Proceedings of the 8th International Conference on Computing and Data Science
DOI: 10.54254/2755-2721/2026.34758

LLM-Driven Evolutionary Program Search: From FunSearch to Automated Scientific Discovery

Jiale Mao

The People's Government of Jiangsu Province, Nanjing Audit University, Nanjing, China
3622321319@qq.com

Abstract. Large language models (LLMs) suffer from logical illusions, while evolutionary algorithms (EAs) face search blindness in program generation. The LLM + EA paradigm addresses these issues.This paper systematically discusses its core technologies: program search space representation, memory-based semantic mutation, hierarchical evaluation feedback, and island population management. Using the car movement problem as a case study, it shows how historical code injection, multi-objective evolution, and distributed islands evolve an effective and interpretable control strategy. It also constructs a multi-dimensional evaluation system (objective performance + subjective interpretability), and compares methods such as AlphaEvolve, ShinkaEvolve, and OpenEvolve in combinatorial optimization, mathematical algorithm discovery, and software engineering, including new matrix multiplication algorithms and GPU kernel optimization.Finally, it identifies challenges (credibility illusion, dimension disaster, black-box interpretability, data leakage) and future directions (formal verification, context expansion, enhanced interpretability). This paper provides a systematic framework for LLM-driven evolutionary program search, pushing automated scientific discovery from "generative" to "verifiable".

Keywords: LLMs, EAs, program search, automated scientific discovery, LLM-driven evolution

1. Introduction

LLMs have reshaped code generation and program understanding. Code-based LLMs such as GPT model, CodeLlama, and DeepSeek-Coder excel in code completion and reasoning[1, 2]. Meanwhile, EAs play an irreplaceable role in program search and optimization-from genetic programming to neural architecture search. However, both face fundamental dilemmas [3].

The integration of LLMs and EAs follows a clear trajectory: traditional genetic programming (1980s)→neural architecture search (2010s)→LLM+EA. In recent years, LLMs have injected new vitality into evolutionary program search. Between 2024 and 2025, Google DeepMind's AlphaEvolve became a landmark work, combining Gemini with evolutionary search to automatically discover multi-agent learning algorithms surpassing human experts. Almost simultaneously, Sakana AI's ShinkaEvolve further improved sample efficiency, discovering a better circular stacking algorithm in only 150 iterations [4]. The open-source community also responded actively: projects such as OpenEvolve support GPU kernel optimization and sorting algorithm

DOI: 10.54254/2755-2721/2026.34758

discovery. At the theoretical level, Wu et al. systematically summarized the bidirectional LLM-EA enhancement. However, existing research still lacks systematic abstraction of general technical paths; LLM mutation operators often rely on heuristic prompt engineering; and credibility mechanisms such as interpretability and formal verification are still in infancy.

In response to these gaps, this paper systematically studies LLM-driven evolutionary program search, covering: overall architecture and core supporting technologies; integration paths and the enhancement effect of advanced prompt engineering; a multi-dimensional quality evaluation system comparing mainstream methods on typical tasks; prospective challenges and future research directions.

The primary innovations and contributions of this work are as follows：

• Systematically organizing the technical map of LLM+EA integration, covering search representation, operator design, evaluation feedback, and population management.

• Classifying LLM mutation operators into "prompt-based semantic mutation" and "feedback-based directional mutation", with applicable scenarios and optimization paths.

• Constructing a credibility analysis framework for program search in scientific discovery, addressing formal verification, interpretability enhancement, and data leakage prevention.

