ALGOEVO: SELF-EVOLVING AGENTIC SEARCH FOR AUTOMATED ALGORITHM DISCOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models have advanced automated algorithm discovery by synthesizing executable code, but existing frameworks trap them in rigid search pipelines with pre-defined control flows. This limitation restricts adaptive reasoning, blocks cross-paradigm transfer, and overlooks richer execution feedback. We propose **AlgoEvo**, a unified agentic framework that transforms automated algorithm discovery into an interactive, knowledge-accumulating process. An autonomous agent dynamically inspects, diagnoses, and edits code based on runtime feedback. A design skill hub decouples paradigm-specific knowledge from the core discovery engine, allowing a unified workflow to seamlessly handle single-heuristic, multi-objective, and multi-component design. Meanwhile, a hierarchical experience bank organizes search trajectories into a task-level tree to guide exploration and consolidates cross-task patterns into reusable skills. Across six representative benchmark tasks, AlgoEvo reaches state-of-the-art performance with as little as 7% of the evaluation budget and reduced token consumption, demonstrating strong intra-task accumulation, cross-task transfer, and the ability to reproduce or exceed the strongest existing methods through flexible skill activation.

1 INTRODUCTION

Automated Algorithm Discovery (AAD) focuses on generating optimization heuristics without manual algorithm engineering (Burke et al., 2013; Stützle & López-Ibáñez, 2019). Integrating Large Language Models into Automated Heuristic Design (LLM-AHD) has enabled direct code synthesis for complex heuristics (Liu et al., 2026b; Wu et al., 2024). From the perspective of search mechanisms, existing methods have explored various mature evolutionary paradigms (Liu et al., 2024a; Ye et al., 2024; van Stein & Bäck, 2025). These include population-based global search, as in EoH (Liu et al., 2024a), island models that maintain high-quality diversity, as in FunSearch (Romera-Paredes et al., 2024), Monte Carlo tree search (MCTS) Zheng et al. (2025), and iterative search over localized code edits (Zhang et al., 2024). As the design target has grown from a single scoring function into a complex multi-stage algorithmic topology, these mechanisms have been extended to meet increasingly diverse design requirements. This progression spans from single-objective optimization to Pareto trade-offs in multi-objective settings, as in MEoH (Yao et al., 2025), and to multi-component operator systems, as in E2OC (Qiu et al., 2026a) and MOTIF (Kiet et al., 2025), achieving strong empirical performance across a variety of optimization and scheduling tasks.

Despite these growing capabilities, current frameworks encounter limitations rooted in their control flow, modularity, and memory management, as illustrated in Figure 1a. First, they typically embed large language models within rigid, **expert-predefined evolutionary pipelines** rather than empowering them as active search orchestrators. In this setup, the model often functions as a passive code sampler driven by static prompt templates, which restricts interactive debugging, diagnostic analysis, and deep structural refactoring (Wang et al., 2024b). Second, **paradigm-specific coupling** causes severe framework fragmentation, requiring practitioners to maintain separate codebases for distinct design tasks. Third, because these pipelines evaluate candidates using scalar performance feedback (Ye et al., 2024; Qiu et al., 2026b), **amnesic trial-and-error search** often discards the broader structured execution details required to understand why a heuristic succeeded or failed. Consequently, many discovery workflows risk reducing each problem to an isolated trial-and-error procedure that struggles to accumulate or transfer reusable algorithmic experience.

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.

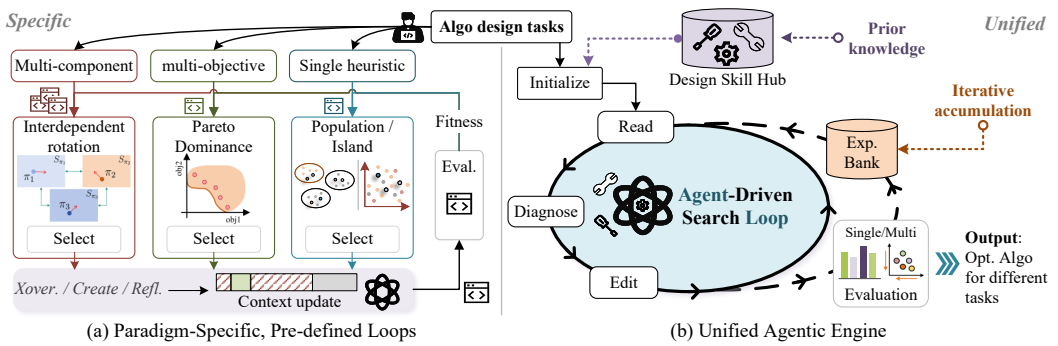


Figure 1: Comparison of search paradigms in LLM-driven AAD. (a) Conventional methods rely on rigid, task-specific pipelines with pre-defined loops and passive LLM sampling, leading to framework fragmentation. (b) AlgoEvo decouples the search loop via a design skill hub and experience bank, enabling an autonomous agent to handle diverse optimization paradigms through interactive tool use and iterative knowledge accumulation.

To address these architectural and memory constraints, we present **AlgoEvo**, an agentic framework that replaces predefined prompt completion with an interactive, self-evolving discovery loop. Rather than following a rigid schedule, AlgoEvo empowers an autonomous agent to direct the discovery process, dynamically determining when to inspect code, run diagnostic analyses, edit components, or evaluate candidates. Operating at the core of this flexible loop, the design skill hub expands the boundaries of automated design by decoupling paradigm-specific knowledge from the search mechanism, allowing a unified agentic process to seamlessly span single-heuristic, multi-objective, and multi-component tasks without structural modification. Concurrently, the hierarchical experience bank deepens optimization within individual tasks and across generations by transforming historical search trajectories into structured experience cards, organizing them into a task-level tree, and retrieving context-relevant insights to guide subsequent iterations. Through this synergy, the framework elevates automated algorithm discovery from isolated trial-and-error routines into a cumulative, self-improving scientific process (Figure 1b).

Our main contributions can be summarized as follows:

- We propose **AlgoEvo**, an end-to-end framework for automated algorithm discovery that integrates an agentic search loop with the design skill hub. Instead of embedding a large language model within a predefined evolutionary pipeline, AlgoEvo delegates the discovery process to an autonomous agent that dynamically reads, edits, diagnoses, and evaluates algorithmic code across single-heuristic, multi-objective, and multi-component settings.
- We develop a hierarchical experience bank that transforms historical search trajectories into structured experience cards and organizes them through a task-level tree for context-aware retrieval. It accumulates and reuses algorithmic experience, ensuring that prior knowledge continually contributes to the evolution and transfer of solutions across discovery processes.
- We conduct extensive experiments across single-heuristic, multi-objective, and multi-component algorithm-design tasks, comparing against specialized methods such as FunSearch, EoH, MEoH, E2OC, and MOTIF. AlgoEvo reaches state-of-the-art quality within 7% of the evaluation budget and matches or surpasses the strongest baseline on all six tasks, by up to 8.8%, and further studies verify continual self-improvement and cross-task knowledge transfer.

2 Automated Algorithm Discovery

Automated algorithm discovery seeks to synthesize executable code or heuristic programs for target optimization problems (Liu et al., 2026b; Hu & Zhang, 2026), ranging from single heuristics to multi-objective strategies and interacting operators for complex systems. The discovery process navigates a candidate program space guided by task-specific evaluations; we formalize this setting as a discovery task $\mathcal{T}$.

Definition 2.1 (Domain and Instance). An optimization domain $\mathcal{D}$ is characterized by an instance space $\mathcal{X}_d$, a solution space $\mathcal{Y}_d$, and a task objective $f_d : \mathcal{X}_d \times \mathcal{Y}_d \to \mathbb{R}$. For an instance $\mathbf{x} \in \mathcal{X}_d$ with feasible set $\mathcal{Y}_d(\mathbf{x}) \subseteq \mathcal{Y}_d$, the underlying optimization problem is

$$\mathbf{y}^* = \arg \min_{\mathbf{y} \in \mathcal{Y}_d(\mathbf{x})} f_d(\mathbf{x}, \mathbf{y}). \tag{1}$$

Definition 2.2 (Solver and Algorithm). A solver s generates a solution using a collection of designable strategies $\mathbf{\Pi} = (\pi_1, \ldots, \pi_K)$ with $\pi_k \in \mathcal{S}_k$, where $\mathcal{S}_k$ is the search space of the k-th strategy. Each strategy may represent a scoring rule, construction policy, neighborhood operator, penalty update mechanism, or another component. The induced algorithm space is $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_K$, where $K = 1$ corresponds to single-heuristic design and $K > 1$ to multi-component design with strategies optimized jointly. The solver's output on instance $\mathbf{x}$ is $s(\mathbf{x} \mid \mathbf{\Pi})$.

Definition 2.3 (Algorithm Optimization). The performance of an algorithm $\mathbf{\Pi}$ on instance $\mathbf{x}$ is measured by

$$F_d(\mathbf{x} \mid \mathbf{\Pi}) = \phi_d \left(f_d \left(\mathbf{x}, s(\mathbf{x} \mid \mathbf{\Pi}) \right) \right), \tag{2}$$

where ϕ_d maps the task-specific evaluation to a scalar measure, i.e., the objective value for single-objective tasks or a scalar indicator such as hypervolume (HV) for multi-objective tasks. We normalize the direction so that smaller F_d is better, and define the discovery objective as

$$\mathbf{\Pi}^* = \arg \min_{\mathbf{\Pi} \in \mathcal{S}} \mathbb{E}_{\mathbf{x} \sim \mathcal{X}_d} \left[F_d(\mathbf{x} \mid \mathbf{\Pi}) \right], \tag{3}$$

subject to a computational budget B bounding evaluation and discovery resources.

3 AlgoEvo: Self-Evolving Agentic Algorithm Discovery

AlgoEvo formulates automated algorithm design as an agentic discovery process over the algorithm space $\mathcal{S}$. Unlike conventional pipelines with fixed generation-and-evaluation cycles, an autonomous agent dynamically inspects, modifies, evaluates, and diagnoses algorithms based on runtime feedback (Figure 2). This process is supported by two complementary components: the design skill hub, which provides paradigm-specific execution contracts to let a single loop span single-heuristic, multi-objective, and multi-component tasks; and the hierarchical experience bank, which stores distilled trajectories and retrieves context-relevant insights to guide exploration. Concretely, the pipeline proceeds in five stages: (1) a design skill is *activated* for the task; (2) an experience tree is *matched* or created; (3) the agent *searches* by inspecting, editing, diagnosing, and evaluating; (4) each evaluation is *summarized* into an experience card; and (5) across tasks, accumulated experience *refines* the skill. Through this synergy, flexible agentic actions and experience-driven selection drive both intra-task optimization and continual cross-task self-evolution.

3.1 Agentic Search Loop

AlgoEvo explores the algorithm space through an *agent-driven loop*. Unlike rigid pipelines with predefined generation-evaluation cycles, the agent dynamically decides its execution trajectory, observing the discovery state, selecting and executing an operation, and processing the feedback until the budget is exhausted or it terminates.

At step t, the discovery state is $s_t = (\mathbf{\Pi}_t, \mathcal{C}_t, \epsilon, F_d(\mathbf{x} \mid \mathbf{\Pi}_t))$, comprising the current algorithm $\mathbf{\Pi}_t$, working context $\mathcal{C}_t$, experience tree ϵ, and performance feedback $F_d(\cdot)$. The working context maintains code, recent modifications, diagnostic observations, and intermediate reasoning.

Given s_{t-1}, the agent selects an action a_t to inspect code, modify a strategy, run a diagnosis, or evaluate a candidate, thus determining both the operation and its timing. Executing a_t yields a transition $\tau_t = (s_{t-1}, a_t, r_t, s_t)$. When a_t updates the algorithm from $\mathbf{\Pi}_{t-1}$ to $\mathbf{\Pi}_t$, the immediate reward is the performance improvement, positive when the update lowers F_d,

$$r_t = F_d(\mathbf{x} \mid \mathbf{\Pi}_{t-1}) - F_d(\mathbf{x} \mid \mathbf{\Pi}_t), \tag{4}$$

so that maximizing the cumulative improvement $\sum_{t=1}^{T} r_t$ reduces to minimizing the final performance $F_d(\mathbf{x} \mid \mathbf{\Pi}_T)$, thereby recovering the global objective in Eq. 3. Rather than following a fixed cycle, the agent interleaves diagnosis, modification, and evaluation, invoking targeted analysis when bottlenecks arise and structural edits when supported by evidence.

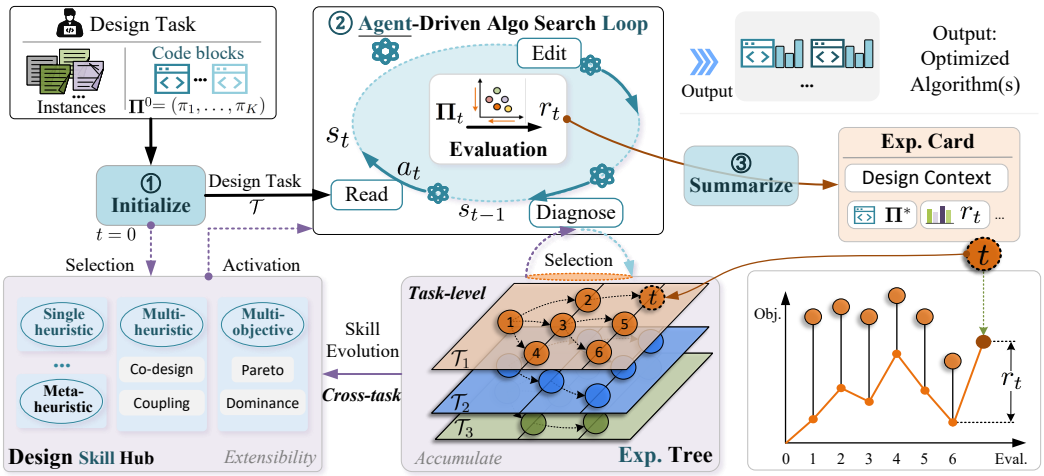


Figure 2: Overview of AlgoEvo. Given a target design task, (1) the design skill hub activates the appropriate algorithm design paradigm; (2) an autonomous agent drives the discovery process through an interactive search loop of code inspection, diagnosis, editing, and evaluation; and (3) performance feedback generates structured experience cards. These cards populate a task-level experience tree that guides ongoing exploration, while successful patterns accumulate across tasks to continually evolve the skill hub.

3.2 Design Skill Hub

The algorithm space $\mathcal{S}$ specifies what can be designed, but its structure is paradigm-dependent. A single-heuristic constructive rule, a multi-objective evolutionary operator, and a coupled destroy-repair system differ in the roles their strategies play, the interfaces they expose, and the principles that make them effective. Encoding these differences into the search procedure would yield a separate pipeline for each paradigm, reproducing the fragmentation we seek to avoid. AlgoEvo therefore externalizes them as pluggable *design skills* in a shared design skill hub $\mathcal{S}_{\text{hub}}$, so that a new paradigm is supported by adapting the same general engine with an additional skill rather than by modifying it.

A design skill declares (i) the roles of the designable strategies in $\mathbf{\Pi} = (\pi_1, \ldots, \pi_K)$, (ii) their code interfaces, (iii) the principles that guide valid modifications, and (iv) the task-specific evaluation conventions. Together, these fix the constraints and inductive bias of the search over $\mathcal{S}$, while the search mechanism remains unchanged. Each skill follows a life cycle, being selected and activated when a task begins and refined once the task completes.

