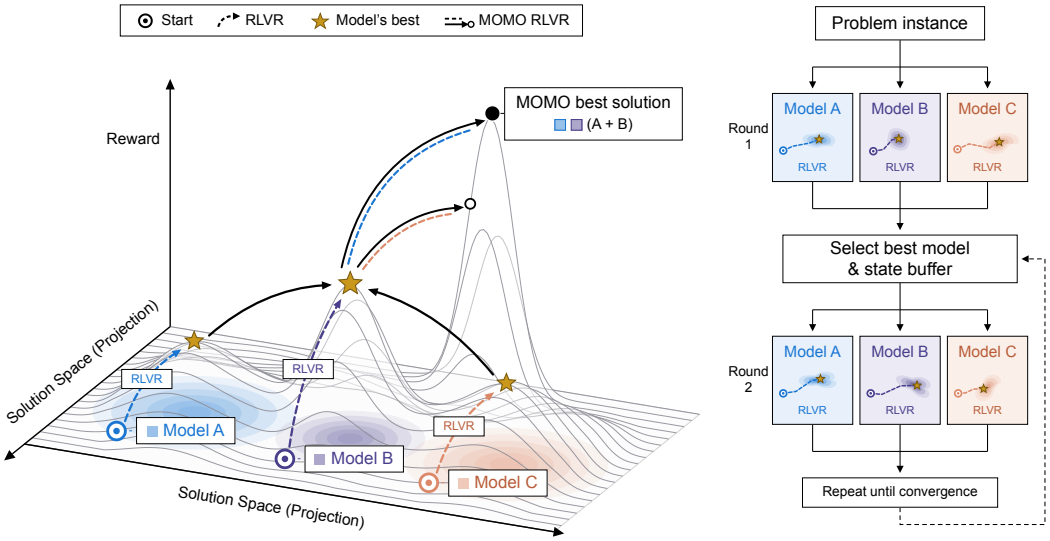
MOMO: MIXTURE OF MODEL OPTIMIZATION

Anonymous authors
Paper under double-blind review



(a) MOMO finds solutions beyond single-model refinement. (b) MOMO search procedure.

Figure 1: **Mixture of Model Optimization (MOMO).** MOMO ensembles models together. When applied to a problem, it can find solutions in the optimization landscape that are unreachable from any single model alone.

ABSTRACT

Reinforcement learning with verifiable rewards (RLVR) can train a language model to improve its solutions on a single problem. But each model refines toward its own set of behaviors, and once it stalls, further training rarely changes its approach. We propose Mixture of Model Optimization (MOMO): instead of training one model longer, we use the solutions it has found as the starting point for training other models. Each recipient keeps its own weights, so it brings different learned behaviors to the same solutions. Across mathematical constructions, circuit placement, and qubit routing with three models of roughly 30B parameters, MOMO consistently improves on single-model training, making progress after the previous model had stalled. MOMO sets a new state of the art on circuit placement, qubit routing, and the Erdős minimum overlap problem, where its gains consistently hold across seeds. Code is available here.

1 INTRODUCTION

We would like language models to do research: run an experiment, read the result, and decide what to try next. Given a goal that can be checked, such an agent should get closer to it as it collects more data. This makes automated research a continual learning problem, and raises a practical question: where does the agent keep what it has learned?

One answer is to keep memory in language, and multi-agent systems have pushed this answer furthest. Their designs vary from problem to problem, though a common primitive is to run agents in parallel and then aggregate their attempts, either by distilling drafts into a shared workspace or by having a model merge candidate solutions (Madaan et al., 2025; Venkatraman et al., 2025; Lee et al., 2026).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

At the largest scale, OpenAI coordinated roughly 10,000 agents through shared channels on the Navier–Stokes problem (OpenAI, 2026). There are many design choices that matter; the best prompts and topologies change from task to task (Zhou et al., 2026), coordination has a cost, and whether it pays off depends on the capability of the base model (Kim et al., 2026).

The other common answer is to keep memory in the weights. Reinforcement learning with verifiable rewards (RLVR) writes into the parameters whatever actions moved the agent toward its goal. Work in this line has mostly varied the advantage function. ThetaEvolve uses the group-relative advantage of GRPO (Wang et al., 2025; Shao et al., 2024), while TTT-Discover and TailRL shift weight toward the tail of the reward distribution (Yuksekgonul et al., 2026; Ramasubramanian et al., 2026). TTT-Discover set new state-of-the-art results in mathematics, GPU kernels, algorithm design, and biology, though search without training has since overtaken some of them (Ye et al., 2026). We ask whether training the weights can reach the state of the art again.

Training several models on these problems, we see the same pattern. Each run converges locally, and each model gets there with its own techniques, regardless of the objective or regularization (Section 3.1). Restarting a run makes its techniques more varied, but throws away what it has found. This led us to continue each model from the best solutions found by a different model.

We call this **Mixture of Model Optimization (MOMO)** (Figure 1). Each model first trains on the problem alone. The buffer with the best solutions is then copied to each of the other models, which continue training from their own weights. We test MOMO with three model families, Qwen3.5-27B, Muse-Glimmer-30B, and Gemma-4-31B, on code-optimization problems in mathematical constructions, circuit placement, and qubit routing. MOMO sets a new state of the art on circuit placement, qubit routing, and the Erdős minimum overlap problem. MOMO is simple, and scales naturally to more model families, since the only thing models exchange is the state buffer. Our experiments suggest that discovery can benefit from model diversity, and MOMO is one simple way to take advantage of it.

2 Problem Setup and Definitions

MOMO is built from one routine, REFINE, which trains a model on a single problem while it searches for better solutions. This section defines the problem and that routine.

The discovery problem. We are given a problem description d and an evaluator that scores candidate solutions. The model writes code, and running the code either produces a valid solution or fails. For a mathematical problem, the solution may be an array that certifies a bound. For a hardware problem, it is a program that must produce a valid result on every instance of a test suite. Our goal is to find the best valid solution within a fixed search budget. Only that one solution matters, not the average quality of what the model generates.

Learning to improve. There is no training set, only the one problem. Instead of sampling fresh answers, we ask the model to improve solutions it has already found. A state holds a candidate program, the construction it returns when there is one, and the evaluator's feedback. The state buffer B stores every scored state and its parent. Given d and a parent state, the model proposes a child state, so it learns an improvement operator rather than a solver. This follows the program-search setting of AlphaEvolve and TTT-Discover (Novikov et al., 2025; Yuksekgonul et al., 2026).

Refinement. Each training step selects 16 parents from the buffer and generates 32 candidates for each, 512 attempts in total. We choose parents with Monte Carlo tree search (MCTS) (Rosin, 2011; Schrittwieser et al., 2020; Yuksekgonul et al., 2026), which favors high-scoring states but returns to less-visited ones. The evaluator scores every candidate, and invalid candidates receive zero reward. The two best valid children of each parent enter the buffer, and a policy-gradient update reinforces the edits that helped. We write this routine as

$$(M', B') = \text{REFINE}(d, M, B, T),$$

which returns the trained model and buffer after T steps. MOMO calls it for every model (Figure 2).

Algorithm 1 REFINE: learn from search

Require: Problem d, model M, buffer B
Require: Refinement steps T
for $t = 1, \ldots, T$ **do**
  Select a batch of parents from B.
  Generate improvements with M.
  Execute and evaluate candidates.
  Add the best valid children to B.
  Compute within-group advantages.
  Update M by policy gradient.
end for
return (M, B)

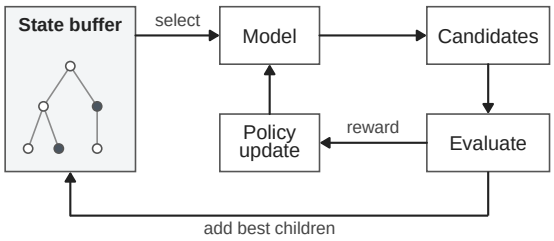


Figure 2: **The REFINE procedure,** shown as pseudocode (left) and a flow diagram (right). Each call returns the trained model parameters and a state buffer of the solutions it generated.

Models and training. We use three open models of roughly 30B parameters (Table 1), chosen for their strong coding, mathematics, and reasoning, which our tasks require. At this size they remain competitive with larger mixture-of-experts models, which are harder to train stably with RL (Ma et al., 2025). For scalability on our TPU infrastructure, we train each model with low-rank adaptation (LoRA), as TTT-Discover does (Yuksekgonul et al., 2026). Schulman (2025) argue that LoRA suffices for RL, because a scalar reward carries little information per episode. LoRA also gives each model its own small adapter, which MOMO keeps across rounds. We use the same recipe for every model, detailed in Appendix A.

Model	Size	Released
Qwen3.5-27B (Qwen) (Qwen Team, 2026)	27B	Feb. 2026
Gemma-4-31B (Gemma) (Gemma Team, 2026)	30.7B	Mar. 2026
Muse-Glimmer-30B (Muse) (Meta, 2026)	29.6B	Aug. 2026

Table 1: Models used in our experiments.

3 MOMO: MIXTURE OF MODEL OPTIMIZATION

MOMO grew out of an observation; models trained on the same problem settle on different techniques. This led us to ask what happens when one model continues from another's solutions, which we formalize as the MOMO algorithm. We then test it on the Erdős minimum overlap problem and study design choices about how to propagate information throughout training.

3.1 MOTIVATION: WHY COMBINE MODELS?

We first studied design choices for RLVR on a single problem. We focus on the Erdős minimum overlap problem (Section 3.3) and compare two advantage functions across four models: GRPO (Shao et al., 2024) and the more recent TTT-Discover objective (Yuksekgonul et al., 2026). On Qwen, we also test how regularization affects training, with a KL penalty or random restarts. To see what each run has learned by the end of training, we take the 32 top-scoring programs in its final state buffer and identify the techniques each one uses by walking the program's abstract syntax tree.

Each model has its own preferred techniques (Figure 3). Qwen and Gemma both settle on a smooth-max surrogate optimized with SLSQP, under either objective. Muse instead compiles its inner loops with Numba to speed up its optimization, which the other models almost never do. GPT-OSS spreads its programs across the widest range of techniques. The objective changes this mix for Muse and GPT-OSS but hardly for Qwen and Gemma, while regularization changes it for Qwen under both objectives.

Each model therefore falls into a few popular modes, and some of these modes exclude each other: Qwen and Gemma rarely use Numba, and Muse's GRPO programs drop smooth max and SLSQP entirely. This made us wonder whether combining models could keep a more consistent diversity of techniques than any single model. Restarts point the same way. Restarting a run was enough to diversify Qwen's techniques, which suggested a different kind of restart: keep the best solutions found so far, but continue from them with a different model. MOMO restarts across models rather than within one.

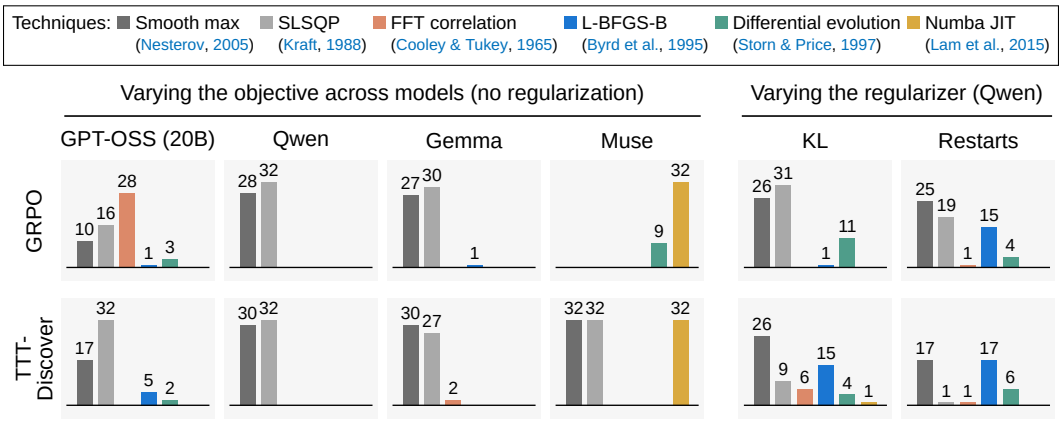


Figure 3: **Models exhibit distinct behavior depending on objective and regularization.** Bars count the techniques used by the 32 top-scoring programs in the final state buffer on Erdős minimum overlap. Left: varying the objective across models; right: varying the regularizer on Qwen.

3.2 MOMO ALGORITHM

MOMO is organized into rounds. In round 1, each model starts from its own state buffer and learns independently with REFINE (RLVR). Each run returns a state buffer, which holds its previous solutions and the edits that improved them, and new model parameters. We pick the model whose buffer holds the best solution. In the next round, every other model continues from a copy of that buffer and from its own trained weights, so it does not discard its earlier learning. The model we picked sits out that round (Algorithm 2).

Algorithm 2 MOMO: retain each model, continue the best states

Require: Problem d, models $M_1, \ldots, M_K$ ($K \geq 2$), initial buffers $B_1, \ldots, B_K$
Require: Round budgets $T_1, \ldots, T_R$
Set the previous winner to "none."
for round $r = 1, \ldots, R$ **do**
  for each model i other than the previous winner **do**
    if $r = 1$ **then**
      $B' \leftarrow B_i$ ▷ start from its own buffer
    else
      $B' \leftarrow$ a copy of the best states in B, with visitation counts reset
    end if
    $(M_i, B_i) \leftarrow \text{REFINE}(d, M_i, B', T_r)$ ▷ keep model i's own weights
  end for
  Select the returned buffer B_j with the best solution.
  $B \leftarrow B_j$; set the previous winner to j.
end for
return the best solution in B.

Two choices determine how information propagates between rounds. **State transfer** decides how to reuse the solutions found by previous models. We copy the donor's best states into each recipient's buffer and reset their visitation counts, so the donor's search does not bias the recipient's. **Weight retention** decides how to keep what each model has internalized. Each recipient keeps its own trained weights, but we reset its optimizer state, since it now optimizes a new set of solutions.

