GUIDED EVOLUTION AND AUTOMATED REFINEMENT FOR LLM-BASED HEURISTIC DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based Automated Heuristic Design (AHD) has shown great promise for combinatorial optimization (CO), discovering competitive heuristics for well-studied problems such as the Traveling Salesman Problem (TSP). However, the quality of LLM-generated heuristics degrades in less-studied CO domains such as airline crew pairing, where LLMs lack domain-specific priors. We hypothesize that the general heuristic-design knowledge of LLMs can still be effective in such domains, and that the bottleneck lies in how the surrounding search loop selects candidates and feeds back information. We study this in a joint multi-component setting, where constructive and refinement heuristics are evolved together; the enlarged space makes scalar-fitness-based selection and feedback particularly uninformative. We propose **G**uided heuristic **E**volution and **A**utomated **R**efinement (GEAR), which uses complementary-gain population management to retain candidates that excel on different instances, and an executable Analyst that writes feature functions over evaluated solutions to produce structured, distributional feedback. Across four less-studied and four classic CO domains, with all baselines adapted to the same joint space, GEAR achieves the highest mean test fitness in all eight domains and consistent gains on scale-up generalization in the classic domains.

1 INTRODUCTION

Heuristics play a critical role in solving complex search and optimization problems, where exact algorithms are often computationally intractable (Lin & Kernighan, 1973; Lourenço et al., 2003; Hromkovič, 2013). Automated Heuristic Design (AHD) aims to reduce the substantial manual effort and domain expertise required to design such heuristics by automatically discovering effective strategies (Burke et al., 2013). However, traditional AHD methods search for the best combination of hand-crafted algorithmic primitives, which restrict their search space and limit their ability to generalize to novel problem settings (O'Neill et al., 2010; Stützle & López-Ibáñez, 2018; Drake et al., 2020).

Recent advancements in Large Language Models (LLMs) have significantly expanded the scope of AHD by enabling the direct generation of executable code from natural language descriptions or partial program specifications (Austin et al., 2021; Chen, 2021). Unlike traditional methods that operate within a fixed, predefined search space, LLM-based approaches explore a far more open-ended space of heuristic programs (Chen et al., 2023a; Meyerson et al., 2024). A growing body of work demonstrates that LLMs can discover competitive heuristics for well-studied CO problems by iteratively proposing, evaluating, and refining heuristic programs, such as the traveling salesman problem (TSP) and flow shop scheduling problem (FSSP) (Romera-Paredes et al., 2024; Liu et al., 2024).

Despite these advances, performance tends to degrade in less-studied domains (Chen et al., 2025; Sun et al., 2026), where well-performing heuristics are less documented and domain-specific priors are scarce in LLM pretraining data. Motivated by the success of LLMs in open-ended code generation and their capacity for zero-shot generalization, we hypothesize that the general knowledge of heuristic design in LLMs can be effective in less-studied domains as well, while the surrounding optimization loop can be the bottleneck for full leverage of the model capability.

Two aspects of existing LLM-based AHD frameworks become particularly limiting in less-studied domains. First, many methods optimize only one solver component, leaving performance constrained

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.

by other, fixed components. Jointly evolving interacting components is a natural way to broaden the search, but the larger space also makes effective exploration more difficult. Second, existing evolutionary loops rely heavily on scalar fitness to manage that search. When used to select candidates, average fitness can overlook heuristics that excel on different instances, discarding complementary strategies that could be combined. When used as feedback for revision, it says little about why a heuristic fails or which component should change. Both problems are more acute when domain-specific priors are scarce and the search must draw more guidance from evaluated candidates.

In this work, we address these challenges by proposing **G**uided heuristic **E**volution and **A**utomated **R**efinement (**GEAR**), a framework that enables richer exploration of heuristic programs together with more informative guidance for their refinement. Concretely, we adopt a joint evolution setting by evolving multiple parts of a solver, allowing more complete heuristic strategies to be constructed and refined over time. To navigate the enlarged search space, its complementary-gain population management uses per-instance performance to favor candidates that perform well where other candidates are weak. An executable Analyst writes feature functions over the solutions produced during evaluation and summarizes their behavior across instances, providing grounded feedback for subsequent revisions. Importantly, this design is modular and can be integrated into existing LLM-based AHD pipelines without requiring substantial changes to their overall structure.

Our contributions are summarized as follows:

- We study joint evolution of constructive and refinement heuristics as a natural extension of LLM-based AHD, and introduce GEAR to address the search and feedback challenges that joint evolution raises in less-studied domains.
- Inside GEAR, we develop complementary-gain population management based on per-instance performance, together with an executable Analyst that produces distributional feedback from measured solution features.
- We conduct comprehensive experiments across four less-studied and four classic CO domains, demonstrating that GEAR outperforms prior LLM-based AHD baselines and yields heuristics that generalize to scale-up evaluations.

2 Related Works

2.1 Automated Heuristic Design (Pre-LLM)

The automation of algorithm design, historically referred to as hyper-heuristic (Burke et al., 2013), aims to reduce the manual effort required to solve combinatorial optimization problems. Early methods (Hutter et al., 2009; Blot et al., 2016; López-Ibáñez et al., 2016; Akiba et al., 2019) primarily focus on selecting and combining predefined algorithmic components and tuning hyperparameters automatically. Beyond simple selection, evolutionary approaches such as genetic programming (Mei et al., 2022; Xu et al., 2023) have also been used to automatically synthesize heuristics and algorithmic components. These methods can yield interpretable, domain-specific heuristics and have shown practical success in real-world applications (Zhu et al., 2023; Zhang et al., 2023b). However, these approaches still require substantial human effort to design the search space. This requirement for domain expertise limits flexibility and can constrain further performance improvements.

2.2 LLM-Based Heuristic and Algorithm Discovery

The emergence of Large Language Models (LLMs) has introduced a new paradigm in algorithm discovery by enabling the direct synthesis of executable algorithms (Yang et al., 2023; Lange et al., 2024). The goal is to evolve a heuristic from a set of instance examples to generalize to solve other problems within the same domain and distribution. Early frameworks such as FunSearch (Romera-Paredes et al., 2024) and EoH (Liu et al., 2024) instantiate this idea through an evolutionary loop: candidate heuristics are executed on training instances, evaluated by scalar fitness, and then used as parents for the LLM to propose new heuristic programs. Subsequent works improve different parts of this evolutionary loop. LLM-LNS (Ye et al., 2025) evolves the prompting strategy in an outer loop, while CALM (Huang et al., 2025) adapts the LLM during the search process. HSEvo (Dat et al., 2025), EoH-S (Liu et al., 2026), and MCTS-AHD (Zheng et al., 2025) further improve population

management and exploration, replacing simple best-performing candidate pools with more structured search strategies.