2. Relevant work

2.1. Overall architecture of the evolutionary program search system driven by LLM

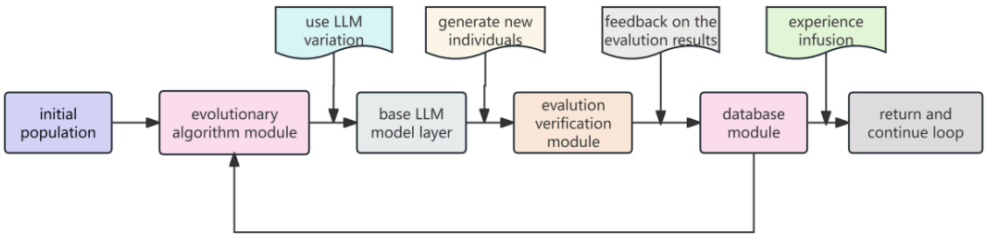

Figure 1. LLM flowchart

As shown in Figure 1, the LLM-driven evolutionary program adopts a modular architecture with four core modules: basic LLM model layer, evolution algorithm module, evaluation and verification module, and database module. The evolution module selects parents; the LLM performs context-aware mutation to generate offspring; the evaluation module checks grammar and performance; the database stores elite individuals and evolutionary trajectories. Historical experiences are re-injected into the evolution module for parent selection and mutation decisions, forming a closed loop that continuously optimizes population performance. The collaborative working mechanism embodies the integration concept of "LLM creative engine + evolutionary search framework" [5]. The specific manifestations are as follows:

Let the population be $P_t = \{p_1, p_2, \ldots, p_N\}$, where p_i represents the i-th program individual. In each generation t, LLM acts as the mutation operator M_{LLM} and generates new individuals based on the historical memory library H :

$$p' = M_{LLM}(p_{elite}, H, C) \tag{1}$$

DOI: 10.54254/2755-2721/2026.34758

Among them, p_{elite} refers to the parent individuals selected from the elite pool, and $\mathscr{C}$ represents the task constraints (such as prompt words). The fitness of the new individual is calculated through the evaluation function F:

$$f(p') = F(p', \mathscr{E}) \tag{2}$$

$\mathscr{E}$ represents the execution environment (such as Gym). The rule for updating the memory library is:

$$H_{t+1} = H_t \cup \{p' \mid f(p') > \theta\} \tag{3}$$

Here, θ represents the selection threshold. Wu et al. classified the interaction between LLM and evolutionary algorithms into two categories: LLM-enhanced evolutionary optimization and evolutionary algorithm-enhanced LLM. The former utilizes the domain knowledge of LLM to guide more intelligent search, while the latter enhances the performance of LLM in closed scenarios through the evolutionary framework.

2.2. Core supporting technologies for the evolutionary program search driven by LLM

The pre-training data construction and domain fine-tuning strategies for LLM code generation are the foundation for ensuring the quality of the generated code. Research has shown that domain-specific fine-tuning (e.g., LoRA) is often necessary for effective program search [6].

The spatial representation of the search space and the design method of operators in the evolutionary program are the technical difficulties of the integrated framework. Traditional Genetic Programming represents programs as syntax trees and conducts searches through crossover and mutation operations. The introduction of LLM provides a new path for semantic-aware mutation operations: The Grammar Variational Autoencoder with Attention-Based Gene Expression Programming (GVAE-ABGEP) method combines the syntax variational autoencoder with the evolutionary algorithm, guides the search direction through neural sampling, and achieves significant performance improvement in symbolic regression tasks [7]. The MCCE framework proposes a multi-LLM collaborative co-evolution mechanism [8].

The integration technology path of LLM and evolutionary algorithms can be divided into two types: embedded integration and external collaboration. Embedded integration embeds LLM as an evolutionary operator directly into the search loop, such as replacing the traditional crossover operation with the mutation operator based on LLM; external collaboration maintains the relative independence of LLM and the evolutionary algorithm, and achieves knowledge transfer through multiple rounds of interaction. The M2N2 algorithm provides a new idea for the deep integration of LLM and evolutionary algorithms [9]. This natural ecological niche model integration method breaks through the limitation of fixed boundaries in integration, providing a new idea for the deep integration of LLM and evolutionary algorithms.

3. The core technical path and optimization strategies of the evolutionary program search driven by LLM

The deep integration of LLM with evolutionary algorithms is not merely a simple combination of technologies, but rather a systematic innovation carried out at three levels: modeling of the search

DOI: 10.54254/2755-2721/2026.34758

space, design of mutation operators, and the architecture of integration. This paper thoroughly analyzes the core technical paths and optimization strategies of the LLM-driven evolutionary program search.

