EVOMEM: MEMORY-AUGMENTED EVOLUTION FOR CODE OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Successful mutation strategies in evolutionary code search may contain reusable knowledge that is useful beyond a single run, and in some cases may transfer across related tasks and domains. However, existing LLM-driven evolutionary frameworks largely discard such knowledge, repeatedly rediscovering similar ideas and limiting opportunities for cross-run and cross-task learning. We introduce EvoMem, a mechanism for transferring mutation knowledge across LLM-based evolutionary program-search runs. EvoMem converts successful mutation events into structured, task-aware advice for future runs. It operates in two phases: after each run, it extracts and stores promising ideas with provenance, and during subsequent evolution, it retrieves a small set of relevant instructions based on the current task and program context to guide mutation. Across geometric optimization, multi-hop question answering, GPU kernel optimization, and related benchmarks, our experiments show positive average improvements in target metrics or search speed for most evaluated settings, while also revealing variability across tasks. Overall, EvoMem provides evidence that persistent memory can reduce some redundant exploration and improve the reuse and adaptation of successful strategies in LLM-driven evolutionary search.

1 INTRODUCTION

LLM-based evolutionary code search has become a promising way to automate program improvement. Systems such as AlphaEvolve combine LLM-generated code mutations with evolutionary selection: they retrieve candidate programs, prompt an LLM to propose a modification, evaluate the result, and retain successful variants in a population (Novikov et al., 2025). This evaluator-in-the-loop structure is especially attractive for research and optimization tasks in which proposed ideas can be expressed as code and tested automatically. AlphaEvolve has already demonstrated this pattern on matrix multiplication, mathematical search, and systems optimization, showing that LLMs can be useful mutation operators when they are embedded in a disciplined search loop.

However, current evolutionary code-search systems mostly treat each run as a self-contained search process. Successful runs produce more than a final program: they also reveal tactics such as useful decompositions, numerical stabilization tricks, pruning rules, data-layout changes, or prompting strategies that may remain valuable beyond the original task. In standard evolutionary pipelines, this knowledge is usually left implicit in the evolved programs and logs. When the system starts a new run, even on a related problem, it must often rediscover similar ideas through broad search.

Naively reusing prior artifacts is not enough. A concrete program is usually tied to a particular task, metric, and interface, while simply mixing many experiments into one population can make retrieval less precise and can expose the mutation model to irrelevant context. What is needed is a mechanism that abstracts useful mutation knowledge from past runs, preserves enough provenance to keep it auditable, and retrieves only a bounded amount of relevant advice during future mutations.

We present EvoMem, a persistent memory architecture for LLM-based evolutionary program search. EvoMem is implemented as an extension to GigaEvo, an open-source implementation of an AlphaEvolve-style optimization pipeline (Khrulkov et al., 2025). After an evolutionary run, EvoMem extracts successful mutation events into structured memory records, deduplicates and enriches them, and stores them with task context and provenance. During later evolution, it retrieves a small set of relevant memory instructions based on the current task and program context and inserts

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.

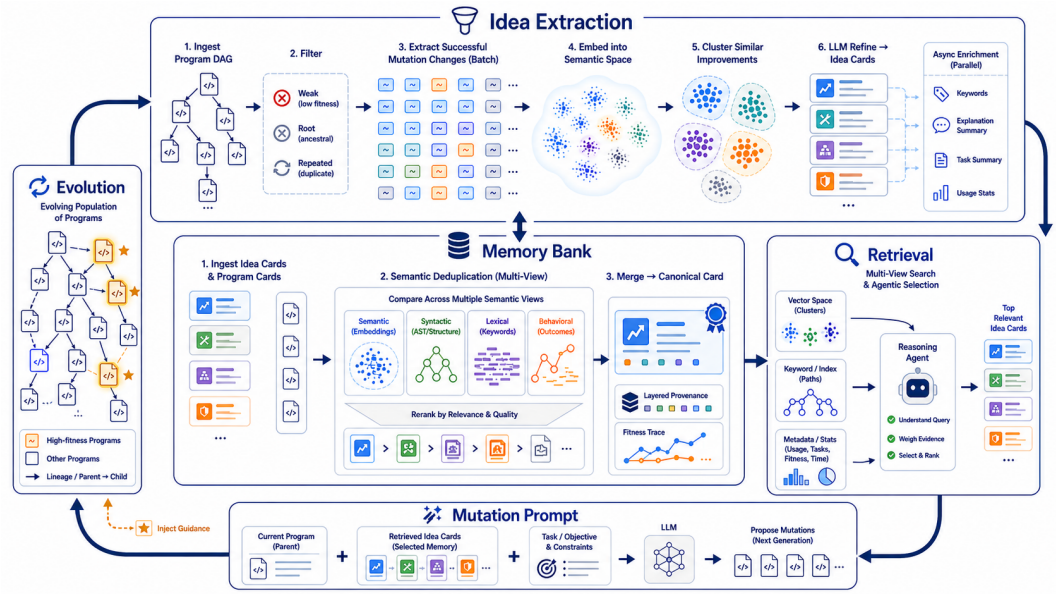

Figure 1: EvoMem pipeline. Successful mutation events from completed runs are distilled into structured memory cards with provenance, stored in a persistent memory layer, and retrieved as bounded advice for future mutations without changing the underlying evolutionary loop.

them into the mutation prompt as advice rather than as a hard constraint. This approach enables a novel mechanism of cross-task memory transfer, allowing for improved convergence on new tasks.

This paper makes the following contributions:

- We formulate cross-run transfer of mutation knowledge in LLM-based evolutionary code search and introduce EvoMem, a mechanism that extracts successful mutation strategies from prior runs and reuses them in future search.
- We describe a two-phase memory pipeline: an offline write phase that extracts, filters, deduplicates, and stores successful ideas with provenance, and an online read phase that retrieves bounded task-relevant advice during mutation.
- We implement EvoMem on top of GigaEvo and evaluate it on transfer settings spanning geometric optimization, multi-hop question answering, GPU kernel optimization, scientific-code tuning, and on three mathematical problems from analysis and geometric measure theory.
- We provide empirical evidence that memory reuse can accelerate some later evolutionary runs and can support cross-problem transfer, with improvements appearing as higher target metrics, faster search, or both depending on the task.

2 Related Work

Recent AI-driven scientific-discovery systems combine code generation, retrieval, external tools, and experimental feedback to automate parts of the research workflow (Eger et al., 2025; Ren et al., 2025; Hasib et al., 2025). Within this broader landscape, the line of work most relevant to this paper is evolutionary program and code search, where systems such as AlphaEvolve and CodeEvolve iteratively modify candidate programs, evaluate them, and retain successful variants (Novikov et al., 2025; Assumpcao et al., 2025). Our focus is narrower than the general scientific-discovery literature: we study how reusable knowledge can be extracted from prior evolutionary runs and reintroduced as persistent memory in later mutations.

Further research on AlphaEvolve capabilities has shown that this approach can rediscover and improve strong existing solutions across multiple domains, including mathematical reasoning, combinatorial search, and systems optimization (Georgiev et al., 2025; Nagda et al., 2026; Li et al., 2026; Vitvitskyi

et al., 2026; Kramár et al., 2026). Recent variants also show evolutionary approaches being adapted to specialized domains such as realistic CAD generation and quantum-circuit T-count optimization, while ImprovEvolve changes the evolved program interface around proposing, improving, and perturbing candidate solutions (Elistratov et al., 2026; Fisher et al., 2026; Kravatskiy et al., 2026). Follow-up work on open or alternative implementations suggests that the search loop itself can be improved: smaller RL-trained models may outperform larger closed ensembles in some settings, task decomposition can make evolution more efficient and interpretable, and adaptive metrics, sampling, or meta-evolution can further improve performance (Wang et al., 2025; Li et al., 2025a; Liu et al., 2026b).

The closest neighboring systems are those that combine evolutionary code search with domain knowledge or explicit memory. AlphaEvolve and related systems such as ShinkaEvolve and MAPPS show that iterative program modification and evaluation can support scientific and algorithmic discovery in specialized settings (Novikov et al., 2025; Lange et al., 2025; Zhou et al., 2025). Nearby examples include GeoEvolve, which combines evolutionary search with domain knowledge, and AgenticSciML, which adds retrieval-augmented method memory to scientific machine-learning workflows (Luo et al., 2025; Jiang & Karniadakis, 2025). Most other domain-specific agent systems focus on chemistry, biology, or materials pipelines rather than on persistent cross-run memory for evolutionary code search (Xiao et al., 2024; Ghafarollahi & Buehler, 2024; Lin et al., 2023).

Agentic memory architectures such as A-MEM and General Agentic Memory likewise retrieve and update agent experience at task time (Xu et al., 2025; Yan et al., 2025). EvoMem differs by storing provenance-grounded advice distilled from successful evolutionary mutations rather than general episodic or document memory.