Despite these advances, existing LLM-AHD methods still face challenges when applied to less-studied CO domains, where domain-specific heuristic priors are weaker and scalar or textual feedback may be insufficient to guide search. A more detailed comparison with closely related LLM-AHD methods is provided in Section B.2.

3 Preliminary: Automated Heuristic Design

3.1 Problem Formulation

We formulate AHD as optimization over a space of executable programs Θ. Consider a distribution of CO problem instances $\mathcal{D}_{target}$. Each problem instance is defined as a tuple $\Phi = (\mathcal{X}, g, \mathcal{C}, \mathcal{T})$, where $\mathcal{X}$ denotes the decision variable space of the CO problem, $g : \mathcal{X} \to \mathbb{R}^+ \cup \{0\}$ is the objective function to be minimized, $\mathcal{C}$ is the set of hard constraints, which a valid solution $x \in \mathcal{X}$ must satisfy, and $\mathcal{T}$ is a natural language description of the problem logic and constraints.

We search for a heuristic policy h_θ, parameterized by executable code $\theta \in \Theta$, that constructs or improves a solution $x = h_\theta(\Phi)$. The quality of a heuristic is measured by a fitness metric

$$f(\Phi, x) = \frac{g(x^*)}{g(x)}, \tag{1}$$

where x^* is the best-known (or optimal) solution. The fitness value is set to 0 when x is not a valid solution. Our goal is to find the optimal program parameters θ^* that maximize the expected fitness over the target distribution:

$$\theta^* = \arg\max_{\theta \in \Theta} \mathbb{E}_{\Phi \sim \mathcal{D}_{target}} \left[f(\Phi, h_\theta(\Phi)) \right] \tag{2}$$

In practice, we select heuristic function $\hat{\theta}$ based on a limited training set $\mathcal{D}_{train} = \{\Phi_1, \ldots, \Phi_K\}$ and evaluate generalization on $\mathcal{D}_{test}$.

3.2 LLM-based AHD

To solve Equation (2), existing LLM-based AHD pipelines (Romera-Paredes et al., 2024; Liu et al., 2024) typically use a population-based evolutionary loop, illustrated by the gray components in Figure 1 (a). The algorithm maintains a population of candidate heuristics $P_t = \{\theta_t^{(i)}\}_{i=1}^N$ over optimization iteration $t \in \{0, 1, \cdots, T\}$.

Starting from a seed population P_0, the algorithm transitions from P_t to P_{t+1}, relying on three stages: proposal, evaluation, and selection. At step t, a subset of parent heuristics $P_{\text{parents}} \subset P_t$ is selected from the population P_t. The LLM generator $\mathcal{G}$ acts as a mutation ($\mathcal{G}$ is prompted with a single program and asked to generate based on it) or crossover ($\mathcal{G}$ is prompted with two parent programs and asked to synthesize a new offspring program) operator, prompted with the parents and their scalar fitness score f_{parents} to generate a batch of new candidates $P_{new} = \{\theta'_1, \ldots\}$:

$$\theta' \sim \text{LLM}_{\mathcal{G}}(P_{\text{parents}}, f_{\text{parents}}, \mathcal{T}), \tag{3}$$

Each candidate $\theta' \in P_{\text{new}}$ is inserted into a solver skeleton and evaluated on $\mathcal{D}_{train}$ to obtain its training fitness $\bar{f}_{\theta'}$, which is the average fitness value over the training set:

$$\bar{f}_{\theta'} = \frac{1}{K} \sum_{k=1}^{K} f(\Phi_k, h_{\theta'}(\Phi_k)). \tag{4}$$

The population is then updated by selecting N heuristics with the highest fitness values from the union of the current population and the new candidates, $P_t \cup P_{new}$:

$$P_{t+1} = \text{Top}_N(\{\theta \mid \theta \in P_t \cup P_{new}\}, f). \tag{5}$$

The final heuristic $\hat{\theta}^*$ is the program with the highest training fitness value in the final population P_T.

This loop yields competitive results in well-documented CO problems (Liu et al., 2024; Ye et al., 2024). However, as shown in Section 5.2, extending it to less-studied domains remains challenging because high-performing heuristics are less documented and domain-specific priors are scarce.

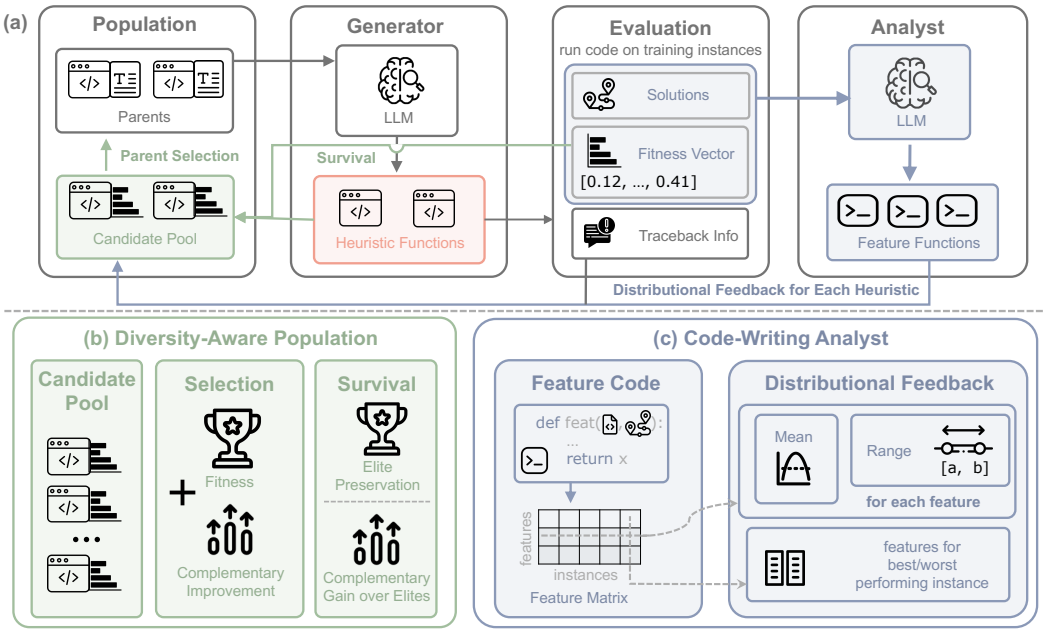

Figure 1: **Overview of GEAR. (a)** At each iteration, the generator proposes a candidate with both constructive and refinement heuristics, which is evaluated on the training instances. The Analyst summarizes the traces into structured feedback, and the Population Manager updates the pool.**(b)** Parent selection and survival favor both high-fitness candidates and candidates that complement the current elites by improving performance on instances where those elites underperform. **(c)** The Analyst writes feature functions over (problem, solution) pairs for each heuristic and summarizes their distributions into structured feedback for the generator.