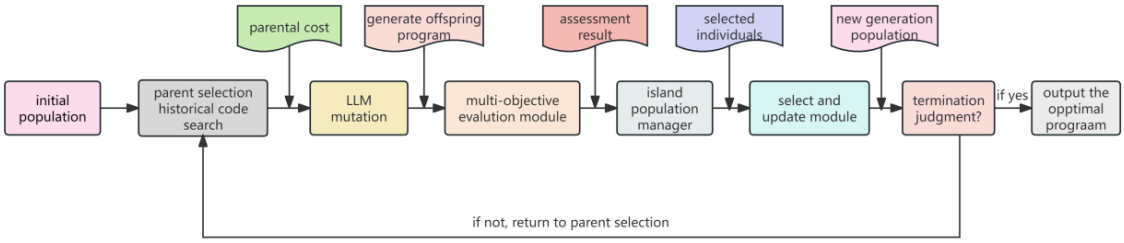

Figure 2. Flowchart of the LLM-driven evolutionary program search process

As shown in Figure 2, the workflow follows a closed-loop mechanism: select parent individuals, mutate via LLM using historical code, evaluate offspring with multi-objective metrics, manage sub-populations via island model, and update the elite library – repeating until termination.

3.1. Evolutionary procedure for search scenario modeling

The typical application scenarios of evolutionary program search can be summarized into three categories: combinatorial optimization scenarios (such as online bin packing, scheduling problems), mathematical algorithm design scenarios (such as extremal combinatorial mathematics, matrix multiplication optimization), and software engineering optimization scenarios (such as code performance optimization, test case generation). Different scenarios have different emphases on the requirements for program search: combinatorial optimization seeks approximate optimal solutions of the algorithm under constraints; mathematical algorithm design emphasizes mathematical rigor and innovation; software engineering optimization pays more attention to the readability and maintainability of the code.

The problem of car movement is a classic benchmark in the field of reinforcement learning, perfectly embodying the common challenges of the aforementioned scenario. The physical modeling of this problem is as follows: The car is in a one-dimensional valley, and the initial position is randomly sampled from pos~Uniform(-0.6, -0.4). The car can perform three discrete actions: accelerate to the left (0), no acceleration (1), and accelerate to the right (2). The core objective is to reach the target position pos≥0.5 within 200 steps and minimize the number of steps required.From the perspective of program search, the solution to the car problem essentially involves discovering a mapping function $\pi : \mathrm{R}^2 \to \{0,1,2\}$, from the state space to the action space. This function must satisfy physical constraints and achieve optimal control. Let the state vector be $\mathrm{s} = (\mathrm{pos}, \mathrm{v})$, and the policy function be expressed as:

$$\mathrm{a}_t = \pi(\mathrm{s}_t, \mathrm{a}_{t-1}) \tag{4}$$

Here, a_t represents the action at time t , and a_{t-1} represents the action at the previous time. The optimization objective can be formulated as:

DOI: 10.54254/2755-2721/2026.34758

$$\min_{\pi} \mathrm{E}_{s_0 \sim \mathscr{U}(-0.6,-0.4)} [\mathrm{T}(\pi, s_0)] \tag{5}$$

$$s.t. \quad \forall t, pos_t \in [-1.2,0.6], v_t \in [-0.07,0.07], a_t \in \{0,1,2\} \tag{6}$$

Here, $\mathrm{T}(\pi, s_0)$ represents the number of steps required for the vehicle to reach a position greater than or equal to 0.5 for the first time starting from the initial state s_0 under the strategy π ; if it fails to reach within 200 steps, then $T = 200$. The complexity of this search task lies in the fact that the vehicle cannot directly accelerate and climb to the right; it must first swing back and forth in the left valley to accumulate sufficient kinetic energy.

```python
# initial_program.py
# EVOLVE-BLOCK-START
"""Initial heuristic for Mountain Car in Gym"""
import numpy as np
def choose_action(pos: float, v: float, last_action: int) -> int:
    """
    Action selection for MountainCar-v0.
    Args:
        pos: position in [-1.2, 0.6]
        v: velocity in [-0.07, 0.07]
        last_action: previous action (0,1,2)
    Returns:
        action: 0 (left), 1 (none), 2 (right)
    """
    # Simple physics-based heuristic
    if pos < -0.5 and v < 0:
     return 0  # accelerate to the left at the bottom of the valley
    elif v > 0:
     return 2  # when charging upwards, accelerate to the right
    else:
     return 0  # underwrite and continue to store energy
```