Tool-augmented systems are relevant insofar as they show how code generation, execution, retrieval, and reflection can be organized into reusable workflows (Ren et al., 2025; Hasib et al., 2025; Li et al., 2025b). Among them, OR-Agent is the closest comparison because it combines structured research loops with memory-based reflection, while SciDataCopilot similarly emphasizes reusable workflow generation (Liu et al., 2026a; Rao et al., 2026). Other systems such as AI Scientist, Paper2Code, CodeScientist, and MLR-Copilot demonstrate broader executable research pipelines, but they are less specifically focused on persistent memory for evolutionary mutation and cross-run reuse (Lu et al., 2024; Seo et al., 2025; Jansen et al., 2025; Li et al., 2024).

3 Memory architecture

GigaEvo follows a standard evolutionary loop: it selects parent programs, asks an LLM-based mutation operator to modify them, executes the resulting candidates, and stores their metrics. In the baseline setting, each mutation sees only local run context such as the parent code, current metrics, recent insights, and lineage. This keeps the search procedure simple and controlled, but it also means that useful tactics discovered in one run are not available in later runs. EvoMem addresses this limitation by converting successful mutation events into reusable, task-aware advice.

The design is intentionally split into two phases. The *write phase* runs after an evolution job finishes, extracts generalizable ideas from the produced programs, and stores them in a memory store. The *read phase* runs before each new mutation, retrieves a small set of relevant memory cards, and injects them into the mutation prompt. This separation is important for efficiency: expensive operations such as clustering, deduplication, and optional LLM-based enrichment happen offline after the run, while mutation-time retrieval remains a small bounded step.

3.1 General pipeline

At a high level, a completed run produces a set of programs, a post-run analysis stage extracts candidate ideas from those programs, the memory layer normalizes and stores them, and future runs query this store before proposing a new mutation. We use the following compact notation to make the data flow explicit:

$$P_r \xrightarrow{\text{extract}} C_r \xrightarrow{\text{write}} M \xrightarrow{\text{select}(p,\tau,\mu)} I_p \hookrightarrow \text{mutation prompt}. \tag{1}$$

Here P_r is the set of programs from run r, C_r is the set of cards extracted from that run, M is the persistent memory bank, and I_p is the small list of instructions selected for parent program p on task τ with metric description μ.

Conceptually, the architecture has three parts. First, a post-run analysis stage reviews completed runs and produces structured records of candidate ideas together with provenance information. Second, a persistent memory layer stores these records, merges near-duplicates, and maintains retrieval indexes. Third, a retrieval layer queries this store at mutation time. In control runs, this layer is simply inactive, so the same evolutionary pipeline can be used both with and without memory.

3.2 Insights extraction and processing

The write phase begins from mutation records attached to evolved programs. During the run, the mutation process records a list of changes; each change has a short description and, when available, a rationale for why it was introduced. After the run, the analysis stage collects the completed programs, filters out root programs and invalid outputs, and converts the remaining candidates into a normalized set of records containing fitness, generation, parentage, task context, mutation strategy, code, and improvement descriptions.

There are two analysis modes. The default mode compares new improvements against a working collection of previously extracted ideas. For each program, an LLM decides whether the recorded improvements correspond to genuinely new ideas, revisions of existing ideas, or simple reformulations of earlier descriptions. The working collection is then updated while preserving provenance: source program references are appended, the latest generation is updated, older descriptions are kept as aliases when wording changes, and new rationales are appended to the explanation history.

The fast analyzer is used when a run contains many similar improvement descriptions. It first groups semantically related candidates using DBSCAN over embedding space, then asks an LLM to refine ambiguous multi-member clusters. This produces a smaller set of reusable idea candidates while preserving the source programs and rationales as provenance. Cross-run consolidation is deferred to the memory insertion stage, where new ideas are compared against the existing corpus.

After analysis, the active memory is enriched with task summaries, keywords, and short explanation summaries. If memory-usage tracking is enabled, the system also uses the selected memory-card identities recorded on parent programs to estimate downstream impact. For a child program, the recorded contribution of a selected memory item is

$$\Delta f = f(\text{child}) - \max_{p \in \text{parents}} f(p),$$

computed only when both child and parent fitness values are valid, i.e., when both passed a task-specific validity check. The stored usage statistics keep per-task counts, all observed deltas, and the median delta. Median aggregation is used because evolutionary fitness changes are typically sparse and heavy-tailed.

3.3 Memory generation

The write pipeline stores two kinds of entries. The first represents an abstract idea or tactic, including its description, task context, explanations, provenance, and usage statistics. The second represents a high-performing program together with its fitness, code, task context, and links to the ideas associated with it. Before insertion, all incoming data are normalized into these structured forms. This normalization step is necessary because the write pipeline consumes heterogeneous inputs—LLM outputs, post-run analysis logs, and historical dumps—so missing fields and legacy aliases must be handled consistently.

Not every extracted idea is promoted into long-term memory with equal status. In addition to the full collection of extracted ideas, the system maintains a smaller snapshot of high-value ideas intended to capture the most reusable findings from a run. After the post-run analysis stage finishes, an origin-analysis pass scores each idea by both how it first appeared in the evolutionary lineage and how it shaped later descendants. For each introduction event, the analysis compares the child program with its strongest parent, with sibling programs generated from the same parent context, and with downstream descendants that inherit the idea.

The promotion rule is deliberately conservative rather than tuned to a particular benchmark. An idea is favored when it has an identifiable introduction point, improves over its strongest available parent, rarely produces worse descendants, and is supported by at least one additional signal such as recurrence across nearby programs, favorable comparison with siblings generated from the same context, or spread into later elite lineages. This filter biases memory toward ideas with lineage-level evidence of usefulness, rather than toward every change described by the LLM.

Once this curated set has been formed, the write pipeline combines the full idea collection, the filtered high-value snapshot, optional high-performing program exemplars, and usage updates. New records are then inserted into persistent memory under a conservative policy:

1. repeated observations of the same memory entity update the existing record and extend its provenance;
2. concrete program exemplars are stored without semantic merging, because they are tied to specific artifacts and fitness values;
3. abstract idea cards are merged only when they describe the same underlying mechanism, not merely the same source task.

To identify possible duplicates, EvoMem compares each incoming idea against multiple textual views of existing cards, including their mechanism descriptions, task summaries, and explanations. Candidate scores are combined by a weighted sum,

$$\mathrm{score}(c) = \sum_{q \in Q} w_q s_q(c),$$

where $s_q(c)$ is the retriever score for candidate card c under query view q. The default weights are largest on the description and description–explanation views, because these are the best signals for whether two cards describe the same mechanism rather than merely the same benchmark.

The highest-scoring candidates are passed to an LLM decision policy that chooses whether to store the new idea, discard it as redundant, or merge it with an existing entry. Ambiguous cases are retained as new cards, favoring recall over accidental loss of useful knowledge. Deduplication is therefore an optimization of corpus quality, not a precondition for preserving a potentially useful idea.

The resulting memory corpus is stored persistently together with provenance and usage statistics, and retrieval indexes are refreshed so that later runs can query the updated memory bank.

3.4 Memory retrieval and usage

Before each mutation, EvoMem optionally augments the candidate context with retrieved memory. In control runs this augmentation is empty; in memory-enabled runs, a retrieval module selects a bounded set of relevant memory cards. This keeps the evolutionary operators fixed across conditions: parent selection, mutation generation, validation, and scoring are unchanged, while memory enters only as an additional advice channel.

The retrieval query is derived from the current task description, metric descriptions, mutation mode, parent program, and local mutation context. EvoMem then selects a small number of concise, actionable ideas. Thus, the intervention is localized: the current program state is used to retrieve advice, but program generation and evaluation remain unchanged.

Before retrieving individual ideas, EvoMem performs an additional task-scope filtering step. An LLM is shown the current task together with the task descriptions present in memory and selects the source tasks that are potentially relevant. Subsequent retrieval is then performed over ideas associated with those selected tasks, rather than over the entire memory corpus. This keeps retrieval focused while still allowing transfer from related but non-identical problems.

Within the selected task scope, EvoMem ranks memory cards using lexical and embedding-based similarity over several semantic views: the card's mechanism description, the source task context, the explanation of why the idea helped, and selected compositions of these fields. This decomposition is important because a useful memory card may match the present mutation through different signals: the mechanism it describes, the task constraints under which it was observed, or the explanation of why it improved performance.

The selection policy merges evidence from multiple retrieval channels and may issue a focused follow-up query when the initial evidence is insufficient. Both the number of retrieval iterations and the number of returned cards are fixed in advance, so retrieval remains a bounded mutation-time operation rather than an open-ended research process.

The retrieval step returns two coupled outputs: a concise advice block inserted into the mutation prompt, and the identities of the selected memory cards. These identities are recorded with the resulting candidate so that, after evaluation, the system can estimate associations between memory use and downstream fitness changes. This record is what enables the usage-tracking pass in the next write phase.

