "Feasible" refers particularly to a *fair* usage of the *shared* LLM-providers Blablador and ChatAI.

process, beyond those characteristics covered by the bespoke evaluation function, i.e. testing for generalization, as well as interpretation and theoretical understanding are still necessary and can be non-trivial. This might become even more important in the context of advanced agentic frameworks with a less rigid structure than OpenEvolve which could be susceptible to reward-hacking-type phenomena; cf., e.g., [WQBa26]. We believe that at least these two tasks, designing the harness as well as carrying out final validation and interpretation, both taking into account the overarching goal(s) and motivation, remain in the human scientists/(research) software engineers responsibility for the foreseeable future.

With regard to the consumption of resources, it should be noted that in addition to the not inconsiderable (energy and financial) costs of LLM use [CCC25], the costs for the evaluation must also be taken into account: here there is a tension between the most comprehensive and good evaluation possible and the higher duration or other resource utilization of such an evaluation that may be necessary for this. Another aspect that deserves attention is the choice of the underlying LLM: at least in our experiments, this strongly influenced the result. This opens up a further area of tension, in that LLMs with stronger capabilities are often, though not necessarily, larger and more expensive to operate, or even only available commercially and operated abroad, which ultimately also potentially touches on the extremely sensitive issues of technological sovereignty, export control and data security. Analogous concerns regarding the threat of a "digital divide" between "AI-have" and "AI-have-nots" in the mathematical sciences were also expressed in [KT26].

To conclude the paper, let us try to put our previous comments in a slightly broader context. Trying to view our setup through the conceptual lens of [FLFa26], the final code for tensor network contraction is only the innermost executable artifact; system message, OpenEvolve workflow, evaluation metric, the set of test examples, timeout configuration, as well as post-hoc validation and interpretation can be seen as elements of the "control systems" ring, as "semi-executable" specifications of the scientific objective(s). In this sense, our experiments with a subset of these specifications can be understood as illustration of the claim made in [FLFa26] for agentic software engineering in general, now of course rephrased to our scientific algorithm-/performance-engineering setting: prompts, agentic workflows/frameworks, evaluation-/scoring-related artifacts and routines, as well as the setup, the results, and their interpretation, of post-hoc validation (experimental or theoretical) become first-class engineering objects.

With regard to the concepts formulated in [BCNa26], we are tempted to categorise our results as follows: the set screws examined in numerous experiments illustrate the central importance of "intentmaking", but also the associated challenges. In our case, we would understand "intentmaking" as the process of condensing an initially open, not precisely formulated algorithm improvement task into an input suitable for OpenEvolve, which definitely requires a set of non-trivial design decisions. We also tackled the step "sensemaking", at least partially, in the form of a more detailed post-hoc analysis of the best generated candidate. In order to achieve a result that is actually relevant for domain science or application, the loop would have to be closed and further steps of "intentmaking" (e.g. by selecting new test cases, including the runtime of the ordering algorithm in the score function, choosing a performance metric beyond pure FLOP count, etc.) and "sensemaking" would have to follow[7]. We also agree with [BCNa26] that working

[7] In fact, a tiny and hidden approach to this has already happened when we chose 2^f in the definition of com-

with OpenEvolve on the problem at hand is better understood as experimenting with, and simultaneously designing, a complex scientific instrument than as using a simple question-answer oracle.

Acknowledgements: The usage of Blablador API, provided by Helmholtz AI Jülich via HIFIS, and SAIA/ChatAI API, provided by GWDG Göttingen via Academic Cloud, is gratefully acknowledged. Moreover, the authors gratefully acknowledge the scientific support and HPC resources provided by the German Aerospace Center (DLR). The HPC system CARO is partially funded by "Ministry of Science and Culture of Lower Saxony" and "Federal Ministry for Economic Affairs and Climate Action". This project was made possible by the DLR Quantum Computing Initiative and the Federal Ministry for Economic Affairs and Climate Action; https://qci.dlr.de/qutenet/. Finally, the authors thank the anonymous reviewers for their feedback that helped to improve the quality of the paper.

Data availability: Scripts used for the generation of tensor networks, an example set up configuration for the OpenEvolve experiments, and the final codes of Experiment 1 are available on Zenodo: https://doi.org/10.5281/zenodo.22108774

Declaration on the use of AI: Besides its obvious usage in the experiments, generative AI has been used for coding, formatting, drafting, and assistance during literature research. AI-generated content created in the course of this was checked, corrected if necessary, and revised by the authors.

Bibliography

[AACa25] B. Atıl, S. Aykent, A. Chittams, et al. Non-Determinism of "Deterministic" LLM System Settings in Hosted Environments. In Akter et al. (eds.), *Proceedings of the 5th Workshop on Evaluation and Comparison of NLP Systems*. Pp. 135–148. Association for Computational Linguistics, Mumbai, India, Dec. 2025. doi:10.18653/v1/2025.eval4nlp-1.12

[AAGa24] E. Agrawal, O. Alam, C. Goenka, et al. Code Compass: A Study on the Challenges of Navigating Unfamiliar Codebases. *CoRR* abs/2405.06271, 2024. doi:10.48550/ARXIV.2405.06271

[AFCa25] H. Assumpção, D. Ferreira, L. Campos, et al. CodeEvolve: an open source evolutionary coding agent for algorithmic discovery and optimization. 2025. doi:10.48550/arXiv.2510.14150

bined_score based on a few experiments as indicated at the beginning of Sect. 3

[AKSa24] V. Aglietti, I. Ktena, J. Schrouff, et al. FunBO: Discovering Acquisition Functions for Bayesian Optimization with FunSearch. 2024.
https://arxiv.org/abs/2406.04824

[ATSa26] L. A. Agrawal, S. Tan, D. Soylu, et al. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. 2026.
https://arxiv.org/abs/2507.19457

[BB17] J. Biamonte, V. Bergholm. Tensor Networks in a Nutshell. 2017.
https://arxiv.org/abs/1708.00006

[BCNa26] A. Bäuerle, A. Connors, A. Novikov, et al. Intentmaking and Sensemaking: Human Interaction with AI-Guided Mathematical Discovery. 2026.
https://arxiv.org/abs/2605.05921

[BHJa25] D. Brown, J. He, H. Jenne, et al. Even with AI, Bijection Discovery is Still Hard: The Opportunities and Challenges of OpenEvolve for Novel Bijection Construction. 2025.
doi:10.48550/arXiv.2511.20987

[BK25] G. Ballard, T. G. Kolda. *Tensor Decompositions for Data Science*. Cambridge University Press, June 2025.
doi:10.1017/9781009471664

[BKB25] J. Beel, M.-Y. Kan, M. Baumgart. Evaluating Sakana's AI Scientist: Bold Claims, Mixed Results, and a Promising Future? *SIGIR Forum* 59(1):1–20, Oct. 2025.
doi:10.1145/3769733.3769747

[BNL26] J. Bhan, N. Nobili, P. Langer. New Bounds for Zarankiewicz Numbers via Reinforced LLM Evolutionary Search. 2026.
https://arxiv.org/abs/2605.01120

[CCC25] F. Caravaca, Ángel Cuevas, R. Cuevas. From Prompts to Power: Measuring the Energy Footprint of LLM Inference. 2025.
https://arxiv.org/abs/2511.05597

[CLPa25] A. Cheng, S. Liu, M. Pan, et al. Let the Barbarians In: How AI Can Accelerate Systems Performance Research. 2025.
doi:10.48550/arXiv.2512.14806

[CZH26] A. Cheng, L. Zhang, G. He. Re4: Scientific Computing Agent with Rewriting, Resolution, Review and Revision. 2026.
https://arxiv.org/abs/2508.20729

[Dec99] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence* 113(1–2):41–85, 1999.
doi:10.1016/S0004-3702(99)00059-4

[DFGa18] E. F. Dumitrescu, A. L. Fisher, T. D. Goodrich, et al. Benchmarking treewidth as a practical component of tensor network simulations. *PLOS ONE* 13(12), 2018. doi:10.1371/journal.pone.0207827

[FBMa24] C. Fernando, D. Banarse, H. Michalewski, et al. Promptbreeder: self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*. ICML'24. JMLR.org, 2024. https://dl.acm.org/doi/10.5555/3692070.3692611

[Fei22] D. G. Feitelson. Considerations and Pitfalls for Reducing Threats to the Validity of Controlled Experiments on Code Comprehension. *Empirical Software Engineering* 27(6):123, Jun 2022. doi:10.1007/s10664-022-10160-3

