CAN TEST-TIME MEMORY EVOLUTION OUTPERFORM TEST-TIME TRAINING FOR SCIENTIFIC DISCOVERY?

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent *test-time training* (TTT) frameworks have enabled small open-weight Large Language Models (LLMs) to outperform inference-only systems powered by frontier models on scientific discovery tasks. However, TTT introduces additional computational cost for online weight updates during the discovery process, making computational efficiency an important consideration when scaling to larger models. This motivates us to explore an efficient *test-time adaptation* method for scientific discovery. Under the same discovery framework, we introduce Test-Time Memory Evolution (TTME), a plug-and-play test-time adaptation method, as an alternative to TTT. TTME continually updates the external memory by distilling the search experience to guide subsequent discovery without updating model weights. By controlling the discovery framework, we have a clean comparison between TTME and TTT on both performance and cost. Our evaluation shows that TTME outperforms TTT in all five mathematical tasks, while reducing computational cost by $82.8 - 84.6\%$ in FLOPs and $47.7 - 58.4\%$ in runtime. These results establish TTME as an effective and computationally efficient alternative to TTT in the evaluated discovery settings.

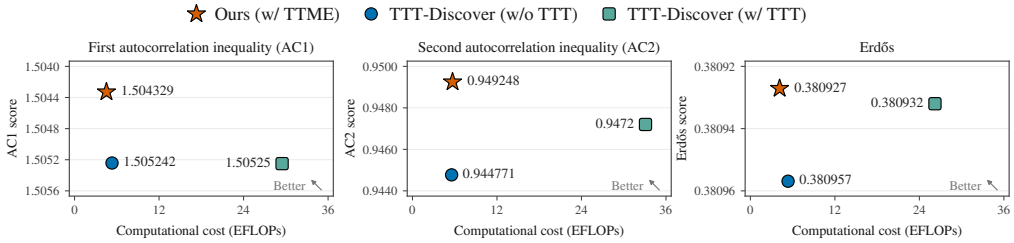


Figure 1: Best-found scores versus computational cost under TTT-Discover with Qwen3-8B across three tasks. **The results demonstrate that our *TTME* method achieves the best scores at low computational cost on all three tasks.**

1 INTRODUCTION

Scientific discovery often involves proposing hypotheses, evaluation, and using the resulting evidence to guide further investigation (King et al., 2009; Lu et al., 2024). Advances in large language models (LLMs) reasoning, coding, and agentic tool use have enabled systems that automate part of this discovery process (Novikov et al., 2025; Lange et al., 2025). In the iterative discovery loop, a failed attempt can reveal an implementation constraint, while a successful modification can suggest an idea worth applying elsewhere. How a discovery system should accumulate and reuse these experiences is therefore central to making progress within a limited search budget.

Recent test-time training (TTT) frameworks, such as TTT-Discover (Yuksekgonul et al., 2026) and ThetaEvolve (Wang et al., 2025), provide a promising solution to this challenge by combining iterative search with test-time reinforcement learning (RL) (Shao et al., 2024; DeepSeek-AI, 2025). Specifically, given a discovery problem, an LLM generates candidate solutions and gets feedback from a verifier. These evaluated attempts provide problem-specific training data to update the model's weights through RL. Through TTT, the model internalizes experience accumulated during the search into its weights, which improves the quality of future candidates. Their results demonstrate that TTT is a powerful *test-time adaptation* method on challenging scientific discovery

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.

tasks. It shows that smaller open-weight models equipped with test-time training can outperform inference-only systems powered by frontier models, such as AlphaEvolve (Novikov et al., 2025). However, TTT requires online model weight adaptation during the whole discovery process, which is usually computationally expensive and time-consuming (see Table 3). It makes TTT hard to scale to larger models for more complex discovery tasks. This motivates us to explore: *under the same discovery framework, is there a more efficient test-time adaptation method that can achieve comparable or even better performance than test-time training?*

To investigate this problem, we compare test-time adaptation methods within the same discovery framework. Specifically, we study whether updating external memory can provide a more efficient alternative to updating model weights. Our core intuition is that LLMs could learn from their discovery experience through in-context learning, without updating their weights (Brown et al., 2020; Shinn et al., 2023; Zhao et al., 2024; Zhang et al., 2026). Building on this intuition, we introduce Test-Time Memory Evolution (TTME), a plug-and-play *self-evolving memory* system for scientific discovery. Instead of internalizing search experience through updating weights, we extract it from evaluated attempts and store it in external memory for LLMs. We continually update the memory using evaluation feedback and use it to guide the next search step.

Specifically, we divide the search experience into two components: practical lessons for implementing candidate solutions and algorithmic ideas for improving task scores. Correspondingly, TTME maintains two complementary memory modules: *Guideline Memory* and *Strategy Memory*. Guideline Memory distills execution and evaluation feedback into shared operational lessons. Strategy Memory extracts effective algorithmic ideas from score-improving modifications and consolidates related experiences into reusable strategies. We select and adapt relevant strategies to guide subsequent exploration, combining ideas from different search attempts.

Our contributions are summarized as follows:

- We investigate whether a self-evolving memory can outperform test-time weight updates in scientific discovery under the same discovery frameworks.
- We introduce TTME, an efficient plug-and-play method that continually updates external memory from evaluated search experience to guide subsequent discovery with a fixed-weight model.
- We evaluate TTME under two TTT frameworks across five mathematical construction tasks and three models spanning 8B and 120B parameters. *We preserve each framework's experimental configuration, replacing only its TTT part with TTME. TTME outperforms the reported TTT results in all evaluated tasks.* Our analysis shows that TTME reduces running time by $47.7-58.4\%$ and computational costs by $82.8-84.6\%$ relative to TTT. These results demonstrate that TTME is a more efficient alternative to TTT for scientific discovery.

2 Related Work

LLM-Based Scientific Discovery. LLM-based discovery systems search for candidates whose quality can be measured by automated evaluators (Lange et al., 2025; Liu et al., 2024). FunSearch (Romera-Paredes et al., 2024) combines a fixed-weight LLM with evolutionary search, using previously evaluated programs as context for generating new candidates. AlphaEvolve (Novikov et al., 2025; Georgiev et al., 2025) extends this approach to scientific and algorithmic discovery through iterative code modification and evaluation. These systems accumulate promising candidates and use them to guide subsequent search. Related work on heuristic design, such as ReEvo (Ye et al., 2024), further incorporates comparative feedback and accumulated reflections into evolutionary search.

Test-Time Training for Scientific Discovery. Test-time training adapts a model to the problem encountered at inference time (Sun et al., 2020; 2023; Hubert et al., 2026; Akyürek et al., 2025; Hardt & Sun, 2024; Hübotter et al., 2025; Gandelsman et al., 2022). For scientific discovery, this allows evaluated search attempts to provide training signals for subsequent generation (Phan et al., 2025; Surina et al., 2025). ThetaEvolve (Wang et al., 2025) incorporates online RL into an evolutionary program-search framework. TTT-Discover (Yuksekgonul et al., 2026) combines problem-specific reinforcement learning with the reuse of previously explored states, targeting the best solution found during search. These approaches update model parameters alongside the evolving search context.

We investigate external memory as an alternative adaptation mechanism within these two frameworks. Specifically, we retain their underlying search procedures and task evaluators, while using feedback to update memory that conditions a fixed-weight model.

Memory for Scientific Discovery. External memory enables LLMs to use accumulated experience through context rather than weight updates (Zhang et al., 2026; Agrawal et al., 2026; Zhang et al., 2025; Yang et al., 2024; Khattab et al., 2024; Fernando et al., 2024; Guo et al., 2024; Madaan et al., 2023). Reflexion (Shinn et al., 2023) stores reflections for later attempts, while ExpeL (Zhao et al., 2024) combines experience-derived insights with retrieved successful trajectories. These studies evaluate question answering, code generation, and domain-specific reasoning, rather than open-ended scientific discovery. In scientific discovery, EvoScientist (Lyu et al., 2026) proposes a multi-agent framework for end-to-end AI research, with persistent memories that accumulate experience. In this paper, we focus on investigating whether a self-evolving memory system can provide a more efficient alternative to test-time training for scientific discovery.

3 METHOD

3.1 BACKGROUND AND PROBLEM FORMULATION

We study scientific discovery as an iterative process of proposing hypotheses, evaluating them, and using the resulting evidence to guide subsequent proposals. We focus on tasks with executable candidate programs and a task-specific verifier, where a program produces a construction whose validity and objective value are assessed by the verifier.