Figure 3. Initial code diagram of the OpenEvolve framework

This heuristic accelerates left when moving left (building momentum) and right when moving right (heading to target). While reflecting human understanding of the physics, it is suboptimal-handling momentum poorly and ignoring the last_action parameter.

3.2. Adaptive optimization strategy for LLM-driven evolutionary program search

The introduction of LLM provides a new solution to the semantic blindness of traditional mutation operators: using LLM as a semantically-aware mutation operator, and leveraging its pre-trained program understanding capabilities on a large amount of code to generate meaningful mutations that are consistent with the problem context.

Wu et al.'s review classified this paradigm as "LLM-enhanced EA", which means LLM-enhanced evolutionary optimization. The AlphaEvolve system of DeepMind first put this idea into practice, using Gemini as genetic operators to rewrite game theory algorithms. Experiments showed that the evolved VAD-CFR and SHOR-PSRO algorithms adopted counter-intuitive mechanisms that humans

DOI: 10.54254/2755-2721/2026.34758

had never thought of, and outperformed the optimal solutions designed by humans in almost all tested games.

In the evolutionary search for the problem of vehicle movement, the design of the LLM mutation operator embodies three levels of optimization strategies:

First: Directed mutation based on the injection of historical excellent code. The system maintains an elite code library. When generating new individuals, the historically optimal code is used as a context injection prompt to guide the LLM to make local improvements based on the excellent code. From the perspective of evolutionary strategy, this process can be regarded as a kind of semantic mutation based on gradients:

$$p' = \mathrm{p}_{\mathrm{elite}} + \Delta_{\mathrm{LLM}}(\mathrm{p}_{\mathrm{elite}}, H) \tag{7}$$

Here, Δ_{LLM} represents the semantic increment generated by LLM, which is different from the random perturbations in traditional evolutionary algorithms. Below is an improved strategy discovered after multiple rounds of evolution:

```python
# initial_program.py
# EVOLVE-BLOCK-START
"""Initial heuristic for Mountain Car in Gym"""
import numpy as np
def choose_action(pos: float, v: float, last_action: int) -> int:
    """
    Action selection for MountainCar-v0.
    Args:
        pos: position in [-1.2, 0.6]
        v: velocity in [-0.07, 0.07]
        last_action: previous action (0,1,2)
    Returns:
        action: 0 (left), 1 (none), 2 (right)
    """
    # Enhanced physics-based heuristic with momentum awareness
    if pos < -0.5 and v < 0:
        # At left valley, accelerate left to build momentum
        return 0
    elif v > 0 and pos < 0.5:
        # Moving uphill toward goal, accelerate right
        return 2
    elif pos > -0.5 and v < 0 and pos < 0.4:
        # Going down from middle, need to regain speed but not too close to g
oal
        return 0
    elif last_action == 0 and v > 0.02:
        # Detecting momentum build-up, maintain direction
        return 0
    elif last_action == 2 and v < -0.02 and pos < 0.4:
        # Detecting oscillation near goal, switch to left
        return 0
    elif pos > 0.45 and v < 0:
        # Near goal but going down, accelerate right to get over the hill
        return 2
    else:
        # Default behavior - slightly more nuanced
        if pos < -0.2 and v < 0:
            return 0  # Build momentum in early valley
        else:
            return 1  # Otherwise, coast
```