[FGGa25] M. Felderer, M. Goedicke, L. Grunske, et al. Investigating Research Software Engineering: Toward RSE Research. *Commun. ACM* 68(2):20–23, Jan. 2025. doi:10.1145/3685265

[FKSa26] D. Fisher, V. Khrulkov, M. Saygin, et al. LLM-Guided Evolutionary Search for Algebraic T-Count Optimization. 2026. https://arxiv.org/abs/2603.29894

[FLFa26] R. Feldt, P. Lenberg, J. Frattini, et al. The Semi-Executable Stack: Agentic Software Engineering and the Expanding Scope of SE. 2026. https://arxiv.org/abs/2604.15468

[FLXa26] R. Fu, Y. Liu, Q. Xu, et al. MappingEvolve: LLM-Driven Code Evolution for Technology Mapping. 2026. https://arxiv.org/abs/2604.26591

[GBC16] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GD04] V. Gogate, R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*. UAI '04, p. 201–208. AUAI Press, Arlington, Virginia, USA, 2004. https://dl.acm.org/doi/10.5555/1036843.1036868

[GGTa25] B. Georgiev, J. Gómez-Serrano, T. Tao, et al. Mathematical exploration and discovery at scale. 2025. https://arxiv.org/abs/2511.02864

[GK21] J. Gray, S. Kourtis. Hyper-optimized tensor network contraction. *Quantum* 5:410, 2021. doi:10.22331/q-2021-03-15-410

[GRSa25] P. W. Goncalves, P. Rani, M.-A. Storey, et al. Code Review Comprehension: Reviewing Strategies Seen Through Code Comprehension Theories . In *2025*

IEEE/ACM 33rd International Conference on Program Comprehension (ICPC). Pp. 589–601. IEEE Computer Society, Los Alamitos, CA, USA, Apr. 2025. doi:10.1109/ICPC66645.2025.00068

[GWDa25] J. Gottweis, W.-H. Weng, A. Daryin, et al. Towards an AI co-scientist. 2025. https://arxiv.org/abs/2502.18864

[IHIa] Y. Imajuku, K. Horie, Y. Iwata, et al. ALE-Bench: A Benchmark for Long-Horizon Objective-Driven Algorithm Engineering. NeurIPS 2025. https://arxiv.org/abs/2506.09050

[ILHa22] C. Ibrahim, D. Lykov, Z. He, et al. Constructing Optimal Contraction Trees for Tensor Network Quantum Circuit Simulation. 2022. https://arxiv.org/abs/2209.02895

[IZLa25] G. Iacovides, W. Zhou, C. Li, et al. Domain-Aware Tensor Network Structure Search. 2025. https://arxiv.org/abs/2505.23537

[JWSa26] J. Jiang, F. Wang, J. Shen, et al. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* 35(2), Jan. 2026. doi:10.1145/3747588

[KGBa25] V. Khrulkov, A. Galichin, D. Bashkirov, et al. GigaEvo: An Open Source Optimization Framework Powered By LLMs And Evolution Algorithms. 2025. https://arxiv.org/abs/2511.17592

[Kjæ90] U. B. Kjærulff. Triangulation of Graphs – Algorithms Giving Small Total State Space. Technical report R 90-09, Aalborg University, 1990. https://cse.unl.edu/~choueiry/Documents/Kjaerulff-TR-1990.pdf

[KSNa26] U. Kumar, A. Saito, H. Niranjani, et al. Evolving Interpretable Constitutions for Multi-Agent Coordination. 2026. https://arxiv.org/abs/2602.00755

[KT26] T. Klowden, T. Tao. Mathematical methods and human thought in the age of AI. 2026. https://arxiv.org/abs/2603.26524

[LGWa26] Z. Liu, X. Guo, X. Wei, et al. Escher-Loop: Mutual Evolution by Closed-Loop Self-Referential Optimization. 2026. https://arxiv.org/abs/2604.23472

[LIC25] R. T. Lange, Y. Imajuku, E. Cetin. ShinkaEvolve: Towards Open-Ended And Sample-Efficient Program Evolution. 2025. https://arxiv.org/abs/2509.19349

[LLLa26] C. Lu, C. Lu, R. T. Lange, et al. Towards end-to-end automation of AI research. *Nature* 651(8107):914–919, Mar 2026. See also https://arxiv.org/abs/2408.06292. doi:10.1038/s41586-026-10265-5

[LMSa26] K.-A. Lie, O. Møyner, E. Svee, et al. Agentic Scientific Simulation: Execution-Grounded Model Construction and Reconstruction. 2026. https://arxiv.org/abs/2603.00214

[LTYa24] F. Liu, X. Tong, M. Yuan, et al. Evolution of heuristics: towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*. ICML'24. JMLR.org, 2024. https://dl.acm.org/doi/abs/10.5555/3692070.3693374

[LYFa26] H. Lin, H. Ye, W. Feng, et al. Can Language Models Discover Scaling Laws? 2026. https://arxiv.org/abs/2507.21184

[LZ] X.-Y. Liu, Z. Zhang. Classical Simulation of Quantum Circuits Using Reinforcement Learning: Parallel Environments and Benchmark. In *NeurIPS 2023*. https://proceedings.neurips.cc/paper_files/paper/2023/file/d41b70011dd21ec3de5e019302279551-Paper-Datasets_and_Benchmarks.pdf

[LZCa25] G. Liu, Y. Zhu, J. Chen, et al. Scientific Algorithm Discovery by Augmenting AlphaEvolve with Deep Research. 2025. https://arxiv.org/abs/2510.06056

[LZX+24] F. Liu, R. Zhang, Z. Xie, R. Sun, K. Li, X. Lin, Z. Wang, Z. Lu, Q. Zhang. LLM4AD: A Platform for Algorithm Design with Large Language Model. 2024. https://arxiv.org/abs/2412.17287

[MMMa] E. A. Meirom, H. Maron, S. Mannor, et al. Optimizing Tensor Network Contraction Using Reinforcement Learning. In *Proceedings of the 39th International Conference on Machine Learning (ICML 2022)*. https://proceedings.mlr.press/v162/meirom22a.html

[MS08] I. L. Markov, Y. Shi. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM Journal on Computing* 38(3):963–981, 2008. doi:10.1137/050644756

[MYCa25] L. Mitchener, A. Yiu, B. Chang, et al. Kosmos: An AI Scientist for Autonomous Discovery. 2025. https://arxiv.org/abs/2511.02824

[MZKa26] V. A. Mazin, M. A. Zorin, D. S. Korzh, et al. LLM-Guided Prompt Evolution for Password Guessing. 2026. https://arxiv.org/abs/2604.12601

[NGWI25] K. Nagaitsev, L. Grbcic, S. Williams, C. Iancu. Optimizing PyTorch Inference with LLM-Based Multi-Agent Systems. 2025. https://arxiv.org/abs/2511.16964

[NRT] D. Neumüller, A. Raschke, M. Tichy. Providing Information About Implemented Algorithms Improves Program Comprehension: A Controlled Experiment. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. EASE '25.
doi:10.1145/3756681.3756968

[NVEa25] A. Novikov, N. Vũ, M. Eisenberger, et al. AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery. 2025.
https://arxiv.org/abs/2506.13131

[O'G] B. O'Gorman. Parameterization of Tensor Network Contraction. In *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*.
doi:10.4230/LIPIcs.TQC.2019.10

[Orú14] R. Orús. A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States. *Annals of Physics* 349:117–158, 2014.
doi:10.1016/j.aop.2014.06.013

[PA+25] O. Press, B. Amos et al. AlgoTune: Can Language Models Speed Up General-Purpose Numerical Programs? 2025.
https://arxiv.org/abs/2507.15887

[RBNa24] B. Romera-Paredes, M. Barekatain, A. Novikov, et al. Mathematical discoveries from program search with large language models. *Nature* 625:468–475, 2024.
doi:10.1038/s41586-023-06924-6

[RBSa25] P. Rajput, A. A. Bonkoungou, Y. Song, et al. Dynamic Stability of LLM-Generated Code. 2025.
https://arxiv.org/abs/2511.07463

[RTL76] D. J. Rose, R. E. Tarjan, G. S. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM Journal on Computing* 5(2):266–283, 1976.
doi:10.1137/0205021