Finally, the retrieved text is inserted into the mutation prompt as a dedicated memory section alongside metrics, recent insights, lineage, evolutionary statistics, and any task-specific artifact context. The mutation agent therefore sees memory as advice rather than as a hard constraint. It may reuse the retrieved tactics when they are relevant, while every candidate program is still evaluated by the same validators and metrics as in non-memory runs.

3.5 Design properties

Three design choices are especially important for experiments. First, memory is *optional*: control and treatment runs differ only in whether the memory advice channel is populated. Second, retrieval is *bounded*: mutation-time memory access is limited by fixed item and iteration budgets. Third, the system is *auditable*: selected memory cards are recorded with the candidates that used them, and post-run analysis can trace those links back to the relevant source programs. Together, these properties make it easier to attribute measured gains to the memory mechanism rather than to unrelated changes elsewhere in the pipeline.

This design is intended to limit negative transfer. EvoMem does not replace the core evolutionary process with memory replay: parent selection, mutation generation, validation, evaluation, and population updates remain the same as in the baseline system. Memory enters only as a bounded suggestion channel inside the mutation context, and retrieved ideas are not enforced as constraints. As a result, the search can ignore irrelevant memories, continue proposing novel mutations, and select against candidates that overfit to stale or task-mismatched advice. The memory layer therefore aims to bias exploration toward previously useful directions without eliminating the discovery of new strategies.

4 Methodology

For memory-mechanism evaluation, we build memories from prior runs but exclude same-benchmark memories at test time. The memory-generation set spans geometry, question answering, and code optimization. It includes Circle Packing (26), Heilbronn-style point placement, Kissing Number (11D), HotpotQA (Yang et al., 2018), HoVer (Jiang et al., 2020), GSM8K (Cobbe et al., 2021), selected AlgoTune tasks {QP, Markowitz, LP-Box, Chebyshev Center, Feedback Controller, LQR, Lyapunov} (Press et al., 2025), and selected KernelBench kernels {L1 P56, L1 P82, L1 P86, L1 P87, L2 P57} (Ouyang et al., 2025). Only part of this set is drawn from established benchmark suites, including AlgoTune for scientific-code tuning and KernelBench for GPU-kernel optimization; the remaining tasks cover geometric optimization and reasoning settings used to probe transfer more broadly.

The evaluation set contains tasks from the same broad families but is treated as a separate testing set for cross-task memory reuse. It includes Circle Packing (26 and 32), Hexagon Packing, Heilbronn-style point placement, Kissing Number (12D), HoVer, HotpotQA, AlgoTune Power Control, AlgoTune Kalman, and KernelBench kernels including L1 P66 and L3 P21, as well as three mathematical problems from Georgiev et al. (2025): Second Autocorrelation, Minimum Autocorrelation, and Kakeya Parallelograms (64). Circle packing and hexagon packing ask the system to arrange shapes as densely as possible inside a bounded region. Heilbronn-style tasks optimize point placements by maximizing the minimum triangle area induced by triples of points. Kissing Number evaluates dense high-dimensional sphere configurations. HotpotQA and HoVer test multi-hop question answering and evidence integration, with QA performance measured by exact match on a fixed test set of 200 samples for each benchmark. AlgoTune evaluation tasks measure speedups of scientific and

algorithmic routines under correctness constraints, and KernelBench evaluation tasks measure low-level GPU-kernel optimization while preserving correctness. The two autocorrelation problems ask for a nonnegative function that maximizes a constant in an autocorrelation inequality, and Kakeya Parallelograms asks for a placement of 64 slanted parallelograms that minimizes the area of their union.

All components of the GigaEvo mutation framework used Gemini 3 Flash as the LLM backbone (Gemini Team, 2025). Prompt-evolution experiments with LLM-based evaluation or generation components, including HotpotQA, HoVer, and GSM8K, used Qwen3-8B as the tested LLM (Yang et al., 2025).

For reproducibility, we provide the full EvoMem implementation code with the submission. Unless otherwise noted, LLM calls used temperature 0.7, the MAP-Elites population used a single island, memory retrieval returned at most $k = 3$ cards per mutation, and embedding-based retrieval used the all-MiniLM-L6-v2 embedding model. To gather statistics, we used three different seeds for each experiment. Across the reported experiments, the evolution phase used on average 7,192,260 input tokens and 892,674 output tokens, while the post-run memory-generation phase used on average 683,470 input tokens and 25,225 output tokens. At the idea level, each memory bank session extracted 99.71 ± 38.68 generated ideas from one completed evolution run, and selected 12.76 ± 7.51 ideas for insertion into the memory bank after filtering and processing.

We build the memory bank from the memory-generation set and then test EvoMem on the evaluation set. When evaluating a benchmark, we exclude all memories produced from that benchmark from the memory bank. Thus, the memory-enabled run can only retrieve memories from other benchmarks, including related tasks, and never from the benchmark currently being tested. For each evaluation benchmark, we compare a baseline condition without memory against the same evolutionary pipeline with retrieved memories enabled. The comparison measures whether reusable ideas extracted from prior tasks improve final performance or search speed on held-out tasks.

5 RESULTS

We evaluate EvoMem on cross-problem transfer experiments spanning geometric optimization, multi-hop question answering, GPU kernel optimization, scientific-code tuning, and on three mathematical problems. In each setting, we compare a baseline target-benchmark run against a memory-enabled run whose memory bank excludes memories from that target benchmark. The results are reported in Table 1. We used relative gain and candidate-efficiency ratio (CER) as primary metrics. Relative gain is calculated separately for each matched pair of runs as $g_i = (F_i^{\text{memory}} - F_i^{\text{baseline}})/|F_i^{\text{baseline}}|$. The reported average, minimum, and maximum gains summarize the resulting pairwise g_i values. Candidate-efficiency ratio (CER) is the ratio between the number of evaluated candidates required by the baseline run and the memory-enabled run to reach the baseline run's final best score. Values greater than 1 indicate that the memory-enabled run reaches the reference score using fewer candidate evaluations. If the memory-enabled run does not reach the baseline final best score, we assign a value of 0. Figure 2 shows the search trajectories on the three mathematical problems.

Table 1: Summary of target-metric relative gains and CER for memory-enabled runs across evaluated benchmarks.

Benchmark	Average gain	Min gain	Max gain	Average CER	Min CER	Max CER
Circle Packing (32)	5.21% ±3.38%	0.48%	9.29%	9.26 ±6.80	3.29	22.00
Circle Packing (26)	5.88% ±5.93%	0.28%	17.13%	5.06 ±3.08	0.31	9.00
Heilbronn	3.86% ±3.54%	0.00%	8.55%	6.01 ±7.20	0.31	16.17
Kissing Number (12D)	0.00% ±0.00%	0.00%	0.00%	6.78 ±0.62	6.17	7.40
Hexagon Packing	3.69% ±2.68%	0.38%	6.93%	5.75 ±4.95	0.75	12.50
HoVer	2.42% ±0.14%	2.28%	2.56%	2.71 ±0.79	1.92	3.50
AlgoTune	7.90% ±1.67%	6.22%	9.57%	8.58 ±7.76	0.82	16.33
KernelBench	16.89% ±25.11%	0.00%	65.93%	3.44 ±3.38	0.05	9.00
HotpotQA	11.79% ±8.38%	5.16%	27.41%	5.75 ±4.71	1.27	12.00
Second Autocorrelation	1.05% ±0.25%	0.76%	1.21%	1.45±0.78	0.56	2.03
Minimum Autocorrelation	0.78% ±1.54%	-0.47%	2.50%	2.27±1.81	0.84	4.30
Kakeya Parallelograms (64)	0.93% ±1.16%	-0.13%	2.18%	1.32±0.58	0.70	1.85

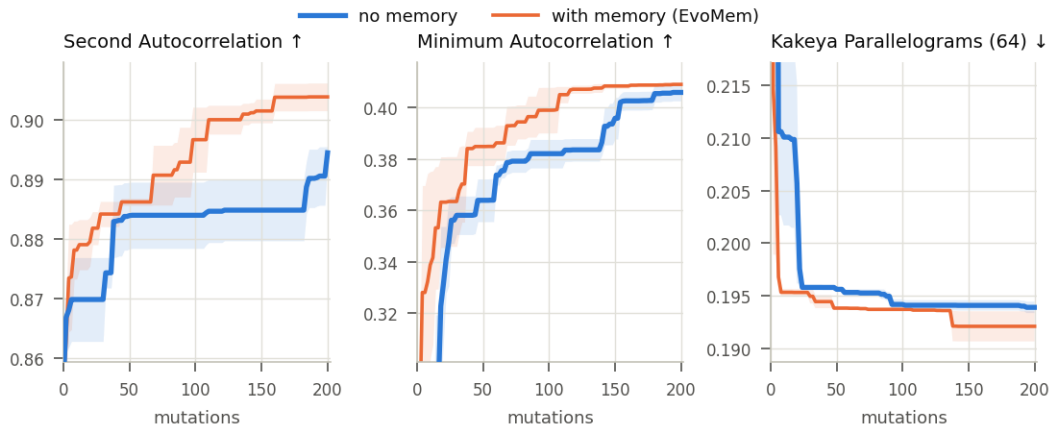

Figure 2: Mean best-so-far value without memory and with EvoMem, on the three mathematical problems of Section 4. Shaded: ± 1 standard error of the mean. An upward arrow marks a score that is maximized, a downward arrow one that is minimized. Table 1 gives these problems in its own measures.