Selection and activation ($t = 0$). At task initialization, the hub matches the task to the skill whose declared conditions it satisfies and activates it in the agent's context. The agent then uses the skill to interpret the design roles, reason about valid modifications, and construct algorithms within the corresponding strategy spaces $\mathcal{S}_1, \ldots, \mathcal{S}_K$. Because the paradigm is carried by the skill rather than the loop, the same engine supports all three design settings by changing the skill alone; the skill is thus what specializes a general agent to a particular design paradigm.

Evolution ($t = T$). At task completion, the evidence accumulated during discovery is consolidated into the experience bank; the skill is not updated within a single task, so evolution is triggered only across tasks. Each skill defines the paradigm under which experiences are accumulated, so that retrieved experiences are interpreted within the corresponding design context. Once a skill owns at least μ experience trees, the agent summarizes their most salient patterns and refines the skill. The hub therefore evolves with the tasks it supports, so that later tasks start from a stronger prior. A skill may also encode methodological knowledge from existing algorithm-design approaches, inheriting their inductive bias while retaining the autonomous execution process.

3.3 HIERARCHICAL EXPERIENCE BANK

The design experiences produced by the agentic loop are organized in a hierarchical experience bank $\mathcal{E}$ with three levels: experience cards, a task-level experience tree, and cross-task consolidation.

Level 1: Experience cards. The agent's actions differ in whether they yield a performance signal. Evaluation produces the reward r_t of Eq. 4, while actions that inspect, edit, diagnose, reason, or reflect change the algorithm or context without producing one. The bank is therefore updated at evaluation events, each distilled into an **experience card** recording the design context, the modification, the reward, and the rationale. Cards are the basic units of the bank and persist across evaluations and tasks.

Level 2: Experience tree. Cards from the same task form an experience tree ϵ, where nodes are cards and edges represent derived-from relationships. The tree is traversed using the four operations of MCTS (Świechowski et al., 2023).

Selection. A card is selected to guide the next design decision. Each card e maintains a mean reward $\bar{r}(e \mid q)$ and a visit count $n(e, q)$ under the current situation q, and selection maximizes the upper confidence bound

$$\mathrm{UCB}(e \mid q) = \bar{r}(e \mid q) + c\sqrt{\frac{2 \ln N}{n(e, q)}}, \tag{5}$$

where N is the tree-level visit statistic and c controls the exploration-exploitation trade-off. The situation q, written q_t at step t, is the set of conditions active in the current search state, drawn from $\mathcal{Q} = \{\textit{stagnation}, \textit{bottleneck}, \textit{coupling}, \textit{sparse front}\}$ and detected by the diagnostic engine from runtime attribution, interaction, and search-history signals (Appendix C.2). Since several conditions may hold at once, a card is scored by the statistics it has accumulated under those it matches. When $n(e, q) = 0$, the global mean of e serves as a prior with an exploration incentive.

Expansion. When a design decision yields a verified candidate, its card is attached as a child of the card it evolved from.

Evaluation. The candidate is evaluated, producing the reward r_t of Eq. 4.

Backpropagation. The reward is propagated to the ancestors of the new card c_t, so that a design decision is credited for the outcomes of the candidates it produced. Each ancestor e updates its mean reward $\bar{r}(e \mid q)$ and visit count $n(e, q)$ under the current situation q, so that design directions that consistently produced strong descendants are preferred in subsequent selections.

Level 3: Cross-task consolidation. The experience tree captures knowledge within a single task; across tasks, the bank accumulates broader evidence. Once a skill owns at least μ experience trees, the agent summarizes their most salient patterns and refines the skill, transferring concrete experience into paradigm-level knowledge.

3.4 EXPERIENCE ACCUMULATION AND SELF-EVOLUTION

The self-evolution of AlgoEvo follows a closed-loop MDP process (Algorithm 1). At initialization (Algorithm 3), the system generates a system prompt from the problem description, then uses this prompt to match a skill and retrieve or create an experience tree. Once initialized, the agent autonomously drives the discovery process, selecting at each decision step t an action a_t based on the current state and updating the working context with the outcome. When a_t is *evaluate*, a new experience card is created and attached to the tree; when a_t is *diagnose* or *retrieve*, the agent retrieves relevant experiences to guide its reasoning.

Beyond single-task adaptation, the framework achieves cumulative evolution via cross-task knowledge consolidation (Algorithm 5). Once a skill owns at least μ experience trees, the agent summarizes their most salient patterns and refines the skill accordingly. This hierarchical propagation allows the overarching discovery engine to continuously enhance its foundational capabilities and transfer expertise to new domains without requiring manual redesign or structural modifications.

Algorithm 1 Agentic Algorithm Discovery.

Require: Design task $\mathcal{T}$, skill hub $\mathcal{S}_{\text{hub}}$, experience bank $\mathcal{E}$, budget B, threshold μ
Ensure: Optimized algorithm $\mathbf{\Pi}^*$

$\sigma \leftarrow$ **Activate**$(\mathcal{T}, \mathcal{S}_{\text{hub}})$ ▷ 1. Activate skill
$\epsilon \leftarrow$ **Match**$(\mathcal{T}, \sigma, \mathcal{E})$ ▷ 2. Match/create tree (Algo. 3)
Initialize $\mathbf{\Pi} \leftarrow \mathbf{\Pi}_0$, $\mathcal{C}_t$, $t \leftarrow 0$
repeat
  $q_t \leftarrow$ **Categorize**(s_t) ▷ 3. Detect situation (Algo. 4)
  $a_t \leftarrow$ **Agent**$(\sigma, q_t, \epsilon, \mathcal{C}_t)$; execute a_t; update $\mathcal{C}_t$ ▷ Inspect/edit/diagnose/retrieve/evaluate
  if $a_t =$ evaluate: update $\mathbf{\Pi}$ and r_t, summarize card to ϵ ▷ 4. Summarize (Algo. 6)
  if $a_t \in \{\text{diagnose}, \text{retrieve}\}$: select card e via Eq. 5 ▷ Use card
  $t \leftarrow t + 1$
until B exhausted or $a_t =$ terminate
Evolve$(\sigma, \mathcal{E}, \mathcal{S}_{\text{hub}}, \mu)$ ▷ 5. Refine skill (Algo. 5)
return $\arg\min_{\mathbf{\Pi}} F_d(\mathbf{x} \mid \mathbf{\Pi})$

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmark. The benchmark comprises six problems across three paradigms, with objectives, instance sets, and code contracts detailed in Appendix D.1: constructive heuristic design for TSP and CVRP, multi-objective design for Bi-TSP and Bi-FJSP, and multi-component co-design for CVRP-DR and FJSP 4-Ops (Liu et al., 2026a; Chen et al., 2023; Kiet et al., 2025; Brandimarte, 1993). Each task features disjoint training and held-out test sets, with all instances, random seeds, and evaluation scripts publicly released.

Baselines and protocol. We compare against each paradigm's strongest method. On TSP and CVRP these are nearest-neighbor heuristics and the LLM-AHD methods EoH (Liu et al., 2024a), ReEvo (Ye et al., 2024), MCTS-AHD (Zheng et al., 2025), and FunSearch (Romera-Paredes et al., 2024); on the multi-objective tasks, MEoH (Yao et al., 2025), NSGA-II (Deb et al., 2002), and MOEA/D (Zhang & Li, 2007); and on the multi-component tasks, MOTIF (Kiet et al., 2025), E2OC (Qiu et al., 2026a), and the EoH variants Synergy and Rotating-EoH. All methods run on the LLM4AD platform (Liu et al., 2024b) under one shared protocol, with identical instance splits and evaluation entries, three seeds, and 500 evaluations per seed, using `gpt-4o-mini` on TSP and CVRP and `DeepSeek-V4-Flash` elsewhere. Because AlgoEvo writes and runs its own code, its budget is guarded against evasion by cross-checked counters, and any run that scores a candidate outside the shared evaluation entry is discarded rather than reported (Appendix D.4). Remaining parameters are in Appendix D.3.

Metrics. For multi-objective results, a shared global normalization basis ensures comparability of HV and Inverted Generational Distance (IGD) within each problem. Appendix D.2 formalizes these metrics and notes that multi-objective rows report training-instance quality since the final algorithm is fixed post-run. In Table 1, the evaluation column reports the mean number of candidate evaluations consumed per design task, and the token column reports cumulative backbone usage in millions. All reported dispersions denote sample standard deviations over random seeds.

4.2 MAIN RESULTS

To test whether a single agentic engine can match or surpass paradigm-specialized baselines, we evaluate AlgoEvo on all six tasks, changing only the activated design skill.

Single-heuristic design. On TSP and CVRP AlgoEvo attains the best training and test quality, ahead of the strongest LLM-AHD baseline by 4 to 7% and of the nearest-neighbour heuristic by 12.3% on TSP. The margins are not bought with budget, as the baselines run to the 500-evaluation cap and remain behind, while AlgoEvo finds its best incumbent by about 35 evaluations, a thirteenth of the allowance.

Multi-objective design. AlgoEvo attains, on training instances, the best HV on both tasks and the best IGD on Bi-TSP, while trailing MEoH on front proximity for Bi-FJSP. The archives reveal that our front reaches low-objective corners missed by the baselines, whereas the MOEA/D front collapses to a point (Figure 11).

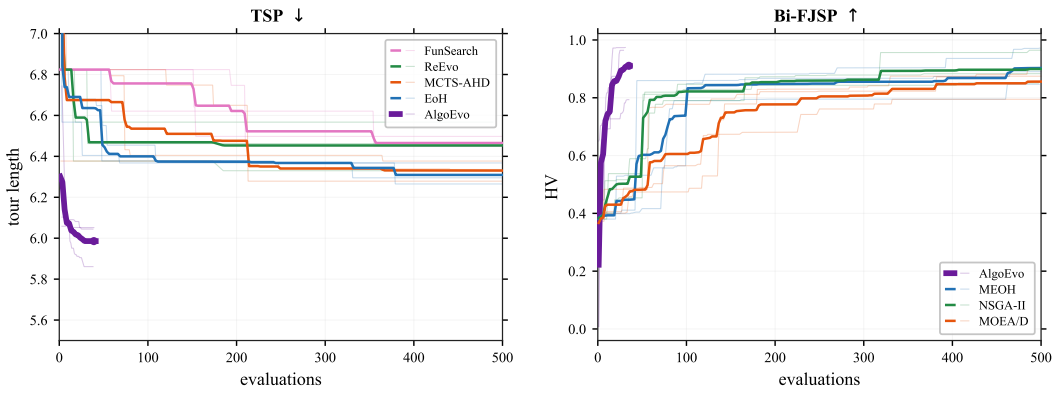
Multi-component design. Here the components are coupled and no single unit suffices. AlgoEvo obtains the best result of any method on both splits of CVRP-DR and the best held-out result on FJSP four-operator co-design, where it also improves on the expert operators. Unlike the four tasks above, it draws on most of the allowance rather than stopping early, so the margins come at or below the budget the baselines spend, and they are widest on the held-out sets, which is where a design has to transfer rather than to fit.

4.3 Analysis and Discussion

Sample efficiency. AlgoEvo reaches final quality within tens of evaluations without further improvement, whereas baselines keep progressing toward full budget without catching up (Figure 3). That curvature is not an artefact of early stopping, since the agent keeps proposing changes until the

Table 1: Performance on all six tasks. On the quality metrics the best result is **bold** and the runner-up underlined; on the Evaluations and Tokens columns, where smaller is better, the best result is **bold**. HV is the only metric for which larger is better. The columns are defined in Section 4.1.

Single heuristic	**Method**	**Train**	**Test**	**Evals**	**Tokens (M)**
TSP	NN	6.824	7.990	–	–
	EoH	6.309 ± 0.043	7.392 ± 0.058	500	4.1
	ReEvo	6.453 ± 0.097	7.699 ± 0.220	500	4.3
	MCTS-AHD	6.331 ± 0.040	7.337 ± 0.028	500	3.8
	FunSearch	6.465 ± 0.023	7.484 ± 0.051	500	3.2
	AlgoEvo	5.986 ± 0.11	7.007 ± 0.04	39	2.9
CVRP	NN	13.611	26.283	–	–
	EoH	13.537 ± 0.028	26.148 ± 0.169	500	4.1
	ReEvo	13.403 ± 0.129	25.950 ± 0.517	500	4.3
	MCTS-AHD	13.236 ± 0.302	25.967 ± 0.447	500	4.1
	FunSearch	13.584 ± 0.030	26.270 ± 0.013	500	3.9
	AlgoEvo	12.698 ± 0.39	24.150 ± 0.40	35	1.9
Multi-objective	**Method**	**HV ↑**	**IGD ↓**	**Evals**	**Tokens (M)**
Bi-TSP	MEoH	0.760 ± 0.020	0.077 ± 0.018	500	4.2
	NSGA-II	0.614 ± 0.067	0.215 ± 0.067	500	3.6
	MOEA/D	0.314 ± 0.180	0.601 ± 0.172	500	3.5
	AlgoEvo	0.827 ± 0.012	0.036 ± 0.003	36	2.6
Bi-FJSP	MEoH	0.903 ± 0.051	**0.143 ± 0.077**	500	6.3
	NSGA-II	0.899 ± 0.048	0.149 ± 0.067	500	5.7
	MOEA/D	0.792 ± 0.002	0.286 ± 0.000	500	5.2
	AlgoEvo	0.911 ± 0.083	0.162 ± 0.125	33	2.4
Multi-component	**Method**	**Train**	**Test**	**Evals**	**Tokens (M)**
CVRP-DR	Standard	10.791	10.558	–	–
	MOTIF	9.156 ± 0.164	9.204 ± 0.114	500	317.1
	E2OC	9.920 ± 0.078	9.837 ± 0.098	500	470.0
	Synergy	8.928 ± 0.075	9.112 ± 0.043	500	289.6
	Rotating-EoH	8.998 ± 0.067	9.178 ± 0.166	500	337.9
	AlgoEvo	8.860 ± 0.037	8.999 ± 0.036	271	139.2
FJSP 4-Ops	Expert	712.5 ± 5.0	2137.9 ± 12.0	–	–
	MOTIF	648.07 ± 3.43	1952.00 ± 23.41	500	584.7
	E2OC	663.47 ± 7.62	2019.20 ± 44.53	500	699.3
	Synergy	646.10 ± 2.95	1965.20 ± 14.81	500	460.7
	Rotating-EoH	**645.10 ± 3.37**	1954.87 ± 26.14	500	518.5
	AlgoEvo	646.4 ± 3.3	1927.5 ± 38.5	275	118.1



(a) TSP constructive heuristic design.

(b) Bi-FJSP multi-objective heuristic design.

Figure 3: Best-so-far quality as a function of the evaluation budget on TSP and Bi-FJSP. Thin lines are individual seeds; bold lines are means. AlgoEvo reaches its final quality within the first tens of evaluations and then stays flat, whereas the baselines consume their full budget.

harness limit is reached and few proposals improve on the incumbent; the flat tail therefore reflects the absence of further improvement rather than a deliberate stop.

Experience verification. Within a task, the experience bank grows as the agent evaluates candidates, reusing earlier design decisions rather than exploring from scratch, so cost decreases as cards accumulate. The agent reaches its final quality well before the budget is exhausted. Experience also carries over between successive runs of the same task, where Table 2 shows that the warm-started agent consistently improves over the cold start while reducing the variance, indicating that transferring executable code is more effective than injecting abstract strategy descriptions.