4 METHOD

To improve the applicability of LLM-based AHD in less-studied domains, we revisit how the evolutionary loop exposes the search problem to the LLM. Our design is based on the observation that, even when domain-specific heuristic priors are weak, LLMs still provide two transferable capabilities relevant to AHD:

- **Generation:** LLMs are exposed during pre-training to a broad range of algorithmic and heuristic code patterns, allowing them to synthesize heuristic components.
- **Diagnosis:** While an LLM may not zero-shot generate an optimal heuristic for a novel domain, it can perform programmatic diagnosis. Given execution traces produced by a heuristic, an LLM can write analysis code to extract behavioral features, summarize failure modes across instances, and identify patterns that may explain poor performance.

These capabilities are not fully exploited when the generator is asked to update a narrow heuristic space using only aggregate fitness values. Therefore, we propose GEAR, a framework that preserves the evolutionary structure of standard LLM-based AHD while upgrading its key interfaces: (1) on the generation side, we search over a richer multi-component heuristic space and GEAR proposes a complementary-gain population management strategy, and (2) on the diagnosis side, GEAR introduces an executable Analyst that guides generation by producing execution-grounded, distributional feedback beyond scalar reward. The following sections describe these two designs in detail.

4.1 HEURISTIC DESIGN: JOINT GENERATION WITH COMPLEMENTARY-GAIN POPULATION MANAGEMENT

To increase the capacity of the generated algorithm and grant LLM more flexibility to improve it, we adopt a joint multi-component heuristic space as the search setting. Targeting frameworks like Large

Neighborhood Search (LNS) and Genetic Algorithms (GA), we represent each generated heuristic program as a set of editable components $\theta^i = \{\theta^i_{\text{cons}}, \theta^i_{\text{ref}}\}$, where θ^i_{cons} is a constructive heuristic that generates an initial solution and θ^i_{ref} is a refinement heuristic that guides subsequent improvement.

By prompting the generator LLM $\mathcal{G}$ to synthesize both components jointly, we grant it more flexibility to modify the algorithmic pipeline. Rather than restricting the LLM to optimizing a refinement step on fixed, potentially sub-optimal initial solutions, joint generation allows $\mathcal{G}$ to leverage its pre-trained coding capabilities to propose diverse combinations of construction and refinement strategies.

However, jointly evolving multiple components expands the search space. Candidates can have similar average fitness while performing well on different training instances, so average fitness alone can hide useful complementary strengths. To preserve and combine these strengths, GEAR uses complementary-gain population management alongside joint generation, as shown in Figure 1(b).

For each candidate θ, we record its per-instance fitness profile on $\mathcal{D}_{\text{train}}$:

$$\mathbf{v}_\theta = [f(\Phi_1, h_\theta(\Phi_1)), \ldots, f(\Phi_K, h_\theta(\Phi_K))]\,. \tag{6}$$

For a nonempty reference set S, we define the complementary gain of θ as

$$G(\theta \mid S) = \sum_{i=1}^{K} \left[v_{\theta,i} - \max_{\tilde{\theta} \in S} v_{\tilde{\theta},i} \right]_+, \qquad [x]_+ = \max(0, x). \tag{7}$$

Thus, a candidate earns gain only when it improves the best performance already represented in set S.

Parent Selection. For parent selection, where we select a pair of parents to seed the generation of new algorithm, we choose the first parent θ_a by a fitness tournament: we sample a small subset of the population and select the candidate with the highest average fitness in that subset. This favors strong starting points while allowing candidates beyond the population's top-ranked member to participate. Conditioned on θ_a, we sample the second parent θ_b with probability proportional to $G(\theta_b \mid \{\theta_a\})^\gamma$, where γ controls the strength of this preference and we set as 1. This favors a partner that performs better on instances where the first parent is weaker. If no partner has positive gain, we choose the highest-fitness remaining candidate. The resulting pair supplies the generator $\mathcal{G}$ with complementary programs for crossover.

Population Survival. After the generator $\mathcal{G}$ proposes a batch of new candidates P_{new}, the population is updated from the candidate pool $P_t \cup P_{\text{new}}$. We first preserve a small elite set with the highest average fitness to avoid discarding strong candidates. The remaining slots are iteratively filled greedily: at each step, we select the eligible candidate with the largest $G(\theta \mid S)$, where S is the set of survivors already chosen. Candidates with non-finite overall fitness cannot claim these gain slots. If no remaining candidate has positive gain, we fill the slots by average fitness.

This rule preserves strong candidates while giving a survivor slot to a program only when it adds an instance-level strength to the selected population. It directly affects which programs are shown to the LLM as parents and which programs survive into the next generation.

4.2 Heuristic Analysis: Grounded Code-Writing Analyst

To guide the generator $\mathcal{G}$ in heuristic synthesis, in addition to the parent program, we require a feedback signal for the structural properties associated with a heuristic's successes and failures. The diagnosis side of GEAR provides structured feedback for heuristic refinement. During heuristic evaluation, each candidate θ is executed on the training set $\mathcal{D}_{train}$, producing execution traces

$$\mathcal{E}_\theta = \{(\Phi_k, x_k, f_k, m_k)\}_{k=1}^{K}\,, \tag{8}$$

where $x_k = h_\theta(\Phi_k)$ is the solution produced on instance Φ_k, $f_k = f(\Phi_k, x_k)$ is its fitness, and m_k is the auxiliary traceback information, such as runtime or timeout errors. The traces contain richer information than the mean fitness $\bar{f}_\theta$, but are too noisy to pass directly to the generator.

To convert traces into useful feedback, GEAR introduces a code-writing LLM-based analyst. Rather than producing only free-form textual comments via pure reasoning, the Analyst writes analysis code that defines feature functions over problem-solution pairs:

$$\Psi_j : (\Phi, x) \to \mathbb{R}, \qquad j = 1, \ldots, M. \tag{9}$$

Each feature is intended to capture a scale-invariant structural property of the problem-solution pair, such as imbalance among routes, unused capacity ratio, constraint slack, or other domain-dependent characteristics. The feature functions are generated from the problem description $\mathcal{T}$, the current execution traces, and the Analyst's previous feedback history.

Once the Analyst proposes a set of feature functions $\{\Psi_j\}_{j=1}^{M}$, these functions are executed on the traces from the training set. For each heuristic θ, this yields a feature matrix

$$Z_\theta = [\Psi_j(\Phi_k, x_k)]_{j=1,k=1}^{M,K}. \tag{10}$$

The Analyst then summarizes this matrix into distributional feedback:

$$\Delta_\theta = \left(\bar{f}_\theta, \mathrm{range}(Z_\theta), \mu(Z_\theta), Z_\theta^{\mathrm{best}}, Z_\theta^{\mathrm{worst}}, \{m_k\}_{k=1}^{K}\right), \tag{11}$$

where $\mathrm{range}(Z_\theta) \in \mathbb{R}^{M\times 2}$ records the value range of each feature across $\mathcal{D}_{train}$, $\mu(Z_\theta) \in \mathbb{R}^M$ records the average value of each feature across $\mathcal{D}_{train}$, and $Z_\theta^{\mathrm{best}}, Z_\theta^{\mathrm{worst}} \in \mathbb{R}^M$ denote the feature vectors of the best- and worst-performing instances for that heuristic.