Figure 4. Improved strategy of the OpenEvolve framewor

DOI: 10.54254/2755-2721/2026.34758

Compared with the initial version, this strategy achieves 100% success rate and reduces average steps to 106.4. Secondly, the multi-objective evolutionary strategy balances code performance and simplicity. The program search not only aims for performance metrics (success rate, average steps), but also needs to consider code readability and logical rigor. Let the target variables be:

$$g(p) = (g_1(p), g_2(p)) \tag{8}$$

Among them, $g_1(p) = avg\_steps(p)$ measures performance, and $g_2(p) = code\_length(p)$ measures simplicity. The multi-objective optimization aims to find the Pareto optimal solution set:

$$P = \{p \mid \exists p's.t.\, g_i(p') \leq g_i(p) \forall i, and\ \exists j, g_j(p') < g_j(p)\} \tag{9}$$

The μMOEA framework is the first to combine LLM with multi-objective evolutionary algorithms. Through adaptive selection and mutation mechanisms, it achieves a balance between search efficiency and population diversity. In the car problem, the evaluation function calculates both the combined score and code length, and selects the elite individuals based on the Pareto dominance relationship.

Third, distributed population management based on the island model. The island model divides the population into multiple subpopulations (islands), each evolving independently, and periodically migrates the superior individuals to maintain search diversity [10]. Let the number of islands be K ,and the population of the k-th island be $P_t^{(k)}$. The island migration can be expressed as:

$$P_{t-1}^{(i)} = P_t^{(i)} \cup Migrate\left(U_{j \neq i} Elite\left(P_t^{(j)}\right)\right) \tag{10}$$

The research on distributed evolutionary algorithms indicates that dynamic adjustment of the number of islands can significantly enhance the search efficiency [11]. In the implementation of OpenEvolve, three evaluators run in parallel (parallel_evaluations: 3), simulating the effect of multi-island parallel evolution.

3.3. Integration path of LLM and evolutionary algorithms

The integrated architecture design of LLM and evolutionary algorithms consists of three core modules: programmatic representation of search space, LLM creative engine, and hierarchical filtering system. The programmatic representation of the search space treats algorithmic code as an evolving genome, the LLM creative engine is responsible for generating candidate variations, and the hierarchical filtering system conducts multi-round evaluations to select high-quality individuals.

The advanced design of the prompt engineering is the key to achieving high-quality variations. In the car problem, the design of the prompt words embodies the following principles:

DOI: 10.54254/2755-2721/2026.34758

```yaml
# Configuration for Mountain Car in Gym environment
max_iterations: 100
checkpoint_interval: 5
# Prompt configuration
prompt:
  system_message: |
    You are an expert reinforcement learning engineer.
    Improve the 'choose_action' function for the Mountain Car problem in OpenAI Gym.
    Environment: MountainCar-v0
    - Initial position: uniformly random in [-0.6, -0.4]
    - Goal: reach position >= 0.5
    - State: (position ∈ [-1.2, 0.6], velocity ∈ [-0.07, 0.07])
    - Actions: 0=accelerate left, 1=no accelerate, 2=accelerate right
    Strategy must:
    - Use momentum: accelerate left when going down left hill to gain energy
    - Accelerate right when moving upward toward the goal
    - Minimize steps to reach the flag
    - Be deterministic and fast (no loops or heavy computation)
    -
    MUST use 'last_action' in at least one conditional branch to detect oscillation or momentum phase
    -
    Only allowed import is 'import numpy as np'. Do not import any other library
    - Never use 'random', 'np.random', or any stochastic operati
    Return ONLY the complete Python function 'choose_action(pos, v, last_action) -> int'.
    Do not include imports, explanations, or extra code.
# LLM configuration
llm:
  primary_model: "qwen3-coder-flash"
  api_base: "https://dashscope.aliyuncs.com/compatible-mode/v1"
  api_key: "sk-c870bb2f508143368d83479330e17ebc"
  temperature: 0.0
  max_tokens: 12000
# Evaluator configuration
evaluator:
  timeout: 60
  cascade_evaluation: false
  parallel_evaluations: 3
# Evolution settings
diff_based_evolution: true
max_code_length: 20000
```

Figure 5. Prompt words of the OpenEvolve framework

This constraint-based prompt integrates role definition, physical constraints, momentum strategy, mandatory last_action usage, and a ban on random operations, effectively narrowing LLM generation space and improving variation hit rate.