Let $v(x) \in \{0, 1\}$ indicate whether candidate program x produces a valid result, and let $s(x)$ denote its evaluated score, oriented so that larger values are better. For the set $\mathcal{X}_{\text{eval}}$ of candidates evaluated within the search budget, the best valid score is

$$s^\star = \max_{\substack{x \in \mathcal{X}_{\text{eval}} \\ v(x)=1}} s(x). \tag{1}$$

Our goal is to find the highest-scoring valid candidate within a given search budget.

We treat two existing TTT frameworks, e.g., TTT-Discover (Yuksekgonul et al., 2026) and ThetaEvolve (Wang et al., 2025), as a common search interface. A *search state* represents a candidate solution from which search can continue. We use *archive* to denote the collection of retained search states and associated records in either framework. A *parent* is a state selected from this archive as the starting point for a new attempt. At each search step, the framework selects parents and constructs generation contexts from the task, selected states, and associated information. The LLM, then, generates programs. Execution and verification produce candidate outcomes, including validity, objective values, and execution feedback. The framework records these attempts in the evaluated search history and updates the archive with states retained for subsequent search. We preserve each framework's parent-selection and archive-update rules. Below, *parent program* refers to the code associated with a selected parent and supplied for modification.

Both frameworks use search feedback to update model weights through test-time RL. *We ask whether evolving external memory can substitute for these weight updates*. Writing $\mathcal{D}_t$ for evaluated search records available to an update, we contrast two adaptation mechanisms:

$$\begin{aligned} &\text{Weight adaptation:} && \theta_{t+1} \leftarrow U_{\text{RL}}(\theta_t, \mathcal{D}_t), \\ &\text{Memory adaptation:} && \mathcal{M}_{t+1} \leftarrow U_{\text{mem}}(\mathcal{M}_t, \mathcal{D}_t), \qquad \theta_t = \theta_0. \end{aligned} \tag{2}$$

Here, U_{RL} updates model parameters, whereas U_{mem} updates external memory used to construct subsequent generation contexts. We update memory every K search steps. We retain the underlying search interface and task verifier while replacing weight adaptation with *self-evolving memory adaptation*. Figure 2 compares these two adaptation mechanisms.

Search records contain both operational feedback and examples of algorithmic improvement. Reusing this experience across parents requires extracting useful changes from their original code and adapting them to the current program. We therefore maintain $\mathcal{M}_t = (\mathcal{L}_t, \mathcal{S}_t)$: *Guideline Memory* $\mathcal{L}_t$ stores shared operational lessons, and *Strategy Memory* $\mathcal{S}_t$ organizes reusable algorithmic

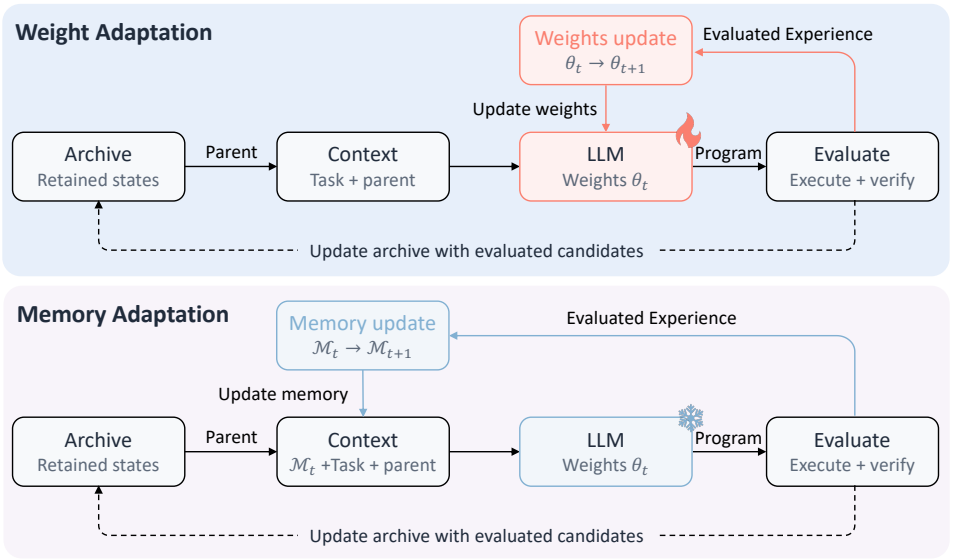


Figure 2: Weight adaptation updates model parameters through test-time RL (top). Memory adaptation updates external memory that conditions generation while keeping model weights fixed (bottom). Both retain the same underlying discovery framework.

experience. We use Strategy Memory to guide search through parent-specific modification instructions, while both generation branches receive the shared guidelines. Together, these memories provide reusable guidance for subsequent search while keeping the model weights fixed. Figure 2 illustrates how new outcomes update the guidance for subsequent search.

3.2 Guideline Memory: Shared Operational Lessons

Evaluating a candidate provides feedback beyond its objective score, including information about execution and constraint satisfaction. This feedback can reveal experience to subsequent attempts. We distill these observations into compact, reusable lessons stored in Guideline Memory. Each lesson specifies when it applies, what was observed, and what action to take. For example, a failed call to an unavailable function can yield a guideline that identifies the unsupported API and suggests an alternative. We provide these guidelines as shared context for subsequent candidate generation.

In every K search step, we summarize the attempts evaluated since the previous update, including valid results and observed failures from different parents. We prompt the same base model with this summary and the current guidelines to produce an updated set of lessons. We parse the output, remove duplicate lessons, and retain the updated guidelines subject to a capacity limit.

3.3 Strategy Memory: Reusable Algorithmic Experience

Combining existing ideas is important in scientific discovery. When a hypothesis and its corresponding implementation achieve a higher evaluation score in one parent program, the underlying idea may also be useful for parents in other parts of the search space. Strategy Memory consolidates these experiences into reusable strategies, and records the conditions under which they were applied.

We update Strategy Memory every K search steps, together with Guideline Memory. We select evaluated cases where a modified program produces a valid result with a higher score than its parent. For each case, we analyze the original and modified programs, their differences, and the evaluation results using the same base model. It produces an *experience note* describing the algorithmic idea reflected in the changes and the conditions under which it was applied.

Different modifications can use the same underlying idea with different implementation details. We group experience notes that share a similar idea into a *strategy family* and maintain a persistent *strategy entry* for each family. The strategy entry consolidates the experience notes into a reusable

Algorithm 1 Scientific discovery with self-evolving memory

Require: Fixed model π_θ, task verifier, search budget; B candidates per parent, N modification instructions per parent; memory update interval $K \geq 1$

Initialize archive, evaluated history, and memories $\mathcal{L}_0, \mathcal{S}_0$
for each search step $t = 0, 1, \ldots$ within the budget **do**
  Sample parents using a selection rule from the underlying framework
  for each parent i, sequentially **do**
    Assign N strategies using Eq. (3)
  end for
  for each parent i **do**
    Form N modification instructions using Eq. (4)
    Generate $B/2$ direct candidates using Eq. (5)
    Generate $B/(2N)$ candidates per instruction using Eq. (6)
  end for
  Evaluate candidates; record outcomes and commit archive updates
  if $(t+1) \bmod K = 0$ **then**
    Update $\mathcal{L}_{t+1}$ from $\mathcal{L}_t$ and evaluated outcomes
    Update $\mathcal{S}_{t+1}$ using $\mathcal{S}_t$ and evaluated improvements
  else
    $\mathcal{L}_{t+1} \leftarrow \mathcal{L}_t, \mathcal{S}_{t+1} \leftarrow \mathcal{S}_t$
  end if
end for
return Best valid result found under the task objective

description of the shared idea, its application conditions, and its implementation variants, with references to the source cases.

At each update, we incorporate new experience notes into existing strategy entries or create new entries for distinct ideas, preserving their implementation details and source references. Thus, Strategy Memory accumulates strategies and their observed applications for subsequent search.

3.4 Strategy-Guided Search and Candidate Generation

With Guideline Memory and Strategy Memory established, we now describe how to use them to guide the next round of candidate generation. Guideline Memory directly provides shared context for all candidate generation. Strategy Memory stores a collection of ideas, but not all ideas are relevant to every parent. Moreover, reusing an idea requires adapting it to the current parent program. We therefore introduce *Strategy-Guided Search* to select relevant ideas from Strategy Memory and adapt them to the current parent program. When no suitable strategy is selected, we use the memory and parent context to propose a new idea. In either case, we produce a modification instruction describing how to modify the current parent program.

Specifically, at search step t, let $p_i^{(t)}$ denote the program associated with the i-th selected parent. A *parent group* consists of the generation attempts associated with that parent within the step. Let $C_i^{(t)}$ denote the input context supplied by the underlying framework for the i-th parent at step t, which includes the task specification, the parent program $p_i^{(t)}$, and available history.