The resulting feedback gives the generator a more grounded view of the conditions under which a candidate works or fails. Instead of observing only that one heuristic has a higher average fitness than another, the generator receives evidence about which structural properties differ between successful and unsuccessful cases. The next generation is then sampled by generator $\mathcal{G}$ as

$$P_{\mathrm{new}} \sim \mathcal{G}\left(P_{\mathrm{parents}}, \{\Delta_\theta\}_{\theta\in P_{\mathrm{parents}}}, \mathcal{T}\right), \tag{12}$$

where $\mathcal{G}$ is prompted to propose coordinated edits to the heuristic components.

This executable form of diagnosis has two advantages. First, it grounds feedback in measured behavior rather than relying only on unverified textual reflection. Second, because the feature functions are code, they can be applied consistently across all training instances, allowing the feedback to describe distributional patterns rather than isolated examples.

5 Experiment

We evaluate GEAR across eight combinatorial optimization domains, focusing on both heuristic quality and generalization beyond the training instances. Our experiments address two primary questions: **RQ1 (Less-studied domains):** Can GEAR discover effective heuristics in domains with relatively few established heuristic designs, and do these heuristics generalize to unseen instances? **RQ2 (Scale-up generalization):** Do the heuristics discovered by GEAR remain effective when transferred to larger problem instances?

We further examine how the individual components of GEAR contribute to performance through a case study, ablations, convergence analysis, and computational-cost analysis.

5.1 Experimental Setup

Benchmarks and datasets. To evaluate performance across domains with varying levels of prior knowledge, we consider two groups of benchmarks:

- **Less-studied domains:** We evaluate four diverse problems from the HeuriGym benchmark (Chen et al., 2025), which targets comparatively under-explored CO domains with fewer established heuristic designs. The problems include: pickup and delivery with time windows (PDPTW), airline crew pairing, technology mapping, and intra-operator parallelism scheduling (intra-op) spanning logistics, scheduling, electronic design automation, and compiler optimization domains. Additional HeuriGym tasks are discussed in Section D.1.
- **Well-studied domains:** We evaluate four classical CO problems with well-established heuristics in prior works: Capacitated Vehicle Routing Problem (CVRP), Flow Shop Scheduling Problem (FSSP), Multi-Demand Multidimensional Knapsack Problem (MDMKP), and Periodic Vehicle Routing Problem (PVRP). Standardized instances and scale-up settings are taken from CO-Bench (Sun et al., 2026).

Table 1: **Comparison of our framework GEAR with baselines on less-studied domains**. Results are reported in terms of AF (higher is better) averaged over 3 runs using `GPT-5.6-Luna` with `low` reasoning effort.

Method	PDPTW		Crew Pairing		Techno Mapping		Intra-op	
	Train	Test	Train	Test	Train	Test	Train	Test
EoH	0.813	0.583	0.666	0.424	0.636	0.600	0.543	0.411
ReEvo	0.807	0.539	0.881	0.533	0.550	0.544	0.596	0.416
LLM-LNS	0.815	0.505	0.756	0.597	0.639	0.614	0.554	0.440
MCTS-AHD	0.754	0.555	0.937	0.768	0.603	0.559	0.520	0.413
HSEvo	0.765	0.624	0.699	0.455	0.609	0.575	0.469	0.395
GEAR	**0.941**	**0.667**	**1.251**	**1.104**	**0.658**	**0.626**	**0.704**	**0.488**

Baselines. We compare against representative population-based LLM-AHD methods: EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), LLM-LNS (Ye et al., 2025), MCTS-AHD (Zheng et al., 2025), and HSEvo (Dat et al., 2025) (selection rationale in Section B). All methods search the same joint two-component space: we adapt each baseline's generation interface to produce both components while retaining its original selection, update, and feedback mechanisms (Section E).

Metrics. We report **Average Fitness (AF)**, the mean of f (Equation (1)) over the K evaluation instances, averaged over 3 runs unless stated otherwise. Since f normalizes cost by the benchmark reference solution and assigns zero to infeasible solutions, AF reflects both solution quality and feasibility rate. During evolution, selection instead uses a penalty-shaped fitness $\tilde{f}$ (Section E) that gives a graded signal to infeasible candidates; all reported results use the strict f.

5.2 Results

GEAR achieves consistently strong observed performance on less-studied domains and unseen instances (RQ1). We first evaluate GEAR on four less-studied CO domains, where LLMs are expected to have weaker domain-specific heuristic priors. As shown in Table 1, GEAR achieves the highest mean train and test AF across domains. Importantly, the empirical baseline frontier is domain-dependent: HSEvo obtains the highest baseline mean on PDPTW, MCTS-AHD on Crew Pairing, and LLM-LNS on Technology Mapping and intra-op scheduling. Thus, no single baseline is uniformly strongest. In contrast, GEAR attains a higher observed mean than the strongest baseline for each domain, indicating robust performance across heterogeneous problem structures rather than an advantage over any one competing method.

The largest improvement occurs on Crew Pairing, while the margins on the other domains are smaller and indicate competitive, consistently favorable performance. Overall, these results suggest that GEAR can navigate distinct joint heuristic spaces effectively and that its relative training performance generally carries over to unseen instances.

Crew Pairing AF can exceed 1 because the benchmark normalization uses expert reference solutions rather than proven global optima. An AF above 1 therefore indicates that the generated heuristic exceeds the reference performance on average.

GEAR shows consistently strong observed performance under scale-up evaluation (RQ2). We next evaluate GEAR on four well-studied CO domains using a scale-up protocol, in which heuristics are trained on smaller instances and tested on larger ones. For example, MDMKP uses instances of sizes 100 and 250 for training and size 500 for testing, while PVRP uses sizes 50 and 75 for training and size 100 for testing. As shown in Table 2, GEAR achieves the highest mean test AF across three runs on all four domains. As in the less-studied setting, the empirical baseline frontier is domain-dependent: the strongest competing method varies across the four problems. In contrast, GEAR is the only evaluated method that achieves the highest observed mean across all four scale-up test sets, indicating consistently strong performance across different problem structures and scales.

The scale-up setting also reveals substantial train–test differences for some methods. As an illustrative example, MCTS-AHD and GEAR have the same reported mean CVRP training AF of 0.961, but

Table 2: **Comparison of our framework with LLM-based heuristic design methods on classic CO domains.** Results are reported as mean Average Fitness (higher is better) over three seeds using `GPT-5.6-Luna`. Test algorithms are evaluated with a 300-second time limit per instance.