[SBK+] C. Staudt, M. Blacher, J. Klaus, F. Lippmann, J. Giesen. Improved Cut Strategy for Tensor Network Contraction Orders. In *22nd International Symposium on Experimental Algorithms (SEA 2024)*.
doi:10.4230/LIPIcs.SEA.2024.27

[SG18] D. G. A. Smith, J. Gray. opt_einsum - A Python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software* 3(26):753, 2018.
doi:10.21105/joss.00753

[SHGa23] S. Schlag, T. Heuer, L. Gottesbüren, et al. High-Quality Hypergraph Partitioning. *ACM J. Exp. Algorithmics* 27, Feb. 2023.
doi:10.1145/3529090

[SISa25] M. L. Siddiq, A. Islam-Gomes, N. Sekerak, et al. Large Language Models for Software Engineering: A Reproducibility Crisis. 2025.
https://arxiv.org/abs/2512.00651

[SJ20] F. Schindler, A. S. Jermyn. Algorithms for tensor network contraction ordering. *Machine Learning: Science and Technology* 1(3):035001, 2020.
doi:10.1088/2632-2153/ab94c5

[SMM24] M. Stoian, R. M. Milbradt, C. B. Mendl. On the Optimal Linear Contraction Order of Tree Tensor Networks, and Beyond. *SIAM Journal on Scientific Computing* 46(5):B647–B668, 2024.
doi:10.1137/23M161286X

[ŠMQa25] A. Šurina, A. Mansouri, L. C. P. M. Quaedvlieg, et al. Algorithm Discovery With LLMs: Evolutionary Search Meets Reinforcement Learning. 2025.
https://arxiv.org/abs/2504.05108

[SS15] J. Siegmund, J. Schumann. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering* 20(4):1159–1192, Aug 2015.
doi:10.1007/s10664-014-9318-8

[Str17] B. Strasser. Computing Tree Decompositions with FlowCutter: PACE 2017 Submission. 2017.
https://arxiv.org/abs/1709.08949

[Tam] H. Tamaki. Positive-instance driven dynamic programming for treewidth. In *25th Annual European Symposium on Algorithms (ESA 2017)*.
doi:10.4230/LIPIcs.ESA.2017.68

[TDPa26] A. Torri, P. Dominikowski, B. Pointal, et al. Near-Optimal Contraction Strategies for the Scalar Product in the Tensor-Train Format. In Nagel et al. (eds.), *Euro-Par 2025: Parallel Processing*. Pp. 63–77. Springer Nature Switzerland, Cham, 2026.
doi:10.1007/978-3-031-99872-0_5

[TGZa24] M. Tian, L. Gao, S. D. Zhang, et al. SciCode: A Research Coding Benchmark Curated by Scientists. 2024.
https://arxiv.org/abs/2407.13168

[TRHC25] N. Thach, A. Riahifar, N. Huynh, H. Chan. RedAHD: Reduction-Based End-to-End Automatic Heuristic Design with Large Language Models. 2025.
https://arxiv.org/abs/2505.20242

[VPC04] F. Verstraete, D. Porras, J. I. Cirac. Density Matrix Renormalization Group and Periodic Boundary Conditions: A Quantum Information Perspective. *Phys. Rev. Lett.* 93:227205, Nov 2004.
doi:10.1103/PhysRevLett.93.227205

[WQBa26] J. Wen, L. Qiu, J. Benton, et al. Automated Weak-to-Strong Researcher. 2026. Anthropic Alignment Science Blog.
https://alignment.anthropic.com/2026/automated-w2s-researcher/

[WSZa25] Y. Wang, S.-R. Su, Z. Zeng, et al. ThetaEvolve: Test-time Learning on Open Problems. 2025.
https://arxiv.org/abs/2511.23473

[XZLa23] J. Xu, H. Zhang, L. Liang, et al. NP-Hardness of Tensor Network Contraction Ordering. 2023.
https://arxiv.org/abs/2310.06140

[YLDa25] J. Yuan, H. Li, X. Ding, et al. Understanding and Mitigating Numerical Sources of Nondeterminism in LLM Inference. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. 2025.
https://arxiv.org/abs/2506.09501

[YLLa25] Y. Yamada, R. T. Lange, C. Lu, et al. The AI Scientist-v2: Workshop-Level Automated Scientific Discovery via Agentic Tree Search. 2025.
https://arxiv.org/abs/2504.08066

[YWCa24] H. Ye, J. Wang, Z. Cao, et al. ReEvo: large language models as hyper-heuristics with reflective evolution. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*. NIPS '24. Curran Associates Inc., Red Hook, NY, USA, 2024.
https://dl.acm.org/doi/10.5555/3737916.3739297

[YZLL24] J. Yang, K. Zhou, Y. Li, Z. Liu. Generalized Out-of-Distribution Detection: A Survey. *International Journal of Computer Vision* 132:5635–5662, 2024.
doi:10.1007/s11263-024-02117-4

[ZLSa24] J. Zeng, C. Li, Z. Sun, et al. tnGPS: Discovering Unknown Tensor Network Structure Search Algorithms via Large Language Models (LLMs). In *Proceedings of the 41st International Conference on Machine Learning*. 2024.
https://proceedings.mlr.press/v235/zeng24b.html

[ZWZ26] B. Zhao, H. Wang, L. Zeng. G-LNS: Generative Large Neighborhood Search for LLM-Based Automatic Heuristic Design. 2026.
https://arxiv.org/abs/2602.08253

A Appendix

A.1 Additional Details on the Setup of the Experiments

Cotengra's function `utils.lattice_equation` is used to generate random tensor networks of type "scalar product of two periodic MPS" with lengths and dimensions within specified bounds to obtain the sets of "small", "middle", and "large" tensor networks, each of them containing 5000 networks; for usage in the different experiments they are saved to `json` files. To reduce computational costs of the evaluation function, we sometimes use smaller sets of TNs in which we only use the first 1000 TNs from the aforementioned sets and refer to them as "reduced" sets, whereas the original ones are referred to as "full" for clarity. For Experiment 4 we also consider a mixed set named "all". A summary of all sets is provided in Table 3. The reference contraction orders are computed by cotengra's HyperOptimizer; see Table 4 for the respective configurations. Hereby, no parallelization is utilized.

The configuration of OpenEvolve for all experiments is shown in Table 5; all parameters not mentioned in this table are set to their default values[8] (e.g., temperature= 0.7 and top_p= 0.95 for the LLM). The system message and the initial program are found as Listings 1 and 2, respectively; the latter implements a highly naive greedy approach in which indices are contracted in the order of their size (descending). An overview over the open-weight LLMs used in the experiments and how they were accessed can be found in Table 6.

Experiment 1 as well as the analysis of the respective outcomes were performed on Linux Workstations under Python 3.10 whereas Experiments 2, 3 and 4 were conducted on nodes of DLRs cluster CARO equipped with a single Nvidia GH200 under Python 3.12.

A.2 Additional Details on Experiment 1

If the evolution trajectories shown in Figure 2 for certain models end before 1000 iterations, this is due to excessive time consumption and/or a high number of restarts from checkpoints[9] due to frequently crashing/timing out evaluations in which case the experiments were aborted manually; the latter issue happened, e.g., for the GLM 4.7 model. While referring to iteration numbers is common and convenient, it should be noted that resource usage per iteration varies considerably between the shown runs: the time for evaluation ranges from a few seconds (for the initial code) to a maximum of 10 minutes in case of a timeout. On the LLM side, the number of output tokens, the hardware required and the time needed for generation depend on the model, the deployment and the system load. The configuration in OpenEvolve, however, at least yields an upper bound of resource utilization per iteration in terms of time and token usage; cf. Table 5.

[8] https://github.com/algorithmicsuperintelligence/openevolve/blob/main/configs/default_config.yaml [accessed 04/2026]

[9] Restarts become necessary if errors/timeouts during evaluation are not catched successfully by the timeout mechanism in OpenEvolve resulting in crashing/freezing of the entire evolution process.