Table 2: Effect of accumulated experience on TSP. **cold** starts from the default heuristic; **warm** from the best algorithm of a previous run.

Seed	Cold (no exp.)		Warm (with exp.)	
	Tour	Evals	Tour	Evals
2025	5.006	45	**4.819**	52
2026	4.879	48	**4.810**	28
2027	4.997	51	**4.794**	41
Mean	4.961	48	**4.808**	40

Multi-round progressive accumulation Table 3 isolates the contribution of the experience bank with identical budgets. Round 0 initializes from a default heuristic with an empty bank, and each subsequent round resumes from the preceding best code, with **Δ** measured against Round 0. Both problems exhibit steady, monotonic improvement, advancing from 4.8928 to 4.7995 in tour length and 0.1549 to 0.1862 in HV. Because the evaluation budget remains fixed, this progressive enhancement confirms that accumulated experience actively guides search toward superior regions. Furthermore, promoting validated patterns into the design skill hub reinforces search stability (Algorithm 5), and ablating the hub degrades performance (Appendix F.3).

Table 3: Performance across successive optimization rounds on TSP and Bi-objective FJSP.

Round	TSP constructive		Bi-objective FJSP	
	Tour ↓	Δ	HV ↑	Δ
0	4.8928	—	0.1549	—
1	4.8011	−1.88%	0.1688	+8.97%
2	4.8011	−1.88%	0.1832	+18.27%
3	4.8003	−1.89%	0.1840	+18.79%
4	**4.7995**	**−1.91%**	**0.1862**	**+20.21%**

Cross-task transfer. The experience bank facilitates knowledge transfer across problem settings. As shown in Table 4, transferring distilled patterns from single-objective CVRP to multi-component CVRP-DR allows AlgoEvo to reuse edge-scoring rules within the constructor. This cross-task transfer enhances solution quality and lowers the mean cost from 10.314 to 9.816 (surpassing the cold-start baseline of 10.791), demonstrating that the hierarchical experience bank enables rapid adaptation and superior performance on complex downstream tasks.

Table 4: Cross-problem experience transfer. A bank of validated design experience from the source task is made available to the target task, against a cold start that differs only in the absence of that experience; the better arm is **bold**.

Source → target	Metric	Baseline	Cold start	With experience	Change
CVRP → CVRP-DR	cost ↓	10.791	10.314 ± 0.345	$\mathbf{9.816 \pm 0.642}$	-4.8%
TSP → Bi-TSP	HV ↑	152.2	336.5 ± 130.6	$\mathbf{352.2 \pm 141.6}$	$+4.7\%$

Action behavior. We instrument the framework to log every tool invocation and analyze the resulting trajectories on CVRP-DR across three seeds. Each call is classified as read, edit, diagnose, or evaluation, and Figure 4 summarizes their evolution over time. The three seeds exhibit different search styles, one edit-driven, another diagnosis-driven, and the third mixing both. The diagnosis-driven seed reaches the best cost, consistent with the diagnosis-guided design principle, in which investing in structural attribution before editing avoids wasted modifications.

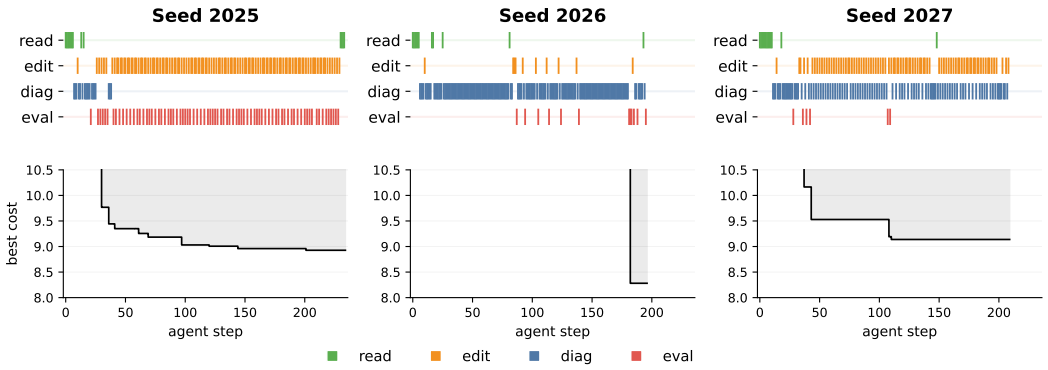


Figure 4: Agent behavior trajectories on CVRP-DR across three seeds. Each row is an action type (evaluation, diagnose, edit, read); the black curve shows the best cost achieved so far.

4.4 Method Skills for Automated Algorithm Design

The design skill abstraction is not limited to knowledge accumulated by AlgoEvo itself; it can also carry methodological knowledge distilled from existing algorithm-design methods. We build method skills from six established approaches spanning single- and multi-objective design, and supply them to AlgoEvo without importing their native search procedures. The skill-based variants match or exceed the native frameworks on solution quality while requiring substantially fewer evaluations, showing that methodological knowledge can be transferred and reused independently of the framework that produced it. The full protocol and results are given in Appendix G.1.

5 Conclusion

In this paper, we introduced AlgoEvo, a unified agentic framework that transforms automated algorithm discovery from rigid, expert-predefined pipelines into an adaptive, knowledge-accumulating search process. By decoupling paradigm-specific knowledge through the design skill hub and organizing discovery trajectories via the hierarchical experience bank, AlgoEvo enables a single engine to seamlessly govern single-heuristic, multi-objective, and multi-component tasks. Extensive empirical evaluations across six representative combinatorial optimization benchmarks demonstrate that our state-aware search control achieves state-of-the-art solution quality while consuming as little as 7% of the standard evaluation budget. Furthermore, our findings highlight the efficacy of intra-task experience accumulation and cross-task knowledge transfer in bootstrapping complex algorithmic co-design. This work establishes a principled and scalable foundation for self-evolving automated algorithm discovery, opening new avenues for intelligent scientific optimization.

AI USE STATEMENT

Generative AI tools were used to aid and polish the writing of this paper. All AI-generated content was reviewed and verified by the authors, who take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We document the parameter settings and the concrete implementation of the framework in the appendix, including the pseudocode of the discovery loop and the benchmark procedures. The code is provided to the reviewers as supplementary material and will be released publicly upon acceptance.

ETHICS STATEMENT

This work studies automated algorithm design on standard combinatorial optimization benchmarks. It does not involve human subjects, personal or sensitive data, or deployment in a safety-critical setting, and we are not aware of ethical concerns that it raises.

REFERENCES

Paolo Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41(3):157–183, 1993.

Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. In *Findings of the Association for Computational Linguistics (ACL)*, pp. 16803–16822, 2026.

Jinbiao Chen, Zizhen Zhang, Zhiguang Cao, Yaoxin Wu, Yining Ma, Te Ye, and Jiahai Wang. Neural multi-objective combinatorial optimization with diversity enhancement. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 39176–39188, 2023.

Nuo Chen, Yicheng Tong, Yuzhe Yang, Yufei He, Xueyi Zhang, Zou Qingyun, Qian Wang, and Bingsheng He. Diversity collapse in multi-agent LLM systems: Structural coupling and collective failure in open-ended idea generation. In *Findings of the Association for Computational Linguistics (ACL)*, pp. 251–306, 2026.

Carlos A. Coello Coello and Nareli Cruz Cortés. Solving multiobjective optimization problems using an artificial immune system. *Genetic Programming and Evolvable Machines*, 6(2):163–190, 2005.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. HSEvo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 39, pp. 26931–26938, 2025.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

Jingzhi Gong, Ruizhen Gu, Zhiwei Fei, Yazhuo Cao, Lukas Twist, Alina Geiger, Shuo Han, Dominik Sobania, Federica Sarro, and Jie M Zhang. SkillMOO: Multi-objective optimization of agent skills for software engineering. *arXiv preprint arXiv:2604.09297*, 2026.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

Qinglong Hu and Qingfu Zhang. Partition to evolve: Niching-enhanced evolution with LLMs for automated algorithm discovery. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, 2026.

Nguyen Viet Tuan Kiet, Dao Van Tung, Tran Cong Dao, and Huynh Thi Thanh Binh. MOTIF: Multi-strategy optimization via turn-based interactive framework. In *arXiv preprint arXiv:2508.03929*, 2025.

Yubo Li. Dynamic agent skills: A lifecycle survey and taxonomy of evolving skill libraries. *arXiv preprint arXiv:2607.10113*, 2026.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pp. 32201–32223, 2024a.

Fei Liu, Rui Zhang, Zhuoliang Xie, Rui Sun, Kai Li, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. LLM4AD: A platform for algorithm design with large language model. In *arXiv preprint arXiv:2412.17287*, 2024b.

Fei Liu, Yilu Liu, Qingfu Zhang, Xialiang Tong, and Mingxuan Yuan. EoH-S: Evolution of heuristic set using LLMs for automated heuristic design. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 40, pp. 37090–37098, 2026a.

Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Zhe Zhao, Xialiang Tong, Kun Mao, Zhichao Lu, Zhenkun Wang, et al. A systematic survey on large language models for algorithm design. *ACM Computing Surveys*, 58(8):1–32, 2026b.

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

Jinghao Luo, Yuchen Tian, Chuxue Cao, Ziyang Luo, Hongzhan Lin, Kaixin Li, Chuyi Kong, Ruichao Yang, and Jing Ma. From storage to experience: A survey on the evolution of LLM agent memory mechanisms. In *Findings of the Association for Computational Linguistics (ACL)*, pp. 41622–41652, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Junhao Qiu, Xin Chen, Liang Ge, Liyong Lin, Zhichao Lu, and Qingfu Zhang. Evolving interdependent operators with large language models for multi-objective combinatorial optimization. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026a.

Junhao Qiu, Haoyang Zhuang, Fei Liu, Jianjun Liu, and Qingfu Zhang. EvoDR: Evolving dispatching rules via large language model for dynamic flexible assembly flow shop scheduling. *arXiv preprint arXiv:2601.15738*, 2026b.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Gabriel Sarch, Lawrence Jang, Michael J Tarr, William W Cohen, Kenneth Marino, and Katerina Fragkiadaki. Vlm agents generate their own memories: Distilling experience into embodied programs of thought. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 75942–75985, 2024.

Shuzheng Si, Haozhe Zhao, Yu Lei, Qingyi Wang, Dingwei Chen, Zhitong Wang, Zhenhailong Wang, Kangyang Luo, Zheng Wang, Gang Chen, et al. From context to skills: Can language models learn from context skillfully? *arXiv preprint arXiv:2604.27660*, 2026.

Hongwen Song and Song Wei. More skills, worse agents? skill shadowing degrades performance when expanding skill libraries. *arXiv preprint arXiv:2605.24050*, 2026.

Thomas Stützle and Manuel López-Ibáñez. Automated design of metaheuristic algorithms. In *Handbook of Metaheuristics*, pp. 541–579. Springer, 2019.

Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3): 2497–2562, 2023.

Darshan Tank and Baran Nama. The regression tax: Decomposing why skills help and hurt LLM agents. *arXiv preprint arXiv:2607.22520*, 2026.

Niki van Stein and Thomas Bäck. LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29 (2):331–345, 2025.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024a.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024c.

Xingyu Wu, Sheng-hao Wu, Jibin Wu, Liang Feng, and Kay Chen Tan. Evolutionary computation in the era of large language model: Survey and roadmap. *IEEE Transactions on Evolutionary Computation*, 29(2):534–554, 2024.

Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. How memory management impacts LLM agents: An empirical study of experience-following behavior. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 623–645, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, pp. 17577–17604, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 50528–50652, 2024.

Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. Multi-objective evolution of heuristic using large language model. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 39, pp. 27144–27152, 2025.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 43571–43608, 2024.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel Agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 27890–27913, 2025.

Chi Zhang, Yimin Liu, Xinze Chen, and Ping Ji. What keeps agent skills from being reusable? evidence from 138k SKILL.md files. In *First Workshop on Agent Skills*, 2026.

Qingfu Zhang and Hui Li. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.

Rui Zhang, Fei Liu, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Understanding the importance of evolutionary search in automated heuristic design with large language models. In *Proceedings of the 17th International Conference on Parallel Problem Solving from Nature (PPSN XVII)*, pp. 185–202, 2024.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 78338–78373, 2025.

Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms—a comparative case study. In *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature (PPSN V)*, pp. 292–301, 1998.

AlgoEvo: Self-Evolving Agentic Search for Automated Algorithm Discovery

APPENDIX CONTENTS

A Questions and Clarifications **15**

B Related Work **17**

C Method Details **19**

C.1 Task Specification and Initialization 19

C.2 Discovery State, Actions, and Diagnosis 20

C.3 Design Skill Hub 22

C.4 Hierarchical Experience Bank 24

C.5 Concrete Artifacts and Prompts 25

D Benchmark and Evaluation Details **29**

D.1 Benchmark Tasks and Instance Sets 29

D.2 Metrics 30

D.3 Baseline Details 30

D.4 Anti-cheating settings 31

E Design Space and Task Instantiation **31**

F Additional Experimental Results **33**

F.1 Convergence Under the Evaluation Budget 33

F.2 Pareto Archives on Multi-Objective Tasks 34

F.3 Ablation of Knowledge Mechanisms 35

G Further Analysis **35**

G.1 Method Skills and Methodological Transfer 35

G.2 Representative Diagnosis-Guided Design Trajectory 37

G.3 Variation in Search Behavior 38

G.4 Cross-Task Experience Transfer 38

G.5 Limitations and Future Work 39

A Questions and Clarifications

This section addresses several questions that may naturally arise when comparing AlgoEvo with existing LLM-based algorithm design methods and general-purpose agentic systems. The purpose is not to restate the framework, but to clarify its methodological positioning, design rationale, and scope. Detailed descriptions of the framework and its individual components are provided in the main paper.

> **Question 1.** *What is the fundamental novelty of AlgoEvo beyond existing LLM-based algorithm design methods?*

The fundamental novelty does not lie in utilizing an LLM for code generation or introducing an agentic execution loop, but in a dual contribution: first, formulating the *algorithm-design process itself as an adaptive search problem*, and second, establishing a *unified framework* that seamlessly spans single-heuristic, multi-objective, and multi-component design tasks. Existing methods typically rely on prescribed, static protocols, such as fixed generation-refinement loops or evolutionary search, and remain fragmented across specific problem paradigms. In contrast, AlgoEvo elevates the search space from generating individual algorithm candidates to orchestrating the discovery process dynamically. Guided by the design skill hub and the hierarchical experience bank, the agent adapts its search trajectory based on runtime evidence across diverse paradigms. The core contribution is thus a state-aware search framework that unifies automated algorithm design under an interactive control process, rather than serving as another task-specific heuristic generator.

> **Question 2.** *How does AlgoEvo differ specifically from existing frameworks such as EoH, ReEvo, FunSearch, and MCTS-AHD?*

While those methods have advanced automated algorithm design significantly by employing LLMs to generate, evaluate, or organize candidates, their underlying search procedures remain bound to fixed, rigid pipelines. AlgoEvo introduces a higher-level control mechanism that operates above candidate generation. Instead of repeatedly executing a uniform schedule, our agent dynamically selects among heterogeneous design actions according to the evolving search state. This enables the search to adapt its granularity and direction in response to runtime feedback. For instance, a stagnant search triggers diagnostic attribution rather than unconditioned code generation; evidence of component-level weakness motivates targeted refactoring; and strong cross-component interactions prompt joint redesign. Thus, AlgoEvo does not replace existing candidate-generation capabilities, but rather orchestrates them within a flexible, state-driven workflow.