Method	CVRP		FSSP		MDMKP		PVRP	
	Train	Test	Train	Test	Train	Test	Train	Test
EoH	0.951	0.858	0.950	0.811	0.692	0.511	0.757	0.672
ReEvo	0.934	0.666	0.967	0.670	0.677	0.493	0.761	0.678
LLM-LNS	0.908	0.768	0.941	0.850	0.731	0.601	0.781	0.688
MCTS-AHD	**0.961**	0.399	0.960	0.524	0.504	0.532	**0.791**	0.685
HSEvo	0.951	0.715	0.692	0.485	0.686	0.774	0.779	0.679
Ours	**0.961**	**0.936**	**0.978**	**0.957**	**0.732**	**0.789**	0.788	**0.726**

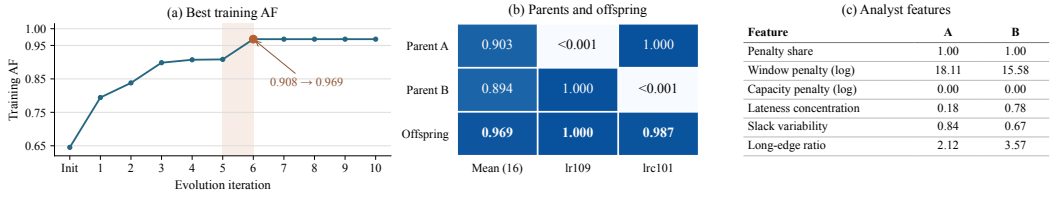

Figure 2: **PDPTW case study for GEAR.** (a) We analyze the highlighted improvement whichraises AF from 0.908 to 0.969 at iteration 6. (b) At the performance jump, the selected parents have complementary strengths: parent A performs well on `lrc101`, while parent B performs well on `lr109`. Their offspring performs well on both. (c) Analyst feature values for each parent on its worst-performing training instance: `lr109` for A and `lrc101` for B.

achieve mean test AFs of 0.399 and 0.936, respectively. Inspection of the evaluation traces suggests two possible failure modes behind such train–test degradation. First, some generated heuristics appear to specialize to smaller training instances and transfer less effectively as problem size increases. Second, several generated programs contain computationally expensive operations whose runtime grows rapidly with instance size, causing timeouts or leaving less time for solution improvement. These behaviors occur despite a 300-second test-time budget, five times the 60-second budget used during training. The observed results are therefore consistent with GEAR providing more reliable transfer to larger instances in the evaluated runs.

5.3 Case Study

We trace one improvement in a PDPTW run (Figure 2). At iteration 6, complementary-gain sampling pairs two parents whose mean training fitness (0.903 and 0.894) is below the incumbent's 0.908, but whose strengths are complementary: parent A solves `lrc101` (1.000) but fails on `lr109` (<0.001), while parent B shows the reverse. The Analyst's features attribute both failures to time-window violations rather than capacity, with lateness concentrated in a single visit for B (78%) but spread out for A (18%). The offspring combines bounded regret insertion with slack-aware removal scoring, reaches 1.000 and 0.987 on the two instances, eliminates lateness on both, and raises the incumbent from 0.908 to 0.969.

5.4 Analysis and Ablation

RQ3: How do individual components of GEAR contribute to performance? We conduct two subtractive ablations across the four less-studied CO domains in Table 3. Removing either the code-writing Analyst or complementary-gain sampling lowers train AF in all four domains and test AF in three, with the remaining domain (complementary gain on Technology Mapping) essentially unchanged. The effect of complementary-gain sampling is most pronounced on PDPTW, where its removal reduces test AF from 0.667 to 0.431, while the Analyst matters more on Technology Mapping and Intra-op. Each component therefore contributes to performance, with relative effects that vary across domains.

Table 3: **Ablation study on four less-studied CO domains.** We compare the full GEAR framework with variants that remove the code-writing Analyst or complementary-gain sampling.

Method	PDPTW		Crew Pairing		Tech. Mapping		Intra-op	
	Train	Test	Train	Test	Train	Test	Train	Test
GEAR	**0.941**	**0.667**	**1.251**	**1.104**	**0.658**	**0.626**	**0.704**	**0.488**
w/o analyst	0.837	0.593	1.094	0.999	0.621	0.575	0.587	0.432
w/o complementary sampling	0.843	0.431	1.091	1.002	0.634	0.624	0.576	0.424

RQ4: How does GEAR progress under joint evolution? We examine the evolution of best-so-far training AF in Figure 3. All methods operate over the same joint search space, evolving both constructive and refinement heuristics. Under this matched setting, GEAR improves rapidly and reaches the highest final training AF across all four less-studied domains. The advantage is attributed to the combined effect of complementary-gain population management and Analyst feedback in guiding the joint search. The component ablations in Table 3 indicate that both complementary-gain population management and Analyst feedback contribute to this advantage.

RQ5: What is the computational cost of GEAR? The additional components in GEAR affect only the offline discovery stage: the code-writing Analyst adds an LLM call and executes the generated diagnostic program to provide structured feedback. Under the GPT-5.6-Luna low-reasoning setting on Crew Pairing, GEAR uses approximately 307k tokens and 25 minutes per optimization iteration. EoH, ReEvo, LLM-LNS, and HSEvo use approximately 158k–250k tokens and 10–20 minutes per iteration, while MCTS-AHD uses approximately 573k tokens and 34 minutes. Thus, GEAR incurs additional cost relative to most baselines, but is not the most expensive method. Convergence as a function of wall-clock time (Figure 4) shows trends consistent with the iteration-matched comparison. After discovery, GEAR outputs an executable heuristic and requires no LLM interaction at test time.

6 Conclusion and Discussion

We presented GEAR, an LLM-based AHD framework for guiding heuristic search in less-studied combinatorial optimization domains. When multiple interacting heuristic components are searched jointly, aggregate fitness provides limited guidance for selection and refinement; GEAR addresses this through complementary-gain population management based on per-instance performance and an executable Analyst that produces structured feedback from evaluation traces. Across four less-studied and four well-studied CO domains, GEAR achieves the highest mean test AF among the evaluated methods, on both unseen and scale-up instances.

GEAR also has several limitations. First, it searches within solver templates specified in advance; while this keeps the search space interpretable and comparisons controlled, it does not determine the algorithmic structure itself. Future work could let the framework decide which modules to introduce, compose, or discard during search. Second, the Analyst summarizes completed executions and final solutions; reasoning over solving trajectories, i.e., how solutions are constructed, improved, or rejected, could give finer-grained diagnosis of which heuristic decisions drive improvement.