Table 1 shows that EvoMem is associated with better target metrics, faster search, or both across several benchmark families. The strongest convergence improvements appear in Heilbronn, Kissing Number, HotpotQA, and Circle Packing, while AlgoTune and KernelBench show positive mean gains with higher variability. Kissing Number illustrates an acceleration-only case: final quality is unchanged on average, but memory-enabled runs reach the baseline best score earlier. Absolute target-metric values are reported in Appendix E.

6 Ablation Studies

We conduct a controlled ablation study on Circle Packing (32) to separate the effect of the complete EvoMem mechanism from simpler forms of prompt and memory augmentation. Circle packing provides a well-understood optimization objective with a known reference solution. Because each condition requires a fresh evaluator-in-the-loop evolutionary run, we allocate the experimental budget to five independent trials per condition on this task rather than to fewer trials spread across several benchmarks. We compare the full default EvoMem configuration with no memory, random memory, generic advice, memory without task filtering, and retrieval-only memory without the complete re-ranking procedure.

Near-optimal improvements are difficult to interpret from raw fitness alone: scores around 2.8 are comparatively easy to reach, whereas small absolute gains become increasingly consequential near the best-known fitness $f^* = 2.9370$. We therefore also report the percentage of the remaining target gap closed. For a fitness value f, this quantity is

$$\mathrm{GapClosed}(f) = 100\,\frac{f - \bar{f}_{\mathrm{base}}}{f^* - \bar{f}_{\mathrm{base}}}, \qquad \bar{f}_{\mathrm{base}} = 2.8151. \tag{2}$$

Under this normalization, the no-memory mean is 0% and the best-known fitness is 100%. Standard deviations are transformed by the same fixed scale; consequently, the baseline can have zero mean gap closure while retaining a nonzero standard deviation.

Table 2 shows that arbitrary memory content or retrieval alone does not reliably reproduce EvoMem's gains. Random memory closes only 5.6% of the target gap on average, and retrieval-only memory closes 18.7% while exhibiting the largest variance; the one run that reaches the best-known fitness is therefore not representative of its overall performance. Generic advice and removing task filtering perform better on average, closing 53.7% and 46.2% of the gap, respectively, but both remain below the full method, and the no-task-filtering condition is particularly unstable.

In contrast, full EvoMem attains a mean fitness of 2.9338 ± 0.0044 and closes $97.4\% \pm 3.6\%$ of the target gap. Even its worst run, at 2.9278, exceeds the no-memory baseline mean of 2.8151. The

Table 2: Circle Packing (32) ablation results over five independent trials per condition. Fitness and target-gap closure are reported as mean ± standard deviation; best and worst denote the extrema across the five trials. Ablation runs use an independent set of trials and are not included in the data reported in Table 1.

Condition	Fitness (mean ± SD)	Best run	Worst run	Target gap closed (mean ± SD)
Baseline (no memory)	2.8151 ± 0.0683	2.9061	2.7388	$0.0\% \pm 56.0\%$
Random memory	2.8219 ± 0.0681	2.8900	2.7538	$5.6\% \pm 55.9\%$
Generic advice	2.8805 ± 0.0530	2.9228	2.7877	$53.7\% \pm 43.5\%$
No task filtering	2.8714 ± 0.1148	2.9352	2.6683	$46.2\% \pm 94.2\%$
Retrieval-only memory	2.8379 ± 0.2210	2.9370	2.4426	$18.7\% \pm 181.3\%$
Full EvoMem	2.9338 ± 0.0044	2.9370	2.9278	$97.4\% \pm 3.6\%$

simultaneous improvement in average fitness and run-to-run consistency indicates that the strongest and most reliable gains arise from the complete mechanism—constructing reusable knowledge, selecting task-relevant memories, and integrating them into the evolutionary process—rather than from additional context or retrieval machinery alone.

7 Conclusion

We introduced EvoMem, a persistent memory architecture for LLM-based evolutionary code optimization. EvoMem converts successful mutation events from previous runs into structured, auditable memory cards and retrieves a bounded set of relevant cards during later mutations. This lets an evolutionary search process reuse previously discovered tactics without replaying old programs directly or changing the evaluator, parent-selection policy, or fitness computation.

Across the evaluated geometry, reasoning, scientific-code, and GPU-kernel benchmarks, EvoMem is associated with improved efficiency or final target metrics in most of the tested settings. The results suggest that some useful mutation knowledge is more general than the single program in which it first appeared: strategies discovered in one run can help guide exploration in related and sometimes substantially different domains. These gains provide initial support for the central hypothesis of this work, namely that LLM-driven evolution can benefit from persistent cross-run memory because it may reduce redundant rediscovery and bias search toward previously productive directions while preserving evaluator-based selection.

As evolutionary agents become more capable, the artifacts produced by a run include not only final code but also reusable design decisions, optimization heuristics, and failure-avoidance patterns. Capturing these artifacts in a compact and retrievable form may provide a practical path toward evolutionary systems that accumulate experience across tasks instead of starting each optimization process from scratch.

8 Limitations and Future Work

The benefits of EvoMem vary across runs, as reflected in the wide range between metrics values observed for several benchmarks. This variance is partly inherent to evolutionary search: useful exploration often requires high-temperature LLM sampling, so repeated runs can follow different trajectories. At the same time, too little diversity, or overly rigid use of retrieved memory, can cause premature convergence. EvoMem is also sensitive to the quality of its memory pipeline, including idea extraction, clustering, summarization, retrieval, and their associated hyperparameters. Poorly formed or poorly retrieved memories can reduce the relevance of mutation advice and limit downstream gains.

The present study also uses a relatively small memory bank, a fixed memory influence during mutation, and a simple code-evolution setting. Future work should scale memory across more runs and domains, adapt memory reliance to the task and population state, and study whether persistent memory can reduce the cost of repeated LLM calls, execution, validation, and selection. Another important direction is repository-scale evolution, where memories may capture project conventions, interface constraints, recurring bug patterns, and cross-file optimization strategies.

AI USE STATEMENT

In this work, we used generative AI tools for aid or polish writing and to draft sections of the paper. We have not used generative AI tools for generating synthetic datasets, developing theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in writing proofs, propositions, or refining hypotheses, designing or providing feedback on research methodology or experiments, implementing methods, assisting with translation, cleaning and reformatting datasets, supporting qualitative and thematic data analysis or interpreting results.

Additionally, we used generative AI tools for creating scientific figures and images, drafting parts of a research paper, and editing a research paper to improve readability. We have reviewed all AI-assisted work. All sections of the paper were manually reviewed by at least two authors after the paper draft was completed and polished. All citations were also manually verified. Generated images were validated against the data they were based on. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Henrique Assumpcao, Diego Ferreira, Leandro Augusto Lacerda Campos, and Fabricio Murai. CodeEvolve: an open source evolutionary coding agent for algorithmic discovery and optimization, 2025. URL https://arxiv.org/abs/2510.14150.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Steffen Eger, Yong Cao, Jennifer D'Souza, Andreas Geiger, Christian Greisinger, Stephanie Gross, Yufang Hou, Brigitte Krenn, Anne Lauscher, Yizhi Li, Chenghua Lin, Nafise Sadat Moosavi, Wei Zhao, and Tristan Miller. Transforming science with large language models: A survey on AI-assisted scientific discovery, experimentation, content generation, and evaluation, 2025. URL https://arxiv.org/abs/2502.05151.

Maksim Elistratov, Marina Barannikov, Gregory Ivanov, Valentin Khrulkov, Anton Konushin, Andrey Kuznetsov, and Dmitrii Zhemchuzhnikov. CADEvolve: Creating realistic CAD via program evolution, 2026. URL https://arxiv.org/abs/2602.16317.

Daniil Fisher, Valentin Khrulkov, Mikhail Saygin, Ivan Oseledets, and Stanislav Straupe. LLM-Guided evolutionary search for algebraic T-Count optimization, 2026. URL https://arxiv.org/abs/2603.29894.