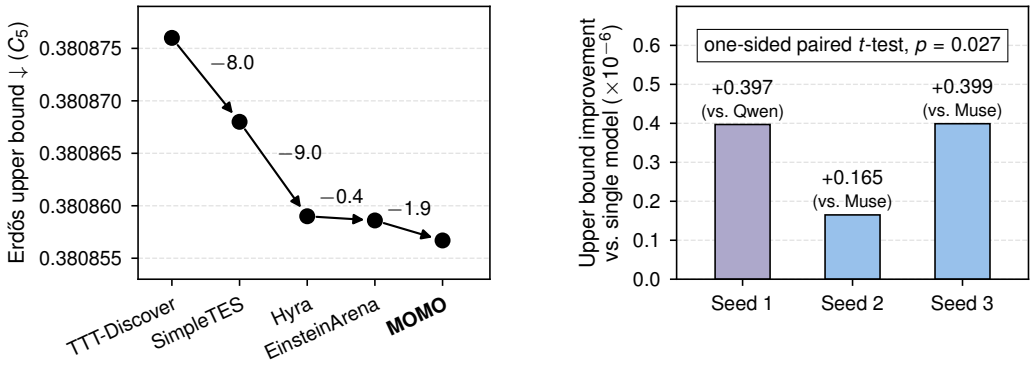
3.3 TESTING ON ERDŐS MINIMUM OVERLAP

Erdős's minimum overlap problem (Erdős, 1955) asks how evenly the integers $\{1, \ldots, 2n\}$ can be split into two halves A and B. Shift B by some amount and count how many of its elements land on elements of A. The worst shift gives the overlap of the split, and C_5 is the limit of the smallest possible overlap, divided by n, as n grows. Its exact value is unknown. Following Georgiev et al. (2025), any

function f on $[-1, 1]$ with values in $[0, 1]$ and $\int f = 1$ gives an upper bound,

$$C_5 \leq \sup_{x \in [-2,2]} \int_{-1}^{1} f(t)\, g(x+t)\, dt,$$

where $g = 1 - f$ on $[-1, 1]$ and both are zero outside it. The function f is a continuous function representing a partition, and the bound is its largest overlap with its complement over all shifts; lower is better. Programs return a step function f, and the evaluator checks the constraints and rewards valid ones with $1/(10^{-8} + C_5)$.



(a) State of the art on Erdős minimum overlap.

(b) MOMO consistency over seeds.

Figure 4: **MOMO sets a new state of the art on Erdős minimum overlap.** It improves on the previous best agentic result, from EinsteinArena, and on single-model refinement in each of three seeds, though the best solution comes from the first seed. Arrow labels and bars are in units of 10^{-6}.

Figure 4a shows the upper bound over recent methods. TTT-Discover (Yuksekgonul et al., 2026), which trains the model at test time, improved the bound beyond AlphaEvolve V2 (Georgiev et al., 2025). SimpleTES (Ye et al., 2026) then scaled up search without training, and agents lowered it further (the Hyra research agent (Hyra Team, 2026) and EinsteinArena (Bianchi et al., 2026)). MOMO sets a new record, $C_5 = 0.3808567$, and is the first method since TTT-Discover to hold it by training weights. When tested across seeds, MOMO improves on training individual models (Figure 4b; one-sided paired t-test, $p = 0.027$).

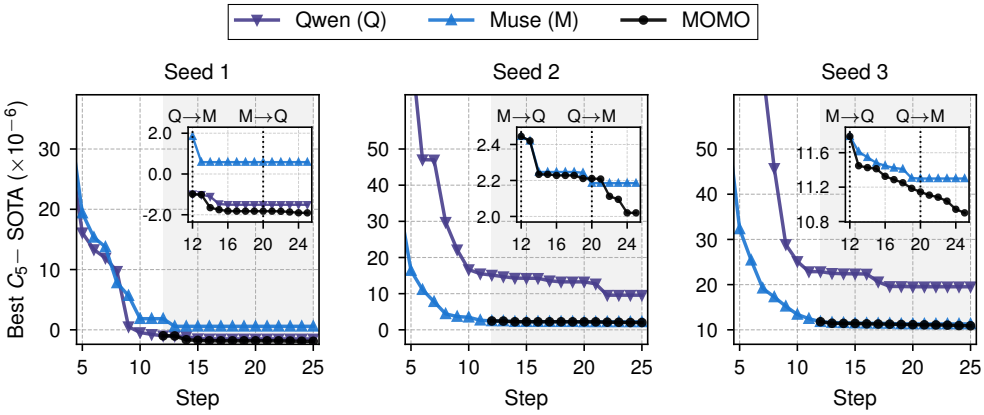


Figure 5: **MOMO improves on individual models across seeds.** Best C_5 per step, relative to EinsteinArena's previous best bound; lower is better. MOMO runs 12, 8, and 5 steps in rounds 1, 2, and 3; dotted lines in the insets mark each switch of model. Insets zoom into steps 12–25; Qwen is off-scale in seeds 2 and 3.

Figure 5 shows the seeded analysis over steps. Here MOMO uses only Qwen and Muse, with rounds of 12, 8, and 5 steps: 37 model-steps in total, against 25 per single model. The stronger single model changes with the seed: Qwen is better in seed 1, and Muse in seeds 2 and 3. MOMO is robust to this, and within the 25 steps of the comparison it ends ahead of the better single model in every seed.

3.4 What should cross a round boundary?

Our design, as tested on Erdős minimum overlap, transfers the state buffer and retains each model's weights. We also ablated other ways to continue across a round, with Qwen as the donor and Muse and Gemma as recipients (Table 2). For the starting point, we compare the donor's state buffer with a short prompt summary of the techniques and scores that worked in the previous round. For the weights, we compare fresh weights with the weights retained from the previous round.

	State continuation		Weight continuation	
Model	Prompt summary	State buffer	Fresh	Retained
Muse	-40.25	$\mathbf{+0.81}$	$+0.44$	$\mathbf{+0.64}$
Gemma	-53.79	$\mathbf{+0.41}$	$+0.50$	$\mathbf{+0.57}$

Table 2: **Continue from the states and keep the weights.** Change in best C_5 relative to Qwen's round 1 result ($\times 10^{-6}$); positive is better. Based on this, MOMO transfers the state buffer and retains weights across rounds.

Starting from Qwen's strong buffer, a prompt summary of what worked is not enough: both recipients end well above Qwen's solution, far worse than when they continue from the states in the buffer. Fresh weights still give a gain, but a smaller one than retained weights. This is why MOMO transfers the state buffer and retains each model's weights across rounds.

4 Experimental Results

We run MOMO for three rounds on two mathematical construction problems, Erdős minimum overlap and the first autocorrelation inequality (AC1), and two hardware optimization problems, circuit placement and qubit routing. On each problem, we compare against the prior state of the art and report the best result of each round. Figure 6 summarizes the winning solutions.

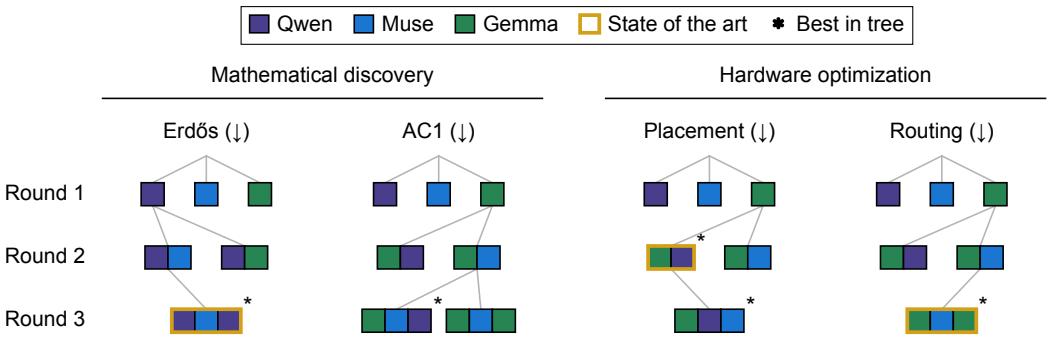


Figure 6: **MOMO finds state-of-the-art solutions on three of four problems.** Each row is a round; in each round, the other models continue from the state buffer of the previous round's best model, with their own weights. An asterisk marks each tree's best solution, a gold outline a new state of the art. On placement, Round 3 gives no further reduction and ties Round 2, where the state of the art stays.

4.1 Mathematical discovery

Both problems ask for a step function that satisfies a bound. Section 3.3 defines Erdős minimum overlap. The first autocorrelation inequality (AC1) concerns nonnegative functions f supported on $[-1/4, 1/4]$: C_1 is the largest constant such that

$$\max_{|t|\le 1/2} (f * f)(t) \;\ge\; C_1 \Big(\int f \Big)^2$$

for all such f. Any valid f therefore gives the upper bound $C_1 \le \|f * f\|_\infty / \|f\|_1^2$, and lower is better. Intuitively, $(f * f)(t)$ measures how much f overlaps a mirrored copy of itself shifted by t, so a good f spreads its mass so that no shift produces a tall peak.

Method	Model	Erdős $C_5 \downarrow$	AC1 $C_1 \downarrow$
Existing Best Solutions			
TTT-Discover	GPT-OSS-120B	0.3808760	1.502870
SimpleTES	GPT-OSS-120B	0.3808680	1.503871
Hyra	Agent (model not reported)	0.3808590	1.502850
EinsteinArena	Agent (custom: CodexProLong)	0.3808586	**1.502744**
MOMO			
Round 1	Qwen (Q)	0.3808571	1.506568
Round 1	Gemma (G)	0.3808608	1.504545
Round 1	Muse (M)	0.3808592	1.505513
Best Round 2	Mixed config (Erdős: Q→M | AC1: G→M)	0.3808568	1.503496
Best Round 3	Mixed config (Erdős: Q→M→Q | AC1: G→M→Q)	**0.3808567**	1.503463

Table 3: **MOMO's results on mathematical constructions.** MOMO achieves the best bound on Erdős minimum overlap; lower is better. Bold marks the state of the art, and underline the best of MOMO.

Table 3 compares MOMO with prior methods, from TTT-Discover to EinsteinArena. MOMO achieves the best bound on Erdős minimum overlap. On AC1, it improves on SimpleTES but stays behind TTT-Discover, Hyra, and EinsteinArena. On both problems, the bound improves in every round under the same fixed procedure. Figure 5 and Appendix B show the two searches step by step.

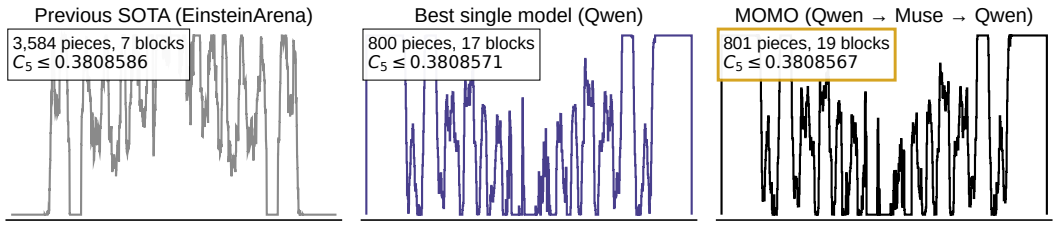


Figure 7: **MOMO finds a different, coarser construction than the previous state of the art.** The previous state of the art (left) uses many pieces in a few blocks (intervals where $h > 0$); MOMO's new state of the art (right) uses far fewer pieces spread over more blocks. Middle: the best single model.

Erdős minimum overlap. MOMO sets a new upper bound, $C_5 \le 0.3808567$, and improves in every round. The best lineage is Qwen → Muse → Qwen, and each model changes something different. Qwen, alone, minimizes a smooth maximum of the overlap with SLSQP. Muse, given Qwen's buffer, resampled the function onto 801 pieces, restarted the optimizer from noisy copies of the best function, and added a random local search. Qwen, given Muse's buffer, replaced the random local search with a single optimization round, which gave the final gain (Figure 7). Appendix C lists the program that produced it.

First autocorrelation inequality. The best lineage is Gemma → Muse → Qwen, and each model adds a different step. Gemma builds the optimizer that every later program keeps. Muse moves the best sequence onto ever finer grids, which Qwen, given the same buffer, never does. Qwen then polishes Muse's fine-grid sequences more aggressively than Gemma. MOMO helps because each model supplies a step the others skip; most of the gain is Muse's re-gridding (Appendix B).

4.2 Hardware Optimization

Both hardware tasks ask the model for a placement algorithm rather than a single solution. Given a configuration, either a chip's circuit blocks or a quantum circuit on a qubit topology, the code must turn it into a placement that minimizes a cost function, scored across a suite of configurations.

Circuit placement. Chip placement is a combinatorial optimization problem: each block on a chip connects to others, so where it sits sets costs such as wire length, density, and routing congestion.

Method	Model	Placement cost $\bar{c}\downarrow$
Existing Best Solutions		
ArchGen	Human-designed	0.9611
AbuPlace	Human-designed	0.9533
MOMO		
Round 1	Qwen (Q)	0.9408
Round 1	Gemma (G)	0.9318
Round 1	Muse (M)	0.9525
Best Round 2	Mixed config (G→Q)	**0.9214**
Best Round 3	Mixed config (G→Q→M)	0.9214*

Table 4: **MOMO's results on circuit placement.** Mean proxy cost over the 17 IBM designs; lower is better. Bold marks the best result, and underline the best of MOMO. *Round 3 gives no further reduction; the best solution stays at Round 2.

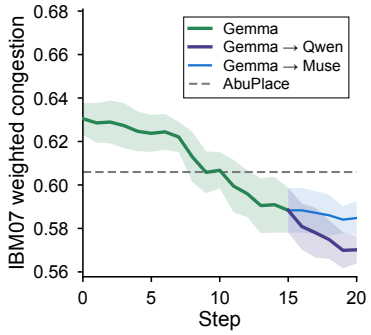


Figure 8: **Qwen on Gemma cuts congestion more than Muse.**

AlphaChip (Mirhoseini et al., 2021) framed placement as a reinforcement learning problem that optimizes a proxy cost for a given chip design. The TILOS AI Institute's MacroPlacement project then replicated this environment on open designs (TILOS AI Institute, 2022; Cheng et al., 2026), and the Partcl/HRT Macro Placement Challenge recently standardized its evaluation (Partcl & Hudson River Trading, 2026), which we use.

We use the challenge's 17 IBM designs from ICCAD04. The program must place each design's macros, large chip elements such as memory blocks, inside the canvas and without overlap. The TILOS evaluator scores a placement by a proxy cost $c_i = W_i + 0.5D_i + 0.5G_i$ that combines wirelength, density, and congestion, averaged over the designs as $\bar{c} = \frac{1}{17}\sum_i c_i$; lower is better. The reward is $1/(1+\bar{c})$ when every design is valid and zero otherwise. To save computation, we skip the challenge's second tier, which ran the top entries through a fuller flow on more designs.

We compare with the challenge's two winners (Table 4), re-running their released code on NVIDIA A10 GPUs. ArchGen (ArchGen AI, 2026) ranks many candidate moves at once with GPU coordinate descent and checks each accepted move with the exact evaluator. AbuPlace (Gottumukkala, 2026) ports the evaluator to C, so it can score every move exactly while refining a global placement. MOMO beats both winners by Round 2, and Round 3 offers no further cost reduction.