> **Question 3.** *Is AlgoEvo simply a general-purpose coding agent applied to heuristic design?*

No. Such a comparison overlooks the fundamental divergence in optimization targets, execution paradigms, and architectural grounding between software engineering and algorithm discovery. A general-purpose coding agent is designed to produce a software artifact satisfying static functional specifications or test cases, where progress is governed by syntactic correctness and binary pass-or-fail test results. In contrast, AlgoEvo addresses an empirical optimization problem over complex instance distributions, where the output is an algorithmic decision rule and feedback manifests as a continuous quantitative performance signal.

Crucially, this structural divide becomes starkly evident in advanced paradigms. For instance, in multi-objective optimization, a generic coding agent lacks the inherent capability to discover, maintain, and balance trade-off Pareto fronts, typically degrading into uncoordinated single-objective outputs. Similarly, in multi-component co-design, generic agents fail to manage intricate cross-component dependencies without exhaustive and brittle prompt engineering. AlgoEvo overcomes these barriers through domain-specific architectural design. Specifically, the design skill hub and the hierarchical experience bank are not generic text prompt modules, but specialized components that interface directly with the task evaluation harness and execution contracts. Through this integration, the agent continuously leverages external performance feedback to refine its internal harness interactions, accumulating hierarchical experience across the trajectory to progressively elevate single-task performance and control search directions. The agent does not merely decide how to edit code based on general instructions, but systematically determines which algorithmic component requires

structural redesign and what type of action to execute over an empirical performance landscape. Consequently, viewing the framework as a wrapper around a general-purpose coding agent fundamentally misinterprets its nature, as its efficacy stems from principled, harness-integrated search control and continuous experience accumulation rather than ad-hoc text manipulation.

> **Question 4.** *How is AlgoEvo different from recent skill-based or skill-evolving agent frameworks?*

AlgoEvo shares with skill-based agents the premise that reusable knowledge enhances problem-solving efficiency, but the semantics and role of a skill are fundamentally different. In general-purpose agent systems, skills typically denote reusable executable workflows for task execution, such as tool invocation or API interaction, and skill evolution focuses on completing task-oriented procedures. In contrast, a design skill here represents reusable algorithm-design knowledge. It encapsulates the mechanism and rationale of an optimization paradigm, its applicable conditions, design principles, and characteristic failure modes. It is not an executable workflow, but abstract knowledge regarding how an algorithmic component should be structured or tuned. Furthermore, skills are coupled dynamically to the optimization state rather than accumulated indiscriminately. Relevant knowledge is invoked when runtime diagnostics indicate a matching scenario, and successful trajectories are consolidated into paradigm-level knowledge. The novelty thus lies in grounding reusable skills within the specialized algorithm-design search space to govern state-aware search control.

> **Question 5.** *Are the design skill hub and the experience bank merely forms of prompt engineering or retrieval-augmented generation?*

No. While both mechanisms supply context to the LLM, their functional roles extend far beyond prompt augmentation or standard text retrieval, and they serve distinct purposes. The design skill hub maintains overarching methodological knowledge, including paradigm principles, instantiation templates, and convergence characteristics, whereas the experience bank records validated empirical trajectories paired with their specific contexts and observed rewards. The former provides generalized design principles, while the latter offers concrete empirical evidence. More importantly, both actively participate in search control rather than passive context assembly. Their outputs directly dictate structural decisions, such as which component to isolate, whether to perform targeted versus joint modifications, and which operational action to execute next, while successful empirical experiences are systematically abstracted back into reusable skills. These mechanisms thus establish a rigorous closed-loop pathway from empirical discovery to methodological knowledge and adaptive search control.

> **Question 6.** *Why is adaptive search control necessary, and why not use a fixed evolutionary loop with the same evaluation budget?*

A fixed evolutionary loop implicitly assumes that a uniform search action remains appropriate throughout the entire optimization process. This assumption is overly restrictive, as the required design granularity evolves as the search progresses. Early stages benefit from broad exploration of alternative paradigms, whereas later stages often demand targeted refactoring of a single bottleneck, diagnostic analysis of persistent failure modes, or coordinated co-design of interacting components. AlgoEvo leverages the current discovery state to dynamically select the most suitable action, avoiding the inefficiency of repeatedly applying suboptimal operations. Under identical evaluation budgets, our ablation studies confirm that removing these experience- and skill-driven control mechanisms leads to marked performance degradation.

> **Question 7.** *Does diagnosis provide information beyond ordinary evaluation or LLM reflection?*

Yes, because evaluation and diagnosis serve fundamentally different roles in the discovery process. Evaluation reveals whether a candidate improves the global objective, whereas diagnosis identifies where and why a current design underperforms. Specifically, diagnosis isolates component-level attribution, analyzes interaction effects, and pinpoints structural failure modes to directly guide subsequent action selection. It goes beyond generic LLM self-reflection by grounding reasoning

in quantitative counterfactual probes, such as no-op substitution. For instance, in our CVRP-DR experiments, diagnosis first pinpointed a flawed destruction operator and subsequently uncovered severe coupling between destruction and repair, prompting a transition from isolated editing to joint co-design. In short, evaluation provides the scalar search signal, whereas diagnosis provides the structural search direction.

> **Question 8.** *Could the performance gains simply result from a stronger LLM, more evaluations, or a larger computational budget?*

No. Empirical evidence refutes this across three distinct aspects. First, AlgoEvo does not rely on a larger computational budget. Rather, it achieves state-of-the-art performance using only 7% of the standard evaluation budget, requiring just 35 evaluations compared to the baseline 500. Second, under identical backbone models, AlgoEvo actually reduces cumulative token consumption. By replacing redundant and amnesic trial-and-error sampling with targeted agentic search, it avoids token waste. Third, controlled ablation studies confirm the source of these gains. When we systematically disable the hierarchical experience bank, design skill hub, or agentic loop while keeping base models, task prompts, and evaluation limits strictly identical, performance drops significantly. Thus, the empirical improvements originate entirely from structural algorithmic orchestration rather than model scaling or computational excess.

> **Question 9.** *How does the protocol ensure that the reported evaluations reflect genuine algorithm design?*

To guarantee that reported performance gains stem from genuine algorithmic intelligence rather than artifacts, loopholes, or reward hacking, AlgoEvo enforces a multi-layer verification and isolation protocol. First, all agent-synthesized heuristic codes are executed within a strictly sandboxed, resource-bounded environment that blocks arbitrary system calls, network access, and unauthorized caching. Second, enforced by the Design Skill Hub, candidates must strictly adhere to immutable interface contracts and type signatures; any code attempting to bypass standard data structures or inject malicious overrides fails structural validation pre-evaluation. Third, to prevent hardcoding or instance-level overfitting, evaluation harnesses compute performance over a randomized, disjoint set of validation instances with hidden evaluation seeds. Finally, every evaluation is immutably tracked by a harness-managed budget counter (detailed in Section C.1 and Table 5) that is cross-validated against raw execution logs, ensuring complete auditability and automatically invalidating any run attempting to circumvent the evaluation pipeline via private pathways (fully documented in Section D.4).

> **Question 10.** *How well does AlgoEvo generalize beyond the evaluated tasks and settings?*

We evaluate generalization across multiple dimensions. AlgoEvo is instantiated across six diverse tasks spanning single-heuristic, multi-objective, and multi-component optimization paradigms, involving varied representation spaces and problem scales between training and testing instances. Additionally, we validate knowledge transfer through cross-task experience reuse, as detailed in Section G.4, demonstrating that insights from prior optimization runs accelerate and enhance subsequent discovery processes. While our current scope focuses on combinatorial optimization and heuristic design, extending these search-control principles to broader scientific and engineering domains presents a promising direction for future work.

B Related Work

AlgoEvo lies at the intersection of automated algorithm design, agentic software engineering, and knowledge-guided search. We therefore review related work from both the algorithm-design and general agent perspectives. The fundamental distinction is not merely whether an approach utilizes language models, tools, memory, or skills, but what is being searched, how the search state is defined, and how empirical feedback governs subsequent actions.

LLM-based Automated Algorithm and Heuristic Design. LLMs have recently enabled a powerful paradigm for automated algorithm and heuristic design (Liu et al., 2026b). FunSearch combines

LLM-based program generation with evolutionary search, demonstrating the capacity of language models to discover programs with improved performance (Romera-Paredes et al., 2024). EoH formulates heuristic design as an iterative process of generating heuristic ideas and implementations and progressively improving them through evaluation (Liu et al., 2024a). ReEvo further introduces evolutionary refinement of LLM-generated heuristics, enabling iterative improvement of previously discovered designs (Ye et al., 2024). MCTS-AHD employs Monte Carlo tree search to structure the exploration of algorithm-design candidates (Zheng et al., 2025). In multi-objective settings, MEoH extends LLM-based heuristic design to multi-objective evolutionary search (Yao et al., 2025). Additional recent approaches include EvoPrompt, which connects LLMs with evolutionary algorithms for prompt optimization (Guo et al., 2024), LLaMEA, which uses LLMs to automatically generate metaheuristics (van Stein & Bäck, 2025), and HSEvo, which combines harmony search with genetic algorithms via LLMs (Dat et al., 2025). EoH-S further extends EoH to heuristic set design (Liu et al., 2026a). These studies establish that LLMs can serve as effective generators and refiners of algorithmic ideas and executable implementations. However, their underlying search procedures and multi-component coordination policies remain bound to fixed, predefined protocols. Traditional automated configuration (e.g., irace (López-Ibáñez et al., 2016)) and recent multi-component frameworks like MOTIF (Kiet et al., 2025) and E2OC (Qiu et al., 2026a) demonstrate that search strategies and cross-component interactions are critical, yet they typically instantiate coordination via static, algorithm-specific rules rather than adaptive control.

Agentic Coding and Program Synthesis. Recent advances in LLM-based coding agents (e.g., SWE-agent (Yang et al., 2024) and MetaGPT (Hong et al., 2024)) demonstrate autonomous code generation, repository exploration, and iterative debugging, while performance-driven evolutionary coding agents such as AlphaEvolve (Novikov et al., 2025) optimize programs against empirical objectives (Wang et al., 2024a). Both, however, operate over concrete programs rather than algorithm-design knowledge: general-purpose software agents are driven by functional correctness, syntactic validity, and test compliance against static specifications, while evolutionary coding agents search over program variants without an explicit model of the algorithm being designed. In algorithm discovery, by contrast, a syntactically flawless program frequently exhibits severe optimization bottlenecks, poor search trajectories, or weak generalization across instances. Bridging across complex algorithmic paradigms requires navigating intricate search semantics and multi-component interactions that cannot be solved simply by employing a strong general-purpose coding agent. AlgoEvo overcomes this limitation by treating code merely as the implementation medium, whereas empirical optimization feedback across diverse problem instances acts as the primary state signal guiding structural redesign. The agentic interface is thus elevated into an optimization-driven design search, rather than functioning as a standard coding assistant.

Skill-based and Experience-driven Agents. A growing body of research equips LLM agents with reusable skills, external memory, and behavioral evolution mechanisms (Li, 2026), ranging from single-agent workflow memory systems such as AWM (Wang et al., 2024c), Gödel Agent (Yin et al., 2025), and Trace2Skill (Ni et al., 2026), to multi-objective skill bundle optimization frameworks such as SkillMOO (Gong et al., 2026). VLM agents have also demonstrated the ability to generate their own memories by distilling experience into embodied programs of thought (Sarch et al., 2024), while procedural memory frameworks enable experience-driven agent evolution through dynamic refinement (Cao et al., 2026). Memory evolution surveys further demonstrate the progression from static storage to experience-driven agent improvement (Luo et al., 2026). For skill creation and transfer, Trace2Skill distills trajectory-local lessons into transferable agent skills (Ni et al., 2026), while Ctx2Skill explores how language models learn context-specific skills through self-play (Si et al., 2026). However, recent empirical studies reveal critical limitations of generic skill representations: skill libraries can degrade agent performance by up to 21% as they expand due to skill shadowing effects (Song & Wei, 2026), over 89% of real-world skills exhibit reusability defects (Zhang et al., 2026), and skills can cause regressions through description osmosis, grounding displacement, and verification displacement (Tank & Nama, 2026). Moreover, indiscriminate memory addition can propagate errors and contaminate agent learning (Xiong et al., 2026), while agentic memory systems that enable dynamic organization show superior performance over static approaches (Xu et al., 2026). These findings confirm that such generic structures fail to capture the specialized semantics required for algorithm optimization. Algorithm discovery is uniquely challenging because design choices exhibit deep structural coupling (Chen et al., 2026), and historical performance trends are non-linear. AlgoEvo addresses this by transforming these mechanisms into

rigorous, domain-specific algorithm-design knowledge. A Design Skill encodes execution mechanisms, applicable conditions, design principles, and characteristic failure or convergence patterns. Similarly, the Experience Bank records validated design decisions alongside their empirical optimization effects, establishing a continuous knowledge cycle connecting experience acquisition, knowledge abstraction, and state-aware design search. This specialized structure ensures that knowledge governs not only text retrieval, but also the dynamic selection and coordination of subsequent algorithmic components.

Positioning of AlgoEvo. The three lines of work above leave three distinct gaps, each addressed by one component of AlgoEvo. First, LLM-based algorithm discovery searches candidate programs under fixed protocols, so the design process itself is never searched; AlgoEvo answers this with its *agentic search loop*, in which the agent chooses among inspection, diagnosis, modification, and evaluation according to the current optimization state rather than following a prescribed cycle. Second, general-purpose coding agents and evolutionary program searchers act on concrete programs without a representation of the algorithm under design, and cross-component coordination is either absent or encoded in static, algorithm-specific rules; AlgoEvo answers this with its *design skill hub*, whose paradigm-specific contracts let a single engine span single-heuristic, multi-objective, and multi-component tasks. Third, generic skill and memory structures, as the empirical studies above indicate, do not capture the semantics of algorithm design and can even degrade as they grow; AlgoEvo answers this with its *hierarchical experience bank*, which records design decisions as situation-conditioned cards, organizes them into a task-level tree, and consolidates validated patterns into transferable skills. Together, these components recast algorithm discovery from a fixed candidate-generation procedure into an optimization-driven, state-aware search over the design process.

C Method Details

This section provides a concrete, implementation-level description of the algorithm discovery process introduced in Section 3. Rather than executing a predefined script of optimization steps, AlgoEvo treats algorithm design as an interactive agentic search process. The framework coordinates agent reasoning, diagnostic feedback, methodological guidance, and hierarchical experience accumulation to explore complex algorithmic spaces under a strict evaluation budget. We detail the task specification and initialization, the discovery state, actions, and diagnosis, the design skill hub, the experience bank, and the concrete artifacts and prompts used in the implementation.

For reference, Algorithm 2 restates the complete discovery loop, expanding the condensed version in the main text (Algorithm 1) into its constituent steps: situation categorization, experience retrieval via UCB, experience-card construction, reward backpropagation, and cross-task skill evolution.