For each parent, we use a *selector* (base model) to analyze and select up to N suitable strategies from Strategy Memory based on their potential to improve the parent program. The selector is provided with the parent context, a summary of each strategy entry, and the strategies already assigned to earlier parents in the current step. Selection proceeds sequentially across parents:

$$\mathbf{a}_i^{(t)} = \mathrm{Select}\big(C_i^{(t)}, \hat{\mathcal{S}}_t, \mathbf{A}_{<i}^{(t)}\big), \qquad \mathbf{a}_i^{(t)} = (a_{i,1}^{(t)}, \ldots, a_{i,N}^{(t)}), \tag{3}$$

where $\mathbf{A}_{<i}^{(t)}$ contains the strategies assigned to parents $1, \ldots, i-1$ at search step t, $\hat{\mathcal{S}}_t$ denotes the strategy summaries in Strategy Memory, and $a_{i,n}^{(t)}$ is the n-th selected strategy or null if no suitable strategy is selected. The selector is instructed to prioritize improvement potential for the current

parent. Among similarly plausible choices, it should prefer a strategy not yet assigned in the step. If fewer than N strategies are selected, we set the remaining elements of $\mathbf{a}_i^{(t)}$ to $\varnothing$.

Then, we produce a parent-specific modification instruction $d_{i,n}^{(t)}$ for each slot:

$$d_{i,n}^{(t)} = \begin{cases} \mathrm{Adapt}\big(C_i^{(t)}, \mathcal{S}_t[a_{i,n}^{(t)}]\big), & a_{i,n}^{(t)} \neq \varnothing, \\ \mathrm{Propose}\big(C_i^{(t)}, \mathcal{S}_t\big), & a_{i,n}^{(t)} = \varnothing. \end{cases} \tag{4}$$

For a selected strategy, one *adaptor* reads its full content and the full parent code, then specifies how to apply the strategy to that parent. For an empty slot, the adaptor uses the parent and available strategy memory content to propose a new idea.

We use the constructed instructions to guide candidate generation. Adapting an explored strategy allows us to combine an idea extracted from a score-improving modification with the parent program's existing approach. However, it may also limit the exploration space. Therefore, we also generate candidates without prescribing a particular strategy. Specifically, let B denote the total number of candidates generated per parent. We allocate $B/2$ candidates to *Direct Generation* and $B/2$ to *Strategy-Guided Generation*, dividing the latter equally among the N modification instructions:

$$x \sim \pi_\theta(\cdot \mid C_i^{(t)}, \mathcal{L}_t), \tag{5}$$

$$x \sim \pi_\theta(\cdot \mid C_i^{(t)}, \mathcal{L}_t, d_{i,n}^{(t)}). \tag{6}$$

Their evaluation outcomes inform subsequent memory updates, which provide guidance for later search steps. The complete procedure is summarized in Algorithm 1. Implementation details and prompts appear in Appendices C and D.

4 Experiments

4.1 Experiment Settings

Models. We use Qwen3-8B (Qwen Team, 2025) and gpt-oss-120b (OpenAI, 2025) in the TTT-Discover framework. For ThetaEvolve, we use DeepSeek-R1-0528-Qwen3-8B (DeepSeek-AI, 2025), abbreviated as Distill-Qwen3-8B. In our method, we keep the model weights fixed throughout the search. The same base model performs candidate generation and all memory-related operations.

Frameworks. We evaluate our method within two frameworks, TTT-Discover (Yuksekgonul et al., 2026) and ThetaEvolve (Wang et al., 2025), that combine iterative search with test-time RL. We compare three settings within two frameworks: w/o TTT, which keeps model weights fixed without our TTME; w/ TTT, which updates model weights through test-time RL; and w/ TTME, which keeps model weights fixed and updates self-evolving memory. For weight adaptation, we report results from the original papers, with model weights updated through test-time RL. Without weight adaptation, we run the underlying search procedure with fixed model weights and its existing generation context. Our method also keeps the weights fixed, but updates Guideline Memory and Strategy Memory to guide subsequent generation. We retain each framework's parent-selection rules, archive-update rules, and task verifier (settings in Appendix B).

Tasks. We evaluate five mathematical construction tasks. (1) `Erdős`: Erdős's minimum overlap problem searches for density functions that certify tighter upper bounds on the minimum-overlap constant. (2) `First autocorrelation inequality (AC1)`: AC1 constructs a nonnegative function f to minimize $\|f * f\|_\infty / \|f\|_1^2$, yielding an upper bound on the first autocorrelation constant. Here, $f * f$ denotes the autoconvolution of f. (3) `Second autocorrelation inequality (AC2)`: AC2 constructs a nonnegative function f to maximize $\|f * f\|_2^2 / (\|f * f\|_1 \|f * f\|_\infty)$, yielding a lower bound on the second autocorrelation constant. (4) `Third autocorrelation inequality (AC3)`: AC3 allows functions with both positive and negative values and seeks a tighter upper bound using the absolute autoconvolution peak. (5) `Maximal determinant problem (Hadamard)`: Hadamard maximizes the normalized absolute determinant of a 29×29 matrix with entries in $\{-1, +1\}$.

Table 1: Results under the TTT-Discover framework for three settings: w/ TTT, w/o TTT, and w/ TTME. Results for w/ TTT are from Yuksekgonul et al. (2026) over 50 search steps.

Method	Model	Erdős ↓	AC1 ↓	AC2 ↑
TTT-Discover (w/ TTT)	Qwen3-8B	0.380932	1.50525	0.9472
TTT-Discover (w/o TTT)	Qwen3-8B	0.380957	1.505242	0.944771
Ours (w/ TTME)	Qwen3-8B	**0.380927**	**1.504329**	**0.949248**
TTT-Discover (w/ TTT)	gpt-oss-120b	0.380876	1.50287	0.9591
TTT-Discover (w/o TTT)	gpt-oss-120b	0.380945	1.505966	0.953682
Ours (w/ TTME)	gpt-oss-120b	**0.380862**	**1.502745**	**0.960784**

Table 2: Results under the ThetaEvolve framework for three settings: w/ TTT, w/o TTT, and w/ TTME. Results for w/ TTT are from Wang et al. (2025) over 65 search steps.

Method	Model	AC2 ↑	AC3 ↓	Hadamard ↑
ThetaEvolve (w/ TTT)	Distill-Qwen3-8B	0.9469	1.4930	0.5764
ThetaEvolve (w/o TTT)	Distill-Qwen3-8B	0.947375	1.519277	0.571085
Ours (w/ TTME)	Distill-Qwen3-8B	**0.948148**	**1.464224**	**0.579938**

4.2 Main Results

Performance across frameworks and models. TTME outperforms both TTT and search without TTT across two discovery frameworks, three models, and five tasks (Tables 1 and 2). The comparison with TTT shows that evolving external memory can yield better discovery results than existing weight-update methods. The comparison with search without TTT shows the benefit of adding memory to the underlying fixed-weight search procedure. Together, these results establish TTME as an effective alternative to test-time weight adaptation across the evaluated settings. Appendix E visualizes the discovered constructions.

Progress during search. We next examine how TTME improves the best-found score over the course of search (Figure 3). Under the TTT-Discover framework, TTME surpasses the TTT scores within 25 steps across all three tasks and both models. Under ThetaEvolve framework, TTME also surpasses the TTT scores on three tasks. These trajectories show that TTME can improve on the final TTT results before exhausting the search budget.

Table 3: Runtime (excluding evaluation) and computation under TTT-Discover with Qwen3-8B. TTME reduces runtime by 47.7–58.4% and FLOPs by 82.8–84.6%.

		Runtime excluding evaluation (h)			Computational cost (EFLOPs)		
Task	Model	TTT	TTME	Reduction	TTT	TTME	Reduction
AC1	Qwen3-8B	11.76	**5.36**	54.5%	29.47	**4.53**	84.6%
AC2	Qwen3-8B	12.73	**6.66**	47.7%	33.09	**5.69**	82.8%
Erdős	Qwen3-8B	11.67	**4.85**	58.4%	26.21	**4.15**	84.2%

Runtime and computational cost. We compare runtime (excluding evaluation) and computational cost under TTT-Discover with Qwen3-8B (Table 3). We include all candidate generation, memory-related LLM calls, and RL training where applicable. Across the three tasks, even though TTT-Discover updates only LoRA parameters (Hu et al., 2021), TTME reduces runtime by 47.7–58.4% and FLOPs by 82.8–84.6% relative to TTT. Combined with the performance results, these reductions show that TTME achieves better discovery outcomes with less runtime and computation.