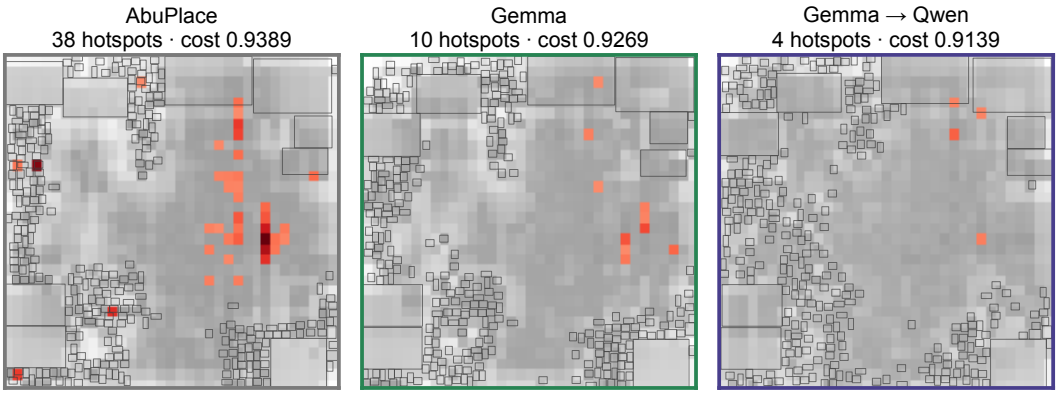


Figure 9: **MOMO removes routing hotspots.** IBM07 layouts over the estimated routing congestion (gray), with hotspots in red. Left to right: AbuPlace, Gemma alone (Round 1), and Qwen continuing Gemma (Round 2).

Gemma is the strongest single model, and MOMO builds on it with Qwen. Qwen puts more emphasis on penalizing congestion, which lowers the mean cost (Figure 8). Figure 9 shows how on IBM07: AbuPlace leaves a band of hotspots through the center of the chip, Gemma removes most of them, and Qwen's continuation removes nearly all the rest.

Qubit routing. A quantum circuit may request an interaction between qubits that are not adjacent on its processor. A routing algorithm chooses an initial placement and inserts SWAP operations to make the required interactions possible. Each SWAP costs three CNOT gates, so fewer SWAPs give

shorter, less noisy circuits. SABRE (Li et al., 2019) and its successor LightSABRE (Zou et al., 2024) are the standard heuristics. We use SimpleTES's routing benchmark (Ye et al., 2026): candidate Rust code sets the initial layout and SWAP choice, and a fixed engine builds and checks the circuit.

The suite contains 24 circuits on each of three processors, Q20, Willow, and Heron. Table 5 reports the SWAPs added on each processor and their total. Training uses the weighted count $S = 0.2S_Q + 0.4S_W + 0.4S_H$ and the reward $B/(B + S)$, where B is SABRE's weighted count. Appendix D describes where each baseline's counts come from.

Method	Model	Added SWAPs per topology ↓			
		Q20	Heron	Willow	Total
Existing Best Solutions					
SABRE	Heuristic	22,714	50,186	38,352	111,252
LightSABRE	Heuristic	20,063	45,827	36,802	102,692
SimpleTES	GPT-OSS-120B	15,147	42,274	32,258	89,679
SimpleTES	Gemini 3 Pro	13,470	42,396	**31,481**	87,347
Hyra	Agent (model not reported)	**12,623**	**41,534**	31,966	86,123
MOMO					
Round 1	Qwen (Q)	13,239	42,871	32,061	88,171
Round 1	Gemma (G)	13,367	41,569	31,728	86,664
Round 1	Muse (M)	16,156	45,024	36,859	98,039
Best Round 2	Mixed config (G→M)	12,897	41,540	31,752	86,189
Best Round 3	Mixed config (G→M→G)	12,810	41,540	31,752	**86,102**

Table 5: **MOMO achieves state of the art on qubit routing.** Q20, Heron, and Willow are quantum processor topologies; we route 24 circuits on each and count the added SWAPs (lower is better; bold is best, underline best of MOMO). Our best configuration sets the state of the art in this setting.

Gemma alone, trained with RLVR, already beats every previous routing program, including SimpleTES with Gemini 3 Pro, but it stays behind the Hyra agent. Muse, continuing Gemma's programs, removes 475 more SWAPs, and Gemma, taking Muse's programs back, removes another 87, all on Q20. With these three rounds, MOMO beats the state of the art by 21 SWAPs, at 86,102. Muse drives most of this gain by weakening the decay penalty in Gemma's programs (Appendix D).

5 Related Work

Tree-based search. Search over a tree of states is a long-studied idea in RL (Silver et al., 2017; Schrittwieser et al., 2020). With LLMs, FunSearch and AlphaEvolve search over programs at inference time (Romera-Paredes et al., 2024; Novikov et al., 2025), OpenEvolve open-sources this approach (Sharma, 2025), and SimpleTES scales it up (Ye et al., 2026). ThetaEvolve and TTT-Discover instead update weights during search, with mean-based (GRPO) and tail-based advantages (Wang et al., 2025; Yuksekgonul et al., 2026). MOMO is also a weight-space method and can use any advantage function inside Refine (Section 2).

Model ensembling. Combining models is a long-studied idea in machine learning, from classic ensembles to Snapshot Ensembles, which collects many models from one training run (Dietterich, 2000; Lakshminarayanan et al., 2017; Huang et al., 2017). When the goal is one best solution, variance matters: maximum-entropy RL keeps policies stochastic (Haarnoja et al., 2018) and Bootstrapped DQN explores with an ensemble (Osband et al., 2016), whereas RL on language models quickly collapses policy entropy (Cui et al., 2025). MOMO is an ensembling method for test-time RLVR.

6 Discussion and Conclusion

MOMO lets trained models continue one another's best solutions: the state buffer shares what has been found, while each model's weights retain what it has learned. With MOMO, we find state-of-the-art solutions on Erdős minimum overlap, circuit placement, and qubit routing, and consistent improvements over single-model training. Given its simplicity, MOMO extends naturally to larger families of models and more rounds, a direct way to test how far model diversity can push discovery.

Ethics statement

We are not aware of ethical concerns raised by this work. It uses public mathematical and hardware optimization benchmarks and involves no human subjects or personal data.

Reproducibility statement

Our code is open source and available in an anonymous repository. Appendix A details all hyperparameters and training configurations, Appendix A.4 includes every prompt the models see, and Appendix C lists the best generated programs.

AI use statement

Generative models were used to execute the experiments and to refine the plots in the paper.

References

ArchGen AI. ArchGen: Macro placement challenge 2026. https://github.com/archgen-ai/Macro-Placement-challenge-2026, 2026.

Federico Bianchi, Yongchan Kwon, Aneesh Pappu, and James Zou. Harnessing the collective intelligence of ai agents in the wild for new discoveries. *arXiv preprint arXiv:2606.10402*, 2026.

Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 1995.

Chung-Kuan Cheng, Andrew B. Kahng, Sayak Kundu, Yucheng Wang, and Zhiang Wang. An updated assessment of reinforcement learning for macro placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2026.

James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 1965.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, et al. The entropy mechanism of reinforcement learning for reasoning language models. *arXiv preprint arXiv:2505.22617*, 2025.

Thomas G. Dietterich. Ensemble methods in machine learning. In *MCS*, 2000.

Paul Erdős. Some remarks on number theory. *Riveon Lematematika*, 1955.

Gemma Team. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

Rishi Gottumukkala. AbuPlace: GPU macro placer. https://github.com/rishivg/AbuPlace, 2026.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. Snapshot ensembles: Train 1, get M for free. In *ICLR*, 2017.

Hyra Team. Hyra: Hunyuan research agent. https://hy.tencent.com/research/hyra, 2026.

Yubin Kim, Ken Gu, Chanwoo Park, et al. Capable language models can outgrow the benefits of collaboration. *Nature Machine Intelligence*, 2026.

Dieter Kraft. A software package for sequential quadratic programming. Technical report, DFVLR, Oberpfaffenhofen, Germany, 1988.

Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *NeurIPS*, 2017.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

Yoonsang Lee, Howard Yen, Xi Ye, and Danqi Chen. Agentic aggregation for parallel scaling of long-horizon agentic tasks. In *COLM*, 2026.

Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for NISQ-era quantum devices. In *ASPLOS*, 2019.

Wenhan Ma, Hailin Zhang, Liang Zhao, Yifan Song, Yudong Wang, Zhifang Sui, and Fuli Luo. Stabilizing MoE reinforcement learning by aligning training and inference routers. *arXiv preprint arXiv:2510.11370*, 2025.

Lovish Madaan, Aniket Didolkar, Suchin Gururangan, John Quan, Ruan Silva, Ruslan Salakhutdinov, Manzil Zaheer, Sanjeev Arora, and Anirudh Goyal. Rethinking thinking tokens: LLMs as improvement operators. *arXiv preprint arXiv:2510.01123*, 2025.

Meta. Introducing muse glimmer: An open agentic model that runs on your device. Meta AI Research Blog, August 2026. URL https://research.meta.ai/blog/introducing-muse-glimmer-open-agentic-model.

Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. A graph placement methodology for fast chip design. *Nature*, 2021.

Yurii Nesterov. Smooth minimization of non-smooth functions. *Mathematical Programming*, 2005.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. On the Navier–Stokes millennium prize problem. https://openai.com/index/navier-stokes-solution/, 2026.

Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. In *NeurIPS*, 2016.

Partcl and Hudson River Trading. Partcl/HRT macro placement challenge. https://github.com/partcleda/macro-place-challenge-2026, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Shrinivas Ramasubramanian, Daman Arora, Fahim Tajwar, Guanning Zeng, Qingyang Wu, Zhongzhu Zhou, Chenfeng Xu, Haiwen Feng, Yuda Song, Aarti Singh, Ruslan Salakhutdinov, J. Andrew Bagnell, Jeff Schneider, and Andrea Zanette. Tail-likelihood reinforcement learning. *arXiv preprint arXiv:2609.02987*, 2026.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 2024.

Christopher D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 2011.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588: 604–609, 2020.

John Schulman. LoRA without regret. https://thinkingmachines.ai/blog/lora/, 2025. September 29.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Asankhaya Sharma. OpenEvolve: an open-source evolutionary coding agent. https://github.com/algorithmicsuperintelligence/openevolve, 2025.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 2017.

Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 1997.

TILOS AI Institute. MacroPlacement: Macro placement benchmarks, evaluators, and reproducible results. https://github.com/TILOS-AI-Institute/MacroPlacement, 2022.

Siddarth Venkatraman, Vineet Jain, Sarthak Mittal, Vedant Shah, Johan Obando-Ceron, Yoshua Bengio, Brian R. Bartoldson, Bhavya Kailkhura, Guillaume Lajoie, Glen Berseth, Nikolay Malkin, and Moksh Jain. Recursive self-aggregation unlocks deep thinking in large language models. *arXiv preprint arXiv:2509.26626*, 2025.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, et al. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

Haotian Ye, Haowei Lin, Jingyi Tang, Yizhen Luo, Rahul Thapa, Caiyin Yang, Chang Su, Rui Yang, Ruihua Liu, Rundao Li, Zeyu Li, Pengwei Sun, Chong Gao, Dachao Ding, Guangrong He, Miaolei Zhang, Lina Sun, Wenyang Wang, Yuchen Zhong, Zhuohao Shen, Puheng Li, Pan Lu, Bianxiao Cui, Di He, Jianzhu Ma, Junfeng Li, Hexi Baoyin, Yejin Choi, Stefano Ermon, Xiaowen Chu, Tongyang Li, Yuzhi Xu, and James Zou. Structured scaling of AI discovery across diverse scientific domains. *arXiv preprint arXiv:2604.19341*, 2026.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. In *ICML*, 2026.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö. Arık. Multi-agent design: Optimizing agents with better prompts and topologies. In *ICLR*, 2026.

Henry Zou, Matthew Treinish, Kevin Hartman, Alexander Ivrii, and Jake Lishman. LightSABRE: A lightweight and enhanced SABRE algorithm. *arXiv preprint arXiv:2409.08368*, 2024.

A TRAINING SETTINGS

A.1 HYPERPARAMETERS

Table 6 lists the settings shared by every run, and Table 7 lists the step budgets of each experiment, all of which use Qwen, Gemma, and Muse for three rounds. Each step samples 512 candidates, so a round of T steps samples $512T$.

Setting	Value
Advantage function	PWC (App. A.2)
PWC weight α	0.5 (Erdős: 1)
LoRA rank	32
Learning rate, Qwen	1.5×10^{-4}
Learning rate, Gemma and Muse	4×10^{-5}
Parents per step	16
Candidates per parent	32
Candidates per step	512
Children kept per parent	2
Reward for an invalid candidate	0
State Buffer with visitation stats reset	top 48
Optimizer state at a handoff	reset

Table 6: **Settings shared by every run.**

Experiment	T_1 / T_2 / T_3
Erdős minimum overlap	12 / 8 / 5
Autocorrelation (AC1)	15 / 25 / 5
Circuit placement	15 / 10 / 5
Qubit routing	20 / 10 / 5

Table 7: **Step budgets per experiment.** T_r is the number of steps in round r.

A.2 ADVANTAGE FUNCTION CHOICE

Section 3.1 compares two advantage functions: the group-relative advantage of GRPO (Shao et al., 2024) and the tail-weighted advantage of TTT-Discover (Yuksekgonul et al., 2026). Neither was best across problems and models. GRPO helped most when a model produced many invalid programs: its negative advantage on the lower tail raised validity over training, which left more valid samples to learn from. TTT-Discover helped most once validity was already high, or when the model found a behavior it could exploit, because it concentrates credit on the best samples. When many samples were invalid, however, it did little to raise validity.

We therefore combine the two in a piecewise-centered advantage (PWC): the GRPO advantage for every sample, plus α times the TTT-Discover advantage computed on the valid samples only and centered to zero mean among them. The second term is scaled by the smallest gap between a valid and an invalid reward in the group, so with $\alpha = 0.5$ every valid sample keeps an advantage at least as large as every invalid one, while the best valid samples receive an exponentially larger bonus. We use this recipe in every experiment, with $\alpha = 0.5$ except on Erdős minimum overlap, which uses $\alpha = 1$.

A.3 HARDWARE

Our codebase (anonymous repository) is a streamlined implementation of the Tinker API that runs on TPUs. We train on v4-64 and v5p-32 slices and scale up inference with smaller v6e-8 or v4-32 reservations. When a v5p-32 slice was available, we used it for colocated training. The framework is fault tolerant, supports spot machines, and has been tested with heterogeneous workloads.

A.4 PER-EXPERIMENT DETAILS

Each experiment asks the model for a program, runs it in a sandbox, and scores what it returns. A program that returns an invalid result, raises an exception, or exceeds its time limit receives reward 0. Below, each problem's inputs, outputs, reward, and limits are followed by the prompt the model sees when it continues a parent program. Angle brackets mark text filled in per parent: its program, its scores, and its output. There is no task-specific system prompt.