C.1 Task Specification and Initialization

A design task is fully specified at initialization by three artifacts: the initial algorithm code, the set of evolvable units, and the evaluation function F_d, together with a budget B equal to the maximum number of evaluations allowed for the task. These are fixed for the entire run and shared by all agent actions, so the search mechanism itself is paradigm-agnostic. A *multi-component* task simply declares an algorithm $\mathbf{\Pi} = (\pi_1, \ldots, \pi_K)$ whose strategies decompose into coupled sub-components; the agent edits those sub-components directly, and the diagnostic engine measures their coupling. A *multi-objective* task differs only in the evaluation function, which returns a vector of objectives rather than a scalar; the reward of Eq. 4 is then computed on the aggregated objective value, which for these tasks is the negative hypervolume of the current Pareto front under the task's reference point. In both cases, supporting the paradigm requires only a different task specification; the loop itself is unchanged.

Initialization. At task initialization, the system generates a system prompt that states the method and the task (Section C.5). Guided by this prompt, the agent activates the design skill whose declared conditions best match the task, and matches an experience tree semantically indexed by (skill, problem, evaluator). If no matching tree exists, a new one is created. Algorithm 3 formalizes this process.

Algorithm 2 Agentic Algorithm Discovery (full)

Require: Design task $\mathcal{T}$, design skill hub $\mathcal{S}_{\text{hub}}$, experience bank $\mathcal{E}$, budget B, threshold μ
Ensure: Optimized algorithm $\mathbf{\Pi}^*$

$\sigma \leftarrow$ **Activate**$(\mathcal{T}, \mathcal{S}_{\text{hub}})$ ▷ Skill activation (Algo. 3)
$\epsilon \leftarrow$ **Match**$(\mathcal{T}, \sigma, \mathcal{E})$ ▷ Experience tree matching (Algorithm 3)
$t \leftarrow 0$ ▷ Step counter
Initialize algorithm $\mathbf{\Pi} \leftarrow \mathbf{\Pi}_0$, working context $\mathcal{C}_t$, and compute $F_d(\mathbf{x} \mid \mathbf{\Pi}_0)$
repeat
    // 1. Observe: detect current situation from global discovery state
    Update global discovery state $s_t \leftarrow (\mathbf{\Pi}, \mathcal{C}_t, \epsilon, F_d(\mathbf{x} \mid \mathbf{\Pi}))$
    $q_t \leftarrow$ **Categorize**(s_t) ▷ Extract conditions (Algorithm 4)
    // 2. Action: The agent selects an action, such as inspect, retrieve, diagnose, edit, or evaluate.
    $a_t \leftarrow$ **Agent**$(\sigma, q_t, \epsilon, \mathcal{C}_t)$
    // 3. Update: process outcome
    if $a_t =$ evaluate **then**
        Obtain the latest algorithm candidate $\mathbf{\Pi}'$ and compute reward r_t ▷ Eq. 4
        $\mathbf{\Pi} \leftarrow \mathbf{\Pi}'$ ▷ Update current incumbent algorithm
        Create card $c_t \leftarrow$ **Summarize**$(\mathcal{T}, \sigma, a_t, F_d(\mathbf{x} \mid \mathbf{\Pi}))$
        Add c_t to ϵ and backpropagate r_t ▷ Card $\leftrightarrow$ node mapping
        Update working context $\mathcal{C}_t$
    else if $a_t =$ diagnose or $a_t =$ retrieve **then**
        Select experience e via UCB: $e \leftarrow \arg\max_{e' \in \epsilon} \mathrm{UCB}(e' \mid q_t)$ ▷ Eq. 5
        Agent reads e and updates working context $\mathcal{C}_t$
    end if
    $t \leftarrow t + 1$
until budget B exhausted or $a_t =$ terminate
$\mathbf{\Pi}^* \leftarrow \arg\min_{\mathbf{\Pi}} F_d(\mathbf{x} \mid \mathbf{\Pi})$ ▷ Select best incumbent over evaluated candidates
Evolve$(\sigma, \mathcal{E}, \mathcal{S}_{\text{hub}}, \mu)$ ▷ Cross-task consolidation (Algorithm 5)
return $\mathbf{\Pi}^*$

Algorithm 3 Task Initialization

Require: Design task $\mathcal{T}$, skill hub $\mathcal{S}_{\text{hub}}$, experience bank $\mathcal{E}$
Ensure: Algorithm design skill σ, experience tree ϵ

Generate system prompt P based on problem description, objectives, and design units
Inject P into agent
Agent analyzes P and activates appropriate skill σ from $\mathcal{S}_{\text{hub}}$ ▷ Activate
Agent matches an experience tree semantically indexed by $(\mathcal{T}.\text{problem}, \mathcal{T}.\text{evaluator}, \sigma)$ in $\mathcal{E}$ ▷ Match
if a matching tree is found **then**
    Retrieve existing tree ϵ
else
    Create new experience tree ϵ and store it
end if
return σ, ϵ

C.2 Discovery State, Actions, and Diagnosis

At each step of the discovery trajectory, the agent maintains access to a comprehensive state space comprising the current algorithm implementation, working context, historical decisions, diagnostic feedback, and performance metrics. The working context logs all operations executed within the current trajectory, including inspected code segments, attempted modifications, evaluation outcomes, and analytical conclusions. This distinction prevents the agent from conflating a fresh optimization problem with the continuation of an unsuccessful branch.

Discovery actions are formulated at the level of high-level design decisions. Table 5 itemizes them and their resource consumption. A critical architectural distinction separates evaluation from an-

alytical actions, in that only evaluation yields a candidate algorithm and consumes the evaluation budget, whereas diagnostic and relational analyses extract structural evidence without producing candidates and without charging that budget. Analytical actions are nevertheless bounded by the overall trajectory limit, so the agent must balance empirical testing against analytical reasoning.

Table 5: Discovery actions available to the agent, their state impacts, and resource consumption.

Action	Effect on the discovery state	Candidate	Charge
Inspect	Reads source files or task execution artifacts	No	—
Retrieve	Queries the experience bank and reasoning chains	No	—
Diagnose	Attributes performance to units and identifies failure modes	No	—
Analyse interaction	Measures coupling strength between two units	No	—
Edit	Modifies the source code of a target design unit	No	—
Evaluate	Executes and scores the current on-disk algorithm	Yes	Sample, eval.
Commit	Persists validated high-performing versions	Yes	Eval.
Terminate	Concludes the active task trajectory	No	—

The resulting trajectory forms a heterogeneous sequence of analytical, exploratory, and evaluative steps. In multi-component tasks, this enables the agent to isolate interacting units, measure their coupling, and perform coordinated edits.

Because the trajectory mixes evaluative and analytical steps, the reward is defined only for steps that update the incumbent algorithm, relative to the candidate from which the update was derived (its parent card when the agent branches from an earlier node), and is taken as zero otherwise. The incumbent tracks the latest evaluated candidate, while the algorithm returned at the end is the best incumbent over the whole trajectory, so that a temporarily inferior candidate does not overwrite the final answer.

Diagnosis and situation detection. Relying solely on scalar performance scores provides insufficient guidance for complex search spaces. While a low evaluation score indicates that an algorithm is suboptimal, it fails to specify which component caused the failure or whether multiple units interact adversely. To overcome this, AlgoEvo incorporates situation-aware retrieval to match historical experiences with the current search context.

At each step, the diagnostic engine supplies two runtime signals, a per-unit attribution A_u quantifying the contribution of design unit u to the current performance gap and a scalar interaction measurement int capturing the coupling between units. Together with the submission history and the Pareto archive, these signals determine the active situation set $q_t \subseteq \mathcal{Q}$, where $\mathcal{Q}$ collects the situation categories (stagnation, bottleneck, coupling, sparse front). Algorithm 4 formalizes this process, in which Part 1 categorizes q_t from the discovery state s_t and Part 2 selects the experience card e that best matches it. Thresholds τ_s, τ_c, and τ_b are listed in Table 6.

The engine computes the two signals by counterfactual evaluation on the training set. The per-unit attribution replaces the body of unit u with a neutral no-op implementation and records the resulting change in the evaluation score,

$$A_u = F_d(\mathbf{x} \mid \mathbf{\Pi}_{u \leftarrow \mathrm{noop}}) - F_d(\mathbf{x} \mid \mathbf{\Pi}), \tag{6}$$

so that the unit whose neutralization degrades performance most is the dominant contributor to the current gap. The interaction measurement neutralizes a pair of units jointly and applies inclusion-exclusion,

$$\mathrm{int}(u_a, u_b) = A_{u_a \wedge u_b} - A_{u_a} - A_{u_b}, \tag{7}$$

where $A_{u_a \wedge u_b}$ is the attribution obtained by neutralizing both units at once; a large magnitude indicates strong coupling, so the two units are better redesigned jointly. Both probes are analysis operations rather than candidate submissions, producing no candidate and consuming no evaluation budget, although the trajectory remains bounded by the step limit. The remaining conditions

are read directly from the search history, namely stagnation after τ_s consecutive non-improving submissions and a sparse front when the normalized archive has at least two points whose largest nearest-neighbour gap exceeds three times the smallest.

Algorithm 4 Situation categorization and experience card selection

Require: Discovery state s_t, experience tree ϵ
Ensure: Active situations q_t, selected experience card e

// Part 1: Categorize the current situation from s_t
$q_t \leftarrow \emptyset$
if last τ_s submissions show no improvement **then**
    $q_t \leftarrow q_t \cup \{\text{stagnation}\}$
end if
if $|\text{int}| > \tau_c$ **then**
    $q_t \leftarrow q_t \cup \{\text{coupling}\}$
end if
if $\max_u A_u / \sum_u A_u > \tau_b$ **then**
    $q_t \leftarrow q_t \cup \{\text{bottleneck}\}$
end if
if front has wide gaps **then**
    $q_t \leftarrow q_t \cup \{\text{sparse front}\}$
end if
// Part 2: Select the experience card
$e \leftarrow \arg\max_{e' \in \epsilon} \text{UCB}(e' \mid q_t)$ ▷ Select the highest-scoring card by Eq. 5
return q_t, e

Algorithm 4 instantiates the **Categorize** step of Algorithm 1: it detects the active situations q_t from the discovery state s_t (Part 1) and then selects the experience card e that best matches them (Part 2). The highest-scoring card by Eq. 5 guides the next design decision.

Table 6: Discovery situations and their activation conditions.

Situation	Activated when
Stagnation	The last $\tau_s = 4$ submissions yield no improvement
Sparse front	The largest nearest-neighbour gap in the normalized archive exceeds $3\times$ the smallest
Bottleneck	A single unit u accounts for $> \tau_b = 0.5$ of the attribution mass
Coupling	The interaction measurement satisfies $\|\text{int}\| > \tau_c = 0.05$

This diagnostic mechanism decouples behavioral understanding from raw objective scoring, ensuring that modifications target verified structural weaknesses.

C.3 Design Skill Hub

Different algorithmic paradigms impose unique structural constraints and interface requirements. A constructive heuristic, a multi-objective operator set, and a coupled destroy-and-repair pipeline require distinct design strategies. The design skill hub provides paradigm-specific methodological knowledge without altering the core agentic search loop.

Skill representation. Each design skill is stored as a Markdown file with YAML frontmatter (Listing 5). The frontmatter declares the skill's metadata, applicable conditions, strategy roles, and operator taxonomy. The Markdown body provides detailed design guidance, failure modes, and convergence patterns.

Skill matching. At task initialization, the agent selects the appropriate skill based on the task description (Algorithm 3). If the task config explicitly specifies a skill name, that skill is used directly. Otherwise, the agent analyzes the task requirements and selects the most suitable skill from the registry.

```yaml
---
name: eoh
version: 0.2.0
paradigm: population-based
applicable:
  problem_type: [single-objective, multi-objective]
  instance_set: [combinatorial, scheduling]
  solution_type: [permutation, assignment]
strategy_roles:
  pi_1:
    name: population_generator
    kind: generator
    file: generate.py
    signature: "def generate(problem, config) -> list"
  pi_2:
    name: solution_evaluator
    kind: evaluator
    file: evaluate.py
    signature: "def evaluate(problem, solution) -> float"
operators:
  perturbation: [random_reset, swap_mutation]
  hybridization: [uniform_crossover]
  repair: [feasibility_repair]
  improvement: [local_search]
---

# EoH Design Skill

## Design Principles
1. Maintain diversity through adaptive mutation rates
2. Balance exploration (perturbation) and exploitation
3. Use evaluation feedback to guide operator selection

## Operator Taxonomy
- E1 (Perturbation): random changes to escape local optima
- E2 (Hybridization): combine features of two solutions
- M1 (Repair): restore feasibility after perturbation
- M2 (Improvement): local search to refine solutions

## Failure Modes
- Over-reliance on perturbation leads to random walk
- Insufficient repair causes infeasible solutions
- Weak improvement operators slow convergence
```

Figure 5: Example skill representation for the EoH paradigm. The YAML frontmatter declares metadata, applicable conditions, and operator taxonomy; the Markdown body provides design guidance.

Skill evolution. Skills are not updated within a single task; evolution is triggered only across tasks. Algorithm 5 collects the experience trees owned by the current skill σ and, once at least μ trees have accumulated ($\mu = 2$ by default), has the agent summarize their most salient patterns and refine the skill with the summary. Because the refinement is grounded in accumulated cross-task evidence rather than a single trajectory, it sharpens the skill's methodological knowledge without overwriting its paradigm-level guidance.

Extending the framework to new optimization paradigms requires defining corresponding design skills rather than restructuring the agent architecture. Furthermore, the hub incorporates distilled methodological insights from established optimization literature, equipping the agent with established design patterns and known failure modes.

Algorithm 5 Cross-task skill evolution

Require: Skill σ, experience bank $\mathcal{E}$, skill hub $\mathcal{S}_{\text{hub}}$, threshold μ
Ensure: Refined skill σ
1: Collect the experience trees in $\mathcal{E}$ owned by σ
2: **if** the number of collected trees is at least μ **then**
3: Agent summarizes the most salient patterns across the collected trees
4: Refine σ with the summary
5: **end if**
6: **return** σ

C.4 Hierarchical Experience Bank

While working contexts manage single-trajectory reasoning, the hierarchical experience bank preserves empirical knowledge across tasks. Evaluated candidates are distilled into structured experience cards recording the design context, applied modifications, empirical performance, and underlying rationale.

Experience card structure. Each experience card e is a tuple $e = (\texttt{problem}, \texttt{situation}, \texttt{mode}, \texttt{unit}, \texttt{skill}, \texttt{content}, \texttt{evidence}, \texttt{code}, \texttt{metrics})$ that captures the full context of a design decision. The fields are defined in Table 7. The `mode` field distinguishes validated strategies (`validated`), known failures (`avoid`), and post-task reflections (`reflection`). The `code` field optionally stores reference implementation that can be injected into the agent's context.

Table 7: Experience card fields and their descriptions.

Field	Type	Description
`problem`	string	Problem name (e.g., "TSP", "CVRP")
`situation`	string	Detection context: initial, stagnation, bottleneck, coupling, sparse front, default
`mode`	string	Validated, avoid, or reflection
`unit`	string	Design unit this experience pertains to
`skill`	string	Owning paradigm skill name
`content`	string	Human-readable description of the design pattern
`evidence`	string	Empirical evidence (e.g., "$\times 3$ improved, swing ~ 0.15")
`code`	string	Optional reference implementation
`metrics`	dict	Optional performance metrics

Situation-aware retrieval. Experience cards are selected by the situation-conditioned upper confidence bound of Eq. 5 (Algorithm 4). Each card maintains a mean reward and a visit count for the current situation, so that experiences validated under similar situations are preferred while cards with few observations retain an exploration incentive.