Gemini Team. Gemini 3 Flash: frontier intelligence built for speed. Google Keyword Blog, 2025. URL https://blog.google/products-and-platforms/products/gemini/gemini-3-flash/. Accessed 2026-05-06.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *ArXiv*, abs/2511.02864, 2025. URL https://api.semanticscholar.org/CorpusId:282758479.

Alireza Ghafarollahi and Markus J. Buehler. ProtAgents: Protein discovery via large language model multi-agent collaborations combining physics and machine learning. *Digital Discovery*, 3(7):1389–1409, 2024. doi: 10.1039/D4DD00013G. URL https://doi.org/10.1039/D4DD00013G.

Md Musaddaqul Hasib, Sumin Jo, Harsh Sinha, Jifeng Song, Arun Das, Zhentao Liu, Hugh Galloway, Huey Huang, Kexun Zhang, Shou-Jiang Gao, Yu-Chiao Chiu, Lei Li, and Yufei Huang. A process-centric survey of AI for scientific discovery through the exhyte framework, 2025. URL https://doi.org/10.21203/rs.3.rs-8370059/v1.

Peter Jansen, Oyvind Tafjord, Marissa Radensky, Pao Siangliulue, Tom Hope, Bhavana Dalvi Mishra, Bodhisattwa Prasad Majumder, Daniel S. Weld, and Peter Clark. CodeScientist: End-to-end semi-automated scientific discovery with code-based experimentation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 13370–13467, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.692. URL https://aclanthology.org/2025.findings-acl.692/.

Qile Jiang and George Karniadakis. AgenticSciML: Collaborative multi-agent systems for emergent discovery in scientific machine learning, 2025. URL https://arxiv.org/abs/2511.07262.

Yichen Jiang, Shikha Bordia, Zheng Zhong, Charles Dognin, Maneesh Singh, and Mohit Bansal. HoVer: A dataset for many-hop fact extraction and claim verification. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 3441–3460, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.309. URL https://aclanthology.org/2020.findings-emnlp.309/.

Valentin Khrulkov, Andrey Galichin, Denis Bashkirov, Dmitry Vinichenko, Oleg Travkin, Roman Alferov, Andrey Kuznetsov, and Ivan Oseledets. GigaEvo: An open source optimization framework powered by llms and evolution algorithms, 2025. URL https://arxiv.org/abs/2511.17592.

János Kramár, Joshua Engels, Zheng Wang, Bilal Chughtai, Rohin Shah, Neel Nanda, and Arthur Conmy. Building production-ready probes for Gemini. *ArXiv*, abs/2601.11516, 2026. URL https://api.semanticscholar.org/CorpusId:284860299.

Alexey Kravatskiy, Valentin Khrulkov, and Ivan Oseledets. ImprovEvolve: Ask AlphaEvolve to improve the input solution and then improvise, 2026. URL https://arxiv.org/abs/2602.10233.

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

Annan Li, Chufan Wu, Zengle Ge, Yee Hin Chong, Zhinan Hou, Lizhe Cao, Cheng Ju, Jianmin Wu, Huaiming Li, Haobo Zhang, et al. The FM agent. *arXiv preprint arXiv:2510.26144*, 2025a.

Ruochen Li, Teerth Patel, Qingyun Wang, and Xinya Du. MLR-Copilot: Autonomous machine learning research based on large language models agents, 2024. URL https://arxiv.org/abs/2408.14033.

Zongwei Li, Zhonghang Li, Zirui Guo, Xubin Ren, and Chao Huang. DeepCode: Open agentic coding, 2025b. URL https://arxiv.org/abs/2512.07921.

Zun Li, John Schultz, Daniel Hennes, and Marc Lanctot. Discovering multiagent learning algorithms with large language models. *ArXiv*, abs/2602.16928, 2026. URL https://api.semanticscholar.org/CorpusId:285788128.

Zeming Lin, Halil Akin, Roshan Rao, Brian Hie, Zhongkai Zhu, Wenting Lu, Nikita Smetanin, Robert Verkuil, Ori Kabeli, Yaniv Shmueli, Allan dos Santos Costa, Maryam Fazel-Zarandi, Tom Sercu, Salvatore Candido, and Alexander Rives. Evolutionary-scale prediction of atomic-level protein structure with a language model. *Science*, 379(6637):1123–1130, 2023. doi: 10.1126/science.ade2574. URL https://doi.org/10.1126/science.ade2574.

Qi Liu, Ruochen Hao, Can Li, and Wanjing Ma. OR-Agent: Bridging evolutionary search and structured research for automated algorithm discovery, 2026a. URL https://arxiv.org/abs/2602.13769.

Shu Liu, S. Agarwal, M. Maheswaran, M. Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, E. Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alexandros G. Dimakis, Koushik Sen, Matei A. Zaharia, and Ion Stoica. EvoX: Meta-evolution for automated discovery. *ArXiv*, abs/2602.23413, 2026b. URL https://api.semanticscholar.org/CorpusId:286171173.

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

Peng Luo, Xiayin Lou, Yu Zheng, Zhuo Zheng, and Stefano Ermon. GeoEvolve: Automating geospatial model discovery via multi-agent large language models, 2025. URL https://arxiv.org/abs/2509.21593.

Ansh Nagda, Prabhakar Raghavan, and Abhradeep Thakurta. Reinforced generation of combinatorial structures: Ramsey numbers, 2026. URL https://arxiv.org/abs/2603.09172.

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, 2025. URL https://arxiv.org/abs/2506.13131.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel K. Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, Jisun Park, Nathanael Bosch, Eli Meril, Albert Steppi, Arman Zharmagambetov, Fangzhao Zhang, David Perez-Pineiro, Alberto Mercurio, Ni Zhan, Talor Abramovich, Kilian Lieret, Hanlin Zhang, Shirley Huang, Matthias Bethge, and Ofir Press. AlgoTune: Can language models speed up general-purpose numerical programs?, 2025. URL https://arxiv.org/abs/2507.15887.

Jiyong Rao, Yicheng Qiu, Jiahui Zhang, Juntao Deng, Shangquan Sun, Fenghua Ling, Hao Chen, Nanqing Dong, Zhangyang Gao, Siqi Sun, Yuqiang Li, Dongzhan Zhou, Guangyu Wang, Lijun Wu, Conghui He, Xuhong Wang, Jing Shao, Xiang Liu, Yu Zhu, Mianxin Liu, Qihao Zheng, Yinghui Zhang, et al. SciDataCopilot: An agentic data preparation framework for AGI-driven scientific discovery, 2026. URL https://arxiv.org/abs/2602.09132.

Shuo Ren, Pu Jian, Zhenjiang Ren, Chunlin Leng, Can Xie, and Jiajun Zhang. Towards scientific intelligence: A survey of LLM-based scientific agents, 2025. URL https://arxiv.org/abs/2503.24047.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2Code: Automating code generation from scientific papers in machine learning, 2025. URL https://arxiv.org/abs/2504.17192.

Alex Vitvitskyi, Michael Boratko, Matej Grcic, Razvan Pascanu, Deep Shah, and Petar Veličković. Mining generalizable activation functions. *ArXiv*, abs/2602.05688, 2026. URL https://api.semanticscholar.org/CorpusId:285303657.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, E. Xu, Liliang Ren, Xinyu Yang, Zeying Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, S. Du, and Yelong Shen. ThetaEvolve: Test-time learning on open problems. *ArXiv*, abs/2511.23473, 2025. URL https://api.semanticscholar.org/CorpusId:283438865.

Yihang Xiao, Jinyi Liu, Yan Zheng, Xiaohan Xie, Jianye Hao, Mingzhi Li, Ruitao Wang, Fei Ni, Yuxiao Li, Jintian Luo, Shaoqing Jiao, and Jiajie Peng. CellAgent: An LLM-driven multi-agent framework for automated single-cell data analysis, 2024. URL https://doi.org/10.1101/2024.05.13.593861.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents, 2025. URL https://arxiv.org/abs/2502.12110.

B. Y. Yan, Chaofan Li, Hongjin Qian, Shuqi Lu, and Zheng Liu. General agentic memory via deep research, 2025. URL https://arxiv.org/abs/2511.18423.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Lianhao Zhou, Hongyi Ling, Keqiang Yan, Kaiji Zhao, Xiaoning Qian, Raymundo Arroyave, Xiaofeng Qian, and Shuiwang Ji. Toward greater autonomy in materials discovery agents: Unifying planning, physics, and scientists, 2025. URL https://arxiv.org/abs/2506.05616.