Erdős minimum overlap. The program defines `run(seed, budget_s)` and returns a step function h on $[0, 2]$ as `n_points` samples, with its bound and length. The harness supplies

the parent's construction as `initial_h_values`. The evaluator checks that $0 \leq h \leq 1$ and $\int h = 1$, recomputes $C_5 = \max(\texttt{np.correlate}(h, 1-h)\,dx)$ from h, and assigns the reward $1/(C_5 + 10^{-8})$. A result of the wrong shape, with non-finite values, outside $[0, 1]$, or whose reported bound differs from the recomputed one by more than 10^{-4} receives reward 0. The program has 1,000 seconds; the process is killed at 1,105 seconds, and a killed or failed run receives reward 0.

```
You are an expert in harmonic analysis, numerical optimization, and mathematical
discovery.
Your task is to find an improved upper bound for the Erdős minimum overlap problem
constant C₅.

## Problem

Find a step function h: [0, 2] → [0, 1] that **minimizes** the overlap integral:

$$C_5 = \max_k \int h(x)(1 - h(x+k)) dx$$

**Constraints**:
1. h(x) ∈ [0, 1] for all x
2. ∫₀² h(x) dx = 1

**Discretization**: Represent h as n_points samples over [0, 2].
With dx = 2.0 / n_points:
- 0 ≤ h[i] ≤ 1 for all i
- sum(h) * dx = 1 (equivalently: sum(h) == n_points / 2 exactly)

The evaluation computes: C₅ = max(np.correlate(h, 1-h, mode="full") * dx)

Smaller sequences with less than 1k samples are preferred - they are faster to optimize
and evaluate.

**Lower C₅ values are better** - they provide tighter upper bounds on the Erdős constant.

## Budget & Resources
- **Time budget**: 1000s for your code to run
- **CPUs**: 2 available

## Rules
- Define `run(seed=42, budget_s=1000, **kwargs)` that returns `(h_values, c5_bound,
  n_points)`
- Use scipy, numpy, cvxpy[CBC,CVXOPT,GLOP,GLPK,GUROBI,MOSEK,PDLP,SCIP,XPRESS,ECOS], math
- Make all helper functions top level, no closures or lambdas
- No filesystem or network IO
- `evaluate_erdos_solution()` and `initial_h_values` (an initial construction, if
  available) are pre-imported
- Your function must complete within budget_s seconds and return the best solution found

**Lower is better**. Current record: C₅ ≤ 0.38092. Our goal is to find a construction that
shows C₅ ≤ 0.38080.

You are iteratively optimizing C₅ bound.
Here is the last code we ran:
<parent program>
Here is the C₅ bound before and after running the code above (lower is better): <before>
-> <after>
Target: 0.3808. Current gap: <gap>. Further improvements will also be generously rewarded.

--- Previous Program Output ---
<last 500 characters of the parent's output>
--- End Output ---

You may want to start your search from the current construction, which you can access
through the `initial_h_values` global variable (n=<n> samples).
You are encouraged to explore solutions that use other starting points to prevent getting
stuck in a local optimum.

Reason about how you could further improve this construction.
Ideally, try to do something different than the above algorithm. Could be using different
algorithmic ideas, adjusting your heuristics, adjusting / sweeping your hyperparemeters,
etc.
Unless you make a meaningful improvement, you will not be rewarded.
```

First autocorrelation inequality (AC1). The program defines `propose_candidate()`, which returns the non-negative heights a of a step function. The harness supplies the parent's construction as `height_sequence_1` and the evaluator `evaluate_sequence`, which computes $C_1 = 2n \max(a * a)/(\sum_j a_j)^2$. The reward is $1/(C_1 + 10^{-8})$. Negative, non-finite, or empty sequences

receive reward 0. Time limits are as for Erdős: 1,000 seconds for the program, which is killed at 1,105 seconds, with two CPUs.

````
Act as an expert software developer and inequality specialist specializing in creating
step functions with certain properties.

Your task is to generate the sequence of non-negative heights of a step function, that
minimizes the following evaluation function:

```python
import numpy as np

def evaluate_sequence(sequence: list[float]) -> float:
    """
    Evaluates a sequence of coefficients with enhanced security checks.
    Returns np.inf if the input is invalid.
    """
    # --- Security Checks ---

    # Verify that the input is a list
    if not isinstance(sequence, list):
        return np.inf

    # Reject empty lists
    if not sequence:
        return np.inf

    # Check each element in the list for validity
    for x in sequence:
        # Reject boolean types (as they are a subclass of int) and
        # any other non-integer/non-float types (like strings or complex numbers).
        if isinstance(x, bool) or not isinstance(x, (int, float)):
            return np.inf

        # Reject Not-a-Number (NaN) and infinity values.
        if np.isnan(x) or np.isinf(x):
            return np.inf

    # Convert all elements to float for consistency
    sequence = [float(x) for x in sequence]

    # Protect against negative numbers
    sequence = [max(0, x) for x in sequence]

    # Protect against numbers that are too large
    sequence = [min(1000.0, x) for x in sequence]

    n = len(sequence)
    b_sequence = np.convolve(sequence, sequence)
    max_b = max(b_sequence)
    sum_a = np.sum(sequence)

    # Protect against the case where the sum is too close to zero
    if sum_a < 0.01:
        return np.inf

    return float(2 * n * max_b / (sum_a**2))
```

A previous state of the art used the following approach. You can use it as inspiration,
but you are not required to use it, and you are encouraged to explore.
```latex
Starting from a nonnegative step function $f=(a_0,\dots,a_{n-1})$ normalized so that
$\sum_j a_j=\sqrt{2n}$, set $M=\|f*f\|_\infty$. Next compute $g_0=(b_0,\dots,b_{n-1})$ by
solving a linear program, i.e.\ maximizing $\sum_j b_j$ subject to $b_j\ge0$ and
$\|f*g_0\|_\infty\le M$; as is standard, the optimum is attained at an extreme point
determined by an active set of binding inequalities, here corresponding to important
constraints where the convolution bound $(f*g_0)(x)\le M$ is tight and limiting. Rescale
$g_0$ to match the normalization, $g=\frac{\sqrt{2n}}{\sum_j b_j}g_0$, and update
$f\leftarrow (1-t)f+t g$ for a small $t>0$. Repeating this step produces a sequence with
nonincreasing $\|f*f\|_\infty$, and the iteration is continued until it stabilizes.
```

Your task is to write a search function that searches for the best sequence of
coefficients. Your function will have 1000 seconds to run, and after that it has to have
returned the best sequence it found. If after 1000 seconds it has not returned anything,
it will be terminated with negative infinity points. All numbers in your sequence have to
be positive or zero. Larger sequences with 1000s of items often have better attack
surface, but too large sequences with 100s of thousands of items may be too slow to
search.
````

```
You may code up any search method you want, and you are allowed to call the
evaluate_sequence() function as many times as you want. You have access to it, you don't
need to code up the evaluate_sequence() function.

You are iteratively optimizing upper bound.
Here is the last code we ran:
<parent program>
Here is the upper bound before and after running the code above (lower is better):
<before> -> <after>
Target: 1.503. Current gap: <gap>. Further improvements will also be generously rewarded.

--- Previous Program Output ---
<last 500 characters of the parent's output>
--- End Output ---
Length of the construction: <n>

You may want to start your search from one of the constructions we have found so far,
which you can access through the 'height_sequence_1' global variable.
However, you are encouraged to explore solutions that use other starting points to prevent
getting stuck in a local minimum.

Reason about how you could further improve this construction.
Ideally, try to do something different than the above algorithm. Could be using different
algorithmic ideas, adjusting your heuristics, adjusting / sweeping your hyperparemeters,
etc.
Unless you make a meaningful improvement, you will not be rewarded.

Rules:
- You must define the `propose_candidate` function as this is what will be invoked.
- You can use scientific libraries like scipy, numpy,
  cvxpy[CBC,CVXOPT,GLOP,GLPK,GUROBI,MOSEK,PDLP,SCIP,XPRESS,ECOS], math.
- You can use up to 2 CPUs.
- Make all helper functions top level and have no closures from function nesting. Don't
  use any lambda functions.
- No filesystem or network IO.
- Do not import evaluate_sequence yourself. Assume it will already be imported and can be
  directly invoked.
- **Print statements**: Use `print()` to log progress, intermediate bounds, timing info,
  etc. Your output will be shown back to you.
- Include a short docstring at the top summarizing your algorithm.

Make sure to think and return the final program between ```python and ```.
```

Circuit placement. The program defines `place(problem, seed, *, time_budget_s)` and returns the centers of all macros and soft clusters. The harness calls it on each of the 17 ICCAD04 IBM designs that have macros, ibm01–ibm04 and ibm06–ibm18, with seed 42 and a 300-second search budget per design. Each `problem` holds the canvas, the block sizes and fixed flags, the nets and pins, a verified legal Xplace placement, and the three Xplace global placements that AbuPlace starts from. An injected `Evaluator` scores moves incrementally with the TILOS proxy cost. The trusted grader recomputes $c_i = W_i + 0.5D_i + 0.5G_i$ for each design, and the reward is $1/(1+\bar{c})$; any design with a macro outside the canvas, a hard-macro overlap, or a missing result makes the reward 0. Each design runs on 4 CPU cores with 4 GiB of memory and no accelerator. The program is stopped at 310 seconds, and grading has a separate 180-second limit. Imports are limited to NumPy, SciPy, NetworkX, JAX, Optax, and listed standard-library modules, with no network, filesystem, or external processes. We re-ran ArchGen and AbuPlace on every design with an NVIDIA A10 GPU on Modal, one run per design within their own one-hour search.

```
Benchmark scope: ibm17-proxy-v1. Every submission is evaluated on ALL 17 IBM cases: ibm01,
ibm02, ibm03, ibm04, ibm06, ibm07, ibm08, ibm09, ibm10, ibm11, ibm12, ibm13, ibm14, ibm15,
ibm16, ibm17, ibm18. The scientific objective is their equally weighted arithmetic mean
proxy cost, lower is better. No subset or baseline normalization. Every case must be legal
with zero hard-macro overlaps. Any invalid or missing case gives reward zero; otherwise
reward = max(1e-6, 1/(1+mean_proxy_cost)).

Design a chip macro-placement algorithm using the provided fast objective evaluator to
guide optimization. Return one complete fenced
Python module implementing:

```python
def place(problem: dict, seed: int, *, time_budget_s: float) -> dict:
    # Return all hard-macro and soft-cluster centers, in input order.
    return {"positions": positions}  # numeric array [M, 2], x then y
```
```

```
Optimize the Partcl/TILOS Tier 1 proxy:

    cost = wirelength + 0.5 * density + 0.5 * congestion

The evaluator computes these normalized components with its pinned TILOS
implementation. Lower cost is better. Across ALL 17 IBM benchmarks (ibm01-ibm04 and
ibm06-ibm18),
reward = max(1e-6, 1 / (1 + mean(cost))). Any invalid case makes the whole
submission's reward zero. A cost of 1 gives reward 0.5; cost 2 gives 1/3.
No baseline division is applied. For selected valid programs, feedback includes
per-case proxy_cost, wirelength_cost, density_cost, congestion_cost,
candidate_wall_seconds, candidate_limit_seconds, grading_seconds, and the
suite's arithmetic mean_proxy_cost, benchmark_suite and case_count. Each case has equal
weight. Every case must pass; a four-case or 16-case subset cannot earn a valid suite
reward. Candidate runtime includes imports, initialization,
JIT compilation, and completed device work; grading time is separate.
Components are normalized costs before applying the weights above. Use this
breakdown to identify which cases and components need improvement, while
optimizing the suite mean and preserving legality on every case.

Provided fast objective evaluator:
Evaluator is already imported and available in your program. Do not import or
implement it yourself. It caches layout state for efficient incremental scoring;
you design the search strategy, proposals, acceptance rules and use of the budget.

    with Evaluator(problem, problem["initial_positions"]) as ev:
        current = ev.score()

Methods:
- score(): return the current layout's score components.
- apply(block_id, xy): temporarily move one block to center xy; return scores.
- commit(): accept the pending move; returns None.
- revert(): undo the pending move; return scores for the restored layout.
- evaluate_moves(ids, positions): return a list of score dictionaries for
  independent single-block alternatives against the current layout, leaving
  it unchanged. ids has shape [K], positions has shape [K,2].
- evaluate_batch(layouts): return a list of score dictionaries for complete
  alternative layouts [K,M,2], leaving the current layout unchanged.
- rebuild(): recompute cached state to limit numerical drift; return scores.

Each score dictionary contains proxy_cost, wirelength_cost, density_cost and
congestion_cost. Lower is better. These score the objective defined above;
there is no need to implement its components yourself.
Keep one evaluator alive across your search loop to reuse its caches. Only one
move may be pending: commit or revert before another move or full-layout scoring.
score() may inspect the pending state. Batch calls run serially on CPU; moves
in evaluate_moves are independent alternatives, not a sequence of accepted moves.
Use a separate evaluator initialized at a different layout for a new search state.
Each evaluator is single-threaded; do not share it across threads.

Scores do NOT certify legality. Check complete rectangle bounds, fixed positions
and hard-macro overlaps yourself. Keep your own copy of your best legal positions;
the evaluator does not retain your best solution. Periodic automatic rebuilds
limit drift; rebuild the state corresponding to your final choice before return.
The independent final grader determines validity and reward.

NumPy and concrete JAX CPU arrays are accepted. Calls are host-side, stateful and
non-differentiable: keep them outside jax.jit, jax.grad and jax.vmap. JAX may generate
proposals before passing them to the evaluator. All helper initialization,
scoring time and native memory count against the existing per-case budget.

All coordinates and sizes are in microns. Positions are continuous rectangle
CENTERS, not grid-cell IDs. The metric grid does not restrict legal centers.
The first H = problem["num_hard"] rows are hard macros; the remaining M-H rows
are soft standard-cell clusters. You may move either kind when not fixed.
There is no automatic soft-cluster completion after your algorithm returns.

Hard constraints:
- Return exactly M finite (x, y) centers in the original order.
- Keep the entire rectangle of EVERY macro and cluster inside the canvas.
- Preserve fixed positions exactly. Ports are fixed and are not returned.
- Hard macros must have zero overlap with every other hard macro, including
  fixed hard macros. Soft-cluster overlap is permitted and affects the proxy.
- Dimensions and the supplied pin offsets are fixed. This interface optimizes
  positions only; do not return flips, rotations, or resized soft clusters.
- Initial positions are a precomputed Xplace placement with deterministic
  legalization, verified legal by the trusted grader. They are provided on
  every case. You may refine them or construct other starting layouts. Keep
  a verified legal fallback and a positive margin against float32 violations.
  Xplace precomputation is fixed and outside your candidate runtime budget.
```

```
Input schema (all arrays are NumPy arrays; all indices are zero based):
- canvas: float[2], width and height.
- initial_positions: float[M,2], the verified Xplace starting layout.
- sizes: float[M,2], width and height.
- fixed: bool[M]; num_hard: integer H.
- port_positions: float[P,2].
- net_offsets: int[K+1], compressed offsets into the endpoint arrays below.
- pin_owner: int[L]; pin_offset: float[L,2]; net_weights: float[K].
  Net k contains endpoints [net_offsets[k]:net_offsets[k+1]], driver first,
  followed by sinks. Preserve this order for routing calculations. An owner
  below M refers to a macro/cluster; owner M+j refers to port j. A pin's
  location is its owner's center plus its offset. Multi-pin nets, repeated
  owners, and non-unit net weights are preserved.
- wirelength_normalizer: positive float used by the native weighted HPWL.
- grid_shape: int[2], rows and columns used for density and congestion.
- routes_per_micron: float[2], horizontal and vertical routing capacities.
- macro_routing_allocation: float[2]; congestion_smoothing_range: integer.
- schema_version: "partcl_cpu_positions_v1".

Allowed imports: numpy, scipy, networkx, jax,
jax.numpy, jax.scipy, optax, and these standard-library modules:
math, random, time, itertools, functools, collections, heapq, bisect, array,
statistics, dataclasses, typing, enum, copy, operator, __future__. Seed your random
choices
from the supplied seed. Use time.monotonic() to manage the supplied budget.
Use JAX JIT compilation, automatic differentiation, and vectorized candidate
evaluation where they improve results within the budget. NumPy/SciPy remain
available for CPU preprocessing and final legalization. Compile outside inner
Python loops, avoid unnecessary host/device transfers, and return completed
finite positions. JIT compilation and device synchronization count against
your time budget. JAX does not compile the trusted Python grader.
No PyTorch, external placer, external compiler process, filesystem, network,
external process, dynamic Python code execution, or instance-name lookup is allowed.

Execution profile: CPU only, with 4 logical CPU cores and 4 GiB total RAM
shared by the candidate process tree. No GPU or TPU is available. JAX runs
on CPU; do not select an accelerator backend. There is no per-library memory
allowance. NumPy/SciPy BLAS and OpenMP pools are capped at four threads;
JAX also runs within the same enforced four-core CPU allocation.
JAX and jaxlib 0.10.1; Optax 0.2.8; NumPy 2.4.6;
SciPy 1.17.1; NetworkX 3.6.1; candidate Python 3.12.12.

Each case supplies 300 seconds of search time to place(). Return your best legal positions
before this budget expires. The external candidate cap is 310 seconds, including startup
and output.
Trusted CPU grading has a separate 180-second cap; each case has a 510-second
worker envelope. All 17 cases are required. Cases run in parallel as CPU slots
become available; queue waiting does not consume your per-case candidate budget.
NumPy/SciPy BLAS and OpenMP pools are capped at four threads, within the same
four-core allocation. Memory is a shared process-tree allowance, not 4 GiB
per library. Respect time_budget_s; reserve time to return your best legal
placement. Final independent grading has a separate allowance.

Runtime below the limit does not directly increase your reward. If useful,
spend available time exploring additional placements or refining promising
candidates. Return your best legal placement before the supplied time_budget_s
expires. Additional computation is optional; preserve improvements rather than
running longer for its own sake. Use time_budget_s to plan your work, not the
larger hard deadline shown in feedback.

The injected Evaluator is available in every execution. Reuse its cached state
for fast move evaluation and undo; your own surrogate objectives are also allowed.
The independent evaluator determines the actual reward. Optimize generally
across problems, without embedded solutions or benchmark-specific branches.


You may use JAX or the other allowed libraries for smooth wirelength/density
objectives, batched move evaluation, or restarts within the CPU budget. Preserve exact
hard-macro legality in
the final result; a better surrogate cost never overrides the trusted score.


You may treat congestion and density as spatial fields and apply image-processing
methods, including Gaussian smoothing, convolutions, multiscale analysis, and
spatial gradients to guide placement. Use the allowed numerical libraries for
these operations. All preprocessing and optimization count against the same
runtime and memory limits. Final placements must satisfy every legality
constraint and are scored by the trusted evaluator.

Implementation checklist (apply before returning your complete module):
```

```
- The supplied Xplace initial_positions are a legal fallback. You may replace
  the starter entirely. Reserve time to return your best legal placement.
- Keep hard-macro legalization on the CPU unless your replacement enforces
  every rectangle constraint. A penalty objective alone does not ensure legality.
- The installed segment reductions are jax.ops.segment_sum, segment_min, and
  segment_max, not jax.lax.segment_*. Set num_segments to the Python integer
  len(problem["net_weights"]) outside JIT. Precompute net IDs with NumPy.
- Inside JIT use fixed array shapes. Avoid int/float/bool conversions of tracers,
  data-dependent Python branches, and boolean indexing with dynamic lengths.
  Use jnp.where masks and precomputed integer indices where appropriate.
- Check broadcasting explicitly: positions and sizes [M,2], fixed [M],
  fixed[:,None] [M,1], pin_owner [L], pin_offset [L,2], net_weights [K].
  A pin owner can refer to a port; concatenate macro centers and port centers.
- Copy input arrays before modifying them; restore fixed rows exactly. Check
  finiteness, full rectangle bounds, and hard-hard overlaps before returning.
- Recheck bounds after converting to your returned array dtype. Rounding can
  move a center outside the canvas; use a small inward margin for movable
  boundary macros, then recheck overlaps and preserve fixed objects exactly.
- Include every helper/import you use. Return one complete Python code fence,
  with the exact place(problem, seed, *, time_budget_s) signature; no patch.

Additional starting layouts: problem["starting_layouts"] has shape [3,M,2], in the order
problem["starting_names"]: off, rudy, rudy_hv. These are AbuPlace's three Xplace
global-placement configurations, independently legalized and verified. They do not include
AbuPlace's later refinement pipeline. problem["starting_scores"] has shape [3,4], with
columns named by problem["starting_score_columns"]: proxy_cost, wirelength_cost,
density_cost, congestion_cost. All starts use the same circuit and object ordering.
initial_positions remains the original verified legal fallback. You may choose, copy, or
refine any start. Do not assume the lowest initial cost has the best potential after
optimization. Consider brief exploration of several starts, then allocate the remaining
time to promising layouts. Preserve your best legal result across all attempts.
Perturbations, swaps and coarse-to-fine moves may escape local minima; check legality and
evaluate their actual benefit. Averaging layouts does not preserve legality. There is ONE
shared 300-second search budget per case across all starts, not 300 seconds per start.
Precomputation is excluded; evaluator construction and search consume your budget. Return
only the single best legal layout you found.

Previous candidate:
'''python
<parent program>
'''
Grader feedback:
<JSON with the reward, message, and per-case scores>
Improve this candidate.
```

Qubit routing. The program is a Python module with one string, `RUST_CODE`, that replaces the policy block of the SimpleTES routing engine. The policy implements `choose_best_initial_layout`, an injective map from logical to physical qubits, and `choose_best_swap`, which returns a coupling edge to swap. The engine compiles it and routes all 72 circuits, keeping for each circuit the fewest SWAPs over 20 initial layouts and 20 routing trials. A trusted verifier checks every routed circuit against the processor's coupling graph and the original operations. With C the weighted count of added CNOTs (three per SWAP; weights 0.2, 0.4, and 0.4 for Q20, Willow, and Heron) and B that of SABRE, the reward is $B/(B + C)$. A compilation error, one failed or incorrect circuit, or a timeout makes the reward 0. Compilation, routing, and verification share one 1,900-second deadline; four case workers run at a time, each with 2 CPUs and 4 GiB of memory. The policy may use only the supplied random number generator, the Rust standard library, and the scaffold's pinned crates, with no network, filesystem, or unsafe code.

```
Grader feedback includes a compact per-case SWAP table with columns named in case_columns,
plus per-topology totals. baseline_swaps is the fixed SABRE reference; delta_swaps =
candidate minus baseline, so negative is better. Case IDs identify diagnostics only; do
not hardcode cases. The single aggregate reward remains the optimization objective. Failed
execution may stop before case counts are available; use the reported error in that
situation.

Design a routing policy that reduces added SWAPs while preserving every required circuit
operation and dependency. The trusted SimpleTES engine handles parsing, mapping updates,
routing and output construction. Replace only the existing policy's EVOLVE-BLOCK using the
exact Rust API below.

Return one fenced Python module containing exactly one literal assignment:
RUST_CODE = r'''<complete replacement EVOLVE-BLOCK contents>'''

Include all policy definitions/helpers. Do not return marker lines or a patch, repeat
```

````
scaffold definitions, change engine code, or add crates. Implement
choose_best_initial_layout with an injective logical-to-physical mapping and
choose_best_swap returning a legal coupling edge according to the supplied interface.

Every case must route correctly. The trusted grader independently checks topology,
original operations/dependencies, and SWAP counts. B=sum(weight*baseline_added_CNOTs)>0,
C=sum(weight*candidate_added_CNOTs), one inserted SWAP costs three CNOTs.
Reward=max(1e-6,B/(B+C)); reference reward=0.5; invalid code, one failed case, or timeout
receives zero. Use only supplied RNG, computational standard library and pinned scaffold
crates. No filesystem/network access, external processes, unsafe code, include macros, or
case-name hardcoding.

Resources: four CPU cores, 8 GiB RAM, 1800 seconds total: up to 900 seconds for
changed-policy compilation and routing all cases with 20 layout/20 routing trials, then up
to 900 seconds for independent verification. No accelerator. Installed dependencies are
prewarmed outside timing.

Implementation checklist:
- Preserve the seed policy's exact hook signatures, return types, and required
  definitions. Use only methods and fields shown in the read-only API below;
  do not invent graph, layout, or RNG methods. Respect Rust borrow/lifetime rules.
- A swap must be a physical coupling edge. Initial layouts must map all logical
  qubits injectively into the available physical qubits. Preserve progress and
  disconnected-component handling from the seed unless replacing them correctly.
- Return the ENTIRE replacement policy, including its helpers, inside RUST_CODE.
  The outer fenced language must be python. RUST_CODE must be one literal string:
  no imports, function calls, string concatenation, extra assignments, or Rust
  code outside that literal. Do not copy the fixed scaffold into the string.
- Invalid bootstrap programs may be returned with compiler/verifier feedback.
  Repair the reported cause first, preserving the complete interface. Valid
  feedback reports weighted added-CNOT cost, SWAP totals, and improvement.

Fixed Rust scaffold (read-only):
```rust
<fixed scaffold: 550 lines of types and read-only context accessors, ending with>
pub trait Policy: Clone {
    fn choose_best_initial_layout(
        &mut self,
        ctx: &InitialLayoutContext<'_>,
        rng: &mut RngState,
    ) -> Result<Vec<usize>, RouterError>;

    fn choose_best_swap(
        &mut self,
        ctx: &SwapSelectionContext<'_>,
        rng: &mut RngState,
    ) -> Option<(usize, usize)>;
}
```

Previous candidate:
```python
<parent program>
```
Grader feedback:
<JSON with the reward, message, and per-case SWAPs>
Improve this candidate.
````

B FULL SEARCHES

Figure 5 shows the Erdős search step by step. Figures 10–12 show every run of the AC1, placement, and routing searches.

What each model changed on AC1. Gemma is the strongest single model; handing its buffer to Muse and then to Qwen lowers the bound in each round, and each model contributes a different technique. (1) Gemma, alone, built the optimizer that every later program keeps: Adam on a log-sum-exp smoothing of the peak of the self-convolution, with FFT-based gradients, a smoothing temperature lowered over the run, and a projection that keeps the heights non-negative with a fixed sum. It reached 1.504545, building each sequence from scratch with at most 5,000 points. (2) Muse, given Gemma's buffer, kept this optimizer and added re-gridding: it interpolates the parent's best sequence onto a finer grid and continues optimizing. A finer grid gives finer control over the shape of the self-convolution, which lowers the bound the optimizer can reach. Muse moved from 1,800 points to 4,096, 8,192, and 16,384, reaching 1.503496. Qwen, given the same buffer, never re-gridded; all of its programs kept 1,800 points, and it stalled at 1.503713. (3) On Muse's 16,384-point sequences, Qwen polished more aggressively than Gemma, with parallel optimization runs, larger perturbations, and restarts after stalls, and reached 1.503463. Gemma, taking the same buffer back, switched to very small learning rates and reached 1.503475. Most of the gain came from Muse's re-gridding; further polishing on a fixed grid gave much less.

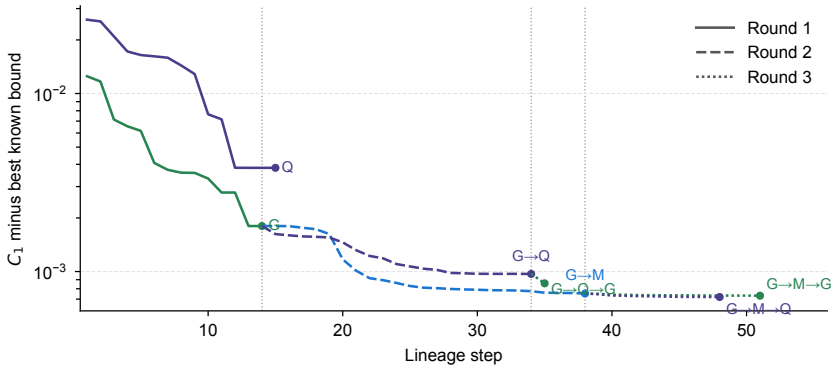


Figure 10: **Full AC1 search.** Each line is one run, colored by the model taking the step and styled by round; labels give the lineage, donor first. Dotted vertical lines mark branch points, where a run's buffer is handed to other models. The y-axis is the gap to the best known bound, 1.50274365 (EinsteinArena; Bianchi et al., 2026), on a log scale. Round 3 is shown up to submission.

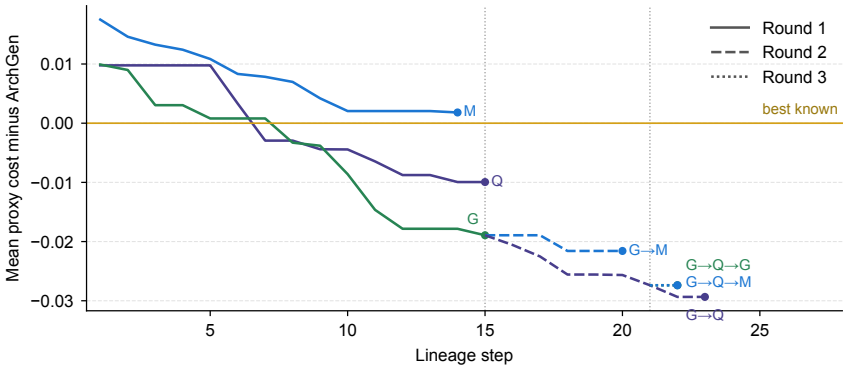


Figure 11: **Full placement search.** Mean proxy cost over the 17 IBM designs, relative to ArchGen's leaderboard score of 0.9507 (ArchGen AI, 2026); below zero is better. Lines, colors, and branch markers as in Figure 10. Round 3 is shown up to submission; G→Q→G and G→Q→M overlap.