Experience tree. Experiences generated within a task are organized into a task-level experience tree, optimized via four Monte Carlo Tree Search stages:

1. *Selection*: Balances exploitation of high-performing design branches with exploration of unvisited directions via the situation-conditioned upper confidence bound (Eq. 5).
2. *Expansion*: Appends newly evaluated design decisions to the tree structure.
3. *Evaluation*: Grounds experience nodes in empirical performance rather than language model heuristics.
4. *Backpropagation*: Propagates performance outcomes up ancestral nodes, crediting foundational design choices that enabled downstream improvements.

Querying the experience tree using active discovery situations ensures that retrieved historical insights match the current operational context.

Experience consolidation and promotion. To enable cumulative learning, the framework converts evaluation outcomes into reusable methodological knowledge. This section details the experience update and the reflection mechanism.

Experience update. The agent updates the experience bank only when it performs an evaluation, each of which drives it to summarize the outcome into an experience card, which is classified as validated or avoid and attached to the experience tree. Algorithm 6 formalizes this process.

Algorithm 6 Experience card construction

Require: Evaluation outcome of candidate $\mathbf{\Pi}$, active situation q_t, experience tree ϵ
Ensure: Experience card c_t
1: Agent summarizes the evaluation into a card c_t, recording the situation q_t, the modification, the reward r_t, and the rationale
2: Set the mode of c_t to validated or avoid
3: Attach c_t to ϵ
4: **return** c_t

Post-task reflection. After the task completes, the agent generates a structured reflection summarizing the design process. Listing 6 shows the reflection template, which captures effective process paths, situation-strategy mappings, and practices to avoid.

```
### Effective Process Path
- step-by-step process that worked (numbered list, max 6 items)

### Situation-Strategy Mapping
- for each situation encountered, which information/strategy helped
  (bullet list: "situation -> strategy")

### Avoid List
- practices that failed or wasted budget (bullet list, max 5 items)
```

Figure 6: Reflection template used after each task. The reflection is stored as an experience card with mode=`reflection`.

Cross-task promotion. A skill is refined only once it owns at least μ experience trees accumulated across tasks. This closes the self-evolving loop, allowing AlgoEvo to accumulate transferable algorithmic intelligence over successive problem instances.

C.5 Concrete Artifacts and Prompts

We close the method details with concrete instances of the three knowledge artifacts the loop produces and consumes: a *paradigm skill*, a *method skill*, and an *experience card*. A paradigm skill is keyed to the design setting (single-heuristic construction, multi-objective design, or multi-component co-design) and fixes how a task of that setting is designed (Section 3.2); a method skill instead carries the methodology of an existing algorithm and is supplied without importing its native search loop (Section 4.4); and an experience card is the unit of empirical evidence (Section 3.3).

Paradigm skill. A paradigm skill declares the roles of the designable strategies, their interfaces, the principles that guide valid modifications, and the evaluation conventions, together with the structural features that decide its activation. Listing 7 shows the skill for the single-heuristic setting on TSP: its `paradigm` field names the setting, its `features` state that the task is single-objective with one function-typed unit in the `tsp` domain, and its body fixes the unit interface and the design knowledge.

```
---
name: tsp-constructive
version: 1.0.0
paradigm:  single-heuristic
triggers: [tsp, travelling salesman, tour]
features:
  n_objectives: 1
  n_units: 1
  unit_kinds: [function]
  domain: [tsp]
---

# TSP Constructive Skill

## Applicability
Single unit, constructive; the evolvable unit is
  def select_next_node(current_node, destination_node,
                       unvisited_nodes, distance_matrix) -> int

## Design Knowledge
- The nearest-neighbour rule is the reference point.
- Combining distance with a lookahead term improves on it.
- Caching and lookahead help at larger instance sizes.

## Failure Modes
- A non-integer or out-of-range node crashes the evaluator.
- Pure distance scoring degenerates to nearest-neighbour.
- Over-complex scoring overfits the training instances.

## Acceptance
- Valid signature; always returns an unvisited node.
```

Figure 7: A paradigm skill for the single-heuristic setting. The frontmatter declares the design setting, applicable conditions, and activation triggers; the body fixes the unit interface, the design knowledge, the failure modes, and the acceptance criteria.

Method skill. A method skill carries the methodological knowledge of an established algorithm and is supplied to the agent without importing its native search loop. Listing 8 shows the EoH method skill, whose body states the operator taxonomy, the recommended parameters, and the execution discipline that the agent follows.

Experience card. Listing 9 shows a validated card created when an evaluation improved the incumbent. Each line corresponds to a field of Table 7: `problem` and `situation` fix the context; `unit` and `skill` identify what was modified and under which skill; `content` records the design pattern; `evidence` and `metrics` summarize its empirical outcome; and `code` stores a reference implementation that can be injected into the agent's context.

```
---
name: eoh
version: 1.2.0
paradigm:  evolutionary
triggers: [eoh, evolution of heuristics]
features:
  n_objectives: 1
  n_units: 1
  unit_kinds: [function]
  domain: [tsp]
---

# Evolution of Heuristics (EoH) Skill

## Operator Taxonomy
- E1 (explore): mutate an operator into a new form
- E2 (exploit): recombine the strongest operators
- M1 (merge): combine two operators into one
- M2 (modify): tune the parameters of an operator

## Parameters
pop_size = 4, selection_num = 2, max_generations = 10

## Execution Guide
- Emit one operator per turn and record which one was used
- Maintain an implicit population of the best combinations
- Rotate E1 -> E2 -> M1 -> M2 to balance exploration
- On stagnation, prefer E1 over M2

## Experience Upgraded
- Validated on cvrp_dr (swing ~0.145)
- Validated on fjsp_multi_operator (swing ~18.0)
```

Figure 8: A method skill. The frontmatter declares the methodology, applicable conditions, and activation triggers; the body states the operator taxonomy, parameters, execution guide, and the accumulated cross-task evidence from skill evolution.

```
problem:   TSP
situation: stagnation
mode:      validated
unit:      select_next_node
skill:     eoh
content:   Replace the distance-only score with a regret-based
           score that also rewards isolated nodes (large gap to the
           nearest unvisited node), escaping the greedy optimum.
evidence:  9 submissions improved, swing ~0.711
code:      def select_next_node(current_node, destination_node,
                                unvisited_nodes, distance_matrix):
               regret = [min(distance_matrix[j][k]
                             for k in unvisited_nodes if k != j)
                         for j in unvisited_nodes]
               score = [distance_matrix[current_node][j] - 0.3 *
   regret[i]
                        for i, j in enumerate(unvisited_nodes)]
               return unvisited_nodes[argmin(score)]
metrics:   {best_score: 6.362, eu: 0.29, situations: [stagnation]}
```

Figure 9: A validated experience card from a TSP run. The card records the situation, the modified unit, the design pattern, the empirical evidence, a reference implementation, and self-measured metrics.

Prompt design. The full discovery pipeline is injected into the agent through a single system prompt that states the method and the current task, while each subtask the agent performs is additionally guided by a dedicated reference instruction. We present both below.

System prompt. The system prompt establishes the agent's role, describes the closed-loop method, and provides the concrete task; it is followed by the active skill, the retrieved experience cards, and the knowledge state. Listing C.5 shows a complete example.

System Prompt Example

```
You are AlgoEvo, an autonomous agent that designs
high-quality algorithms for optimization problems.

## METHOD
You operate in a closed-loop, experience-driven process:
1. Activate. A design skill is activated for this task;
   it fixes the paradigm, the unit roles, and the guidance.
2. Match. An experience tree indexed by (skill, problem,
   evaluator) is matched; if none exists, create one.
3. Search. You iteratively act on the algorithm: read
   code, edit a unit, diagnose, or evaluate a candidate,
   choosing each action yourself from the current state.
4. Summarize. Every evaluation produces an experience
   card recording the situation, the modification, the
   reward, and the rationale.
5. Evolve. Across tasks, the accumulated experience trees
   are summarized and the skill is refined.

## TASK
PROBLEM: TSP
OBJECTIVES: tour length (minimize)
EVOLVABLE UNITS:
- select_next_node (heuristic) file=construct.py
  signature=def select_next_node(...) -> int
BUDGET: max_samples=100, max_evals=500

## DESIGN PRINCIPLES
1. The code you write MUST be callable by the eval entry.
2. Only modify the internal implementation of the units.
3. Keep the public signatures unchanged.
4. Every submitted candidate must be a runnable file.

## ACTIVE SKILL
===== SKILL: eoh (paradigm: population-based) =====
<design principles, operator taxonomy, failure modes>

## EXPERIENCE
PREVIOUS EXPERIENCE ON THIS PROBLEM (TSP):
STRATEGIES THAT WORKED:
  [EXP-0] select_next_node: weighted distance scoring
    REFERENCE CODE:
    def select_next_node(current_node, destination_node,
                         unvisited_nodes, distance_matrix):
        scores = [distance_matrix[current_node][j] * (1 + angle(j))
                  for j in unvisited_nodes]
        return unvisited_nodes[argmin(scores)]
STRATEGIES TO AVOID:
  [EXP-1] select_next_node: random selection (failed x5)

## KNOWLEDGE STATE
- Step 1: tried greedy heuristic, got 7.231
- Step 2: added 2-opt, improved to 6.892
```

System prompt example: the method and the task are stated explicitly, followed by the active skill, the retrieved experience cards, and the knowledge state.

Subtask instructions. For each subtask the agent performs autonomously, a reference instruction specifies its objective, its inputs, and the expected output. Listing C.5 summarizes the four instructions: skill activation, experience tree matching, experience card summarization, and skill refinement.

Subtask Reference Instructions

```
[SKILL ACTIVATION]
Given the problem description and objectives, select the
single most appropriate design skill from the registry
and justify the choice.

[EXPERIENCE TREE MATCHING]
Search the experience bank for a tree indexed by
(skill, problem, evaluator) that matches the current
task. If one exists, load it; otherwise create a new
empty tree. Report the matched or created tree.

[EXPERIENCE CARD SUMMARIZATION]
Summarize the latest evaluation into an experience card
with the fields: problem, situation, mode (validated |
avoid | reflection), unit, skill, content (the design
pattern), evidence (the empirical outcome), and
optional reference code and metrics.

[SKILL REFINEMENT]
Review the experience trees owned by the skill. Summarize
the most salient recurring patterns and refine the skill
document with them, without overwriting its
paradigm-level guidance.
```

D Benchmark and Evaluation Details

D.1 Benchmark Tasks and Instance Sets

To comprehensively evaluate AlgoEvo across distinct algorithmic complexities, we utilize six benchmark tasks spanning single-heuristic construction, multi-objective optimization, and multi-component co-design. Each task incorporates a training set utilized during the discovery process and a disjoint test set to evaluate generalization. All comparative methods operate under identical instance splits. Table 8 summarizes the instance distributions and objectives.

Table 8: Summary of benchmark tasks, objectives, and instance distributions.

Task	Objective	Training set	Test set
TSP	tour length ↓	16 instances (50 cities, uniform)	24 instances (sizes 50/100/200, uniform/clustered), plus 6 TSPLib
CVRP	route cost ↓	16 instances (50 customers, cap 40, demands $U[1, 10]$)	48 instances (16 each of sizes 50, 100, and 200)
Bi-TSP	two tour lengths ↓	1 instance (50 cities, two coordinate matrices)	2 instances (sizes 20 and 100)
Bi-FJSP	makespan ↓, utilization ↑	10 Brandimarte instances (mk01–mk10)	5 Brandimarte instances (mk11–mk15)
CVRP-DR	route cost ↓	10 instances (50 customers, cap 50, demands $[0, 9]$)	64 instances (size 50), plus 5 reserve sets up to size 500
FJSP 4-Ops	makespan ↓	10 Brandimarte instances (mk01–mk10)	5 Brandimarte instances (mk11–mk15)

D.2 METRICS

For single-objective tasks, performance is the mean objective over training instances, $\bar{F}_d(\mathbf{\Pi}) = \frac{1}{|\mathcal{D}_{\text{train}}|}\sum_{\mathbf{x}\in\mathcal{D}_{\text{train}}} F_d(\mathbf{x} \mid \mathbf{\Pi})$, where lower is better. For multi-objective tasks, non-dominated objective vectors form an archive, and objectives are min-max normalized per problem, $\tilde{z}_i = (z_i - z_i^{\min})/(z_i^{\max} - z_i^{\min})$, over all solutions evaluated across methods and seeds. We report HV (Zitzler & Thiele, 1998) and IGD (Coello & Cortés, 2005), with reference point $\mathbf{r} = (1.1, \ldots, 1.1)$; HV is the Lebesgue measure Λ of the region dominated by the archive $\mathcal{P}$,

$$\mathrm{HV}(\mathcal{P}) = \frac{1}{\prod_{i=1}^{M} r_i}\Lambda\left(\bigcup_{\mathbf{z}\in\mathcal{P}}[\mathbf{z}, \mathbf{r}]\right), \tag{8}$$

and IGD is the mean distance from the reference front $\mathcal{F}$ of the aggregated solutions to $\mathcal{P}$,

$$\mathrm{IGD}(\mathcal{P}) = \frac{1}{|\mathcal{F}|}\sum_{\mathbf{u}\in\mathcal{F}}\min_{\mathbf{z}\in\mathcal{P}}\|\mathbf{u} - \mathbf{z}\|_2, \tag{9}$$

with higher HV and lower IGD indicating better convergence and diversity. Both the search reward and the reported metrics use a hypervolume reference point; the reported values are additionally normalized to $[0, 1]$ over the union of evaluated solutions so that they are comparable across methods and problems. As normalization bases are built per problem, these metrics are compared within a problem only.

D.3 BASELINE DETAILS

All baselines are re-implemented from their original publications and evaluated on the LLM4AD platform (Liu et al., 2024b) under the shared protocol: identical instance pools, the same evaluation entry points, a maximum of 500 evaluations per seed, and three random seeds.

Rule-based heuristics. We include simple constructive rules that involve no search, as classical reference points. For TSP, the nearest-neighbour (NN) heuristic builds a tour by repeatedly moving from the current city i to the closest unvisited city $\arg\min_j d(i, j)$. For CVRP, NN is made capacity-aware: from the current customer it visits the nearest unvisited customer whose demand fits the remaining vehicle capacity, and opens a new route only when no such customer exists. For the multi-component CVRP-DR task, the *Standard* baseline runs the shipped destroy-and-repair pipeline, whose three units are a savings-inspired edge score (inverse distance weighted by capacity efficiency and a depot bonus), a worst-customer removal criterion (demand-to-capacity ratio plus mean distance to other customers), and a nearest-insertion repair rule. For FJSP 4-Ops, the *Expert* baseline uses the hand-designed operator set shipped with the task. These rules are the starting points that every search-based method must improve upon.

Search-based baselines. For single-heuristic construction we compare against EoH (Liu et al., 2024a), ReEvo (Ye et al., 2024), MCTS-AHD (Zheng et al., 2025), and FunSearch (Romera-Paredes et al., 2024), each following its original search loop. For multi-objective design we use MEoH (Yao et al., 2025), NSGA-II (Deb et al., 2002), and MOEA/D (Zhang & Li, 2007), whose archives pass through the shared normalization and metric harness. For multi-component co-design we include MOTIF (Kiet et al., 2025) and E2OC (Qiu et al., 2026a), together with the two EoH variants described next.