A.3 Additional Details on Experiment 2

OpenEvolve allows setting seeds that are respected both in its evolution algorithm and fed forward into the LLM-API call; it should be noted that this still might not guarantee reproducibility on the LLM-side [AACa25, YLDa25]. We run the same 3 experiments 20 times with different seeds 0-19 (instead of OpenEvolve's default 42). The decision to perform 250, 500, and 1000 iterations, respectively, is due to the fact that the different reasoning efforts "high", "medium", and "low" for GPT-OSS-20B came with a roughly similar ratio in terms of computing time for the whole evolution process. For a different number of iterations, the explanation is otherwise the same as for Experiment 1 (see above). Nevertheless, to keep the computational effort reasonable we had to switch from the full set of small tensors to the reduced one (see Table 3) and run evaluations on 4 cores only. Under these changes, the fixed timeout of 600s for each evaluation places slightly different demands on the programmes being evaluated than in the previous experiment, of course. In addition to the distribution of average reduction of $\log_{10}$ FLOPs shown in Figure 3, we show here also the evolution trajectories leading there in Figure 7.

A.4 Additional Details on Experiments 3 and 4

To reduce the computational effort, all runs in Experiments 3 and 4 utilize the reduced TNs sets and GPT-OSS-20B with "medium" reasoning effort. For each configuration under consideration —different metrics over "small" TNs in Experiment 3, "avg_log10_speedup" over different sets of TNs in Experiment 4—10 runs with different seeds 0-9 have been done utilizing the same hardware configuration as in Experiment 2. However, each run has been assigned a hard *overall* timeout of 12 hours[10] and in case of crashing before *no* restart has been done. The evolution trajectories and the reasons for their termination are shown in Figure 5. To reduce stochastic effects, for the actual evaluations in Experiment 3 and 4 only the run with the best final score has been selected for each configuration.

A.4.1 Additional Details on Experiment 3

For this experiment, the description of "combined_score" in the system message (cf. Listing 1, line 15) has been adapted accordingly for each chosen metric, of course. In addition to Table 1 showing the final values of the metrics, Figure 6 displays the whole evolution process. In most cases, optimization based on a particular metric yields comparatively good results for that metric (first or second place); in addition, optimization against minimum and average seem to perform rather well, whereas optimization against maximum and median show weaker results. While some observations—such as the rather poor performance of optimization over maximum on the minimum, the mean and the median, are expected and understandable, some others are less: e.g., the comparatively solid performance of "log10_total_flops_reduction" can be viewed as surprising, as it is not obvious that this metric generates a reasonable signal for optimization because different contributions of FLOPs from different TNs in the test set might be averaged out. Similarly surprising is the relatively poor performance of optimizing the median compared to optimizing the mean; one could in principle suspect a stochastic effect here, an explanation which,

[10] including 20 minutes reserved for starting vllm to host the LLM

however, is not necessarily supported by the comparatively low variation in the corresponding results shown in Figure 5.

A.4.2 Additional Details on Experiment 4

In this experiment, optimization takes place over the *reduced* "small", "middle", "large" TNs set (i.e., 1000 networks each), whereas evaluation uses the respective *full* sets (i.e., 5000 networks each). One additional configuration not yet mentioned in the main text also uses the mixed set "all" (cf. Table 3) for optimization. In general, the results of this experiment should be interpreted with the caveat that the corresponding distributions in Figure 5 appear to be relatively broad. Figure 8 shows histograms of the $\log_{10}$ number of FLOPs on the three evaluation sets for the final code of the four different configurations vs the initial code and the baseline "cotengra cheap". As expected, the programmes perform best on the set of TNs on which they were optimized. This is most obvious in the case where optimization on "small" or "medium" TNs resulted in programmes that could not even be executed within a reasonable time on the "large" TNs set. Moreover, the scores computed on the reduced and the full sets roughly coincide which can be interpreted in that direction that no "overfitting-like" effect took place in the in-distribution regime. The fact that optimization on the "all" set yielded a competitive result only on the "large" TNs set suggests that the three sets of TNs may exhibit characteristics too different for joint optimization, and that the difficulties associated with "large" TNs might be dominant.

A.5 Additional Details on Code Complexity and Comprehension

As the amount of generated code is by far too large for manual inspection, we utilize the following three simple metrics in order get at least a rough impression on comprehensibility of the codes generated in Experiments 1 and 2: (i) the number of lines of code in the block to evolve (i.e., excluding the doc string), (ii) the percentage of comment lines within these lines, (iii) the code complexity computed by complexipy[11] normalized by the number of code lines. While the total length of the code and the relative complexity appear to grow over the number of iterations, no such obvious trend can be identified for the percentage of comment lines; cf. Figure 9. Notably, a high number of lines of code does not necessarily imply better performance as, e.g., the comparison Qwen3-Coder-Next vs Qwen3.5-122B and Devstral-Small-2 vs MiniMax M2.7 shows. Similarly, the overall best program (GPT-OSS-20B "high") has a relative complexity of ≈ 0.80 while its close competitors MiniMax M2.7 and Gemma-4-31B have values ≈ 0.92 and ≈ 0.44, respectively.

For the codes from Experiment 2 we display the results in Figure 10. The number of code lines grows with increased level of reasoning and, as one may expect, also relative code complexity does so. Surprisingly, the percentage of comment lines decreases heavily with higher reasoning level, although one might have expected increased code quality from increasing the reasoning level; one could speculate that a higher reasoning level leads to a higher adherence to the system message in which there is unintentionally no explicit reference to comments. Further analysis showed that also in this setting there is no obvious relation between complexity and performance of the code.

[11] https://rohaquinlop.github.io/complexipy/, Version 5.3.0.

A.6 Additional Details on the "best" Outcome

Figure 11 shows the same five metrics as considered in Experiment 3 during the run of Experiment 1 that produced the "best" Outcome. It can be seen that except for "min_log10_speedup" in all metrics an improvement over the baseline "cotengra cheap" is achieved. To put this into more context, Table 7 also shows the full experimental evaluation of the results from Experiment 1 for a number of other models. It can be seen that the run with GPT-OSS-20B and reasoning level "high" is indeed the only one that improved over the baselines on "small" TNs at all and the only one improving over the more expensive baselines for "middle" TNs. It still leads on "large" TNs, for which other models reach improvements over one or more baselines, too; however, the absolute improvements might not be significant due to the small number of test cases.

Whereas Figure 4 showed distributions of log10 FLOPs and runtimes over the respective test sets for the best programm vs different baselines, Figure 12 additionally illustrates the distributions of per-network improvements in log10 FLOPs (top) and runtime (bottom) over the test sets. Despite standard deviations that are, in some cases, relatively large, the empirical distributions for those cases with clearly positive averages appear to suggest at least a slight bias towards the positive side.

Name	**lengths**	**dimensions**	**number of TNs**
small (full)	16 to 32	2 to 8	5000
middle (full)	3 to 64	2 to 8192	5000
large (full)	3 to 1024	2 to 8	5000
small (reduced)	first 1000 TNs from small (full)		
middle (reduced)	first 1000 TNs from middle (full)		
large (reduced)	first 1000 TNs from large (full)		
all	first 334/333/333 TNs from small/middle/large (full), respectively		

Table 3: Sets of TNs used in this study.

Name	`methods=`	`max_repeats=`	`optlib=`
cotengra cheap	`"greedy"`	32	`"random"`
cotengra+cmaes	`["greedy","kahypar"]`	1024	`"cmaes"`
cotengra+optuna	`["greedy","kahypar"]`	1024	`"optuna"`

Table 4: Configuration of cotengra's `HyperOptimizer` used for references/baselines.

Parameter	value(s)
checkpoint_interval	1
max_code_length	32000
system_message	see Listing 1
population_size	40
num_islands	4
migration_interval	20
feature_dimensions	"avg_log10_speedup", "median_log10_speedup", "max_log10_speedup", "min_log10_speedup", "log10_total_flops_reduction", and "combined_score"
Evaluator timeout	600
parallel_evaluations	3 (Experiments 3 and 4) / 1 (else)
cascade_evaluation	false
LLM max_tokens	64000 (for GPT-OSS-120B "high") / 32000 (else)
LLM timeout	1500 (for GPT-OSS-120B "high") / 1200 (else, if reasoning "high") / 720 (else)

Table 5: Configuration of OpenEvolve for our experiments. All parameters not shown here were set to the default ones.