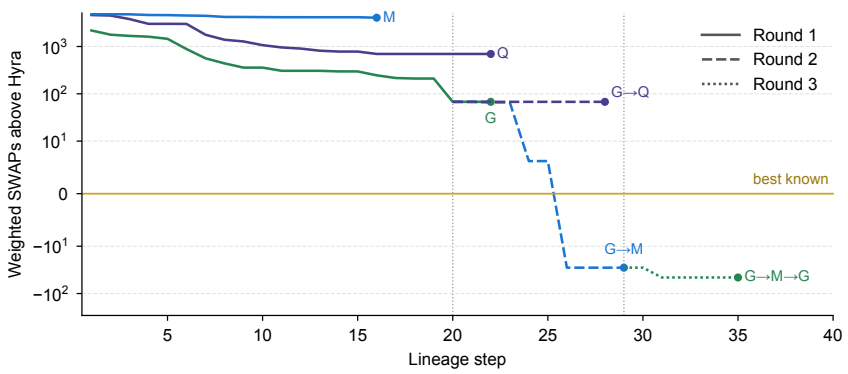


Figure 12: **Full routing search.** Best weighted SWAP count S at every lineage step, as the gap to Hyra's $S = 31{,}924.6$ on a symmetric log scale (linear between -10 and 10); lower is better, and below zero beats Hyra. Line styles, labels, and markers as in Figure 11.

C BEST GENERATED CODE

The programs below are the best found on each problem, reproduced as the models wrote them.

C.1 ERDŐS MINIMUM OVERLAP: QWEN → MUSE → QWEN

```
# Best Round 3: Qwen -> Muse -> Qwen. C5 = 0.3808566917, 801 pieces.
import numpy as np
import time
from scipy.optimize import minimize

def stable_smooth_max(values, beta):
    """
    Smooth approximation of the max function.
    """
    c_max = np.max(values)
    if beta < 1e-12:
        return float(c_max)
    shifted = values - c_max
    exp_vals = np.exp(np.clip(beta * shifted, -500, 500))
    return float(c_max + np.log(np.sum(exp_vals)) / beta)

def compute_raw_overlap(h):
    """
    Compute the raw maximum overlap integral value.
    """
    corrs = np.correlate(h, 1.0 - h, mode='full')
    return np.max(corrs)

def project_sum(h, target_sum):
    """
    Project h onto the constraint sum(h) = target_sum and box constraints [0, 1].
    """
    h = np.array(h, dtype=float).copy()
    h = np.clip(h, 0.0, 1.0)
    s = np.sum(h)

    for _ in range(100):
        scale = target_sum / s if s != 0 else 1.0
        h = h * scale
        h = np.clip(h, 0.0, 1.0)
        s = np.sum(h)
        if np.abs(s - target_sum) < 1e-8:
            break

    return np.clip(h, 0.0, 1.0)

def run(seed=42, budget_s=1000, **kwargs):
    np.random.seed(seed)
    start_time = time.time()

    n = 801
    dx = 2.0 / n
    target_sum = n / 2.0

    h0 = None
```

```
    try:
        init = np.asarray(initial_h_values, dtype=float)
        if len(init) == n:
            h0 = init.copy()
        else:
            h0 = np.interp(
                np.linspace(0, 1, n),
                np.linspace(0, 1, len(init)),
                init,
            )
    except Exception:
        h0 = np.full(n, 0.5)
        h0 += 1e-4 * np.random.normal(0, 1, n)

    h0 = np.clip(h0, 0.0, 1.0)
    h0 = project_sum(h0, target_sum)

    best_h = h0.copy()
    best_val = compute_raw_overlap(best_h) * dx
    print(f"Initial C5: {best_val:.7f}")

    bounds = [(0.0, 1.0) for _ in range(n)]
    cons = [{'type': 'eq', 'fun': lambda x, ts=target_sum: np.sum(x) - ts}]

    betas = [1e6]

    for i, beta in enumerate(betas):
        elapsed = time.time() - start_time
        if elapsed > budget_s * 0.85:
            print("Time limit approaching, breaking optimization loop.")
            break

        opts = {'maxiter': 300, 'ftol': 1e-12, 'disp': False}
        try:
            res = minimize(
                lambda x: stable_smooth_max(np.correlate(x, 1.0 - x, mode='full'), beta),
                h0,
                method='SLSQP',
                bounds=bounds,
                constraints=cons,
                options=opts,
            )
            h0 = res.x
            h0 = project_sum(h0, target_sum)

            val = compute_raw_overlap(h0) * dx
            if val < best_val:
                best_val = val
                best_h = h0.copy()

            print(f"Iteration {i+1}, Beta {beta}: C5 = {val:.7f}, Time elapsed: {elapsed:.2f}s")

        except Exception as e:
            print(f"Optimization error at beta {beta}: {e}")

    elapsed = time.time() - start_time
    if elapsed < budget_s * 0.95:
        opts = {'maxiter': 200, 'ftol': 1e-15, 'disp': False}
        try:
            res = minimize(
                lambda x: stable_smooth_max(np.correlate(x, 1.0 - x, mode='full'), 1e7),
                best_h,
                method='SLSQP',
                bounds=bounds,
                constraints=cons,
                options=opts,
            )
            h0 = res.x
            h0 = project_sum(h0, target_sum)
            val = compute_raw_overlap(h0) * dx
            if val < best_val:
                best_val = val
                best_h = h0.copy()
            print(f"Final Polish: C5 = {val:.7f}")
        except Exception as e:
            print(f"Final polish error: {e}")

    elapsed = time.time() - start_time
    best_h = project_sum(best_h, target_sum)
    c5_bound = compute_raw_overlap(best_h) * dx

    print(f"Final Result: C5 = {c5_bound:.7f}, n = {n}, time = {elapsed:.2f}s")

    return best_h.astype(float), float(c5_bound), int(n)
```

C.2 AUTOCORRELATION (AC1): GEMMA → MUSE → QWEN

```
"""
Complextity Reduction and Improved Local Search.
This function optimizes a step function sequence by minimizing the normalized maximum convolution.
It uses a warm start from `height_sequence_1` and employs a multi-trajectory optimization
with adaptive noise injection and momentum-based gradient descent (Adam).
Key improvements over previous attempts:
1. Adjusted hyperparameters (higher learning rate, different rho schedule) to better escape local
     minima.
2. Increased restart exploration.
3. Focus on effective use of 1000s budget.
"""
import numpy as np
import time

def project_simplex(v, s):
    """Project vector v onto simplex with sum s."""
    n = len(v)
    if n == 0:
        return v
    u = np.sort(v)[::-1]
    cssv = np.cumsum(u)
    div = np.arange(1, n + 1)
    boundaries = (cssv - s) / div
    mask = u > boundaries
    rho_indices = np.where(mask)[0]

    if len(rho_indices) == 0:
        v_pos = np.maximum(0.0, v)
        sp = np.sum(v_pos)
        if sp < 1e-12:
            return np.ones(n) * (s / n)
        return v_pos * (s / sp)

    idx = rho_indices[-1]
    theta = (cssv[idx] - s) / (idx + 1)
    return np.maximum(v - theta, 0.0)

def optimize_trajectory(f_curr, budget_s, sum_target, seed, config=None, rank=0):
    """
    Single optimization trajectory with configurable hyperparameters.
    Uses Adam optimizer with adaptive rho and noise injection.
    """
    rng = np.random.default_rng(seed)
    if config is None:
        config = {
            'rho_start': 0.4, 'rho_end': 1e-5,
            'lr_start': 0.004, 'lr_end': 1e-7,
            'beta1': 0.995, 'beta2': 0.9999,
            'streak_limit': 100, # Increased limit to allow longer smoothing
            'evaluate_interval': 500
        }

    rho_start, rho_end = config['rho_start'], config['rho_end']
    lr_start, lr_end = config['lr_start'], config['lr_end']
    beta1, beta2 = config['beta1'], config['beta2']
    streak_limit = config['streak_limit']
    evaluate_interval = config['evaluate_interval']

    f = np.array(f_curr, dtype=np.float64)
    n = len(f)
    f = project_simplex(f, sum_target)

    m = np.zeros(n)
    v = np.zeros(n)
    # Use a slightly larger FFT size to handle circular convolution aliasing if any
    # though standard linear conv is implied by problem.
    # For correlation gradient, we use zero padding FFT.
    n_fft = 1 << (n).bit_length()
    # Ensure at least n, but for correlation of length n with itself, size n is enough if handled
     carefully
    # But standard practice: n_fft >= 2n-1 for convolution. Here we do correlation.
    # Correlation of f with w (length 2n-1) projected to n terms.
    # The FFT size in the provided solution was 2*n-1 bit length. Let's stick to working logic.
    n_fft = 1 << (2 * n - 1).bit_length()

    best_f = f.copy()

    try:
        best_score = evaluate_sequence(f.tolist())
    except:
        best_score = 1e20

    start = time.time()
```

```
    it = 0
    score_no_change = 0

    # Pre-allocate buffers to reduce overhead
    conv_full = np.empty(n_fft * 2, dtype=np.float64) # Just in case, irfft returns 2n FFT pts
    # Actually n_fft is power of 2 covering 2n-1. irfft returns n_fft // 2 + 1 real components?
    # No, rfft input size n_fft -> n_fft/2+1 outputs. irfft(n_fft) -> n_fft.
    # We only need first n elements for the gradient of f.

    while time.time() - start < budget_s:
        elapsed = time.time() - start
        prog = min(1.0, elapsed / budget_s) if budget_s > 0 else 1.0

        # Linear rho decay for smoother exploration in early phase
        rho = rho_start * (1.0 - prog ** 0.5) + rho_end * (prog ** 0.5)

        # FFT-based gradient computation
        f_fft = np.fft.rfft(f, n_fft)

        # Compute convolution f*f to find maxima
        # Using pad 0
        # f*f is real symmetric if f real?
        # conv_full corresponds to linear convolution values.
        # irfft(f_fft * f_fft) gives convolution result.
        conv_temp = f_fft * f_fft
        # We need the full convolution for the weights w.
        # Note: irfft of size M returns M points. We pass n_fft.
        # The valid linear convolution range is 0 to 2n-2.
        # With n_fft >= 2n-1, we are safe from aliasing.

        # Using smaller size for gradient calculation? No, need weights w based on conv.
        # Optimization: We only need to know w to compute grad.
        # However, conv_full is size n_fft.
        # We slice it to 2*n-1 to match linear conv.

        irr = np.fft.irfft(conv_temp, n_fft)
        # Valid part is first 2n-1 elements
        # Wait, irfft returns n_fft points (or n_fft//2 + 1 depending on input type).
        # With rfft(irfft(fft)) -> identity.
        # n_fft is large enough.

        conv_full_vec = irr[:2 * n - 1]
        c_max = np.max(conv_full_vec)

        # Compute weights
        safe_rho = max(rho, 1e-8)
        diff = (conv_full_vec - c_max) / safe_rho
        diff = np.clip(diff, -50.0, 0.0)

        e = np.exp(diff)
        # Avoid division by zero
        w_sum = np.sum(e) + 1e-12
        w = e / w_sum

        # Gradient calculation
        # We need gradient of sum_k w_k (f*f)_k
        # Gradient is 2 * (w * f) coronary? No, 2 * correlation(w, f).
        # Using FFT for correlation: fft(w) * conj(fft(f))

        # w has length 2n-1. We need to pad it or handle carefully.
        # Correlation (w, f) -> indices i corresponds to sum w_k f_{k-i}.
        # If w is at k, and f at k-i.
        # We can treat this as convolution if we reverse w? No.
        # Let's rely on the previous logic: grad is correlation.
        # For FFT, we need w and f to be of same length.
        # Pad w with zeros to n_fft?
        # w (length 2n-1) -> padded to n_fft.
        # But we only need first n components of correlation.

        w_padded = np.zeros(n_fft, dtype=np.float64)
        w_padded[:2*n-1] = w

        w_fft = np.fft.rfft(w_padded, n_fft)

        # Gradient
        # grad_full corresponds to correlation.
        grad_full = np.fft.irfft(w_fft * np.conj(f_fft), n_fft)
        # Take first n elements corresponding to f
        grad = 2.0 * grad_full.real[:n]

        # Center gradient to maintain sum
        grad -= np.mean(grad)
        # Normalize
        std_grad = np.std(grad) + 1e-8
        grad /= std_grad
```

```
        it += 1
        # Adam update
        m = beta1 * m + (1 - beta1) * grad
        v = beta2 * v + (1 - beta2) * (grad * grad)
        m_hat = m / (1 - beta1 ** it)
        v_hat = v / (1 - beta2 ** it)

        # Cosine decay learning rate
        lr = lr_end + 0.5 * (lr_start - lr_end) * (1 + np.cos(np.pi * prog))

        # Update
        f_new = f - lr * m_hat / (np.sqrt(v_hat) + 1e-8)
        f = project_simplex(f_new, sum_target)

        # Periodic evaluation
        if it % evaluate_interval == 0:
            try:
                sc_exact = evaluate_sequence(f.tolist())
            except:
                sc_exact = 1e20

            if sc_exact < best_score - 1e-10:
                best_score = sc_exact
                best_f = f.copy()
                score_no_change = 0
            else:
                score_no_change += 1

        # Adaptive noise injection
        if score_no_change > streak_limit:
            # Aggressive restart
            noise_scale = 0.05  # Fixed reasonably high scale

            noise = rng.normal(scale=noise_scale, size=n)
            # Perturb best_f to escape basin
            f = best_f * (1.0 + noise)
            f = project_simplex(f, sum_target)

            # Reset momentum and counter
            m = np.zeros(n)
            v = np.zeros(n)
            score_no_change = 0

    return best_f, best_score

def propose_candidate():
    """
    Main candidate proposal function using multi-trajectory optimization.
    """
    start_total = time.time()
    total_budget = 980.0

    # Load warm start
    try:
        f_warm = np.array(height_sequence_1, dtype=np.float64)
        n = len(f_warm)
    except NameError:
        # Fallback if height_sequence_1 not defined (though prompt says it is)
        n = 16384
        rng = np.random.default_rng(0)
        f_warm = np.ones(n, dtype=np.float64)
        # Try to generate a decent random start if needed, but better to rely on warm
        # For safety, just ones.

    best_f = f_warm.copy()
    try:
        best_score = evaluate_sequence(best_f.tolist())
    except:
        best_score = 1e20

    print('Initial best score {:.6f}'.format(best_score))
    sum_target = float(np.sum(best_f))

    # Define optimization trajectories with varied hyperparameters for exploration
    # We run them sequentially.
    trajectories = [
        {'seed': 1234, 'rho_start': 0.4, 'lr_start': 0.004, 'beta1': 0.995, 'streak_limit': 100},
        {'seed': 5678, 'rho_start': 0.2, 'lr_start': 0.003, 'beta1': 0.99, 'streak_limit': 80},
        {'seed': 9012, 'rho_start': 0.5, 'lr_start': 0.005, 'beta1': 0.995, 'streak_limit': 100},
    ]

    # Allocate time
    budget_per_trajectory = total_budget / len(trajectories)
    print('Running {} optimization trajectories, budget {:.1f}s each'.format(
```

```
        len(trajectories), budget_per_trajectory))

    for rank in range(len(trajectories)):
        config = {
            'rho_start': trajectories[rank]['rho_start'],
            'rho_end': 1e-5,
            'lr_start': trajectories[rank]['lr_start'],
            'lr_end': 1e-8,
            'beta1': trajectories[rank]['beta1'],
            'beta2': 0.9999,
            'streak_limit': trajectories[rank]['streak_limit'],
            'evaluate_interval': 500
        }

        seed = trajectories[rank]['seed']
        elapsed_pre = time.time() - start_total
        remaining = total_budget - elapsed_pre

        actual_budget = min(budget_per_trajectory, remaining - 10.0) # Keep buffer

        print('Trajectory {} (seed={}, budget={:.1f}s), elapsed={:.1f}s'.format(
            rank, seed, actual_budget, elapsed_pre))

        # Perturb best_f
        rng = np.random.default_rng(seed + rank * 100)
        perturbation_scale = 0.04
        f_init = best_f * (1.0 + rng.normal(scale=perturbation_scale, size=len(best_f)))
        f_init = project_simplex(f_init, sum_target)

        f_opt, sc_opt = optimize_trajectory(f_init, actual_budget, sum_target, seed, config, rank)

        # Update global best
        try:
            sc_final = evaluate_sequence(f_opt.tolist())
            if sc_final < best_score - 1e-10:
                best_score = sc_final
                best_f = f_opt.copy()
                print('  Improved to {:.6f} at {:.1f}s'.format(sc_opt, time.time() - start_total))
            else:
                # If slightly worse but we are close?
                # No, we want strict improvement
                pass
        except:
            pass

    # Final verification
    try:
        final_score = evaluate_sequence(best_f.tolist())
        print('Final score: {:.6f} at {:.1f}s'.format(final_score, time.time() - start_total))
    except:
        final_score = best_score

    # Small refinement step if time permits
    refine_budget = 50.0
    if time.time() - start_total < total_budget - refine_budget:
        print('Running refinement...')
        # Refine with conservative parameters
        refine_config = {
            'rho_start': 0.1, 'rho_end': 1e-5,
            'lr_start': 0.001, 'lr_end': 1e-8, 'beta1': 0.995,
            'beta2': 0.9999, 'streak_limit': 50, 'evaluate_interval': 200
        }
        # Fresh seed
        f_refine, sc_refine = optimize_trajectory(
            best_f, refine_budget, sum_target, 7890,
            config=refine_config, rank=0
        )

        try:
            sc_refine_final = evaluate_sequence(f_refine.tolist())
            if sc_refine_final < best_score - 1e-10:
                best_score = sc_refine_final
                best_f = f_refine.copy()
                print('  Refinement improved to {:.6f}'.format(sc_refine_final))
        except:
            pass

    print('Final result score={:.6f} length={}'.format(final_score, len(best_f)))
    return best_f.tolist()

# Helper function to calculate score if needed (though we use evaluate_sequence)
# No need to add more global functions.
```