Component-selection variants. Synergy and Rotating-EoH adapt EoH to multi-component tasks by deciding which component to edit at each step, and they differ precisely in that decision. *Synergy* keeps an EoH population over complete component combinations and, at each iteration, draws the component to edit at random with a probability weighted by stagnation, so that a component whose recent evaluations have failed to improve the baseline receives a higher weight and effort is redirected toward the weakest component. *Rotating-EoH* instead rotates through the components in a fixed order, spending a small candidate budget on each component in turn and keeping a change

only if it improves the incumbent, which amounts to greedy coordinate descent over the components. Both use the same EoH operators and the shared evaluation harness, and neither measures cross-component coupling, which is the signal AlgoEvo exploits through its diagnostic probes.

Parameter setting. Across all methods the controlled quantity is the sampling budget, i.e. the maximum number of candidate evaluations; each search-based baseline otherwise keeps the hyperparameters reported in its original publication, such as population size, archive capacity, and its selection or variation operators, so that every method runs in the configuration its authors found effective. The surrounding environment is fixed and follows the LLM4AD benchmark (Liu et al., 2024b): identical instance pools and splits, the same evaluation entry points, the same initial code templates for each evolvable unit, and the same feasibility and objective routines. The comparison therefore isolates the search procedure rather than environment-specific tuning.

D.4 Anti-cheating settings

Because the agent writes and runs its own code, each run is guarded so that a reported result reflects genuine algorithm design rather than a shortcut around the intended search.

Workflow control. Each run is driven by the system prompt and subtask reference instructions of Appendix C.5, which specify the task, the evolvable units, and how the design skill hub (Section 3.2) and the experience bank (Section 3.3) are used at each stage of Algorithm 1. The agent records every action and its outcome in the working context, so the trajectory is auditable and the task is carried to completion.

Exception handling. Every step is checked against its returned result and the log of tool invocations. A step that raises an execution error, returns a malformed result, or yields an invalid candidate is discarded, recorded in the working context, and re-executed, so that a transient failure is neither counted as progress nor allowed to erase the accumulated context.

Budget integrity. A run could otherwise score candidates through a private script and evade the budget B of Appendix C.1, which counts only calls to the shared evaluation entry point. Counters are therefore installed at both the instance layer and the evaluation entry point: a candidate is counted only when it is evaluated through the evaluation action of Table 5, and the counter is cross-checked against the number of individuals the algorithm generated. A mismatch, or any direct call to the objective that bypasses the evaluator, invalidates the run, which is discarded rather than reported.

Isolation. Runs execute in isolated sandboxes with separate working directories and process spaces, so that files and state cannot leak across runs or between methods.

E Design Space and Task Instantiation

To isolate the contribution of agentic algorithm discovery, AlgoEvo keeps the surrounding solver, instance handling, feasibility checks, and objective evaluation fixed, restricting optimization exclusively to pre-defined evolvable algorithmic units. Table 9 summarizes the design space configuration across the six benchmark tasks.

Table 9: Evolvable algorithmic units across the six benchmark tasks.

Task	Number of units	Designed components
TSP	1	Next-city selection rule
CVRP	1	Capacity-aware next-customer rule
Bi-TSP	1	Bi-objective next-city selection rule
Bi-FJSP	1	Dispatching rule
CVRP-DR	3	Construction, destruction, and repair rules
FJSP 4-Ops	4	Two crossover and two mutation operators

Single-Unit Design Tasks. For TSP, the designed component is a scoring rule that selects the next unvisited city. The surrounding constructive execution and termination logic remain fixed.

Evolvable Unit: Next-Node Scoring Function

```
def select_next_node(current_node, destination_node,
                     unvisited_nodes, distance_matrix) -> int
```

For CVRP, the designed component governs capacity-aware customer selection and depot-return timing. The fixed evaluator tracks remaining vehicle load constraints and filters out infeasible choices.

Evolvable Unit: Capacity-Aware Next-Node Selection

```
def select_next_node(current_node, depot, unvisited_nodes,
                     rest_capacity, demands, distance_matrix) -> int
```

For Bi-TSP, the same constructive rule is evaluated across two distinct distance objectives simultaneously, enabling the generation of Pareto archives through multi-objective feedback.

Evolvable Unit: Bi-Objective Next-City Selection

```
def select_next_city(current_node, unvisited_nodes,
                     dist_1, dist_2) -> int
```

For Bi-FJSP, the designed component acts as a priority dispatching rule determining task execution order at each scheduling step. The fixed evaluator enforces precedence constraints and machine eligibility.

Evolvable Unit: Operation Dispatching Rule

```
def select_operation(job_status, machine_status, feasible_ops) -> tuple
```

CVRP-DR: Sequential Component Coupling. CVRP-DR instantiates a multi-component design space comprising three sequential stages: construction, destruction, and repair. Algorithm 7 outlines the evaluation procedure.

Evolvable Units: Destroy-and-Repair Components (F1/F2/F3)

```
def edge_score(i, j, distances, demands, capacity) -> float
def customer_badness(customer_idx, permutation, distances,
                     demands, capacity) -> float
def insert_position(customer, permutation, distances,
                    demands, capacity) -> int
```

Algorithm 7 CVRP-DR evaluation procedure

Require: Instance, construction rule Π_c, destruction rule Π_d, repair rule Π_r, removal fraction ρ

Construct an initial solution using Π_c
Evaluate customer badness using Π_d
Remove the $\lceil \rho n \rceil$ customers with the highest badness scores
for each removed customer c **do**
    Select an insertion position using Π_r
    Insert c into the selected position
end for
Apply the fixed local-improvement procedure
return Solution cost

These three components exhibit sequential coupling: the output of construction bounds the scope of destruction, and the partial state left by destruction directly dictates the restoration target for

repair. Consequently, optimizing these strategies requires coordinated reasoning rather than isolated component tuning.

FJSP 4-Ops: Representation-Based Coupling. FJSP 4-Ops implements a multi-component genetic programming structure that jointly optimizes four distinct operators: priority-segment crossover, priority-segment mutation, machine-assignment crossover, and machine-assignment mutation. Chromosomes are structured as concatenated segments:

$$\mathbf{c} = [\mathbf{c}^{\text{priority}} \mid \mathbf{c}^{\text{machine}}]. \tag{10}$$

The priority segment determines operation sequences within jobs, while the machine segment dictates resource assignments. Algorithm 8 details the simplified evaluation loop.

Evolvable Units: Genetic Operators (op/ma $\times$ crossover/mutation)

```
def op_crossover(parent1, parent2, n_vars) -> (child1, child2)
def op_mutation(solution, n_vars) -> solution
def ma_crossover(parent1, parent2, n_vars) -> (child1, child2)
def ma_mutation(solution, n_vars) -> solution
```

Algorithm 8 FJSP four-operator genetic algorithm evaluation

Require: Population and four evolvable operators
Initialize chromosomes
for each generation **do**
  Evaluate population fitness and retain elite solutions
  while population is not refilled **do**
    Select parent chromosomes
    Apply priority crossover with probability 0.5
    Apply priority mutation with probability 0.3
    Apply machine crossover with probability 0.5
    Apply machine mutation with probability 0.2
    Legalize and evaluate offspring
  end while
end for
return Best makespan

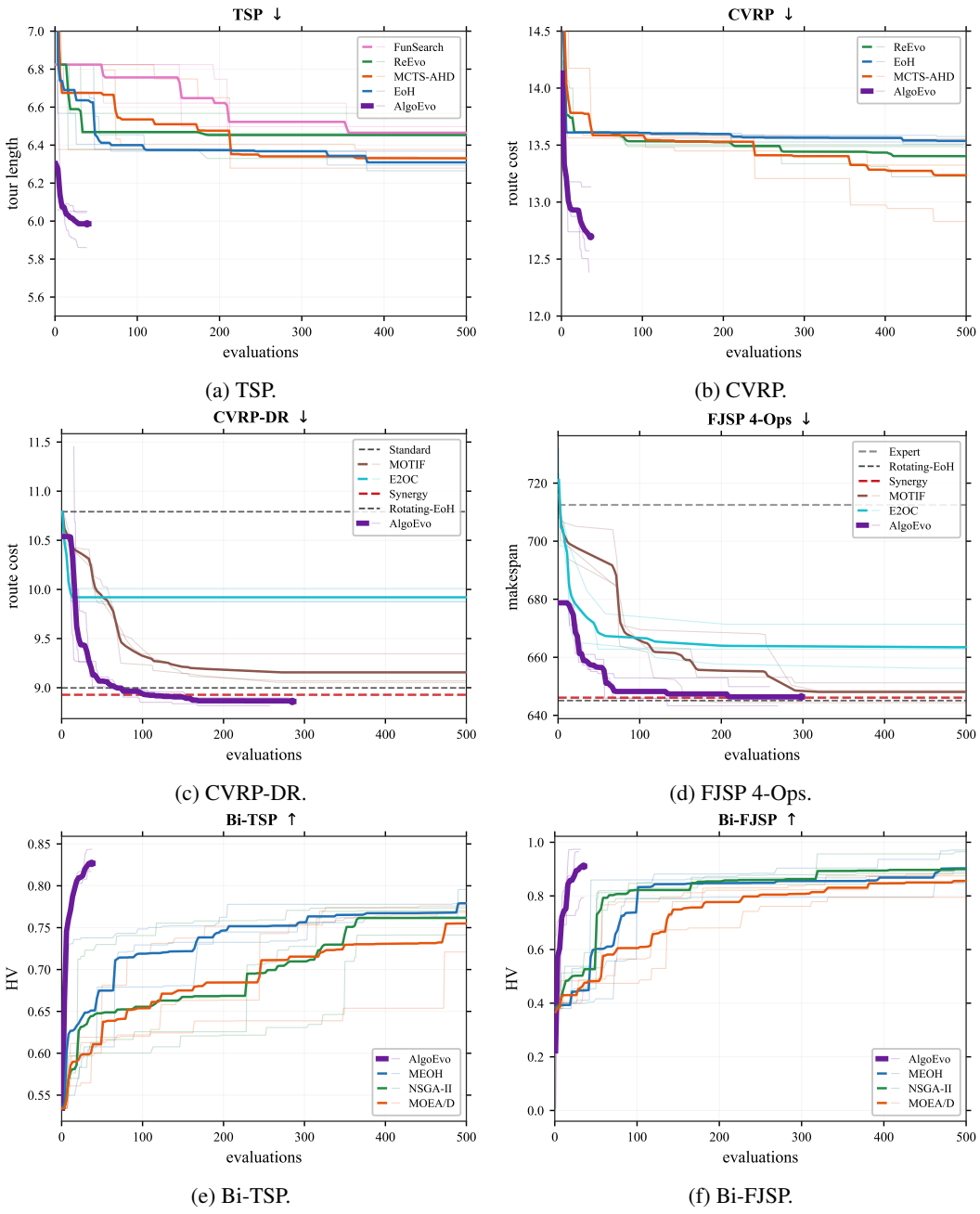
Unlike the pipeline coupling in CVRP-DR, FJSP 4-Ops couples components through a shared chromosomal representation and a joint decoding mechanism. Together, these multi-component tasks rigorously evaluate the framework's ability to coordinate interdependent strategies across diverse structural paradigms.

F Additional Experimental Results

F.1 Convergence Under the Evaluation Budget

Figure 10 illustrates the best-so-far performance as a function of the evaluation budget across all six tasks, complementing the final results in the main text by showing how design quality evolves during the search process.

In single-heuristic and multi-objective tasks, AlgoEvo consistently identifies competitive designs within a small fraction of the available budget. Conversely, multi-component tasks demand longer search trajectories, reflecting larger solution spaces and complex interactions among multiple evolvable components. These trajectories demonstrate the framework's capacity for adaptive resource allocation without prioritizing raw evaluation counts.



(a) TSP. (b) CVRP.

(c) CVRP-DR. (d) FJSP 4-Ops.

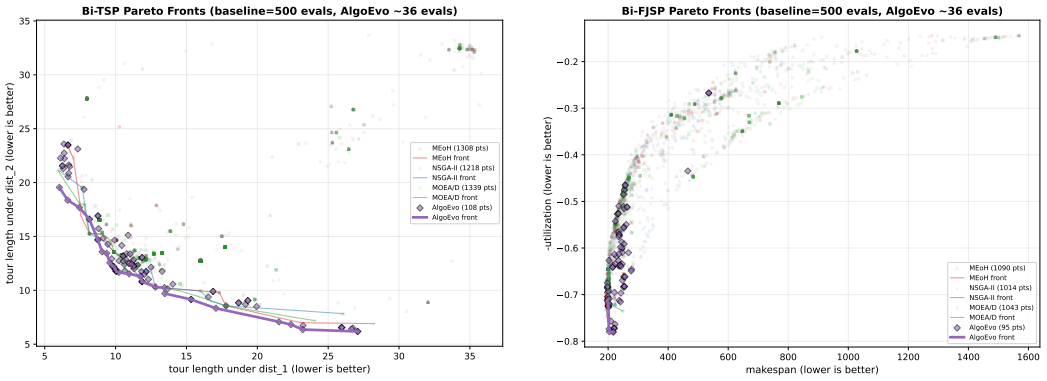
(e) Bi-TSP. (f) Bi-FJSP.

Figure 10: Best-so-far performance against the evaluation budget across the six design tasks.

F.2 Pareto Archives on Multi-Objective Tasks

Figure 11 presents the Pareto archives corresponding to the multi-objective experiments. To display the broader distribution of search outcomes alongside final non-dominated sets, evaluated candidates across all three random seeds are included.

The visualized archives align with the aggregate HV and IGD outcomes in the main text. AlgoEvo explores a wider region of the objective space and yields non-dominated solutions closer to optimal trade-off frontiers, supporting multi-objective algorithm design where discovering complementary design sets is paramount.



(a) Bi-TSP. (b) Bi-FJSP.

Figure 11: Pareto archives on multi-objective tasks, displaying all evaluated candidates across three random seeds.

F.3 Ablation of Knowledge Mechanisms

To assess the contributions of core architectural components, we conduct an ablation study on CVRP-DR under standard evaluation protocols, summarized in Table 10. Because this experiment runs as an independent batch, absolute values may vary slightly from the primary benchmark table.

Table 10: Ablation results on CVRP-DR over three random seeds.

Variant	Cost ↓	Deviation
AlgoEvo (complete framework)	$\mathbf{9.040 \pm 0.031}$	—
without the experience bank	9.100 ± 0.132	$+0.66\%$
without the design skill hub	9.092 ± 0.153	$+0.58\%$
without situation-triggered injection	9.059 ± 0.008	$+0.21\%$

Removing either the experience bank or the design skill hub incurs a larger performance drop than omitting situation-triggered injection. This confirms that persistent memory and reusable design abstractions serve as the primary drivers of discovery performance, whereas situation-triggered injection primarily refines knowledge selection efficiency.

G Further Analysis

G.1 Method Skills and Methodological Transfer

The design skill hub encodes not only paradigm-level design knowledge but also methodological insights distilled from existing algorithms. To examine whether such knowledge remains effective when decoupled from its original search framework, we construct six method skills from established methods and supply them to AlgoEvo without importing their native search procedures.

We evaluate EoH (Liu et al., 2024a), ReEvo (Ye et al., 2024), and FunSearch (Romera-Paredes et al., 2024) for single-heuristic design, and MEoH (Yao et al., 2025), NSGA-II (Deb et al., 2002), and MOEA/D (Zhang & Li, 2007) for multi-objective design, ensuring native and skill-based variants share identical seeds and instance sets.