Name	**by**	**size (active)**	**release**	**deployment**
GPT-OSS-120B	OpenAI (US)	117B (5.1B)	08/2025	ChatAI/Blablador
GPT-OSS-20B		21B (3.6B)	08/2025	self-hosted (vllm*)
Qwen3-235B-A22B-Instruct	Alibaba (CN)	235B (22B)	04/2025	ChatAI/Blablador
Qwen3-30B-A3B-Instruct		30.5B (3.3B)	04/2025	ChatAI
Qwen3-30B-A3B-Thinking		30.5B (3.3B)	04/2025	ChatAI
Qwen3-Coder-30B-A3B-Instruct		30B (3B)	07/2025	ChatAI
Qwen3-Coder-Next-80B-A3B		80B (3B)	02/2026	Blablador
Qwen3.5-122B-A10B-Instruct	Alibaba (CN)	122B (10B)	02/2026	Blablador
Qwen3.5-35B-A3B-Instruct		35B (3B)	02/2026	Blablador
MiniMax-M2.5	MiniMax (CN)	230B (10B)	02/2026	Blablador
MiniMax-M2.7		230B (10B)	04/2026	Blablador
GLM-4.7	Zhipu AI (CN)	355B (32B)	12/2025	ChatAI
Mistral-Large-3	Mistral AI (FR)	675B (41B)	12/2025	ChatAI
Devstral-2		123B (dense)	12/2025	ChatAI
Devstral-2-small		24B (dense)	12/2025	self-hosted (vllm*)
Gemma-4-31B-it	Google (US)	30.7B (dense)	04/2026	Google AI Studio
Gemma-4-26B-A4B-it		25.2B (3.8B)	04/2026	Google AI Studio

Table 6: Overview over the LLMs used in this study. (*versions: 0.12 for Experiment 1, 0.16 for Experiments 2-4)

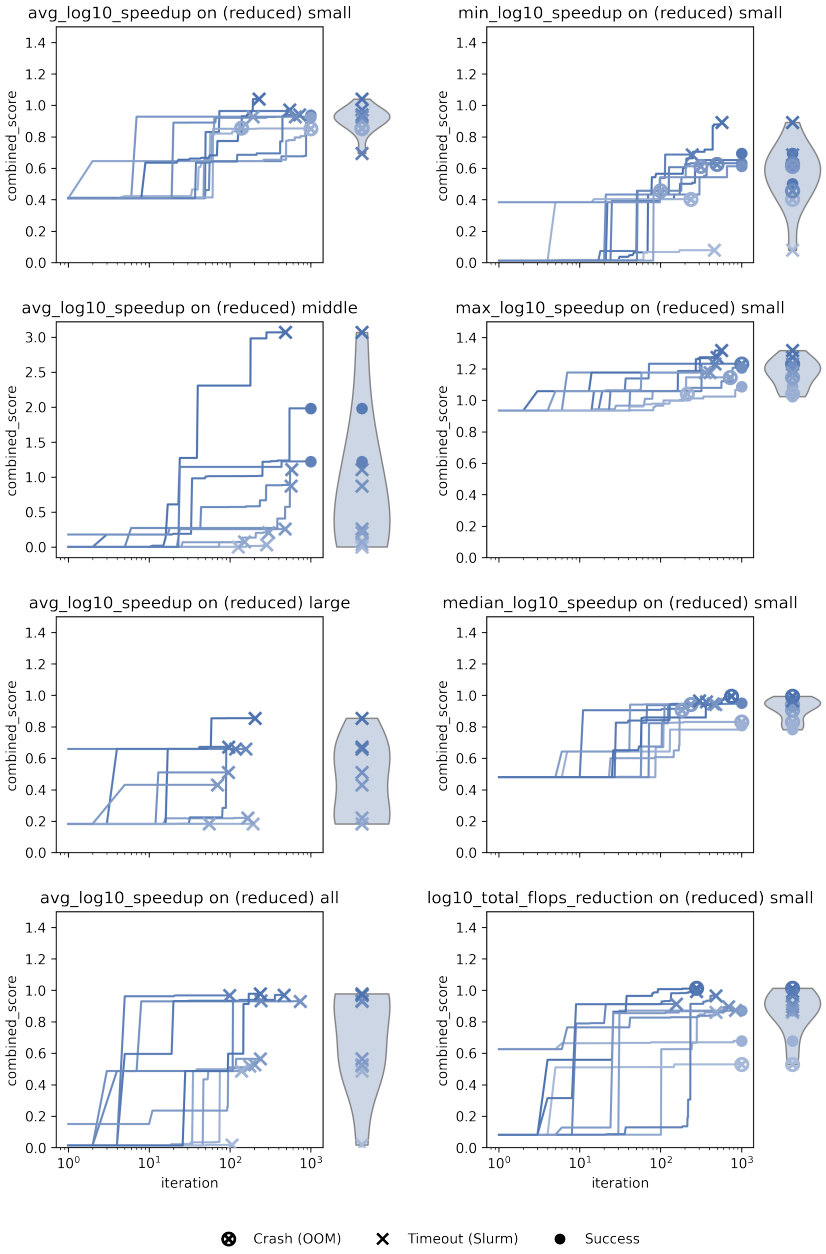


Figure 5: Experiments 3 and 4 — Evolution of "combined_score" for 5 runs of the configurations in Experiments 3 (rhs column) and 4 (lhs column). The different markers at the end of the trajectories indicate different reasons for the end of the iteration.

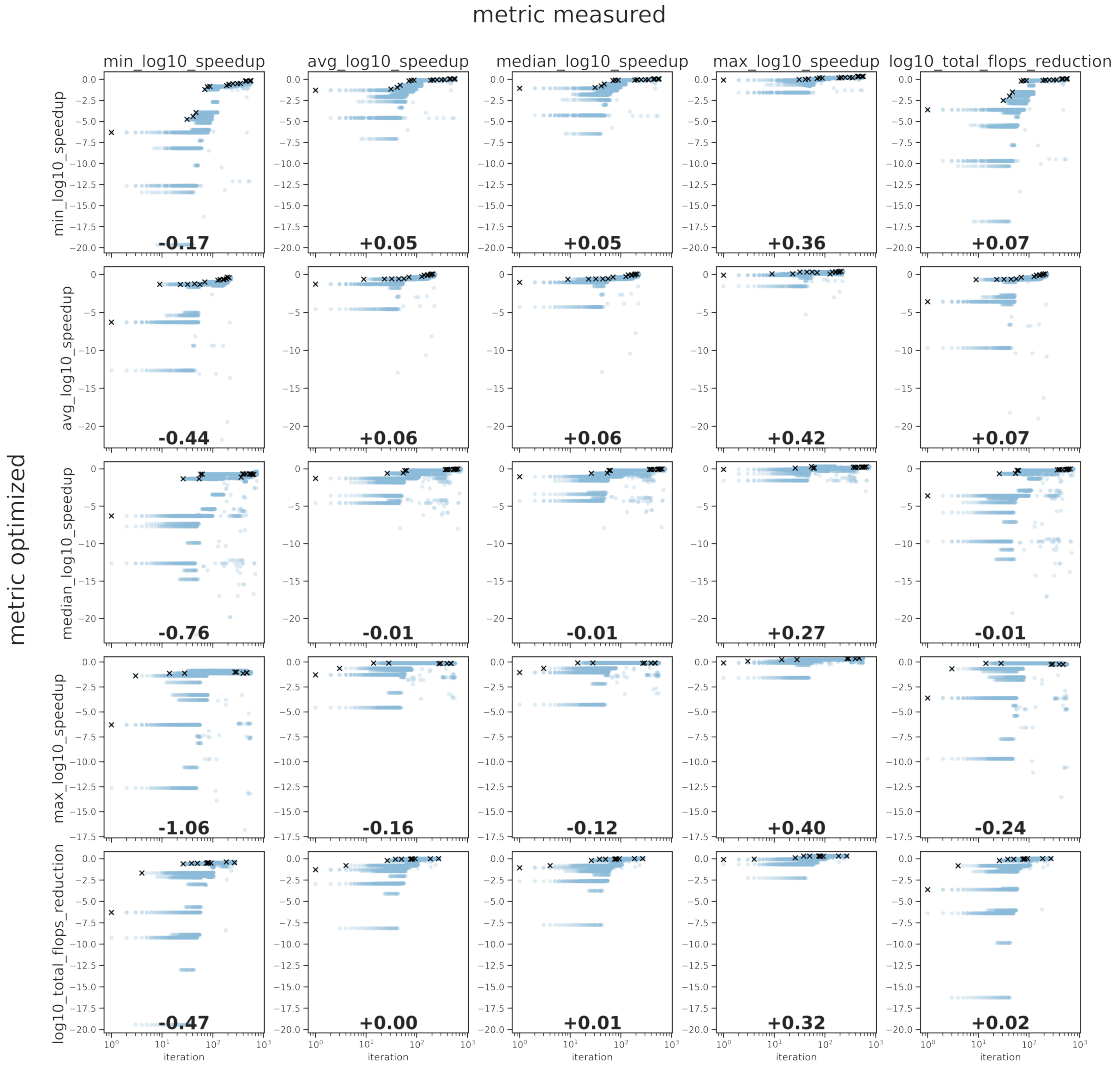


Figure 6: Experiment 3 — Evolution of various metrics ("metrics measured", columns) over the iterations for five experiments with different metric optimized ("metric optimized", rows) with GPT-OSS-20B on the reduced "small" TNs set. Light blue dots indicate individual members of the population, whereas black crosses indicated members of the population that just have become a new best solution (w.r.t. the metric optimized). The number in the lower part of each plot gives the value of the measured metric for the final best solution for the respective experiment.