C.3 CIRCUIT PLACEMENT: GEMMA → QWEN

```python
import time
import numpy as np

def place(problem: dict, seed: int, *, time_budget_s: float) -> dict:
    """
    Optimizes chip macro-placement to minimize proxy cost:
    cost = wirelength + 0.5 * density + 0.5 * congestion.

    Strategy:
    1. Select best starting layout
    2. Simulated Annealing with congestion-aware moves
    3. Strict Legalization enforcement for hard macros
    """
    rng = np.random.default_rng(seed)
    start_time = time.monotonic()
    deadline = start_time + time_budget_s - 4.0

    # Problem data
    canvas = np.asarray(problem['canvas'], dtype=np.float64)
    sizes = np.asarray(problem['sizes'], dtype=np.float64)
    half_sizes = sizes / 2.0
    fixed = np.asarray(problem['fixed'], dtype=bool)
    num_hard = int(problem['num_hard'])
    M = len(problem['initial_positions'])

    net_offsets = np.asarray(problem['net_offsets'], dtype=np.int32)
    pin_owner = np.asarray(problem['pin_owner'], dtype=np.int32)
    pin_offset = np.asarray(problem['pin_offset'], dtype=np.float32)
    net_weights = np.asarray(problem['net_weights'], dtype=np.float32)
    port_positions = np.asarray(problem['port_positions'], dtype=np.float64)

    # Precompute connectivity
    block_to_nets = [[] for _ in range(M)]
    for net_id in range(len(net_weights)):
        start, end = net_offsets[net_id], net_offsets[net_id+1]
        for pin_idx in range(start, end):
            owner = pin_owner[pin_idx]
            if owner < M:
                block_to_nets[owner].append(net_id)

    # Cost function
    def get_cost(s, stage='balanced'):
        if stage == 'congestion':
            return 0.1 * s['wirelength_cost'] + 0.1 * s['density_cost'] + 1.8 * s['congestion_cost']
        else:
            return s['proxy_cost']

    # Explore starting layouts
    start_candidates = [problem['initial_positions']]
    if 'starting_layouts' in problem:
        start_candidates.extend(problem['starting_layouts'])

    best_global_pos = np.array(problem['initial_positions'], dtype=np.float64)
    best_global_proxy = float('inf')

    explore_budget = max(8.0, 10.0 * len(start_candidates))
    explore_start = time.monotonic()

    start_perf = []
    for s_pos_raw in start_candidates:
        if time.monotonic() - explore_start > explore_budget:
            break

        pos = np.array(s_pos_raw, dtype=np.float64)
        movable = np.flatnonzero(~fixed)
        if len(movable) == 0:
            # If no movable, layout is fixed. Use initial score.
            with Evaluator(problem, pos) as ev:
                local_best_proxy = ev.score()['proxy_cost']
            start_perf.append((local_best_proxy, pos))
            continue

        with Evaluator(problem, pos) as ev:
            scores = ev.score()
            local_best_proxy = scores['proxy_cost']
            local_best_pos = pos.copy()

            t_start = get_cost(scores) * 0.03
            s_time = time.monotonic()
            while time.monotonic() - s_time < 6.0:
                sample_ids = rng.choice(movable, size=min(25, len(movable)), replace=False)
                v_ids, v_pos = [], []
                for bid in sample_ids:
```

```
                    xy = pos[bid] + rng.normal(0, canvas.max()*0.015, size=2)
                    if np.any(xy - half_sizes[bid] < 0) or np.any(xy + half_sizes[bid] > canvas):
                        continue
                    if bid < num_hard:
                        diff = np.abs(pos[:num_hard] - xy)
                        overlap_mask = np.all(diff < (half_sizes[:num_hard] + half_sizes[bid]), axis
=1)
                        if np.any(overlap_mask & (np.arange(num_hard) != bid)):
                            continue
                    v_ids.append(bid)
                    v_pos.append(xy)

                if not v_ids:
                    continue
                res = ev.evaluate_moves(np.array(v_ids), np.array(v_pos, dtype=np.float32))

                best_idx = -1
                min_delta = float('inf')
                curr_c = get_cost(scores)
                for i, r in enumerate(res):
                    delta = get_cost(r) - curr_c
                    if delta < min_delta:
                        min_delta, best_idx = delta, i

                if best_idx != -1 and (min_delta < 0 or rng.random() < np.exp(-min_delta/t_start)):
                    tid, txy = v_ids[best_idx], v_pos[best_idx]
                    ev.apply(tid, txy)
                    ev.commit()
                    pos[tid] = txy
                    scores = ev.score()
                    if scores['proxy_cost'] < local_best_proxy:
                        local_best_proxy = scores['proxy_cost']
                        local_best_pos = pos.copy()

        start_perf.append((local_best_proxy, local_best_pos))

if start_perf:
    best_idx = np.argmin([p[0] for p in start_perf])
    best_global_proxy, best_global_pos = start_perf[best_idx]

# Intensive search on best start
pos = best_global_pos.copy()
movable = np.flatnonzero(~fixed)
if len(movable) == 0:
    return {"positions": pos}

with Evaluator(problem, pos) as ev:
    scores = ev.score()

    t_start = scores['proxy_cost'] * 0.1
    t_end = t_start * 0.001

    iter_count = 0
    search_start = time.monotonic()
    while time.monotonic() < deadline:
        iter_count += 1
        elapsed = time.monotonic() - search_start
        progress = min(1.0, elapsed / (deadline - search_start))

        # Stage proportions
        if progress < 0.15:
            stage = 'balanced'
        elif progress < 0.60:
            stage = 'congestion'
        else:
            stage = 'refine'

        temp = t_start * (1.0 - progress)**1.5 + t_end
        radius = canvas.max() * (0.1 * (1.0 - progress**0.5) + 0.001)

        B = 30
        sample_ids = rng.choice(movable, size=min(B, len(movable)), replace=False)
        v_ids, v_pos = [], []

        for bid in sample_ids:
            m_type = rng.random()
            if m_type < 0.5:
                xy = pos[bid] + rng.normal(0, radius * 0.5, size=2)
            elif m_type < 0.75:
                if not block_to_nets[bid]:
                    xy = pos[bid] + rng.uniform(-radius, radius, size=2)
                else:
                    net_id = rng.choice(block_to_nets[bid])
                    start, end = net_offsets[net_id], net_offsets[net_id+1]
                    others = [p for p in range(start, end) if pin_owner[p] != bid]
```

```
                    if not others:
                        xy = pos[bid] + rng.uniform(-radius, radius, size=2)
                    else:
                        q_idx = rng.choice(others)
                        q_owner = pin_owner[q_idx]
                        target_ref = pos[q_owner] if q_owner < M else port_positions[q_owner - M]
                        target_center = target_ref + pin_offset[q_idx]
                        xy = target_center + rng.normal(0, radius * 0.1, size=2)
                elif m_type < 0.95:
                    if not block_to_nets[bid]:
                        xy = pos[bid] + rng.uniform(-radius, radius, size=2)
                    else:
                        net_id = rng.choice(block_to_nets[bid])
                        start, end = net_offsets[net_id], net_offsets[net_id+1]
                        others = [p for p in range(start, end) if pin_owner[p] != bid]
                        if not others:
                            xy = pos[bid] + rng.uniform(-radius, radius, size=2)
                        else:
                            centroid = np.zeros(2)
                            count = 0
                            for p_idx in others:
                                owner = pin_owner[p_idx]
                                centroid += (pos[owner] if owner < M else port_positions[owner - M])
                                count += 1
                            centroid /= count
                            vec = pos[bid] - centroid
                            dist = np.linalg.norm(vec)
                            if dist < 1e-6: vec = rng.standard_normal(2)
                            xy = pos[bid] + (vec / (dist + 1e-6)) * rng.uniform(radius * 0.1, radius *
  0.3)
                else:
                    low = half_sizes[bid] + 1e-4
                    high = canvas - half_sizes[bid] - 1e-4
                    xy = rng.uniform(np.minimum(low, high), np.maximum(low, high))

                if np.any(xy - half_sizes[bid] < 0) or np.any(xy + half_sizes[bid] > canvas):
                    continue
                if bid < num_hard:
                    diff = np.abs(pos[:num_hard] - xy)
                    overlap_mask = np.all(diff < (half_sizes[:num_hard] + half_sizes[bid]), axis=1)
                    if np.any(overlap_mask & (np.arange(num_hard) != bid)):
                        continue
                v_ids.append(bid)
                v_pos.append(xy)

            if not v_ids:
                continue

            res = ev.evaluate_moves(np.array(v_ids), np.array(v_pos, dtype=np.float32))

            best_idx = -1
            min_delta = float('inf')
            current_search_cost = get_cost(scores, stage)

            for i, r in enumerate(res):
                delta = get_cost(r, stage) - current_search_cost
                if delta < min_delta:
                    min_delta, best_idx = delta, i

            if best_idx != -1:
                if min_delta < 0 or (temp > 0 and rng.random() < np.exp(-min_delta / temp)):
                    tid, txy = v_ids[best_idx], v_pos[best_idx]
                    ev.apply(tid, txy)
                    ev.commit()
                    pos[tid] = txy
                    scores = ev.score()
                    if scores['proxy_cost'] < best_global_proxy:
                        best_global_proxy = scores['proxy_cost']
                        best_global_pos = pos.copy()

            if iter_count % 1200 == 0:
                ev.rebuild()

    # Final Legalization
    final_pos = best_global_pos.copy()
    for i in range(M):
        if fixed[i]:
            final_pos[i] = problem['initial_positions'][i]
        else:
            low = half_sizes[i] + 1e-6
            high = canvas - half_sizes[i] - 1e-6
            final_pos[i] = np.clip(final_pos[i], low, high)

    return {"positions": final_pos}
```