Historical excellent code injection retrieves successful code as few-shot examples. The hierarchical filtering system ensures the quality of the generated code through multiple rounds of evaluation. The evaluation module first conducts grammar checks and security validations (such as prohibiting illegal imports), and then performs 100 independent experiments in the Gym environment, calculating success rate, average steps, and comprehensive score. This "early failure principle" and "multi-round screening" mechanism effectively prevents common runtime errors and logical flaws in the code generated by LLM.

4. Performance evaluation and application of evolutionary programs driven by LLM

4.1. Quality evaluation system for LLM-driven evolutionary programs search

The quality assessment of LLM-driven evolutionary programs requires the construction of a multi-dimensional index system, taking into account both objective performance and subjective interpretability. The objective assessment indicators mainly include the accuracy of algorithm solution, progressive complexity, running efficiency, generalization ability, and search diversity. Let

DOI: 10.54254/2755-2721/2026.34758

the evaluation task set be T , for the program p generated by algorithm A , the performance score is defined as:

$$\mathrm{Score}_{\mathrm{Bd}}\left(p\right)=\frac{1}{|\mathscr{T}|}\sum_{\mathrm{p,t}}\left(\alpha\cdot\mathbb{I}\mathrm{success}\left(\mathrm{p,t}\right)+\beta\cdot\frac{1}{1+\mathrm{steps(p,t)}}+\gamma\cdot\frac{1}{\mathrm{complexity(p)}}\right) \tag{11}$$

Here, α,β,γ are weight coefficients. The evaluation of ShinkaEvolve indicates that a new advanced circular arrangement solution can be discovered with only 150 sampling attempts. The sample efficiency has improved by several orders of magnitude compared to traditional methods. The evaluation of the SOAR framework on the ARC-AGI benchmark shows that through self-evolution cycles, the model's solution rate has increased to 52%, fully verifying the effectiveness of the search method [12]. The subjective evaluation indicators focus on the rationality of the generation algorithm's principle, the readability of the code, and the logical rigor. The interpretability score is defined as:

$$\mathrm{Score}_{\mathrm{inhj}}(p)=\lambda_1\cdot R(p)+\lambda_2\cdot readability(p)+\lambda_3\cdot logic(p) \tag{12}$$

Among them, $\mathrm{R(p)}$ represents the principle score, which can be provided by experts or through automatic evaluation based on concepts. The research on Multimodal LLM-assisted Evolutionary Search emphasizes that the code-based strategies offer better interpretability than neural networks [13]. The concept-based scoring method proposed by ConceptSearch achieves a search efficiency 30% higher than the Hamming distance by evaluating the program's ability to capture the transformation concepts behind the input and output examples [14].

4.2. Performance analysis of LLM-driven evolutionary program search

The performance comparison of mainstream LLM-driven evolutionary program search methods in typical evaluation tasks has increasingly become a research focus. To systematically demonstrate the characteristics and advantages of different technical paths, Table 1 compares the architectures, optimization strategies, and key quantitative results of the current representative methods.

Table 1. Comparison of mainstream LLM-driven evolutionary program search methods

Method	Core innovation	LLM Role	Evolutionary strategy	Typical evaluation task
AlphaEvolve	Use LLM as a genetic operator for semantic-level code rewriting	Mutation operator	Population evolution + Elite retention	Game theory algorithms, matrix multiplication, cluster scheduling

DOI: 10.54254/2755-2721/2026.34758

Table 1. (continued)