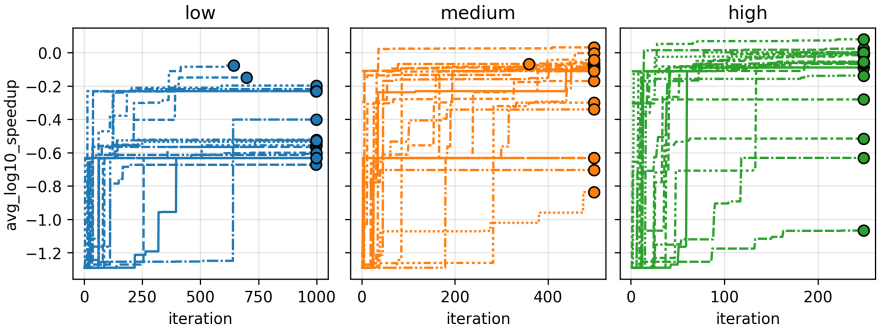


Figure 7: Experiment 2 — Evolution of the average reduction of $\log_{10}$ FLOPs on the "reduced small" TNs set over 20 runs with different random seeds. The LLM is GPT-OSS-20B with different levels of reasoning effort ("low" to "high").

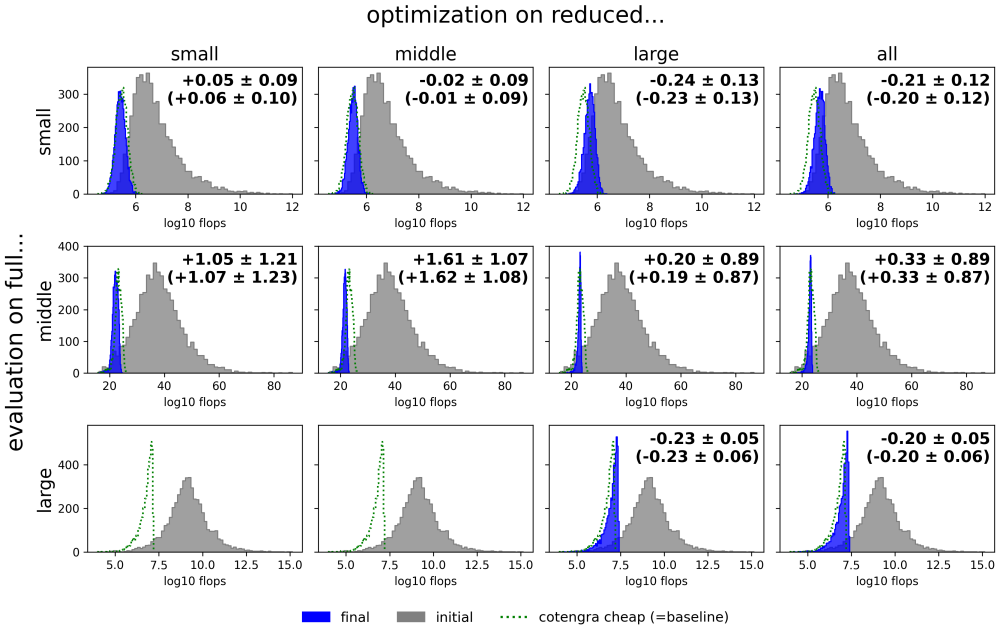


Figure 8: Experiment 4 — Distribution of $\log_{10}$ FLOPs for the final best solution of four experiments with GPT-OSS-20B (evolution on reduced "small", "middle", "large" and "all" TNs sets; columns), evaluated on the full "small" to "large" TNs sets (rows). The gray and blue histograms show the distribution of FLOPs of the initial and the final best code, respectively. The dotted green line indicates the distribution observed for the baseline "cotengra cheap" and the numbers in the upper right corner give the average improvements compared to this baseline (higher is better) on the full and the reduced (in brackets) sets; $\pm$ indicates standard deviations.

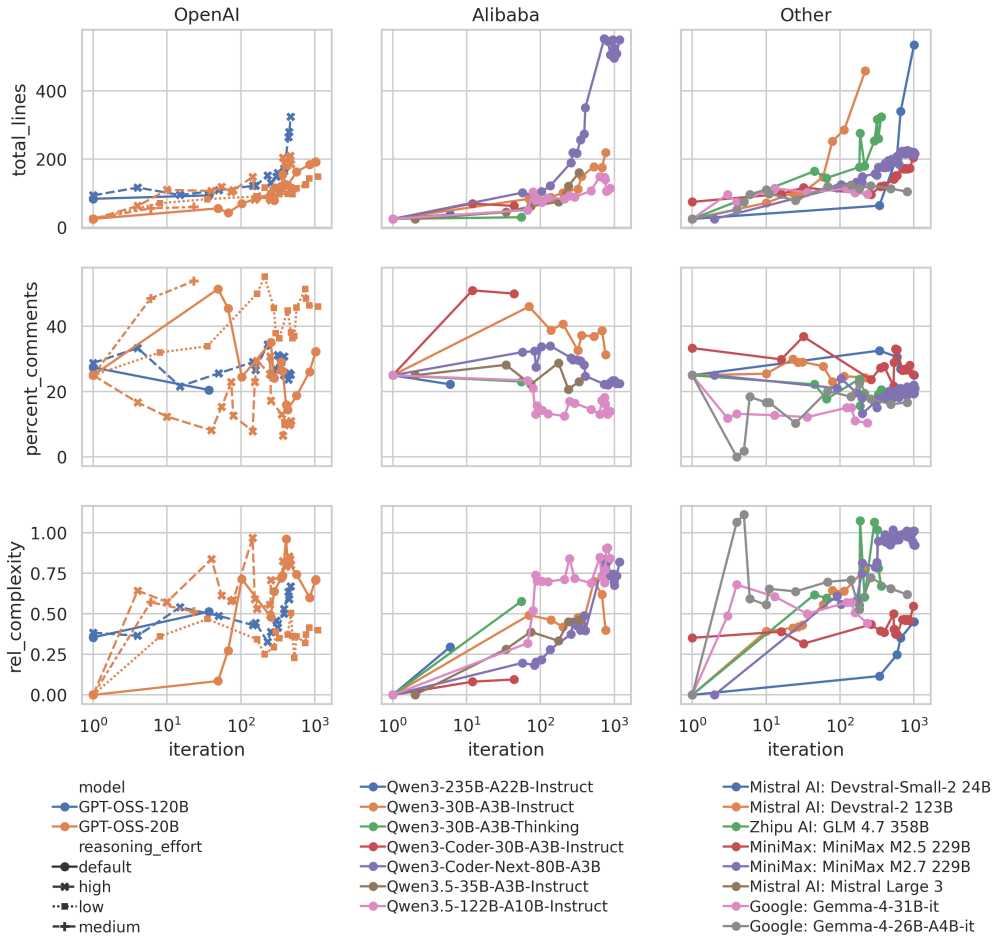


Figure 9: Experiment 1 — Number of lines of codes, percentage of comments, and relative code complexity for the codes leading at times in Experiment 1; cf. Figure 2.