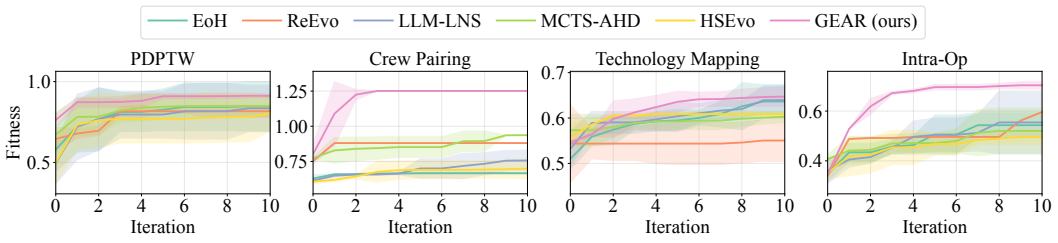

Figure 3: **Training convergence under matched joint evolution.** All methods jointly evolve the constructive and refinement heuristics under the same iteration budget. The curves show mean best-so-far training AF across runs, with shaded regions indicating run-to-run variability.

REPRODUCIBILITY STATEMENT

We provide the information needed to reproduce the experimental methodology and evaluate the reported findings. The problem formulation, fitness metric, and complete GEAR procedure are specified in Equation (2) and Sections C, 4.1 and 4.2. Benchmark selection, problem definitions, training and test protocols, model configurations, population size, iteration budget, and evaluation time limits are described in Sections D.1 and E. The prompts used by the Generator and Analyst are reported in Section F, and representative generated heuristics and Analyst feature functions are provided in Section G. We report results over three independent runs and provide the corresponding sample standard deviations in Section E.

AI USE STATEMENT

Generative AI was used in two distinct capacities in this work. First, LLMs are components of the methods studied: they generate and refine candidate heuristic programs, produce executable diagnostic feature functions in GEAR, and implement the corresponding generation or reflection mechanisms in the evaluated baselines. The models, prompts, and experimental configurations used for these purposes are described in the main paper and appendix.

Second, the authors used generative AI tools to assist with research implementation and manuscript preparation. Specifically, these tools assisted in refining human-written code, implementing adapters for baseline methods and datasets, processing and summarizing experimental outputs, and editing the manuscript for clarity and presentation. All AI-generated or AI-modified code was reviewed and tested by the authors. Reported numerical results were checked against the underlying experimental outputs, and AI-assisted text was reviewed for technical accuracy and consistency with the cited literature and empirical evidence.

The authors made all final decisions regarding the research methodology, experimental design, interpretation of results, and claims presented in the paper, and take full responsibility for the final content.

REFERENCES

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2623–2631, 2019.

Erik Andersson, Efthymios Housos, Niklas Kohl, and Dag Wedelin. Crew pairing optimization. In *Operations research in the airline industry*, pp. 228–258. Springer, 1998.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Aymeric Blot, Holger H Hoos, Laetitia Jourdan, Marie-Éléonore Kessaci-Marmion, and Heike Trautmann. Mo-paramils: A multi-objective automatic algorithm configuration framework. In *International Conference on Learning and Intelligent Optimization*, pp. 32–47. Springer, 2016.

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. doi: 10.1057/jors.2013.71. URL https://doi.org/10.1057/jors.2013.71.

Angelica Chen, David Dohan, and David So. Evoprompting: Language models for code-level neural architecture search. *Advances in neural information processing systems*, 36:7787–7817, 2023a.

Deming Chen and Jason Cong. Daomap: A depth-optimal area optimization mapping algorithm for fpga designs. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pp. 752–759. IEEE, 2004.

Hongzheng Chen, Yingheng Wang, Yaohui Cai, Hins Hu, Jiajie Li, Shirley Huang, Chenhui Deng, Rongjian Liang, Shufeng Kong, Haoxing Ren, Samitha Samaranayake, Carla P. Gomes, and Zhiru Zhang. Heurigym: An agentic benchmark for llm-crafted heuristics in combinatorial optimization. *arXiv preprint arXiv:2506.07972*, 2025. URL https://arxiv.org/abs/2506.07972.

Mark Chen. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023b.

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*, volume 39, pp. 26931–26938, 2025.

John H Drake, Ahmed Kheiri, Ender Özcan, and Edmund K Burke. Recent advances in selection hyper-heuristics. *European Journal of Operational Research*, 285(2):405–428, 2020.

Yvan Dumas, Jacques Desrosiers, and Francois Soumis. The pickup and delivery problem with time windows. *European journal of operational research*, 54(1):7–22, 1991.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Yilun Hao, Yongchao Chen, Yang Zhang, and Chuchu Fan. Large language models can solve real-world planning rigorously with formal verification tools. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3434–3483, 2025.

Juraj Hromkovič. *Algorithmics for hard problems: introduction to combinatorial optimization, randomization, approximation, and heuristics*. Springer Science & Business Media, 2013.

Ziyao Huang, Weiwei Wu, Kui Wu, Jianping Wang, and Wei-Bin Lee. Calm: Co-evolution of algorithms and language model for automatic heuristic design. *arXiv preprint arXiv:2505.12285*, 2025.

Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of artificial intelligence research*, 36:267–306, 2009.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. *arXiv preprint arXiv:2303.00001*, 2023.

Robert Lange, Yingtao Tian, and Yujin Tang. Large language models as evolution strategies. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 579–582, 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

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 *International Conference on Machine Learning (ICML)*, 2024. URL `https://arxiv.org/abs/2401.02051`.

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

Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. Fully autonomous programming with large language models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1146–1155, 2023.

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.

Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pp. 320–353. Springer, 2003.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

Yi Mei, Qi Chen, Andrew Lensen, Bing Xue, and Mengjie Zhang. Explainable artificial intelligence by genetic programming: A survey. *IEEE Transactions on Evolutionary Computation*, 27(3): 621–641, 2022.

Elliot Meyerson, Mark J Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K Hoover, and Joel Lehman. Language model crossover: Variation through few-shot prompting. *ACM Transactions on Evolutionary Learning*, 4(4):1–40, 2024.

Michael D Moffitt and Pratik Fegade. The asplos 2025/eurosys 2025 contest on intra-operator parallelism for distributed deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 5–17, 2025.

Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3):339–363, 2010.

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

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. Can language models solve olympiad programming? *arXiv preprint arXiv:2404.10952*, 2024.

Thomas Stützle and Manuel López-Ibáñez. Automated design of metaheuristic algorithms. In *Handbook of metaheuristics*, pp. 541–579. Springer, 2018.

Weiwei Sun, Shengyu Feng, Shanda Li, and Yiming Yang. Co-bench: Benchmarking language model agents in algorithm search for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 33126–33134, 2026.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, et al. Debugbench: Evaluating debugging capability of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 4173–4198, 2024.

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, 2024.

Meng Xu, Yi Mei, Fangfang Zhang, and Mengjie Zhang. Genetic programming for dynamic flexible job shop scheduling: Evolution with single individuals and ensembles. *IEEE Transactions on Evolutionary Computation*, 28(6):1761–1775, 2023.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

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*, 2024. `https://github.com/ai4co/reevo`.