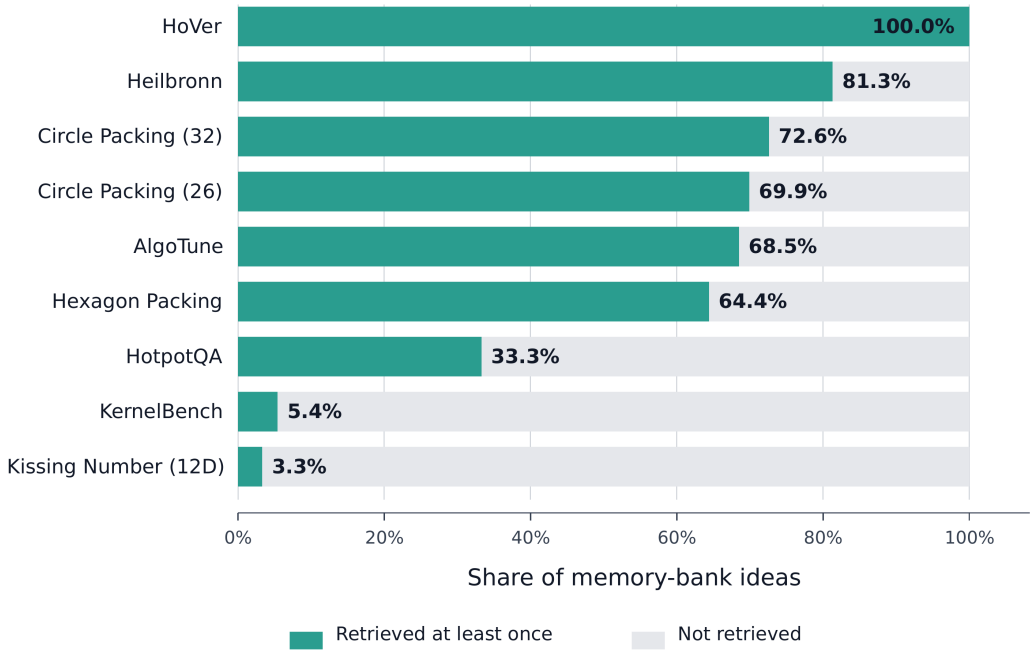

Figure 3: Breadth of memory bank use in the completed transfer experiments for which usage reports were available. Each bar gives the percentage of the available memory bank retrieved at least once, with the two KernelBench targets grouped into one benchmark-family bar. This measures retrieval coverage, not retrieval frequency or implementation.

A Memory Quality and Utilization

The benchmark results show whether memory-enabled evolution improves an end metric, but they do not by themselves show how broadly the bank is queried or whether the retrieved advice is actually incorporated into generated code. We therefore complement the outcome analysis with two mechanism-level audits: memory bank coverage in the transfer-target experiments and an acceptance audit that compares the default retrieval mechanism with random memory selection.

A.1 Memory utilization across transfer targets

The memory-usage reports associate every generated program with the identifiers of the memory cards selected for its mutation. We count a bank idea as *retrieved* when its identifier occurs at least once in a candidate's metadata and matches a card in the bank snapshot loaded for that experiment. This is a breadth measure and does not increase when the same card is retrieved repeatedly.

Figure 3 shows substantial variation in how much of each bank was explored. The only benchmark with full memory bank coverage was HoVer. The completed Heilbronn, Circle Packing (26), and Hexagon Packing reports retrieved 81.3%, 69.9%, and 64.4% of their available bank ideas, respectively. HotpotQA retrieved 33.3%. The KernelBench coverage was 5.4%, while Kissing Number (12D) showed the lowest coverage, at 3.3%.

B Examples of retrieved memory reuse

To make the effect of memory more concrete, Table 3 summarizes representative cases where retrieved memories were incorporated into successful mutations. These examples illustrate that the stored memories were not limited to copying task-specific code. Instead, they often expressed reusable execution strategies, search heuristics, or reasoning patterns that could be adapted to new target tasks.

Across these cases, retrieved memories usually encoded transferable execution ideas—such as caching repeated structure, exploiting sparsity, reducing convergence checks, or limiting custom kernels to profitable fusion—rather than narrow task-specific tricks.

Table 3: Representative examples of memory ideas reused during evolution.

Target task	Source or inferred source	Retrieved memory idea	How it was used
Circle Packing	Heilbronn-style point placement	Use annealed force-directed repulsion and low-discrepancy initialization to improve geometric dispersion.	Replaced the static grid with force-directed motion and adaptive radius expansion.
Hexagon Packing	Circle/geometric optimization	Use basin hopping, simulated annealing, and temperature-scaled repulsive forces to escape local packing optima.	Added hex-lattice starts, Gaussian jitter, and boundary-aware relaxation.
HotpotQA	Multi-hop QA traces	Extract only claim-relevant facts, preserve intermediate entities, and maintain query-refinement continuity across steps.	Added entity extraction and shorter final synthesis to reduce context drift.
HoVer	GSM8K / arithmetic reasoning	Make latent relationships explicit and verify them with concrete checks rather than generic reasoning instructions.	Added explicit reasoning questions, missing-fact lists, and targeted query examples.
KernelBench L1 P66	Kernel optimization traces	Avoid over-customized kernels when vendor primitives are faster; use custom Triton kernels only for profitable fusion.	Used `F.conv3d` for Conv3D and kept Triton only for fused bias/ReLU.
Power Control	Markowitz / QP-style optimization	Precompute invariant matrix scaling, use in-place linear algebra, and reduce convergence-check overhead.	Precomputed scaled coefficients, reused buffers, and checked convergence less often.
Battery Scheduling	LP-Box / Chebyshev Center / QP-style optimization	Cache repeated constraint structure, use sparse matrices for structured LPs, tune HiGHS options, and avoid repeated conversions.	Cached horizon templates, used CSR matrices, tuned HiGHS, and delayed conversions.

C LINEAGE DYNAMICS OF MEMORY USE

The run logs also allow a finer-grained observational analysis inside memory-enabled evolution. For each candidate, we mark whether the mutation directly used retrieved memories and whether the candidate belongs to a descendant lineage of any previous memory-use event. We then compare these groups within the logs that contain both memory and non-memory post-root mutations. Because the underlying tasks use different objective scales, the analysis reports within-run objective ranks among valid post-root programs, where larger ranks are better. These statistics are associative rather than randomized causal estimates, but they test whether memory-guided variants are preferentially retained by the evolutionary process.

Table 4: Lineage-level dynamics of retrieved memory usage. Objective values are converted to within-run ranks among valid post-root programs, where 1 is the best program in that run. Comparisons are made within the evolution logs between memory-associated programs and contemporaneous non-memory programs.

Statistic	Memory-associated	Comparator	Effect
Valid direct mutations	47.9%	40.2%	+7.7 pp
Direct children per mutation	0.87	0.73	1.20×
Descendants per mutation	2.57	2.21	1.16×
Mean objective rank of valid lineage members	0.504	0.429	matched +0.075
Best lineage objective rank	0.997	0.667	+0.330

Table 4 reports that direct memory-use mutations are more likely to produce valid candidates and have more reproductive output than mutations without retrieved memory. The descendant view is also positive: programs in memory-descended lineages have higher average objective rank, and the strongest memory-descended lineages reach higher average best objective ranks than the strongest non-memory lineages. This is consistent with the intended role of EvoMem as a soft evolutionary bias: retrieved advice can be used by mutations whose descendants remain competitive under selection.

D Memory Card Schema and Algorithms

D.1 Memory card schema

The schema below summarizes the normalized fields used by EvoMem memory cards. It is designed to keep each stored item both actionable at mutation time and auditable after a run. Not every field is required for every card type: abstract idea cards emphasize reusable tactics and explanations, while program cards additionally store concrete code artifacts and fitness values.

Memory-card schema.

Field	Typical type	Purpose
`card_id`	string	Stable identifier used for retrieval, merging, and usage tracking.
`card_type`	enum	Distinguishes abstract idea cards from concrete program cards.
`description`	text	Concise statement of the reusable tactic or program-level finding.
`task_summary`	text	Short description of the source task, constraints, and objective.
`metric_context`	text	Description of the metric or fitness signal under which the card was observed.
`explanations`	list of text	Rationales, mechanism descriptions, and explanation summaries collected during analysis.
`keywords`	list of strings	Search terms used for exact and hybrid retrieval.
`source_program_ids`	list of strings	Program identifiers from which the idea or artifact was extracted.
`provenance`	structured record	Run identifier, generation, parentage, introduction events, and lineage evidence.
`aliases`	list of text	Earlier or alternative phrasings preserved during deduplication and merging.
`links`	list of strings	Optional links from an idea card to associated high-performing program cards.
`usage_stats`	structured record	Counts, observed downstream fitness deltas, per-task summaries, and median delta.
`retrieval_views`	structured text	Indexed views such as description, task summary, explanation summary, and their compositions.
`code`	optional text	Concrete evolved program body, present only for program cards.
`fitness`	optional number	Fitness value associated with a stored program card.

D.2 Write-phase pseudocode

Algorithm 1: WriteMemory.