To characterize how the agent behaves when executing a transferred methodology, Figure 12 traces a complete run on TSP (top) and Bi-TSP (bottom). Colored bands mark the action type of each step, and the curve traces the best-so-far quality against the number of evaluations. The agent does not repeatedly rewrite code at random: it first inspects the implementation to understand the problem structure, then produces a candidate, and thereafter conditions each edit on the feedback of the previous evaluation. The trajectory therefore resembles a paced design process of inspection,

implementation, evaluation, and revision, rather than trial-and-error, indicating that the agent follows the methodological workflow encoded in the skill instead of merely emitting code.

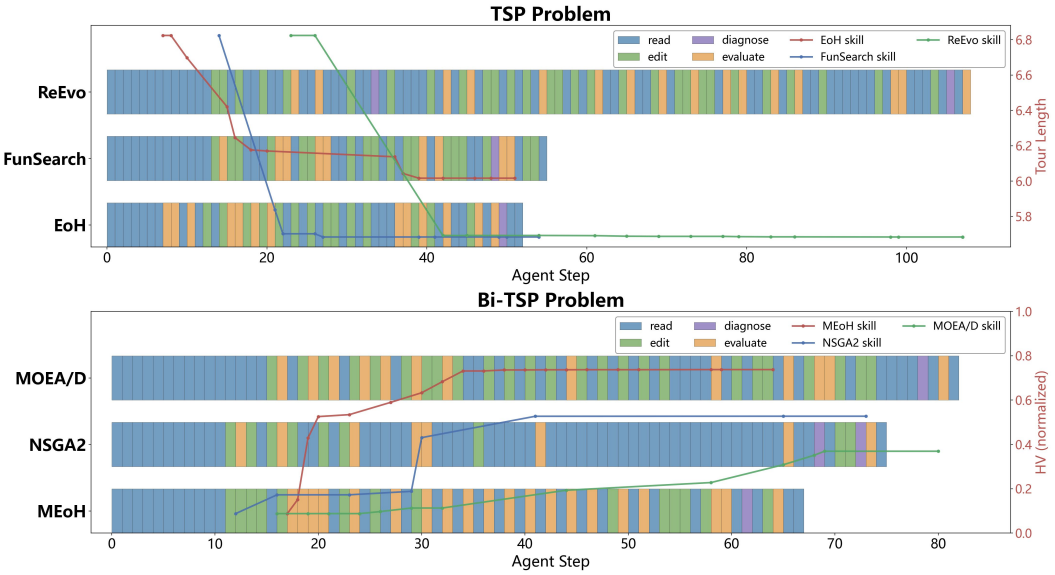


Figure 12: Agent behavior and convergence trajectory on TSP (top) and Bi-TSP (bottom). Colored bands indicate the action type of each step, and the curve traces the best-so-far quality against the number of evaluations. The agent follows a structured design process of inspecting, implementing, and revising in response to feedback, rather than random trial-and-error.

Table 11: Comparison between native methods and AlgoEvo equipped with corresponding method skills.

	Quality		Evaluations	
Method skill	**Native**	**With skill**	**Native**	AlgoEvo
TSP: tour length ↓				
EoH	7.1529 ± 0.0263	$\mathbf{7.0414 \pm 0.0923}$	500	**22**
ReEvo	7.3478 ± 0.3112	$\mathbf{7.1889 \pm 0.3678}$	500	**19**
FunSearch	7.6111 ± 0.1377	$\mathbf{7.1815 \pm 0.1057}$	500	**21**
Bi-TSP: HV ↑				
MEoH	0.760 ± 0.025	$\mathbf{0.829 \pm 0.004}$	500	**316**
NSGA-II	0.614 ± 0.082	$\mathbf{0.828 \pm 0.006}$	500	**391**
MOEA/D	0.314 ± 0.221	$\mathbf{0.829 \pm 0.006}$	500	**284**

Table 11 reports that the skill-based versions achieve solution quality comparable to or exceeding their native counterparts. On TSP, transferred methodologies reach superior average performance with substantially fewer evaluations, illustrated in Figure 13. On Bi-TSP, the primary benefit is enhanced archive quality and stability, while evaluation reductions are less pronounced. These findings indicate that methodological knowledge can be effectively represented and utilized independently of its original search framework. Figure 14 presents the per-seed comparison.

Note that the two experimental blocks use different problem scales (100-city instances for TSP versus 50-city instances for Bi-TSP), and absolute values should not be compared across tasks.

For Bi-TSP, the corresponding Pareto fronts in Figure 16 show that skill-based runs cover a broader favorable region of the objective space, consistent with their higher HV values and with the convergence curves in Figure 15.

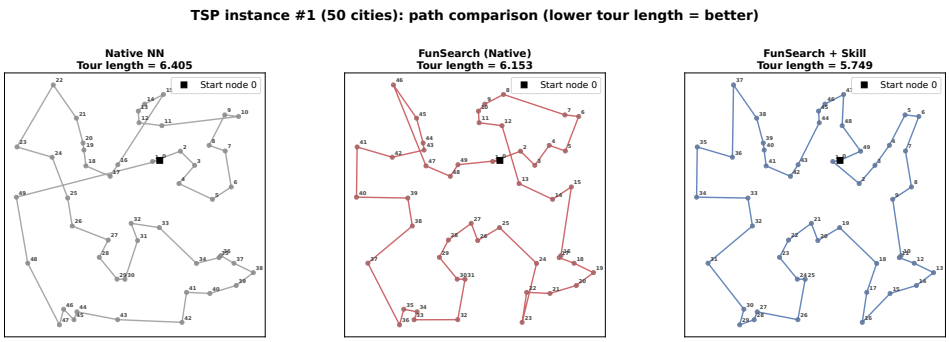


Figure 13: TSP path visualization on a 50-city instance. Left: native nearest-neighbor heuristic (tour length 6.405); center: FunSearch native method (6.153); right: AlgoEvo with FunSearch skill (5.749). The skill-based variant discovers a more efficient tour structure while using fewer evaluations.

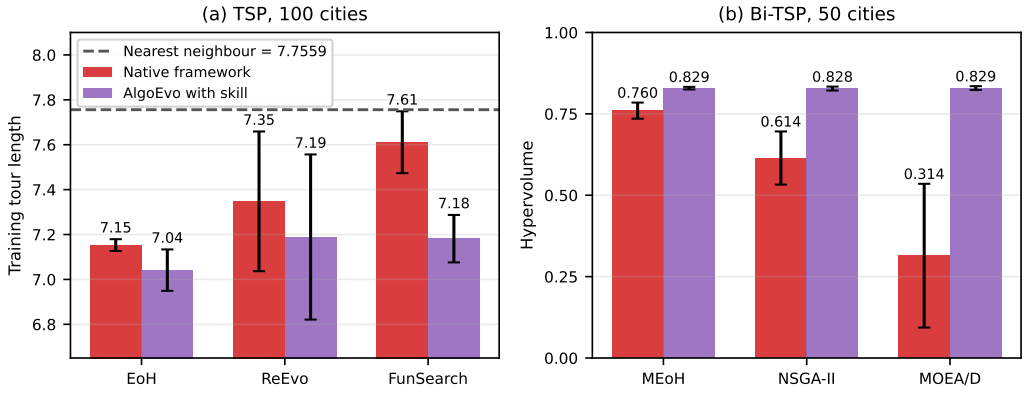


Figure 14: Per-seed quality comparison between native methods and AlgoEvo with corresponding method skills.

G.2 Representative Diagnosis-Guided Design Trajectory

To illustrate how diagnostic feedback adjusts the granularity of multi-component search, Table 12 details a representative CVRP-DR trajectory.

Table 12: Representative diagnosis-guided design trajectory on CVRP-DR.

Step	Operation	Observation	Cost
1	Diagnose	The destroyer exhibits the largest marginal effect	—
2	Analyse interaction	Destroyer and repairer display strong interaction	—
3	Edit	Replace the destroyer's distance-based criterion with absorption cost	9.310
4	Diagnose	Long edges remain insufficiently destroyed	—
5	Edit	Jointly modify destroyer and repairer	**9.088**

This trajectory highlights two key behaviors enabled by the diagnostic mechanism: first, component attribution identifies the destroyer as the current bottleneck, prompting targeted modification; second, interaction analysis reveals coupling between operators, causing the search to transition from isolated component editing to joint design. This qualitative progression supports the system-level co-design results in the main text.

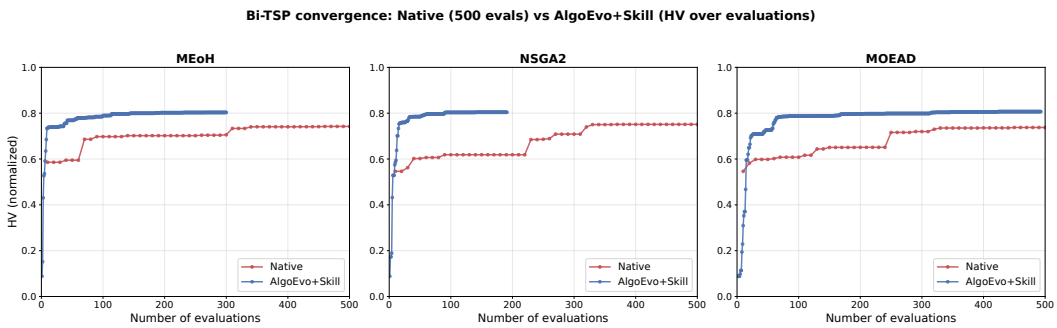


Figure 15: Bi-TSP convergence curves: native methods (red, 500 evaluations) versus AlgoEvo with method skills (blue). AlgoEvo+Skill achieves higher HV within significantly fewer evaluations across all three methods.

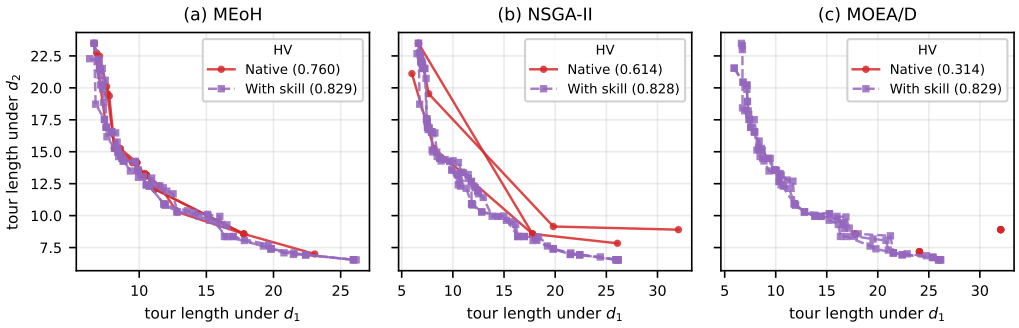


Figure 16: Bi-TSP Pareto fronts for native methods and AlgoEvo equipped with corresponding method skills.

G.3 Variation in Search Behavior

Different random seeds can discover competitive designs through distinct search trajectories. To study this behavior directly, we additionally select one CVRP-DR instance and log the operation breakdown of three runs in Table 13.

Table 13: Search operation counts and outcomes on CVRP-DR across different random seeds.

Seed	Read	Edit	Diagnose	Evaluate	Best cost
2025	14	122	17	81	8.927
2026	11	10	163	12	**8.281**
2027	12	89	102	6	9.138

While the runs allocate effort differently, one skewing toward code modification, another toward diagnostic analysis, and the third mixing both, the diagnosis-intensive run yields the best final result in this sample. Although consistent with the diagnosis-guided search principle, a larger sample size would be required to establish a definitive correlation between operation distributions and final performance.

G.4 Cross-Task Experience Transfer

Figure 17 presents a per-seed evaluation of the warm-start experiment. Initialized with a transferred design yielding a tour length of 5.017, every warm-started seed outperforms its cold-started counterpart. Mean performance improves from 4.961 to 4.808, accompanied by reduced cross-seed variance.

This demonstrates that previously discovered algorithms serve as effective starting points for subsequent search rather than being discarded upon task completion, complementing the main findings on hierarchical experience accumulation.

Note that this warm-start analysis uses the plain TSP constructive task (shipped nearest-neighbor baseline of 5.017) rather than the standard TSP protocol used in the main evaluation (nearest-neighbour baseline of 6.824); consequently, absolute values between these configurations should not be directly compared.

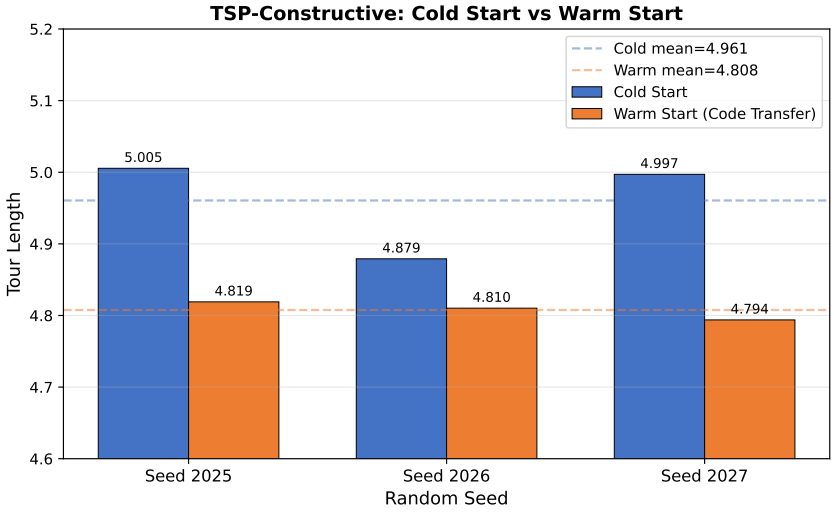


Figure 17: Cold-start and warm-start performance on the plain TSP constructive task. Each group corresponds to one seed; dashed lines indicate group means.

G.5 Limitations and Future Work

Despite strong empirical performance across complex combinatorial optimization tasks, AlgoEvo presents three limitations that motivate future work.

Hand-Designed Situation Awareness. Situations governing retrieval rely on discrete probes over a predefined category set. Consequently, these cues are hand-crafted rather than learned, meaning failure modes falling outside these categories may go undetected.

Skill Overfitting. Distilled skills derive from prior problem outcomes, exposing them to overfitting within aligned problem distributions and limiting zero-shot transfer to different domains.

Rigid Evaluation and Anti-Cheating Boundaries. To guarantee evaluation integrity, our current protocol enforces strict execution harnesses, immutable entry points, and hard sandboxing boundaries. While this prevents reward hacking and private-pathway shortcuts, it can inadvertently suppress unconventional yet valid structural explorations that lie outside the predefined code contracts.

These limitations outline three promising avenues for future research.

Learned Situation Awareness. Replacing hand-designed categorical probes with a continuous, learned representation of the search state could enable more robust, adaptive perception.

Optimized and Validated Skills. Evolving skill refinement from posterior summarization to explicit optimization against held-out validation objectives, complete with rigorous versioning and testing before adoption, will actively mitigate overfitting.

Flexible Yet Secure Validation Protocols. Moving beyond rigid static sandboxes toward verifiable open-ended exploration represents a key frontier. Future systems could dynamically adapt safety and anti-cheating mechanisms to safely accommodate novel architectural designs, ensuring absolute integrity without stifling the creative boundaries of automated discovery.