Huigen Ye, Hua Xu, An Yan, and Yaoyang Cheng. Large language model-driven large neighborhood search for large-scale milp problems. In *Forty-second International Conference on Machine Learning*, 2025.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. In *First Conference on Language Modeling*, 2024.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023a.

Zi-Qi Zhang, Fang-Chun Wu, Bin Qian, Rong Hu, Ling Wang, and Huai-Ping Jin. A q-learning-based hyper-heuristic evolutionary algorithm for the distributed flexible job-shop scheduling problem with crane transportation. *Expert Systems with Applications*, 234:121050, 2023b.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. *arXiv preprint arXiv:2501.08603*, 2025.

Lei Zhu, Yusheng Zhou, Shuhui Sun, and Qiang Su. Surgical cases assignment problem using an efficient genetic programming hyper-heuristic. *Computers & Industrial Engineering*, 178:109102, 2023.

A Overview of Appendix

This Appendix contains several sections, each addressing a specific aspect of the experimental setups and results. Below is a brief overview of each section:

- Section B: Additional discussion on related works.
- Section C: How does the whole system work.
- Section D.1: More details about the problems we include in Table 1, and the reasons we select them from the benchmark while excluding the rest.
- Section D.2: Details about well-studied domains.
- Section E: More experiment details and results:
  - Experiment detail;
  - Convergence under same computation budget;
  - Standard deviation across runs;
  - Robustness to backbone LLM;
  - More ablation.
- Section D.3 More details of dataset.
- Section F: The prompts we use in our system.
- Section G: Example of our system output, including found heuristics and analyst feature functions.

B Additional Related Works

B.1 Code Generation with Execution Feedback

Beyond the specific domain of heuristics, there is a broader literature on how LLMs can iteratively improve executable code through interaction with execution environments Wang et al. (2024). This paradigm has been explored in diverse settings, including debugging Chen et al. (2023b); Tian et al. (2024), algorithmic competition challenges Li et al. (2022); Shi et al. (2024), and general planning problems Zhang et al. (2023a); Hao et al. (2025). A common theme in these approaches is the use of execution-based feedback, such as test outcomes, traces, or scalar rewards, to drive iterative refinement Liventsev et al. (2023); Guo et al. (2024). This idea has proven effective in applications ranging from reward design for reinforcement learning Ma et al. (2023); Kwon et al. (2023) to algorithm self-improvement Zelikman et al. (2024). However, despite their success, most existing methods treat the optimization loop as a tightly coupled, end-to-end process, where evaluation, diagnosis, and code generation are intertwined. In the specific context of automated heuristic design for CO, where solution quality depends on subtle structural interactions and constraint satisfaction, this design makes it difficult to reason about the role of individual components, compare methods in a principled manner, or systematically extend them to more structured domains.

B.2 Discussion of Closely Related LLM-AHD Works

Existing LLM-based automatic heuristic design (LLM-AHD) methods can be broadly grouped according to which part of the evolutionary loop they improve. We summarize the most closely related categories below.

- **Basic evolutionary LLM-AHD.** FunSearch Romera-Paredes et al. (2024) and EoH Liu et al. (2024) represent early and widely used LLM-AHD frameworks. These methods evolve a single heuristic component and rely on mutation or crossover under fixed prompts. The heuristic generation does not provide the generator with any feedback beyond the fitness value of candidate programs.
- **Feedback and diagnosis.** ReEvo Ye et al. (2024) introduces textual reflection into the backward pass, allowing the LLM to revise heuristics based on natural language feedback derived from previous candidates. However, the feedback remains textual and is not produced by executable analysis programs over instance-level execution traces.

- **Outer-loop adaptation.** LLM-LNS Ye et al. (2025) improves the search process by evolving prompts in an outer loop, while CALM Huang et al. (2025) updates the LLM weights online.
- **Population management.** HSEvo Dat et al. (2025), EoH-S Liu et al. (2026), and MCTS-AHD Zheng et al. (2025) improve exploration by changing how candidate heuristics are organized or selected. HSEvo and EoH-S are closest to GEAR because they also use diversity to maintain a heterogeneous population. However, their diversity metrics are different and the diversity mechanisms aren't applied to both survival and parent selection for crossover. MCTS-AHD instead improves exploration through tree-structured search, which is orthogonal to our population-based design.

Our experiments select representative baselines from these categories: EoH Liu et al. (2024) for basic evolutionary LLM-AHD, ReEvo Ye et al. (2024) for reflection-based feedback, LLM-LNS Ye et al. (2025) for outer-loop prompt adaptation, HSEvo Dat et al. (2025) for diversity-aware population management, and MCTS-AHD Zheng et al. (2025) for tree-structured search. This selection covers the main algorithmic axes most relevant to GEAR while keeping the empirical comparison focused.

C Steps of the Whole System

Algorithm 1 The GEAR Framework

Require: Problem description $\mathcal{T}$, training set $\mathcal{D}_{\text{train}}$, generator $\mathcal{G}$, Analyst $\mathcal{A}$, population size N, elite size E, iterations T, gain exponent γ

Ensure: Best heuristic $\hat{\theta}$

$P_0 \leftarrow \textsc{ZeroShotInit}(\mathcal{G}, \mathcal{T})$
Evaluate P_0 to obtain traces $\mathcal{E}_\theta$, fitness profiles $\mathbf{v}_\theta$, and mean fitness $\bar{f}_\theta$
Initialize Analyst features and feedback Δ_θ from the traces of P_0
for $t = 0, 1, \ldots, T-1$ **do**
  // Complementary-gain parent selection
  Select first parents θ_a by fitness tournaments
  Sample second parents θ_b with probability proportional to $G(\theta_b \mid \{\theta_a\})^\gamma$
  If all gains are zero, choose the highest-fitness remaining candidate
  // Joint generation and evaluation
  $P_{\text{new}} \sim \mathcal{G}(P_{\text{parents}}, \{\Delta_\theta\}_{\theta \in P_{\text{parents}}}, \mathcal{T})$
  Evaluate P_{new} to obtain $\mathcal{E}_\theta$, $\mathbf{v}_\theta$, and $\bar{f}_\theta$
  // Analyst feedback
  Update the shared feature functions using the new traces
  Recompute Δ_θ for $\theta \in P_t \cup P_{\text{new}}$
  // Complementary-gain survival
  $S \leftarrow$ top-E candidates by $\bar{f}_\theta$
  while $|S| < N$ **do**
    Add the eligible candidate maximizing $G(\theta \mid S)$
    If no candidate has positive gain, fill the remaining slots by $\bar{f}_\theta$
  end while
  $P_{t+1} \leftarrow S$
end for
return $\hat{\theta} = \arg\max_{\theta \in P_T} \bar{f}_\theta$

D Dataset Detail

D.1 Combinatorial Problems in HeuriGym

There are nine real-world combinatorial optimization problems from HeuriGym Chen et al. (2025). In this paper, we include four problems from the benchmark and exclude the remaining five problems. The main criterion is the quality of the greedy algorithm's solution. Simple greedy algorithms can find optimal or near-optimal solutions for all the test cases of Operator Scheduling, E-Graph Extraction, Protein Sequence Design, Global Routing, and Mendelian Error Detection. The problems, hence, are

not sufficiently challenging to evaluate the performance of LLM-based heuristic evolution pipelines using the latest LLMs (achieving >0.95 fitness value in the very first iteration for all methods, leaving little headroom to compare). Here we introduce the problems included in Table 1.