4.3 Ablation Studies

Combining TTME with TTT. Having established the effectiveness of TTME with fixed model weights, we examine whether adding TTT further improves performance. We compare four settings under the TTT-Discover framework with Qwen3-8B: neither TTT nor TTME, TTT alone, TTME

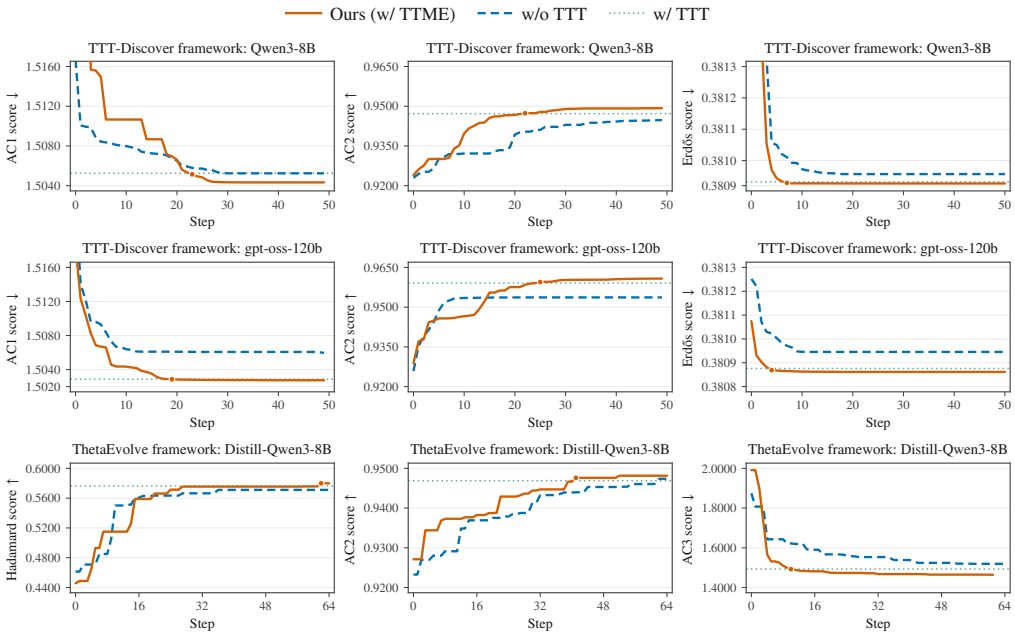


Figure 3: Best-found scores during search under TTT-Discover and ThetaEvolve frameworks. TTME achieves better final scores in every evaluated setting. Orange dots mark the first step at which TTME exceeds the corresponding final TTT score.

alone, and their combination (Table 4). In the combined setting, the RL-updated model performs both candidate generation and memory-related operations. TTME alone achieves the best scores on AC2 and Erdős, while adding TTT worsens both results. This comparison further supports using TTME as an alternative to TTT rather than requiring both forms of adaptation.

Table 4: Comparison of TTT, TTME, and their combination against a baseline with neither. TTME alone achieves the best scores on both tasks; adding TTT reduces performance.

TTT	TTME	AC2 ↑	Erdős ↓
TTT-Discover framework: Qwen3-8B			
✗	✗	0.944771	0.380957
✓	✗	0.9472	0.380932
✗	✓	**0.949248**	**0.380927**
✓	✓	0.944141	0.380944

Table 5: Ablation of Guideline Memory and Strategy Memory. Full TTME achieves the best scores on all three tasks, demonstrating the benefit of combining both memory components.

Configuration	AC1 ↓	AC2 ↑	Erdős ↓
TTT-Discover framework: gpt-oss-120b			
Full TTME	**1.502745**	**0.960784**	**0.380862**
w/o Guideline Memory	1.505644	0.958506	0.380865
w/o Strategy Memory	1.505981	0.956493	0.380864
w/o TTME	1.505966	0.953682	0.380946

Contributions of the two memories. We examine the contributions of Guideline Memory and Strategy Memory under the TTT-Discover framework with gpt-oss-120b (Table 5). We compare full TTME with variants that remove either memory component or both. Full TTME achieves the best scores on all three tasks, and removing either component worsens performance. These results support retaining both shared operational lessons and reusable algorithmic strategies to guide discovery.

4.4 Search Behavior Analysis

Candidate validity and discovery performance. Successful discovery requires candidates that are both valid and capable of improving the task score. We consider a program valid if it compiles successfully and receives a score from the evaluator. Figure 4 compares mean validity over 50 steps and best-found scores under the TTT-Discover framework with Qwen3-8B. Both TTT and TTME improve the mean program validity rate over search without TTT. On AC1 and Erdős, TTT and TTME achieve similar validity rates, while TTME achieves better best-found scores. This suggests TTME generates more effective ideas for improving task scores.

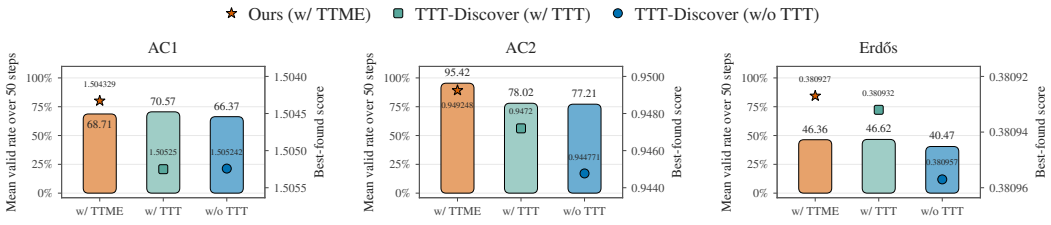


Figure 4: Implementation validity rate and final performance under the TTT-Discover framework with Qwen3-8B across three tasks. Bars show mean validity rates over 50 steps (left axes); markers show best-found scores (right axes), with better scores plotted higher.

Combining ideas for further search. Beyond validity, we examine how search combines and builds on previously explored ideas under the TTT-Discover framework with Qwen3-8B. We count a candidate as a combination if it retains an idea from its parent and adds an idea previously explored outside its ancestry. The added idea must be absent from the parent. We measure how often these combinations are selected as parents for further search.

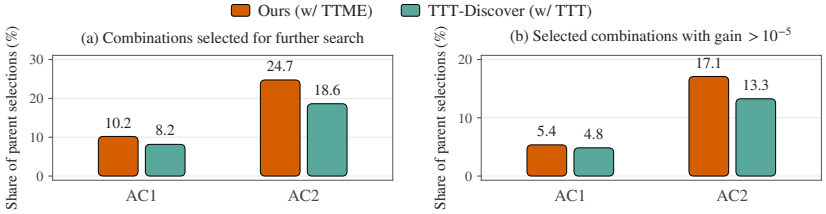


Figure 5: Combination of ideas in subsequent search under TTT-Discover with Qwen3-8B. TTME has a higher combination share on both tasks, with and without the 10^{-5} improvement requirement.

TTME selects combinations more frequently than TTT on both AC1 (10.2% versus 8.2%) and AC2 (24.7% versus 18.6%; Figure 5). We also count selections of combinations that improve on their own parents by more than 10^{-5}. TTME again has higher shares on AC1 (5.4% versus 4.8%) and AC2 (17.1% versus 13.3%). Thus, TTME not only selects combinations of existing ideas more frequently for further search, but also more frequently selects combinations that improve task scores. These observations support using external memory to combine ideas from different search branches and carry productive combinations into subsequent exploration.

5 Conclusion

We investigate whether evolving external memory can outperform test-time training for scientific discovery within the same discovery frameworks. We introduce TTME, which distills search experience into operational lessons and algorithmic strategies to guide a fixed-weight model. Across two frameworks, three models, and five mathematical tasks, TTME achieves better scores than TTT in every evaluated setting. Under TTT-Discover with Qwen3-8B, it reduces runtime excluding evaluation by 47.7–58.4% and FLOPs by 82.8–84.6%, including all memory-related calls. These results establish self-evolving memory as an effective and computationally efficient alternative to test-time weight updates for the discovery tasks we study.

AI USE STATEMENT

We used generative AI tools to assist with experimental code development. AI tools also assisted with manuscript review and language editing, figure preparation, and LaTeX formatting. The authors take responsibility for the final manuscript, code, analyses, and reported results.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2507.19457.

Ekin Akyürek, Mehul Damani, Adam Zweiger, Linlu Qiu, Han Guo, Jyothish Pari, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for few-shot learning. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 942–963. PMLR, 2025. URL https://proceedings.mlr.press/v267/akyurek25a.html.

Richard C. Barnard and Stefan Steinerberger. Three convolution inequalities on the real line with connections to additive combinatorics. *Journal of Number Theory*, 207:42–55, 2020. doi: 10.1016/j.jnt.2019.07.001. URL https://arxiv.org/abs/1903.08731.