C.4 QUBIT ROUTING: GEMMA → MUSE → GEMMA

```
#[derive(Debug, Clone)]
struct InteractionWindow {
    weights: Vec<Vec<(usize, f64)>>,
}

impl InteractionWindow {
    fn new(num_qubits: usize) -> Self {
        Self {
            weights: vec![Vec::new(); num_qubits],
        }
    }

    fn add_interaction(&mut self, p1: usize, p2: usize, weight: f64) {
        self.weights[p1].push((p2, weight));
        self.weights[p2].push((p1, weight));
    }

    fn score_swap(&self, a: usize, b: usize, topology: TopologyView<'_>) -> f64 {
        let mut delta = 0.0;

        if let Some(w_a) = self.weights.get(a) {
            for &(other, w) in w_a {
                if other == b { continue; }
                delta += w * (topology.distance(b, other) as f64 - topology.distance(a, other) as f64)
     ;
            }
        }

        if let Some(w_b) = self.weights.get(b) {
            for &(other, w) in w_b {
                if other == a { continue; }
                delta += w * (topology.distance(a, other) as f64 - topology.distance(b, other) as f64)
     ;
            }
        }

        delta
    }
}

fn build_interaction_window(ctx: &SwapSelectionContext<'_>, depth: usize) -> InteractionWindow {
    let mut window = InteractionWindow::new(ctx.topology().num_qubits());
    let mut pred_counts = ctx.remaining().remaining_predecessor_counts().to_vec();
    let mut current_layer = ctx.front_layer().node_ids().to_vec();
    let mut seen_pairs = std::collections::HashSet::new();

    let weights = [150.0, 60.0, 30.0, 15.0, 8.0, 4.0, 2.0, 1.0, 0.5, 0.2];

    for d in 0..depth {
        if current_layer.is_empty() { break; }
        let base_w = if d < weights.len() { weights[d] } else { 0.1 };

        let mut next_layer = Vec::new();
        for &node_id in &current_layer {
            let node = ctx.circuit().node(node_id);
            if let Some((la, lb)) = node.two_qubit_pair() {
                let (q1, q2) = if la < lb { (la, lb) } else { (lb, la) };
                if seen_pairs.insert((q1, q2)) {
                    let criticality = 1.0 + (node.block_count() as f64 * 0.5);
                    window.add_interaction(
                        ctx.layout().physical_of_logical(la),
                        ctx.layout().physical_of_logical(lb),
                        base_w * criticality,
                    );
                }
            }
            for &successor in node.successors() {
                if pred_counts[successor] > 0 {
                    pred_counts[successor] -= 1;
                    if pred_counts[successor] == 0 {
                        next_layer.push(successor);
                    }
                }
            }
        }
        current_layer = next_layer;
    }
    window
}

#[derive(Debug, Clone)]
pub struct CandidatePolicy {
    decay_state: Vec<f64>,
```

```
}

impl Default for CandidatePolicy {
    fn default() -> Self {
        Self { decay_state: Vec::new() }
    }
}

impl Policy for CandidatePolicy {
    fn choose_best_initial_layout(
        &mut self,
        ctx: &InitialLayoutContext<'_>,
        rng: &mut RngState,
    ) -> Result<Vec<usize>, RouterError> {
        let circuit = ctx.circuit();
        let topology = ctx.topology();
        let num_logical = circuit.num_logical_qubits();
        let logical_components = circuit.logical_interaction_components();
        let physical_components = topology.connected_components();

        let mut interactions = vec![vec![0usize; num_logical]; num_logical];
        let mut logical_degree = vec![0usize; num_logical];
        for node_id in circuit.node_ids() {
            if let Some((la, lb)) = circuit.node(node_id).two_qubit_pair() {
                interactions[la][lb] += 1;
                interactions[lb][la] += 1;
                logical_degree[la] += 1;
                logical_degree[lb] += 1;
            }
        }

        let mut phys_score = vec![0u32; topology.num_qubits()];
        let distances = topology.distances();
        for p in 0..topology.num_qubits() {
            phys_score[p] = distances[p].iter().sum::<u32>();
        }

        let mut mapping = vec![usize::MAX; num_logical];
        let mut used_physical = vec![false; topology.num_qubits()];
        let mut p_comp_used_count = vec![0usize; physical_components.len()];

        for l_comp in logical_components {
            if l_comp.is_empty() { continue; }

            let l_seed = *l_comp.iter().max_by_key(|&&l| logical_degree[l]).unwrap();

            let mut candidates: Vec<usize> = (0..physical_components.len())
                .filter(|&pc_idx| physical_components[pc_idx].len() >= l_comp.len() +
     p_comp_used_count[pc_idx])
                .collect();

            if candidates.is_empty() {
                candidates = (0..physical_components.len())
                    .filter(|&pc_idx| physical_components[pc_idx].len() > p_comp_used_count[pc_idx])
                    .collect();
            }

            if candidates.is_empty() {
                return Err(RouterError::Routing("No physical capacity".to_string()));
            }

            let pc_idx = candidates[rng.gen_index(candidates.len())];
            let p_comp = &physical_components[pc_idx];

            let mut p_seeds: Vec<usize> = p_comp.iter()
                .filter(|&&p| !used_physical[p])
                .cloned()
                .collect();
            p_seeds.sort_by_key(|&p| phys_score[p]);
            let p_seed = p_seeds[rng.gen_index(std::cmp::min(p_seeds.len(), 2))];

            mapping[l_seed] = p_seed;
            used_physical[p_seed] = true;
            p_comp_used_count[pc_idx] += 1;

            let mut queue = VecDeque::new();
            queue.push_back(l_seed);

            while let Some(l_u) = queue.pop_front() {
                for &l_v in l_comp {
                    if mapping[l_v] == usize::MAX && interactions[l_u][l_v] > 0 {
                        let mut best_p = usize::MAX;
                        let mut min_cost = u32::MAX;

                        for &p_w in p_comp {
```

```
                        if used_physical[p_w] { continue; }
                        let mut cost = 0u32;
                        for &l_mapped in l_comp {
                            if mapping[l_mapped] != usize::MAX && interactions[l_v][l_mapped] > 0
  {
                                cost += (interactions[l_v][l_mapped] as u32) * topology.distance(
p_w, mapping[l_mapped]);
                            }
                        }
                        if cost < min_cost {
                            min_cost = cost;
                            best_p = p_w;
                        }
                    }

                    if best_p != usize::MAX {
                        mapping[l_v] = best_p;
                        used_physical[best_p] = true;
                        p_comp_used_count[pc_idx] += 1;
                        queue.push_back(l_v);
                    }
                }
            }
        }
    }

    let mut free_physical: Vec<usize> = (0..topology.num_qubits()).filter(|&p| !used_physical[p]).
collect();
    for l in 0..num_logical {
        if mapping[l] == usize::MAX {
            let mut best_p = usize::MAX;
            let mut min_cost = u32::MAX;
            for &p in &free_physical {
                let mut cost = 0u32;
                for m in 0..num_logical {
                    if mapping[m] != usize::MAX && interactions[l][m] > 0 {
                        cost += (interactions[l][m] as u32) * topology.distance(p, mapping[m]);
                    }
                }
                if cost < min_cost {
                    min_cost = cost;
                    best_p = p;
                }
            }
            if best_p != usize::MAX {
                mapping[l] = best_p;
                used_physical[best_p] = true;
                free_physical.retain(|&x| x != best_p);
            } else {
                return Err(RouterError::Routing("Out of space".to_string()));
            }
        }
    }
    Ok(mapping)
}

fn choose_best_swap(
    &mut self,
    ctx: &SwapSelectionContext<'_>,
    rng: &mut RngState,
) -> Option<(usize, usize)> {
    let num_qubits = ctx.topology().num_qubits();
    if self.decay_state.len() != num_qubits {
        self.decay_state = vec![0.0; num_qubits];
    }

    if ctx.swaps_since_progress() == 0 {
        self.decay_state.fill(0.0);
    } else if let Some((a, b)) = ctx.last_applied_swap() {
        self.decay_state[a] += 1.0;
        self.decay_state[b] += 1.0;
    }

    for val in &mut self.decay_state {
        *val *= 0.95;
    }

    let window = build_interaction_window(ctx, 20);
    let mut candidates = std::collections::HashSet::new();
    for p in 0..num_qubits {
        if !window.weights[p].is_empty() {
            for &neighbor in ctx.topology().neighbors(p) {
                let edge = if p < neighbor { (p, neighbor) } else { (neighbor, p) };
                candidates.insert(edge);
            }
```

```
            }
        }

        if candidates.is_empty() { return None; }

        let mut best_swaps = Vec::new();
        let mut min_score = f64::INFINITY;
        let epsilon = 1e-7;

        for swap in candidates {
            let delta = window.score_swap(swap.0, swap.1, ctx.topology());

            let mut front_layer_bonus = 0.0;
            for &node_id in ctx.front_layer().node_ids() {
                if let Some((la, lb)) = ctx.front_layer().logical_pair(node_id) {
                    let p_la = ctx.layout().physical_of_logical(la);
                    let p_lb = ctx.layout().physical_of_logical(lb);
                    let old_dist = ctx.topology().distance(p_la, p_lb);
                    let mut new_dist = old_dist;
                    if p_la == swap.0 { new_dist = ctx.topology().distance(swap.1, p_lb); }
                    else if p_la == swap.1 { new_dist = ctx.topology().distance(swap.0, p_lb); }
                    else if p_lb == swap.0 { new_dist = ctx.topology().distance(p_la, swap.1); }
                    else if p_lb == swap.1 { new_dist = ctx.topology().distance(p_la, swap.0); }

                    if new_dist < old_dist {
                        front_layer_bonus -= 55.0 * (old_dist - new_dist) as f64;
                        if new_dist == 1 { front_layer_bonus -= 110.0; }
                    } else if new_dist > old_dist {
                        front_layer_bonus += 85.0 * (new_dist - old_dist) as f64;
                    }
                }
            }

            let penalty = (self.decay_state[swap.0] + self.decay_state[swap.1]) * 0.55;
            let score = delta + penalty + front_layer_bonus;

            if score < min_score - epsilon {
                min_score = score;
                best_swaps.clear();
                best_swaps.push(swap);
            } else if (score - min_score).abs() <= epsilon {
                best_swaps.push(swap);
            }
        }

        if best_swaps.is_empty() { return None; }
        Some(best_swaps[rng.gen_index(best_swaps.len())])
    }
}
```

D QUBIT ROUTING DETAILS

Baselines. Every routed circuit must respect the processor's connectivity and remain equivalent to its input. As in SimpleTES and Hyra (Hyra Team, 2026), Table 5 reports the SWAPs added on each processor and their total; the methods rank in the same order under the weighted count S as under the total. SimpleTES discovered routing programs with GPT-OSS-120B and with Gemini 3 Pro; we take its reported counts. Hyra releases routed circuits for 61 of the 72 circuits rather than a routing program. We replayed all 61 with the SimpleTES checker, and all pass; its published totals equal these counts plus SABRE's counts for the 11 missing circuits, which add one SWAP. Because Hyra releases circuits rather than a program, we could check its circuits but not rerun its method.

Performance over the search. Figure 13 follows the winning lineage. Gemma, alone, passes SimpleTES with Gemini 3 Pro at step 11 but stays below Hyra. Muse, continuing Gemma's programs from step 20, passes Hyra at step 26. Gemma, taking Muse's programs back at step 29, raises the reward to 0.556233 at step 31, the result in Table 5.

Decay penalty behavior. Every program is a Rust policy with two functions: one chooses the initial layout, and the other chooses the next SWAP. Gemma's best programs score each candidate SWAP with three terms: the distance change for gates in the next 20 layers, weighted by depth; a decay penalty on qubits that were swapped recently, which keeps the router from undoing its own moves; and a front-layer pull that rewards moving the next executable gates closer. Reading the programs shows why the handoff helps. (1) Gemma, alone, kept raising the decay penalty. Across its

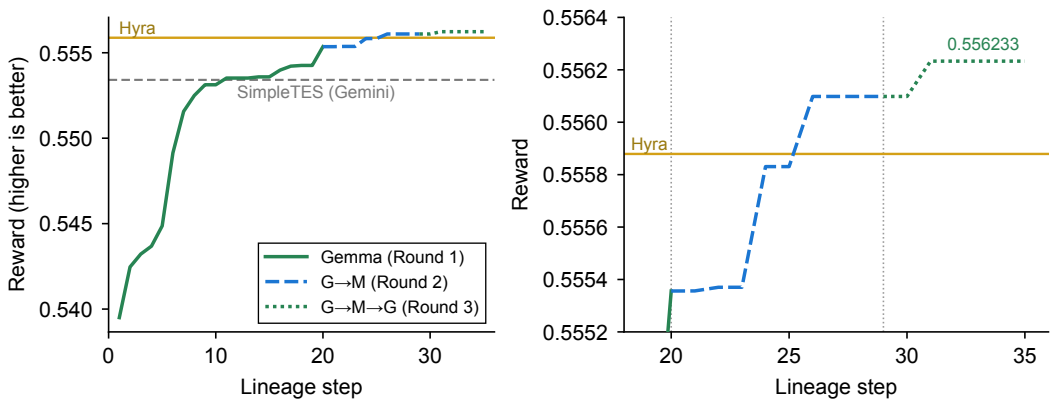
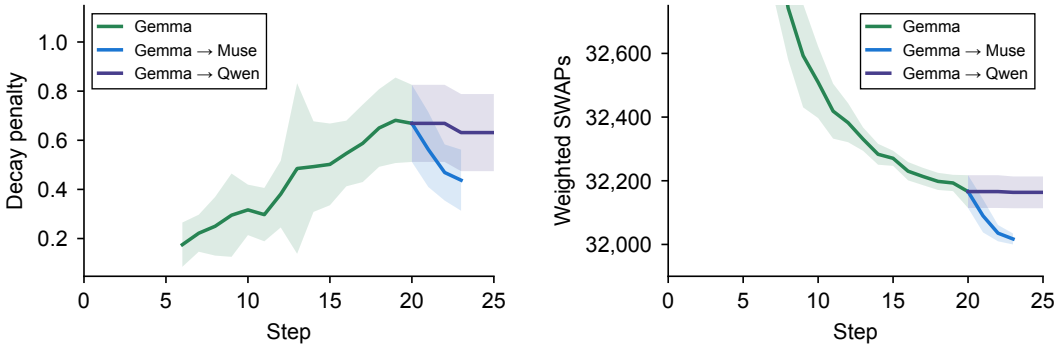


Figure 13: **MOMO passes Hyra after the handoff to Muse.** Best reward $B/(B+S)$ along the winning lineage (Gemma, then Muse, then Gemma), higher is better. Horizontal lines give Hyra's and SimpleTES's rewards under the same formula. Left: the whole search. Right: the handoffs (dotted lines), zoomed.

16 best programs, the mean rose from 0.18 at step 6 to 0.67 at step 20 (Figure 14a). (2) Muse, alone, wrote a different policy. It weights the front layer nearly three times as heavily as the lookahead, adds a large bonus for any SWAP that makes a gate executable, and leaves decay off. (3) Given Gemma's programs, Muse kept their structure and moved them toward its own policy. Within three steps, it lowered the mean decay penalty to 0.44 and doubled the front-layer pull, from 34 to 68. Over the same steps, the 16 best programs fell from 32,166 to 32,017 weighted SWAPs, more than Gemma gained in its last six steps (Figure 14b). Among Muse's programs, 63% of those that weakened decay beat their parent, against 21% of those that left it unchanged. Muse's best program starts from one of Gemma's weaker programs, 150 weighted SWAPs behind Gemma's best; two edits cut its decay penalty from 0.5 to 0.3, raise its front-layer pull from 50 to 70, and pass Gemma's best. Most of the gain is on Q20, while Willow loses 79 SWAPs. Qwen, given the same programs, left the decay penalty and front-layer pull nearly unchanged, and its best programs did not improve.



(a) Decay penalty.

(b) Weighted SWAPs.

Figure 14: **Muse lowers the decay penalty, and the best programs improve.** The 16 best programs at each step, mean and standard deviation; Gemma's programs are handed off at step 20. (a) Weight of the penalty on recently swapped qubits in their SWAP score, over the programs that use it (at least three); Muse's own Round 1 policy uses no decay. (b) Their weighted SWAP count S, lower is better; Gemma's first steps lie above the plotted range.