ShinkaEvolve	Novelty rejection sampling + multi-arm gambling machine LLM integrated selection	Mutation operator + Selector	Sample efficient evolution (150 iterations)	Circular packing problem, AIME mathematical reasoning, ALE-Bench
OpenEvolve	Open-source framework, supporting differentiated evolution and multi-task expansion	Mutation operator + evaluator	Island Model Parallel Evaluation (3 Parallel Evaluators)	GPU core optimization, sorting algorithm, car movement control
RankEvolve	The retrieval algorithm automatically discovers: seed program + LLM variation	Mutation operator	Iterative optimization + generalization verification	BEIR, BRIGHT (12 retrieval datasets), TREC DL
SOAR	Self-evolutionary cycle: Alternation of evolutionary search and post-event learning	Creative Engine + Learner	Self-evolutionary cycle	ARC-AGI Benchmark
MLES	Programmed control strategy search assisted by multimodal LLM	Strategy Generator	Multi-objective Evolution	Standard control tasks (such as Mountain Car, CartPole, etc.)

DOI: 10.54254/2755-2721/2026.34758

Table 1. (continued)

FunSearch	Program space search based on LLM and code comment-driven evolution	Creative Generator	Evolutionary iteration + Program library	Combinatorial optimization problems (online bin packing, scheduling)

RankEvolve outperforms base algorithms on BEIR and BRIGHT datasets [15]. The enhancement effects of different optimization strategies on the search ability can be verified through ablation experiments. ShinkaEvolve's three core innovations—balanced parent sampling, novelty rejection sampling, and multi-armed bandit-based LLM selection—enable a high-performance agent for AIME mathematical reasoning and improved solutions to ALE-Bench problems, including a novel expert mixture load balancing loss. Combined with NSGA-II and prompt grammar, it optimizes task accuracy and labeling efficiency, producing a Pareto front adaptable to various constraints [16].

4.3. Typical application scenarios of LLM-driven evolutionary program search

The combinatorial optimization scenario is an important testing ground for LLM evolutionary search. AlphaEvolve has released 0.7% of the computing resources in the optimization of cluster scheduling algorithms, discovered a GPU core configuration that accelerates attention computations by 32%, and found a partitioning strategy for matrix multiplication that increases speed by an average of 23%, directly reducing the training time of the Gemini model. These achievements demonstrate that evolutionary search can discover engineering optimization solutions that surpass the intuitive understanding of human experts.

The field of mathematics and theoretical computer science witnessed the most exciting breakthroughs in this area. AlphaEvolve discovered a new algorithm for multiplying complex 4×4 matrices, requiring only 48 multiplication operations, breaking the 56-year record set by the Strassen method;ShinkaEvolve also made significant progress in the circular arrangement problem.

The application of actual software engineering scenarios is expanding rapidly. The SOAR framework integrates LLM into the self-evolving search loop, alternately performing evolutionary search and post-event learning, continuously enhancing the model's program sampling and optimization capabilities. The Multimodal LLM-assisted Evolutionary Search method achieves comparable performance to Proximal Policy Optimization (PPO) on standard control tasks, while providing transparent control logic and traceable design processes.

5. Future research directions and challenges

Credibility and hallucination remain primary obstacles. LLMs generate "reasonable" rather than "correct" outputs, often with logical flaws. Existing methods handle hallucinations after generation, lacking active pre-generation verification. Future research should introduce formal verification to embed mathematical correctness checks into the evolutionary cycle, shifting from generative to verifiable AI.

The "dimension disaster" of search space is another bottleneck. For long code or multi-function systems, LLM context windows become the main obstacle. LongRoPE2 extends context to 128K via

DOI: 10.54254/2755-2721/2026.34758

EA-guided rescaling [17]. This shows that evolutionary algorithms can optimize LLM architecture, forming a beneficial cycle.

Black-box optimization limits interpretability: LLMs know what but not why. The AEGIS method decomposes training examples into "atomic edits". Future work should combine conceptual hierarchies with evolutionary trajectory visualization to make algorithm discovery auditable.

Data leakage threatens fairness. If pre-training corpora contain test set solutions, evaluation scores inflate and generalization degrades. The AntiLeakBench framework achieves pollution-free evaluation. Future research needs standardized data traceability and de-identification to ensure verifiable originality.