Christopher Boyer and Zane Kun Li. An improved example for an autoconvolution inequality. *Experimental Mathematics*, pp. 1–7, 2026. doi: 10.1080/10586458.2025.2607423. URL https://doi.org/10.1080/10586458.2025.2607423.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 13481–13544. PMLR, 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

Yossi Gandelsman, Yu Sun, Xinlei Chen, and Alexei A. Efros. Test-time training with masked autoencoders. In *Advances in Neural Information Processing Systems*, volume 35, pp. 29374–29385, 2022. doi: 10.52202/068431-2130. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/bcdec1c2d60f94a93b6e36f937aa0530-Abstract-Conference.html.

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

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations*, pp. 34133–34156, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/9156b0f6dfa9bbd18c79cc459ef5d61c-Abstract-Conference.html.

Moritz Hardt and Yu Sun. Test-time training on nearest neighbors for large language models. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/f02f1185b97518ab5bd7ebde466992d3-Abstract-Conference.html.

Jan Kristian Haugland. The minimum overlap problem revisited. *arXiv preprint arXiv:1609.08000*, 2016. URL https://arxiv.org/abs/1609.08000.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026. doi: 10.1038/s41586-025-09833-y. URL https://www.nature.com/articles/s41586-025-09833-y.

Jonas Hübotter, Sascha Bongni, Ido Hakimi, and Andreas Krause. Efficiently learning at test-time: Active fine-tuning of LLMs. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/ba942323c447c9bbb9d4b638eadefab9-Abstract-Conference.html.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, pp. 54928–54958, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/f1cf02ce09757f57c3b93c0db83181e0-Abstract-Conference.html.

Ross D. King, Jem Rowland, Stephen G. Oliver, Michael Young, Wayne Aubrey, Emma Byrne, Maria Liakata, Magdalena Markham, Pinar Pir, Larisa N. Soldatova, Andrew Sparkes, Kenneth E. Whelan, and Amanda Clare. The automation of science. *Science*, 324(5923):85–89, 2009. doi: 10.1126/science.1165620. URL https://doi.org/10.1126/science.1165620.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025. URL https://arxiv.org/abs/2509.19349.

Fei Liu, Rui Zhang, Zhuoliang Xie, Rui Sun, Kai Li, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. LLM4AD: A platform for algorithm design with large language model. *arXiv preprint arXiv:2412.17287*, 2024. URL https://arxiv.org/abs/2412.17287v1.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI Scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024. URL https://arxiv.org/abs/2408.06292.

Yougang Lyu, Xi Zhang, Xinhao Yi, Yuyue Zhao, Shuyu Guo, Wenxiang Hu, Jan Piotrowski, Jakub Kaliski, Jacopo Urbani, Zaiqiao Meng, Lun Zhou, and Xiaohui Yan. EvoScientist: Towards multi-agent evolving AI scientists for end-to-end scientific discovery. *arXiv preprint arXiv:2603.08127*, 2026. URL https://arxiv.org/abs/2603.08127.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Máté Matolcsi and Carlos Vinuesa. Improved bounds on the supremum of autoconvolutions. *Journal of Mathematical Analysis and Applications*, 372(2):439–447.e2, 2010. doi: 10.1016/j.jmaa.2010.07.030. URL https://doi.org/10.1016/j.jmaa.2010.07.030.

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. URL https://arxiv.org/abs/2506.13131.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

William P. Orrick, Bruce Solomon, Roland Dowdeswell, and Warren D. Smith. New lower bounds for the maximal determinant problem. *arXiv preprint arXiv:math/0304410*, 2003. URL https://arxiv.org/abs/math/0304410.

Peter Phan, Dhruv Agarwal, Kavitha Srinivas, Horst Samulowitz, Pavan Kapanipathi, and Andrew McCallum. MiGrATe: Mixed-policy GRPO for adaptation at test-time. *arXiv preprint arXiv:2508.08641*, 2025. URL https://arxiv.org/abs/2508.08641.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

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. URL https://arxiv.org/abs/2402.03300.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023. URL https://arxiv.org/abs/2303.11366.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 9229–9248. PMLR, 2020. URL https://proceedings.mlr.press/v119/sun20b.html.

Yu Sun, Xinhao Li, Karan Dalal, Chloe Hsu, Sanmi Koyejo, Carlos Guestrin, Xiaolong Wang, Tatsunori Hashimoto, and Xinlei Chen. Learning to (learn at test time). *arXiv preprint arXiv:2310.13807*, 2023. URL https://arxiv.org/abs/2310.13807.

Anja Surina, Amin Mansouri, Lars Quaedvlieg, Amal Seddas, Maryna Viazovska, Emmanuel Abbe, and Caglar Gulcehre. Algorithm discovery with LLMs: Evolutionary search meets reinforcement learning. *arXiv preprint arXiv:2504.05108*, 2025. URL https://arxiv.org/abs/2504.05108.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. ThetaEvolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, pp. 12028–12068, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/3339f19c5fcee3ad74502947a32be9e6-Abstract-Conference.html.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, pp. 43571–43608, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/4ced59d480e07d290b6f29fc8798f195-Paper-Conference.pdf.

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. *arXiv preprint arXiv:2601.16175*, 2026.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2510.04618.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):155:1–155:47, 2025. doi: 10.1145/3748302. URL https://doi.org/10.1145/3748302.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024. URL https://arxiv.org/abs/2308.10144.

APPENDIX

A TASK DEFINITIONS AND EVALUATION

We evaluate five mathematical construction tasks from TTT-Discover (Yuksekgonul et al., 2026) and ThetaEvolve (Wang et al., 2025): three autocorrelation inequalities (AC1–AC3), Erdős' minimum overlap problem, and the maximal determinant problem (Hadamard). The scores below retain the task's original objective direction; Section 3.1 uses a higher-is-better convention for $s(x)$.

A.1 FIRST AUTOCORRELATION INEQUALITY (AC1)

For a nonzero, nonnegative step function f supported on $[-\frac{1}{4}, \frac{1}{4}]$, its autoconvolution is

$$(f * f)(t) = \int_{\mathbb{R}} f(x)\, f(t-x)\, dx.$$

AC1 minimizes the ratio (Matolcsi & Vinuesa, 2010; Barnard & Steinerberger, 2020)

$$S_{\mathrm{AC1}}(f) = \frac{\|f * f\|_\infty}{\|f\|_1^2}. \tag{7}$$

For heights $a_0, \ldots, a_{n-1}$ on n equal-width intervals, this score is

$$S_{\mathrm{AC1}}(a) = \frac{2n\, \max_k (a * a)_k}{\left(\sum_{i=0}^{n-1} a_i\right)^2}, \tag{8}$$

where $a * a$ denotes discrete convolution.

A.2 SECOND AUTOCORRELATION INEQUALITY (AC2)

For the same class of step functions, AC2 maximizes the ratio (Matolcsi & Vinuesa, 2010; Boyer & Li, 2026)

$$S_{\mathrm{AC2}}(f) = \frac{\|f * f\|_2^2}{\|f * f\|_1 \|f * f\|_\infty}. \tag{9}$$

We compute the squared L^2 norm by integrating over each linear segment of the autoconvolution. A segment of width Δ with endpoint values u and v contributes

$$\frac{\Delta}{3}\left(u^2 + uv + v^2\right).$$

A.3 THIRD AUTOCORRELATION INEQUALITY (AC3)

AC3 optimizes over real-valued step functions f satisfying

$$\mathrm{supp}(f) \subseteq \left[-\tfrac{1}{4}, \tfrac{1}{4}\right], \qquad \int_{-1/4}^{1/4} f(x)\, dx \neq 0.$$

The heights may be positive or negative. The objective is to minimize (Wang et al., 2025)

$$S_{\mathrm{AC3}}(f) = \frac{\|f * f\|_\infty}{\left(\int_{-1/4}^{1/4} f(x)\, dx\right)^2}. \tag{10}$$

For heights $a_0, \ldots, a_{n-1}$ on n equal-width intervals, the score is

$$S_{\mathrm{AC3}}(a) = \frac{2n\, \max_k |(a * a)_k|}{\left(\sum_{i=0}^{n-1} a_i\right)^2}. \tag{11}$$

A.4 ERDŐS' MINIMUM OVERLAP PROBLEM

The continuous formulation (Haugland, 2016) uses a step function h and its complement g:

$$h : [-1, 1] \to [0, 1], \qquad \int_{-1}^{1} h(x)\, dx = 1,$$
$$g(x) = 1 - h(x), \qquad x \in [-1, 1].$$

We extend both functions by zero outside this interval and minimize the maximum overlap

$$S_{\text{Erdős}}(h) = \sup_{t \in \mathbb{R}} \int_{\mathbb{R}} h(x) g(x + t)\, dx. \tag{12}$$

For heights $h_0, \ldots, h_{n-1}$ on n equal-width intervals, the constraints are

$$0 \le h_i \le 1 \quad (i = 0, \ldots, n - 1), \qquad \sum_{i=0}^{n-1} h_i = \frac{n}{2}.$$

The score is

$$S_{\text{Erdős}}(h) = \frac{2}{n} \max_{k=-(n-1),\ldots,n-1} \sum_{\substack{0 \le i < n \\ 0 \le i+k < n}} h_i (1 - h_{i+k}). \tag{13}$$

A.5 MAXIMAL DETERMINANT PROBLEM (HADAMARD)

The maximal determinant problem searches for a matrix with the largest absolute determinant (Orrick et al., 2003). We use the search space

$$H \in \{-1, +1\}^{n \times n}, \qquad n = 29.$$

We maximize the normalized score used in ThetaEvolve (Wang et al., 2025):

$$S_{\text{Had}}(H) = \frac{|\det H|}{2^{28} \cdot 7^{12} \cdot 342}. \tag{14}$$

The evaluator computes the determinant using exact integer arithmetic.