1. **Input:** completed run trace R, existing memory bank M.
2. Collect completed programs from R and remove root programs, invalid outputs, and entries without usable fitness or mutation metadata.
3. Normalize each remaining program into a candidate record containing fitness, generation, parentage, task context, mutation descriptions, rationales, and code.
4. Extract candidate ideas from the normalized records using either the default analyzer or the fast clustering analyzer.

5. Enrich extracted ideas with task summaries, keywords, explanation summaries, aliases, and source-program identifiers.
6. Estimate observational usage statistics for previously selected memories by comparing each child fitness with the strongest parent fitness.
7. Run origin analysis to identify high-value ideas supported by introduction gains, sibling comparisons, descendant spread, and downside-rate checks.
8. Normalize abstract idea cards and optional high-performing program cards into the memory-card schema.
9. For each normalized card:
   (a) If its identifier already exists in M, update the existing entity and merge provenance and usage statistics.
   (b) Else if the card is a concrete program, insert it directly.
   (c) Else retrieve near-neighbor idea cards using multiple textual views and combine candidate scores.
   (d) Ask the deduplication policy to store, discard, or merge the incoming idea.
   (e) If the policy fails, store the incoming idea as a new card.
10. Refresh retrieval indexes and persist the updated memory bank.
11. **Output:** updated memory bank M with write statistics and refreshed indexes.

D.3 READ-PHASE PSEUDOCODE

Algorithm 2: RetrieveMemory.

1. **Input:** parent program p, task description τ, metric description μ, mutation context x, memory bank M.
2. Validate the current program state and construct a structured memory request from (p, τ, μ, x).
3. If memory is disabled, return an empty memory section and an empty identifier list.
4. Initialize the bounded selector loop with the request, the available memory abstracts, and the maximum number of retrieval iterations.
5. While the selector has remaining iterations:
   (a) Plan one or more retrieval actions using exact keyword search, direct identifier lookup, or field-aware vector search.
   (b) Execute each retrieval action with the configured top-k limit.
   (c) Merge hits by card identifier and keep the strongest evidence for each card.
   (d) Reflect on the collected evidence and either stop with a final selection or issue a focused follow-up query.
6. Convert the selected cards into concise mutation advice and truncate the resulting text to the prompt budget.
7. Attach the selected card identifiers to the parent program metadata for later usage tracking.
8. Insert the memory text into the mutation prompt as advice, leaving generation, validation, and scoring unchanged.
9. **Output:** mutation prompt with a bounded memory section and selected memory identifiers.

E BENCHMARKS ABSOLUTE VALUES

For completeness, Table 5 reports the mean final target-metric values, on their original task-specific scales, across completed matched baseline–memory run pairs for each benchmark family summarized in the main text. Relative gains reported in Table 1 are calculated separately for each matched baseline–memory run pair and then averaged. Consequently, the mean relative gain need not equal the relative gain computed directly from the two mean values reported here.

Table 5: Absolute target-metric values for baseline and memory-enabled runs across benchmark families. Higher values indicate better performance for the reported task-specific target metric, except for Kakeya Parallelograms (64), which is minimized. ± gives the standard deviation over runs.

Benchmark	Target metric baseline	Target metric with memory
Circle Packing (32) ↑	2.7870 ±0.0954	2.9289 ±0.0081
Circle Packing (26) ↑	2.4218 ±0.1230	2.5577 ±0.0525
Heilbronn ↑	0.0304 ±0.0025	0.0315 ±0.0017
Kissing Number (12D) ↑	840.0000 ±0.0000	840.0000 ±0.0000
Hexagon Packing ↑	-4.3424 ±0.0886	-4.1799 ±0.0427
HoVer ↑	0.7900 ±0.0050	0.8080 ±0.0040
AlgoTune ↑	461.5126 ±154.5847	495.3686 ±159.0712
KernelBench ↑	2.4941 ±3.2564	2.7049 ±3.1980
HotpotQA ↑	0.5389 ±0.0381	0.6015 ±0.0510
Second Autocorrelation ↑	0.8944±0.0018	0.9038±0.0040
Minimum Autocorrelation ↑	0.4058±0.0057	0.4090 ±0.0009
Kakeya Parallelograms (64) ↓	0.1939±0.0009	0.1921±0.0025

F EVOLUTION TREE AND IDEAS VISUALIZATION

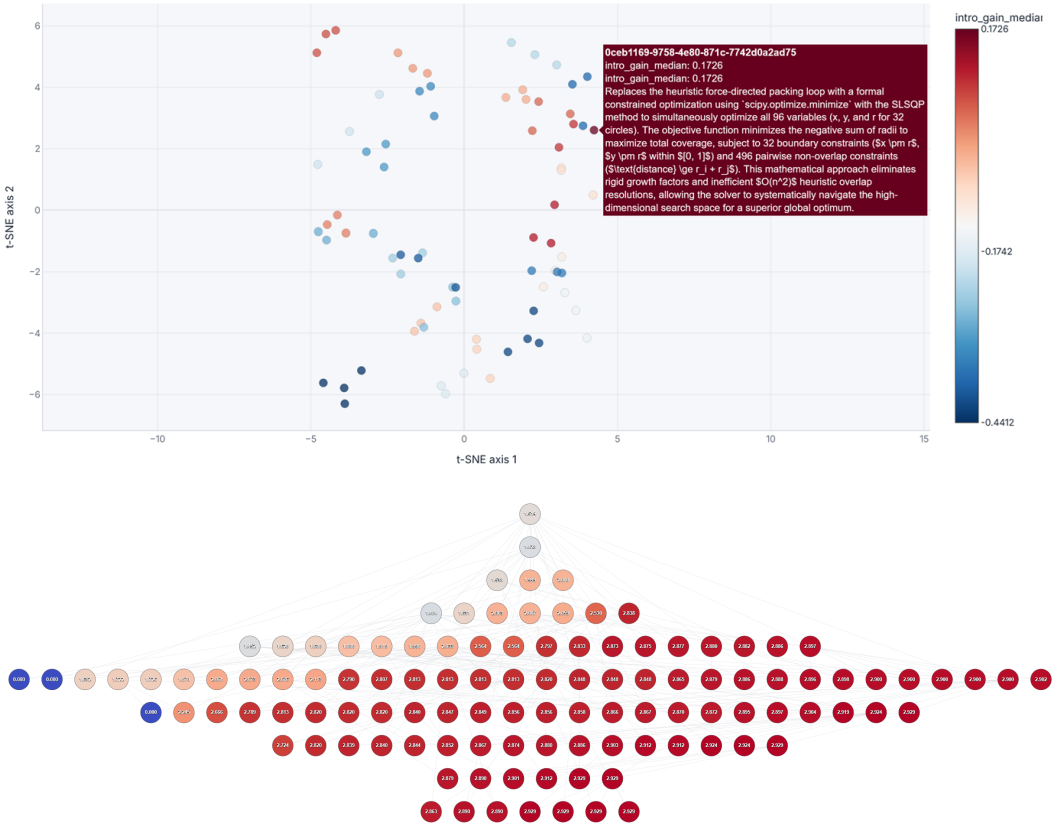

Figure 4: Top: visualization of extracted ideas on a two-dimensional plane obtained by applying t-SNE to embeddings of their textual descriptions. Bottom: evolution tree showing how programs branch and improve over the course of search.

G EXPERIMENTAL PARAMETERS

- **Population size:** MAP-Elites island archive maximum size 75; initial population from all `*.py` files in each problem's `initial_programs/` directory; 5 elites per generation; 8 mutants per generation / epoch; at most 8 steady-state in-flight mutants.
- **Evaluation budget:** no fixed total evaluation budget by default; `max_generations` is null / unlimited by default, with most experiments using 30; per-generation mutant budget is 8, so the approximate mutant budget is `max_generations` $\times 8$ plus initial and archive-refresh evaluations; `optimization_time_budget` is $0.75\times$ `dag_timeout`.
- **Memory top-k:** `MemoryProvider` uses `max_cards=3`; memory runtime `search_limit=5`; Memory `ResearchAgent` allowed tools are `page_index` and `vector`; Memory `top_k_by_tool` uses `keyword=5`, `vector=3`, `vector_description=3`, `vector_task_description` ≥ 1, `vector_explanation_summary=3`, `vector_description_explanation_summary=3`, `vector_description_task_description_summary=3`, and `page_index=5`; card update deduplication retrieval uses `top_k_per_query=10`, `final_top_n=10`, and `min_final_score=0.05`.
- **Retrieval iterations:** Memory`ResearchAgent` uses `max_iters=3`.
- **Embeddings and clustering:** `IdeaTracker` uses `all-MiniLM-L6-v2`; memory uses embeddings-model `all-MiniLM-L6-v2`; DBSCAN uses `min_samples_for_dbscan=4`, `eps=0.25`, and `min_samples=2`; the fast analyzer uses `batch_size=32`, `max_attempts=20`, `max_rounds=100`, `recompute_center=false`, `refine_subgroup_size=10000`, and `llm_max_concurrent=100`.
- **Models and API:** main mutation, insights, and lineage model is `Gemini3Flash`; ideas-tracker analyzer model is `Gemini3Flash`; API base URL is `https://openrouter.ai/api/v1`; LLM defaults are `temperature=0.7`, `max_tokens=81920`, `top_p=0.95`, and `top_k=20`; many chain/prompt evaluations use `Qwen/Qwen3-8B` through local OpenAI-compatible endpoints.
- **Hardware:** NVIDIA A100-SXM4-80GB, driver 560.35.03; Intel Xeon Platinum 8358, 128 logical CPUs; Linux 5.15.0-1044-nvidia.
- **Runtime:** `stage_timeout=2400` s default; `dag_timeout=7200` s default; `dag_concurrency=16`; `max_concurrent_dags=10`; LLM request timeout 3200 s for all phases; stage and DAG timeouts are 7200 s for phase 1 and 18000 s for phases 2–3; generation timeout is 10800 s for phase 1 and 14400 s for phases 2–3.
- **Dataset splits and subsets:** no single universal dataset split; prompt tasks default `load_context n_samples=300`; HotpotQA chain defaults to the first 300 training samples, full validation uses `n_samples=200`, and optional train/validation and pool/held-out-validation splits use `seed=42`; GSM8K chain defaults to the first 200 training samples; test helpers commonly default to `n_samples=3` unless overridden.