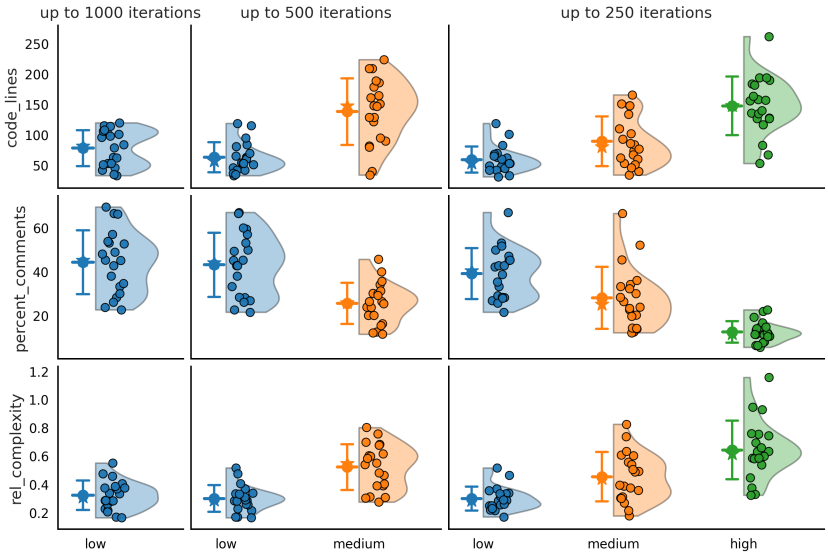


Figure 10: Experiment 2 — Number of lines of codes, percentage of comments, and relative code complexity; cf. Figure 3.

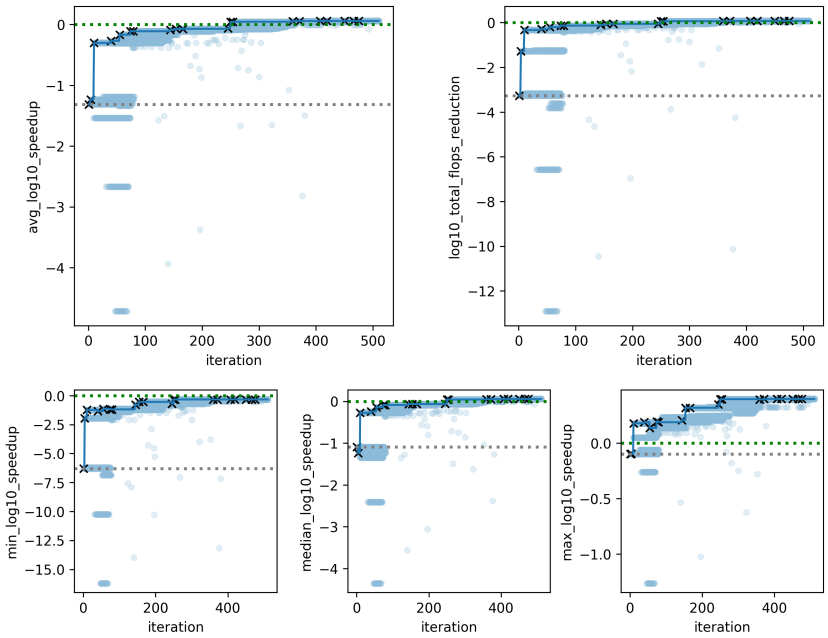


Figure 11: Experiment 1 — Evolution of various metrics for the best run (GPT-OSS-20B on the "full small" TNs set with reasoning level "high"). Light blue dots indicate individual members of the population, whereas black crosses indicated members of the population that just have become a new best solution. The gray dotted lines indicate the level of the initial code, whereas the green dotted line indicates the threshold for an improvement over the baseline "cotengra cheap".

	small			middle			large		
	cheap	cmaes	optuna	cheap	cmaes	optuna	cheap	cmaes	optuna
					average $\log_{10}$ FLOP reduction				
GPT-OSS-20B-high	**+0.06** (±0.08)	-0.02 (±0.06)	-0.02 (±0.06)	**+1.67** (±1.04)	**+0.82** (±0.73)	**+0.75** (±0.70)	**+0.15** (±0.04)	**+0.11** (±0.04)	**+0.11** (±0.04)
GPT-OSS-120B-high	-0.02 (±0.08)	-0.10 (±0.07)	-0.09 (±0.08)	-1.10 (±1.19)	-2.05 (±1.30)	-2.12 (±1.34)	**+0.03** (±0.03)	-0.01 (±0.04)	-0.01 (±0.04)
Qwen35-122B	-0.02 (±0.09)	-0.10 (±0.08)	-0.09 (±0.08)	-0.90 (±1.49)	-1.79 (±1.68)	-1.86 (±1.71)	**+0.06** (±0.04)	**+0.02** (±0.04)	**+0.02** (±0.04)
Qwen35-35B	-0.22 (±0.12)	-0.30 (±0.12)	-0.30 (±0.12)	**+0.26** (±0.78)	-0.68 (±0.46)	-0.75 (±0.49)	-0.20 (±0.05)	-0.24 (±0.04)	-0.24 (±0.04)
MiniMax-M25	-0.14 (±0.11)	-0.22 (±0.09)	-0.22 (±0.09)	-0.66 (±0.85)	-1.55 (±0.48)	-1.62 (±0.50)	-0.12 (±0.05)	-0.16 (±0.03)	-0.16 (±0.03)
MiniMax-M27	-0.01 (±0.08)	-0.09 (±0.07)	-0.09 (±0.07)	⋆	⋆	⋆	**+0.04** (±0.03)	+0.00 (±0.03)	+0.00 (±0.03)
Gemma-4-31B	-0.09 (±0.11)	-0.16 (±0.11)	-0.16 (±0.11)	-2.14 (±1.86)	-3.23 (±2.03)	-3.30 (±2.03)	-0.07 (±0.04)	-0.11 (±0.04)	-0.11 (±0.04)
Gemma-4-26B	-0.10 (±0.12)	-0.17 (±0.11)	-0.17 (±0.11)	-2.24 (±1.90)	-3.13 (±2.01)	-3.21 (±2.01)	-0.07 (±0.04)	-0.11 (±0.04)	-0.11 (±0.04)
GLM47	-0.21 (±0.17)	-0.29 (±0.14)	-0.29 (±0.14)	-2.47 (±2.07)	-3.36 (±2.18)	-3.43 (±2.16)	-0.24 (±0.08)	-0.28 (±0.07)	-0.28 (±0.07)
Devstral-2-123B	-0.24 (±0.13)	-0.33 (±0.18)	-0.33 (±0.17)	⋆	⋆	⋆	-0.23 (±0.06)	-0.27 (±0.05)	-0.27 (±0.05)
Devstral-Small-2	-0.23 (±0.13)	-0.32 (±0.12)	-0.31 (±0.12)	**+0.20** (±0.89)	-0.72 (±0.47)	-0.79 (±0.50)	-0.21 (±0.05)	-0.26 (±0.05)	-0.26 (±0.05)
					average run time speedup				
GPT-OSS-20B-high	**1.91** (±1.43)	540 (±350)	777 (±619)	**4.37** (±6.62)	**2280** (±4640)	**3600** (±7350)	***0.0629*** (±0.320)	**17.9** (±123)	**23.4** (±173)
GPT-OSS-120B-high	1.32 (±0.689)	384 (±178)	544 (±355)	1.93 (±4.02)	919 (±2420)	1410 (±3230)	***0.0407*** (±0.134)	9.73 (±48.3)	11.1 (±67.8)
Qwen35-122B	7.52 (±2.83)	2120 (±653)	2920 (±1490)	9.54 (±13.5)	4150 (±7800)	6430 (±11800)	***0.419*** (±1.19)	**96.5** (±437)	**104** (±614)
Qwen35-35B	2.18 (±1.31)	637 (±275)	891 (±532)	**4.57** (±8.95)	2240 (±6110)	3610 (±8410)	0.0676 (±0.357)	20.1 (±141)	25.6 (±199)
MiniMax-M25	8.11 (±3.77)	2300 (±937)	3210 (±1900)	8.70 (±9.86)	4050 (±5840)	6290 (±10400)	0.670 (±0.933)	124 (±291)	102 (±411)
MiniMax-M27	4.19 (±2.03)	1210 (±552)	1680 (±1020)	⋆	⋆	⋆	***0.203*** (±0.519)	44.3 (±182)	47.3 (±257)
Gemma-4-31B	20.3 (±9.63)	5980 (±2430)	8280 (±4800)	24.3 (±27.9)	12100 (±21600)	19100 (±37500)	1.80 (±2.36)	335 (±799)	276 (±1140)
Gemma-4-26B	20.6 (±10.2)	5850 (±2600)	8070 (±5060)	24.2 (±27.0)	10800 (±15500)	16700 (±27500)	2.14 (±4.32)	443 (±1580)	415 (±2230)
GLM47	2.64 (±1.38)	775 (±355)	1090 (±694)	3.36 (±5.24)	1760 (±3840)	2920 (±6380)	0.149 (±0.542)	37.3 (±204)	43.6 (±287)
Devstral-2-123B	35.6 (±16.2)	9690 (±3650)	12800 (±6250)	⋆	⋆	⋆	69.3 (±24.4)	10500 (±3590)	4340 (±4270)
Devstral-Small-2	86.4 (±36.5)	26800 (±10000)	36200 (±18400)	**76.7** (±35.7)	26500 (±12600)	34600 (±28000)	32.6 (±29.9)	5570 (±8140)	3790 (±11200)

Table 7: Experiment 1 — Average reduction of $\log_{10}$ FLOPs (upper table) and average run time speedup (lower table) of the final best programs for different LLMs, on different TN sets and versus different baselines; standard deviations are indicated by ±. **Bold** entries in the upper table indicate improvement over the baseline (in average) with the same entries also highlighted in the lower table; here, *italic* entries indicate higher, i.e., worse, runtime (in average). Failing evaluations are indicated by ⋆.