Table 6: Evaluation metrics. Arrows indicate the direction of improvement. We compute best-found scores over all valid candidates evaluated up to the current search step.

Task	Score	Best-found
AC1 (↓)	Upper bound	Minimum
AC2 (↑)	Lower bound	Maximum
AC3 (↓)	Upper bound	Minimum
Erdős (↓)	Upper bound	Minimum
Hadamard (↑)	Normalized determinant	Maximum

B EXPERIMENTAL SETUP

Within each framework, w/ TTME and w/o TTT use the same fixed-weight model, task initialization, evaluator, and primary candidate budget. Each run maintains its own search archive. Section C.1 defines the memory conditions.

B.1 THETAEVOLVE FRAMEWORK

We use the ThetaEvolve (Wang et al., 2025) inference framework on AC2, AC3, and the $n = 29$ Hadamard maximal-determinant problem, with DeepSeek-R1-0528-Qwen3-8B (Distill-Qwen3-8B) frozen throughout.

Search and generation. At each step, we select 32 parents from the MAP-Elites archive with islands and generate 16 primary candidates per parent. The w/ TTME setting allocates eight candidates to each generation branch. We construct prompts from the released expert context, parent program, and `SEARCH/REPLACE` rules, then update the archive with the evaluation results. Table 7 compares settings with the released inference-only configuration.

Calls for memory maintenance, strategy selection, and strategy adaptation add to the 33,280 primary candidate attempts.

Table 7: Settings for fixed-weight Distill-Qwen3-8B on AC2, AC3, and Hadamard. Original denotes ThetaEvolve's 65-step early inference-only setting (Wang et al., 2025). Counts are per task and seed.

Setting	Original ThetaEvolve	Ours	
	w/o TTT	w/o TTT	w/ TTME
Search budget and archive			
Search steps	65	65	65
Parents × candidates per parent	$32 \times 16 = 512$	$32 \times 16 = 512$	$32 \times 16 = 512$
Primary candidates per run	33,280	33,280	33,280
Random seeds	42, 1234, 3407	3407	3407
Population / islands / archive	10,000 / 10 / 1,000	10,000 / 10 / 1,000	10,000 / 10 / 1,000
Generation			
Separate thinking-token cap	—	×	×
Output format	`SEARCH/REPLACE`	`SEARCH/REPLACE`	`SEARCH/REPLACE`
Evaluation			
Runtime / eval. timeout (s): AC2, Hadamard	300 / 350	300 / 350	300 / 350
Runtime / eval. timeout (s): AC3	60 / 70	60 / 70	60 / 70
Learning and memory			
Weight updates	×	×	×
Guideline Memory	×	×	✓
Strategy Memory	×	×	✓
Joint memory update interval, K (steps)	n/a	n/a	2

✓: enabled; ×: disabled; —: not reported; n/a: not applicable. The original paper also reports 100-step inference runs. Code reference: ThetaEvolve, commit `7c12898f`.

B.2 TTT-DISCOVER FRAMEWORK

We evaluate Qwen3-8B and gpt-oss-120b on AC1, AC2, and Erdős using TTT-Discover (Yuksekgonul et al., 2026). Both models use the search and evaluation settings below; Table 8 compares the gpt-oss-120b settings with the original configuration.

Search and generation. PUCT selects eight parents from the reuse buffer per step, and the model generates 64 candidates per parent. TTME allocates 32 to each generation branch. We retain the released task prompts, construction-state reuse, and verifier. Each rollout produces one complete Python program. The buffer admits up to two children per parent and retains the initial states.

Completion and evaluation. Both models use vLLM with a 32,768-token context, a 50-token reserve, temperature 1.0, top-p 1.0, and no top-k filtering. For gpt-oss-120b, the released sampler allows 26,000 tokens for the prompt and thinking, then continues in the final-answer channel when needed. Qwen3-8B uses its chat template with an open `<think>` prefix and the remaining context for the response, without forced final-answer continuation. Programs have a 1,000-second runtime budget and a 1,100-second evaluator timeout; failed programs receive no retry with error feedback. Each run has 25,600 primary candidate attempts. Memory updates, strategy selection, and adaptation require additional calls in w/ TTME.

Table 8: Settings for `gpt-oss-120b` on AC1, AC2, and Erdős. Original denotes TTT-Discover with reinforcement learning (Yuksekgonul et al., 2026); both of our conditions use frozen weights. Counts are per run.

Setting	Original TTT-Discover	Ours	
	w/ TTT	w/o TTT	w/ TTME
Search budget and reuse buffer			
Search steps	50	50	50
Parents × candidates per parent	$8 \times 64 = 512$	$8 \times 64 = 512$	$8 \times 64 = 512$
Primary candidates per run	25,600	25,600	25,600
Buffer capacity / PUCT coefficient c	1,000 / 1.0	1,000 / 1.0	1,000 / 1.0
Maximum admitted children per parent	2	2	2
Generation			
Context window (tokens)	32,768	32,768	32,768
Prompt + thinking limit (tokens)	26,000	26,000	26,000
Reasoning effort	high	high	high
Forced final-answer continuation	✓	✓	✓
Output format	Full program	Full program	Full program
Evaluation and serving			
Runtime / eval. timeout (s)	1,000 / 1,100	1,000 / 1,100	1,000 / 1,100
One retry with failure feedback	×	×	×
Model serving	Tinker	vLLM	vLLM
Learning and memory			
Weight updates	✓	×	×
LoRA rank	32	n/a	n/a
Guideline Memory	×	×	✓
Strategy Memory	×	×	✓
Joint memory update interval, K (steps)	n/a	n/a	2

✓: enabled; ×: disabled; n/a: not applicable. Code reference: TTT-Discover, commit `6c40e82d`.

C MEMORY IMPLEMENTATION DETAILS

Each run starts with both memories empty. We use one modification instruction per parent ($N = 1$). Section D provides the prompt templates for distillation, consolidation, and adaptation.

C.1 MEMORY OVERVIEW

Full TTME enables both components in Table 9. Each component ablation removes one memory component and retains the other.

Table 9: Construction and use of the two memories. We update both from evaluated search outcomes every $K = 2$ steps.

Aspect	**Guideline Memory**	**Strategy Memory**
Input evidence	Valid outcomes and execution or validity failures collected across parents.	Evaluated parent–child pairs in which the child is valid and improves on the parent.
Model input	Outcome summaries and the current guidelines.	Parent and child code, their diff, and evaluation results.
Extracted content	An operational lesson: a trigger, a mechanism, and a concrete action.	An experience note: the algorithmic idea expressed by a change and its application conditions.
Memory update	Merge new lessons with current guidelines, combine duplicates, and retain at most 24.	Group notes into strategy families. Create an entry for a new family, or consolidate new notes with the existing entry.
Use during generation	Include current guidelines in the prompt for both generation branches.	Select a strategy for the parent, then adapt its entry into a modification instruction for Strategy-Guided Generation.

C.2 MEMORY UPDATES

We update both memories every $K = 2$ search steps using outcomes collected since the previous update.

Guideline Memory.

1. **Summarize recent outcomes.** The outcome summary covers recent attempts across parents, including valid results, their scores, and execution or validity failures.
2. **Distill guidelines.** The same fixed-weight base model reads the outcome summary and current guidelines, then proposes up to three lessons per update, at most one for failure avoidance. Each lesson specifies a trigger, explains the mechanism, and gives a concrete action. The parser extracts the lesson text from the response.
3. **Match and merge.** For each proposed lesson, we compute its Jaccard similarity to stored guidelines using lowercased word-token sets. A similarity of at least 0.5 counts as a match: we retain the stored text, increment its recurrence count, and refresh its last reaffirmed step. An unmatched lesson enters memory with count one and the current step.
4. **Remove stale entries and enforce capacity.** Guidelines expire when more than 20 search steps have passed since their last reaffirmation. Among the remaining entries, we retain at most 24, ranked by recurrence count with recency breaking ties.
5. **Include guidelines in generation prompts.** Both Direct Generation and Strategy-Guided Generation receive the retained guidelines as shared context. Each prompt displays up to 240 characters per guideline; memory stores the full lesson text.