H HIERARCHICAL TRANSFER STUDY

This section gives the setting and the statistical tests of the experiments on the three mathematical problems of Section 4, and compares EvoMem with hierarchical memory, a variant of EvoMem that changes only how cards are selected.

Setting. On each problem we compare three kinds of runs: baseline runs without memory, runs with EvoMem, and runs with hierarchical memory. EvoMem and hierarchical memory read the same memory bank (Section 4), which contains no memory produced from the problem being tested; the rest of the pipeline is the same in all runs. In hierarchical memory, the cards of the bank are organized into a hierarchy of families of similar cards, and selection first chooses a family and then a card within it. Three hierarchical arms differ in how a family is described to the model: by LLM-written

meta-ideas (Hf), by the titles of its cards (Ht) or by its most central card (Hm). Every arm has three runs on each problem; baseline and EvoMem runs with the same seed form three matched pairs, which give the rows of Table 1.

Measures. To average over problems with different scales, a best-so-far value is mapped to a band score in $[0, 1]$: 0 is the median best-so-far value of all runs of the problem after 20 mutations, and 1 is the best value any run of the problem reached. The final result is a run's band score after 200 mutations; early progress and the whole run are its mean band score over mutations 0–50 and 0–200. The per-step gain is the relative gain of Table 1 computed at every mutation and averaged over mutations 1–200 or 20–200.

Table 6: Effects on the three mathematical problems, relative to the baseline and averaged over the problems: difference in band score for the final result, early progress and the whole run, and the per-step gain in percent (measures defined in the setup). p: one-sided permutation test that repeats the selection of the three problems. Problems $+$: problems with a positive effect. Hierarchical: Hf, Ht and Hm pooled (nine runs per problem), without a test.

	EvoMem vs. baseline			Hierarchical (Hf, Ht, Hm) vs. baseline	
Measure	Effect	p	Problems $+$	Effect	Problems $+$
Final result	+0.207	0.167	3/3	+0.092	3/3
Early progress, mutations 0–50	+0.141	0.097	3/3	+0.139	2/3
Whole run, mutations 0–200	+0.264	0.023	3/3	+0.180	3/3
Per-step gain, mutations 1–200	+3.00%	0.322	3/3	+2.74%	3/3
Per-step gain, mutations 20–200	+1.99%	0.276	3/3	+1.78%	3/3

EvoMem vs. hierarchical memory. Both lead the baseline on every measure of Table 6, EvoMem on all three problems and hierarchical memory on all but early progress on Second Autocorrelation, where it is level with the baseline. EvoMem leads more in the final result ($+0.207$ against $+0.092$) and over the whole run ($+0.264$ against $+0.180$), the two are level in early progress ($+0.141$ against $+0.139$), and their per-step gains differ by less than 0.3 percentage points. For EvoMem, the mean curve lies above that of the baseline, with non-overlapping ± 1 standard error bands, on 59–78% of the mutations and below it on none (curves of Figure 2); after accounting for the selection, its lead over the whole run is significant ($p = 0.023$), while the other measures, with three runs per arm, do not reach $p < 0.05$ (p between 0.097 and 0.322). EvoMem also reaches good solutions sooner, though only in the second half of the run: by mutation 100, 22% of the runs of each group have reached the median final value of the baseline runs, and by mutation 150, 67% of the EvoMem runs and 44% of the hierarchical runs, against 33% of the baseline runs (Figure 5). Between the two there is no consistent order: hierarchical memory has the larger lead over the whole run on one of the three problems. The three hierarchical arms do not differ from each other (Figure 6): no pairwise difference, at most 0.078 in band score, comes close to significance ($p \geq 0.35$), and the best arm over the entire run changes from problem to problem. It shows that organizing the bank into a hierarchy brings no detectable advantage here, not that flat selection is better.

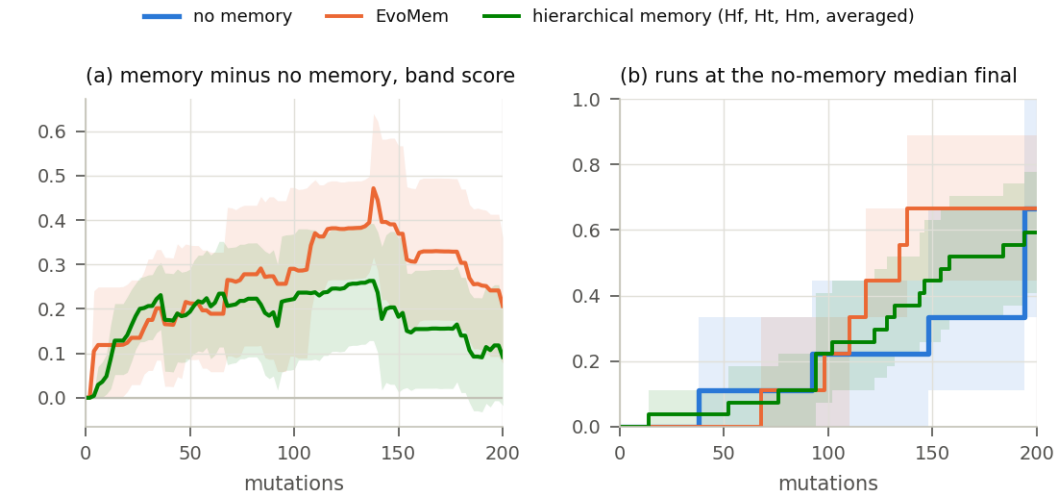

Figure 5: EvoMem and hierarchical memory against the baseline on the three problems of Figure 2. (a) Difference in band score from the baseline at every mutation, mean over the problems; the hierarchical arms are pooled. (b) Share of runs whose best-so-far value has reached the median final value of the baseline runs of their problem, averaged over the problems. Shaded: 95% bootstrap bands, runs resampled within each problem and arm.

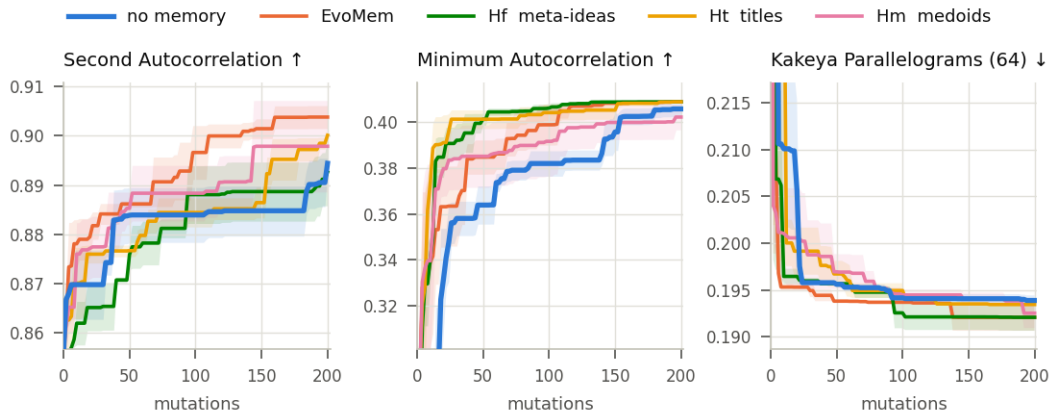

Figure 6: Mean best-so-far value over three runs per arm, ± 1 standard error, on the three problems of Figure 2: baseline without memory, EvoMem, and the hierarchical arms Hf, Ht and Hm. An upward arrow marks a score that is maximized, a downward arrow one that is minimized.