6. Conclusion

This paper systematically reviews the LLM+EA paradigm for program search and automated scientific discovery. Starting from the semantic blindness of traditional genetic programming, it elaborates the core value of LLM as a semantic-aware mutation operator, and constructs a technical system covering search space representation, adaptive optimization, and integrated architecture. Using the car movement problem as a case study, the evolved strategy achieves high success rate, low step count, and good interpretability. Performance evaluation shows that this paradigm surpasses human experts in multiple domains.

However, challenges remain: credibility illusion, dimensionality disaster, black-box interpretability, and data leakage. Future work should introduce formal verification, context expansion, and interpretability mechanisms to push automated scientific discovery from "generative" to "verifiable". This paper provides a technical roadmap for systematic research in this field.

References

[1] Yang D, Liu T, Zhang D, et al. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms [C]//Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing. 2025: 2586-2616.

[2] Billah M M. Large Language Models for Code Generation and Program Comprehension: Exploring Capabilities, Context, and Developer Adaptation [D]. , 2025.

[3] Wu X, Wu S H, Wu J, et al. Evolutionary computation in the era of large language model: survey and roadmap 2024 [EB/OL]. arXiv preprint arXiv: 2401.10034

[4] Lange R T, Imajuku Y, Cetin E. Shinkaevolve: Towards open-ended and sample-efficient program evolution [J]. arXiv preprint arXiv: 2509.19349, 2025.

[5] Abhyankar N, Kabra S, Desai S, et al. Accelerating materials design via LLM-guided evolutionary search [J]. arXiv preprint arXiv: 2510.22503, 2025.

[6] Ozdemir S. Quick start guide to large language models: strategies and best practices for using ChatGPT and other LLMs [M]. Addison-Wesley Professional, 2023.

[7] Lu Q, Yuan Q, Li D, et al. Embedding neural sampling and adversarial bandit into gene expression programming for symbolic regression [J]. Applied Soft Computing, 2025: 113718.

[8] Ran N, Li Z, Wang Y, et al. MCCE: A Framework for Multi-LLM Collaborative Co-Evolution [J]. arXiv preprint arXiv: 2510.06270, 2025.

[9] Beel J, Kan M Y, Baumgart M. Evaluating Sakana's AI Scientist: Bold Claims, Mixed Results, and a Promising Future? [C]//ACM SIGIR Forum. New York, NY, USA: ACM, 2025, 59(1): 1-20.

[10] Abed-alguni B H, Paul D. Island-based Cuckoo Search with elite opposition-based learning and multiple mutation methods for solving optimization problems [J]. Soft Computing, 2022, 26(7): 3293-3312.

[11] Dziurzanski P, Zhao S, Przewozniczek M, et al. Scalable distributed evolutionary algorithm orchestration using Docker containers [J]. Journal of Computational Science, 2020, 40: 101069.

DOI: 10.54254/2755-2721/2026.34758

[12] Pourcel J, Colas C, Oudeyer P Y. Self-improving language models for evolutionary program synthesis: A case study on ARC-AGI [J]. arXiv preprint arXiv: 2507.14172, 2025.
[13] Hu Q, Tong X, Yuan M, et al. Multimodal LLM-assisted Evolutionary Search for Programmatic Control Policies [J]. arXiv preprint arXiv: 2508.05433, 2025.
[14] Singhal K, Shroff G. ConceptSearch: Towards Efficient Program Search Using LLMs for Abstraction and Reasoning Corpus (ARC) [C]//Proceedings of the AAAI Conference on Artificial Intelligence. 2025, 39(19): 20506-20513.
[15] Nian J, Li F, Park D H, et al. RankEvolve: Automating the Discovery of Retrieval Algorithms via LLM-Driven Evolution [J]. arXiv preprint arXiv: 2602.16932, 2026.
[16] Lopes C L V, da Silva L M. Assessing an evolutionary search engine for small language models, prompts, and evaluation metrics [C]//Brazilian Conference on Intelligent Systems. Cham: Springer Nature Switzerland, 2025: 120-134.
[17] Shang N, Zhang L L, Wang S, et al. Longrope2: Near-lossless llm context window scaling [J]. arXiv preprint arXiv: 2502.20082, 2025.