Strategy Memory.

1. **Select valid improvement cases.** We select evaluated parent–child pairs in which the child passes the task verifier and improves on its parent. Both scores must be finite; improvement follows the task's objective direction, consistent with $s(\text{child}) > s(\text{parent})$ in Section 3.3.
2. **Extract experience notes.** The same fixed-weight base model reads the parent and child programs, their code diff, and evaluation results. It writes an experience note describing the algorithmic idea behind the change and its application conditions. Each note retains a reference to its source case; a change containing independent ideas can yield separate notes.
3. **Group notes into strategy families.** The model compares new notes with existing strategy entries and groups notes that share an underlying algorithmic idea into the same family. It creates a new family for a distinct idea and preserves implementation differences as variants within each family.
4. **Consolidate and store strategy entries.** For each family receiving new notes, the model combines those notes with the previous entry, if one exists, to update the description of the shared idea, application conditions, and implementation variants. We store the entry, its summary, and the supporting notes with source references. Families without new notes retain their existing entries.

C.3 Memory Hyperparameters

The limits on guideline count and text length are configurable hyperparameters. Table 10 lists our settings and candidate allocation.

Table 10: Memory hyperparameters and candidate allocation. We measure K in search steps; N is the number of modification instructions per parent. Character budgets apply to prompt-visible text.

Setting	TTT-Discover	ThetaEvolve
Joint memory update interval, K (steps)	2	2
Maximum stored guidelines	24	24
Maximum proposed guidelines per update	3	3
Guideline expiry window (search steps)	20	20
Guideline matching threshold (Jaccard)	0.5	0.5
Display budget per guideline (characters)	240	240
Modification instruction budget (characters)	700	700
Candidates per parent, B	64	16
Modification instructions per parent, N	1	1
Direct Generation candidates	32	8
Strategy-Guided Generation candidates	32	8
Candidates per modification instruction	32	8

D PROMPTS

The following prompts cover five memory operations. Section D.6 provides the input layouts supplied by the caller. Prompt wording retains the original terms *memo*, *method memory*, and *direction*; subsection titles follow the Method terminology. The example outputs were regenerated with `gpt-oss-120b` from historical AC1 and AC2 inputs. Examples 1–3 are excerpts, and Example 5 is complete. The guideline label is standardized, and `[...]` marks an omission.

D.1 GUIDELINE MEMORY UPDATE

The caller appends the current guidelines and a summary containing the objective direction, best score so far, representative successes across parents, and available failure summaries. The template below shows the base instruction.

Prompt 1

You maintain a compact Guideline Memory of transferable lessons for an iterative program-search over a numerical construction problem. From the step outcomes you are given, distill AT MOST 3 guidelines — fewer, including none, when the outcomes support fewer. A guideline must CHANGE what the program does; if the failing code already satisfies your action, the action is not the fix and repeating it will reproduce the failure.

Each guideline must be ONE line in exactly this form:

`Guideline <n>: <trigger> -> <mechanism> -> <action>`

Number the guidelines consecutively from 1. Guidelines may be scientific what-drives-score lessons or failure-avoidance lessons. At most one failure-avoidance guideline. The action must be a concrete, executable instruction (an exact quantity, function, or step), never a vague suggestion. Do not quote these instructions back; emit only the guideline lines inside a <guidelines>...</guidelines> block.

Example 1

Guideline 1: normal() got an unexpected keyword argument 'dtype' → `np.random.normal` does not accept a `dtype` parameter → replace each `np.random.normal(..., dtype=DTYPE)` with `np.random.normal(...).astype(DTYPE)` (or simply omit the `dtype` argument).

D.2 Strategy Memory: Experience Note Extraction

The caller appends the complete parent and child programs, their scores formatted to ten decimal places, and their code diff.

Prompt 2

You are reading one step of an automated program-search run on a numerical optimisation problem. A parent program was modified into a child program, and the child scored better.

Answer these four questions about THIS pair, briefly and concretely:

1. What substantive algorithmic change does the child make relative to the parent? Describe the change itself, not the names of functions.
2. Where in the program does the change take effect — which loop or phase?
3. Which part of the change is most plausibly connected to the score change?
4. Write ONE short memo (2-4 sentences) that would help someone making the next modification. Do not refer to any function or variable by name; a different implementation of the same idea must still match your memo.

Most parent-to-child modifications make ONE change. If this modification genuinely bundles two or three INDEPENDENT changes -- each would work, and could have been made, without the others -- answer all four questions for each change separately, and separate the answer blocks with a line containing only ===. Interlocking parts of one idea (a new stage plus the wiring that feeds it) are ONE change, not two.

Write plain prose. Do not use section headers. Do not speculate beyond what the two programs show.

Example 2

1. When multiple convolution entries share the current maximum, the child forms the descent direction by averaging the individual gradients for all those active indices, whereas the parent simply picks the single argmax entry.
2. This averaging is performed after the convolution is obtained and before the step-size scaling, inside the inner descent loop.

[...]

4. **Memo:** if the max convolution value occurs at several positions, combine their gradients (e.g. by averaging) to obtain the sub-gradient instead of using a single arbitrary index.

D.3 STRATEGY MEMORY: STRATEGY ENTRY CONSOLIDATION

The caller places an existing-entry or new-entry prefix before the shared instruction, then appends the corresponding inputs. The family label uses its quoted string representation.

Prompt 3

Write ONE method memory for this family, with these four parts:

1. MECHANISM - what the method actually does, in its own terms.
2. OBSERVED EXECUTION MODES - every distinct way it has been applied. Two modes are distinct when they differ in WHERE they run (before the loop, inside it, replacing the update, after it), in their SCHEDULE, in how the WORKING SET is chosen, or in their UPDATE / STATE-RESET semantics. They are the SAME mode when they differ only in wording or in ordinary numeric settings such as a step size or an iteration count.
3. WHEN EACH MODE SUITS A PROGRAM - the structural condition that makes a mode applicable to the program being improved.
4. WHAT MAY BE ADAPTED - which parts a per-parent rewrite is free to change.

Rules:

- Do NOT restate the same execution mode more than once, however differently the evidence worded it.
- Do NOT delete a genuinely distinct mode to be brief. There is no length limit; a simple family may need two sentences and a complex one a long paragraph.
- Do NOT invent a numeric threshold or hard rule that the evidence does not support.
- Output the memory only. No preamble, no JSON, no headings beyond the four parts above.

Example 3

OBSERVED EXECUTION MODES

1. **Post-up-sampling L-BFGS-B + final Adam polish** – The quasi-Newton refinement is called immediately after the two up-sampling loops (which already produced a high-resolution candidate) and is followed directly by a brief Adam polish; no further up-sampling occurs.
2. **Post-up-sampling + fine-tune L-BFGS-B $\rightarrow$ extra up-sampling $\rightarrow$ final Adam polish** – The refinement is inserted after the up-sampling and the preceding Adam fine-tuning stage, then its output is fed into an additional up-sampling step before the concluding Adam polish.

D.4 STRATEGY-GUIDED SEARCH: STRATEGY SELECTION

In this template, `{k}` corresponds to the number of strategies per parent, N, rather than the update interval K. The caller requests the smaller of N and the number of available strategy families. The program and available-methods blocks are always included; other input blocks appear only when their context is available. Score history includes the current program and up to six recent ancestors, and each strategy summary contains at most 400 characters.

Prompt 4

You are steering an automated program-search run on a numerical optimisation problem. Below is the program you must improve next, how it and its recent ancestors scored, what has already been tried on this line of descent, and a list of METHODS this run has already found to work somewhere.

Select up to {k} methods from the list most likely to IMPROVE this program's score — including methods the program does not yet contain: similarity to what the program already does is not by itself a reason to choose or to avoid a method. The {k} must be DIFFERENT methods that lead to {k} different changes to this program: do not choose two methods that would make the same change. Take account of what this line of descent has already been given: repeating something that was tried here and did not move the score is usually worse than trying something this program has not seen. If several methods are equally plausible, prefer ones not already assigned to another group this step.

Return ONLY a JSON object, no prose before or after, with the ids in order of preference:

```
{"method_ids":  ["<id from the list>", "<id from the list>"]}
```

D.5 STRATEGY-GUIDED SEARCH: STRATEGY ADAPTATION

The caller appends the parent program and the selected strategy's full entry. It omits the execution-summary block when unavailable and uses the strategy's short direction only when the full entry is empty.