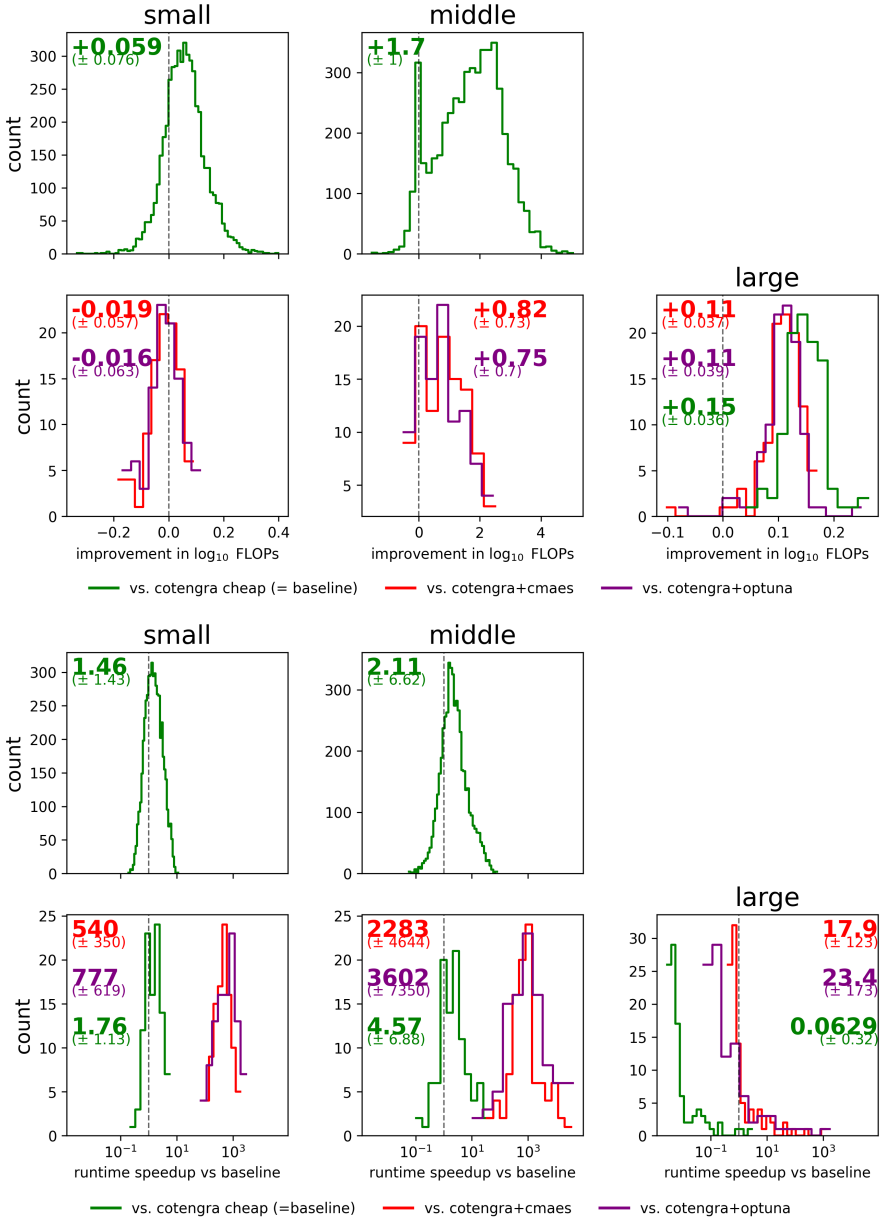


Figure 12: Experiment 1 — Distribution of improvements in log10 FLOPs (top two rows) and distribution of runtime speedups (bottom two rows) for the setting of Figure 4.

```
You are an expert programmer and expert in tensor networks. Your goal is to evolve and improve the code of the function `find_edge_path` in between the markers "EVOLVE-BLOCK-START" and "EVOLVE-BLOCK-END".

CONTEXT:
The function `find_edge_path` gets a tensor network as input, described by the following input parameters:
   `inputs`: list of tensors' index tuples, e.g. [('a','b'), ('b','c'), ('a','c', 'd')]
   `output`: tuple of output indices, e.g. ('d',)
   `size_dict`: map index -> dimension, e.g. {'a':16, 'b':32, 'c':8, 'd': 12}
It returns a list of indices representing the contraction path, e.g. ['c','a','b'].

* Objective: Improve the efficiency of the contraction path found by `find_edge_path`, minimizing the total number of FLOPs associated with the contract path.
* Performance metric: The number of FLOPs of the contraction path found by `find_edge_path` (the lower the better).

The evaluation framework will compute the number of FLOPs of the contraction path returned by your function and compare it for every test case against a baseline implementation. It will provide you with the average, median, maximum, and minimum (over all test cases) of log10 of FLOP **reduction** as well as with the log10 of the total FLOP reduction summed over all test cases.

Your main metric, "combined_score" (the higher the better), is computed from the average log10 improvement.

IMPORTANT HINTS:
* The tensor networks to be contracted in the evaluation have a very specific structure: They are **scalar products of two cyclic/periodic tensor trains / matrix product states. Consider checking for this structure and optimizing the contraction path accordingly.**

* You are free, in fact even encouraged, to experiment with creative ideas and heuristics to optimize the contraction path for this very special tensor network structure.
* You can implement any algorithm or heuristic you see fit, including but not limited to greedy algorithms, dynamic programming, graph-based methods, etc.
* Feel free to try out different ideas, heuristics, and algorithms in whatever combination. Do not hesitate to innovate and think outside the box!
* You are not restricted to existing algorithms; you can design your own methods tailored to the "scalar product of two cyclic/periodic TTs/MPS"-structure.
* You are not restricted to approaches with a theoretical guarantee; heuristic and approximate methods are welcome if they yield better practical performance.

REQUIREMENTS:
* The function must return a valid contraction path.
* Focus on algorithmic improvements and code optimization. You may not use external tensor network libraries to compute the contraction path.
* You may however use standard libraries such as itertools, numpy, scipy, networkx, etc. However, if you use additional libraries, ensure they are included in the import statements.
* Ensure that the code runs correctly and efficiently within the provided function signature.
```

Listing 1: System Message for OpenEvolve

```python
import numpy as np

# EVOLVE-BLOCK-START
def find_edge_path(inputs, output, size_dict):
    """
    This functions determines an edge-based contraction path for a given tensor network, that consists of the scalar product of two cyclic/periodic tensor trains / matrix product states.

    The tensor network is defined by the variables `inputs`, `output`, and `size_dict` that specify the tensors' indices, the output indices, and the dimensions of each index, respectively. These three variables are passed to the function as input arguments.

    The function returns an edge-based contraction path, which is a list of index labels that specifies the order in which the indices should be eliminated during the contraction process.

    Inputs
    --------
    inputs: list of tensors' index tuples, e.g. [('a','b'), ('b','c'), ('a','c', 'd')]
    output: tuple of output indices, e.g. ('d',)
    size_dict: map index -> dimension, e.g. {'a':16, 'b':32, 'c':8, 'd': 12}

    Returns
    -------
    edge_path: sequence of original index labels to eliminate, e.g. ['b', 'a', 'c']
    """
    # simplest idea: eliminate in order of descending size
    sorted_edges = sorted(size_dict.keys(), key=lambda k: size_dict[k], reverse=True)
    return sorted_edges
# EVOLVE-BLOCK-END
```

Listing 2: Initial Program for OpenEvolve