Prompt 5

You are steering an automated program-search run on a numerical optimisation problem. A method has been chosen for the program below. You are given the accumulated memory of that method: what it does, every distinct way it has been applied, when each way suits a program, and what may be adapted.

Write a direction for THIS specific program: 2-4 sentences saying what to change and what would count as having done it. Pick the way of applying the method that suits the structure this program actually has -- where it runs, its schedule, how its working set is chosen -- and commit to that one choice rather than listing alternatives.

The direction is the only thing the next author will see. Do not name any function or variable, do not mention scores or this run, and do not mention that a method list or a memory exists. Output the direction only -- plain prose, no preamble, no JSON, no headings.

Example 5

Insert a bounded L-BFGS-B optimisation step immediately after the second up-sampling and its fine-tuning, using the exact gradient of the negative lower-bound and enforcing non-negativity box constraints, and run it for a modest iteration budget (e.g., a few hundred steps) until the convergence tolerance is met. Then apply a brief Adam polishing pass (a few thousand steps) to re-apply the projection and clean any residual noise. The program is complete when the final candidate passed to the scorer is the result of this L-BFGS-B followed by the short Adam refinement.

D.6 INPUT TEMPLATES

The caller fills the placeholders below and appends each input block to its corresponding fixed instruction. Prompt 3 also receives a prefix before the instruction; its two variants are shown below.

Inputs for Prompt 1

```
## CURRENT playbook
{current_guideline_memory}
## THIS STEP
{step_outcome_summary}
Emit the updated lessons now.
```

Inputs for Prompt 2

````
## PARENT PROGRAM (score {parent_score})
```python
{parent_program}
```
## CHILD PROGRAM (score {child_score}, better)
```python
{child_program}
```
## DIFF (parent -> child)
```diff
{parent_to_child_diff}
```
````

Inputs for Prompt 3

Existing entry: prefix before Prompt 3

```
Here is the current method memory for the family
{family_label_repr}, and the notes that arrived at this boundary.
Rewrite the memory so it covers both.  Keep every execution mode
already in it unless a new note shows it was wrong.
```

Inputs after Prompt 3

```
## CURRENT MEMORY
{current_strategy_entry}
### new note 1
{experience_note_1}
### new note 2
{experience_note_2}
...
```

New entry: alternative prefix before Prompt 3

```
You are starting the method memory for the family
{family_label_repr} from nothing.
```

For a new entry, append only the numbered new notes and omit the `CURRENT MEMORY` block. The number of notes varies.

Inputs for Prompt 4

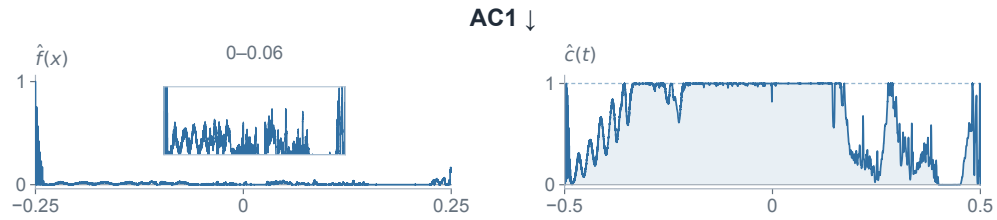
```
## PROGRAM TO IMPROVE
{parent_program}
## WHAT THIS PROGRAM ACTUALLY EXECUTED (self-reported)
{self_reported_execution_summary}
## SCORE OF THIS PROGRAM AND ITS RECENT ANCESTORS
{Higher_or_LOWER} is better.  {scores_most_recent_first} (most
recent first)
## WHAT THIS LINE OF DESCENT HAS ALREADY BEEN GIVEN
{lineage_assignment_history}
## WHAT HAS GONE WRONG BEFORE
{memory_context}
## METHODS ALREADY ASSIGNED TO OTHER GROUPS THIS STEP
- {already_assigned_method_1}
- {already_assigned_method_2}
...
## METHODS THIS RUN HAS FOUND
- id `{method_id_1}`:  {strategy_summary_1}
- id `{method_id_2}`:  {strategy_summary_2}
...
```

Inputs for Prompt 5

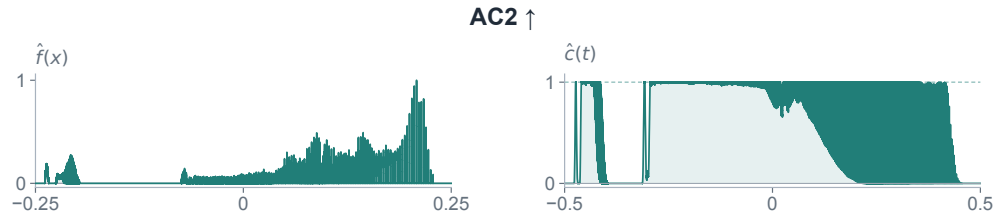
```
## PROGRAM TO IMPROVE
{parent_program}
## WHAT THIS PROGRAM ACTUALLY EXECUTED (self-reported)
{self_reported_execution_summary}
## THE CHOSEN METHOD'S ACCUMULATED MEMORY
{strategy_label_or_id}
{full_strategy_entry_or_fallback_direction}
```

E Discovered Constructions

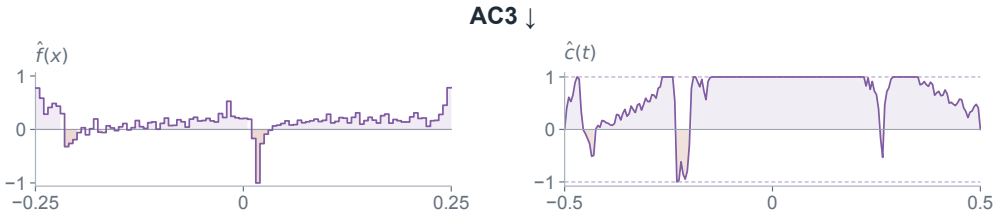
Figure 6 shows constructions for the five tasks defined in Appendix A.



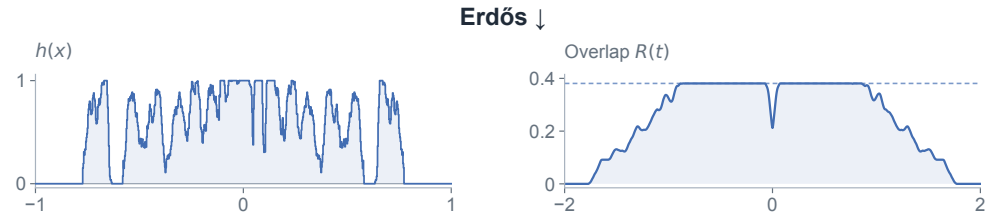
(a) *gpt-oss-120b* produced a 32 000-bin AC1 construction with an objective value of 1.502745082. The plots show the normalized step function $\hat{f}(x)$ (left) and autoconvolution $\hat{c}(t)$ (right).



(b) *gpt-oss-120b* produced a 65 536-bin AC2 construction with an objective value of 0.960784. The plots show the normalized step function $\hat{f}(x)$ (left) and autoconvolution $\hat{c}(t)$ (right).

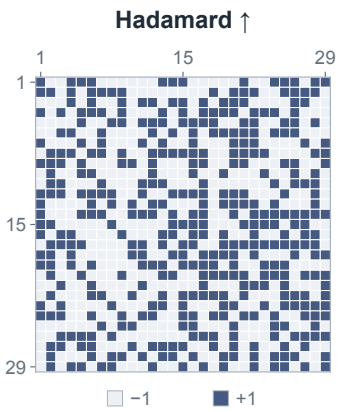


(c) *Distill-Qwen3-8B* produced a 100-bin AC3 construction with an objective value of 1.464224300. The plots show the normalized step function $\hat{f}(x)$ (left) and autoconvolution $\hat{c}(t)$ (right).



(d) *gpt-oss-120b* produced the 801-bin Erdős construction with a maximum overlap of 0.380861541. The left plot shows its density $h(x)$ on $[-1, 1]$, and the right plot shows the corresponding overlap $R(t)$.

Figure 6: Discovered constructions for AC1–AC3 and Erdős. The AC1 inset shows normalized heights 0–0.06. Arrows indicate the direction of improvement.



(e) *Distill-Qwen3-8B*, with memory enabled, produced the 29×29 sign matrix H, attaining an objective value of 0.579938205. Dark and light cells denote $+1$ and -1, respectively.

Figure 6: Discovered constructions (continued). Panel (e) preserves the original row and column order of the Hadamard construction.