Pickup and Delivery Problem with Time Windows The Pickup and Delivery Problem with Time Windows (PDPTW) Dumas et al. (1991) is a generalization of the Capacitated Vehicle Routing Problem (CVRP), in which a fleet of vehicles must serve a set of transportation requests, each consisting of a pair of pickup and delivery locations. The core challenge is to construct a set of routes that minimizes travel cost (distance or time) and fleet size while strictly adhering to pairing and precedence constraints—specifically, that a request's pickup must precede its delivery on the same vehicle. Additionally, service at every location must commence within a specified time window, and the vehicle's load must never exceed its capacity. This problem is widely used to benchmark solvers on their ability to handle precedence-constrained routing and tight temporal feasibility checks, which are central to applications like on-demand logistics, paratransit services, and autonomous ride-sharing.

Airline Crew Pairing The Airline Crew Pairing Problem (CPP) Andersson et al. (1998) is formulated as a cost-minimization problem over a set of flight legs, modeled as a large-scale Set Partitioning Problem (SPP) or Set Covering Problem (SCP). The objective is to construct a set of "pairings", e.g., sequences of flight legs starting and ending at the same crew base, to cover every flight leg exactly once. This problem is characterized by highly constrained search spaces due to complex validity regulations, such as FAA safety rules on maximum flying time and minimum rest periods. The total costs are a complex function of factors such as wages, hotel accommodations, and deadhead travel. We utilize CPP instances to evaluate the solver's ability to handle high-dimensional combinatorial constraints and non-convex objective landscapes.

Technology Mapping Technology Mapping Chen & Cong (2004) is a fundamental step in the Electronic Design Automation (EDA) logic synthesis flow, responsible for binding a technology-independent Boolean network (typically an And-Inverter Graph) to a specific library of physical gates. The problem is formally defined as a graph covering problem: finding a valid cover of the subject graph using a set of K-input subgraphs. The optimization goal is to minimize the total number of subgraphs. This problem serves as a rigorous benchmark for combinatorial optimization under large input size.

Intra-Operator Parallelism The Intra-Operator Parallelism (IOPDDL) Moffitt & Fegade (2025) is a combinatorial optimization challenge introduced in the ASPLOS'25 contest track, focusing on the efficient distribution of large-scale deep learning models across multiple hardware accelerators. The problem is formulated on a large computational graph, where nodes represent operations and edges represent tensor dependencies. For each node, the solver must select a single execution strategy from a set of discrete candidates, each defined by specific computational costs and memory requirements. The objective is to minimize the aggregate cost—comprising both operation execution (node cost) and communication overhead (edge cost)—while strictly adhering to peak memory usage constraints on every device. Due to the diversity of model topologies and the scale of operations involved, this problem serves as a rigorous benchmark for constrained graph optimization, where leading solutions currently rely on specialized meta-heuristic strategies.

D.2 Well-studied Combinatorial Problem

We follow the problem definitions and reference values provided by CO-Bench Sun et al. (2026). Because the original domains include both minimization and maximization problems, we express each objective using a positive minimization-oriented cost $g_\Phi(x)$. For a problem whose original objective $o_\Phi(x) > 0$ is minimized, we set $g_\Phi(x) = o_\Phi(x)$. For a problem whose original objective is maximized, we use the reciprocal transformation

$$g_\Phi(x) = \frac{1}{o_\Phi(x)}.$$

This transformation preserves the ordering of feasible solutions. Consequently, the fitness definition in the main paper, $f(\Phi, x) = g_\Phi(x^{\mathrm{ref}})/g_\Phi(x)$, becomes $o_\Phi(x^{\mathrm{ref}})/o_\Phi(x)$ for minimization problems

and $o_\Phi(x)/o_\Phi(x^{\text{ref}})$ for maximization problems. Here x^{ref} denotes the fixed reference solution supplied by the benchmark; it need not be a proven global optimum. Infeasible solutions receive zero fitness.

Capacitated Vehicle Routing Problem (CVRP). CVRP requires a collection of vehicle routes that start and end at a common depot, visit every customer exactly once, and respect the capacity of each vehicle. The original objective is to minimize the total Euclidean distance traveled across all routes. We therefore use the total route distance directly as $g_\Phi(x)$. If $d(x)$ is the distance of a feasible solution and d^{ref} is the benchmark reference distance, its fitness is $f(\Phi, x) = d^{\text{ref}}/d(x)$.

Flow Shop Scheduling Problem (FSSP). In FSSP, all jobs must be processed on the same ordered sequence of machines, and the heuristic outputs a permutation of the jobs. Given a permutation, its makespan $C_{\max}(x)$ is the completion time of the final job on the final machine. Since the objective is to minimize makespan, we set $g_\Phi(x) = C_{\max}(x)$. Following CO-Bench, the reference value is the instance-specific lower bound or reference makespan supplied with the benchmark, and the resulting fitness is $f(\Phi, x) = C^{\text{ref}}_{\max}/C_{\max}(x)$.

Multi-Demand Multidimensional Knapsack Problem (MDMKP). MDMKP is naturally a maximization problem. A solution is a binary vector $x \in \{0, 1\}^n$ that must satisfy both upper-bound resource constraints and lower-bound demand constraints. Its original objective is to maximize

$$p(x) = \sum_{j=1}^{n} c_j x_j,$$

where the benchmark includes both positive and mixed objective coefficients. For the feasible instances used in our experiments, the evaluated objective and reference values are positive. We express the problem in the common minimization convention using $g_\Phi(x) = 1/p(x)$. Therefore,

$$f(\Phi, x) = \frac{g_\Phi(x^{\text{ref}})}{g_\Phi(x)} = \frac{p(x)}{p^{\text{ref}}},$$

which is exactly the direction-aware normalization used for this maximization problem. Thus, a higher-profit feasible solution always receives higher fitness.

Periodic Vehicle Routing Problem (PVRP). PVRP extends vehicle routing over a multi-day planning horizon. For each customer, the solver first selects one admissible service schedule and then constructs the required daily routes. Each route must start and end at the depot, satisfy vehicle-capacity constraints, and collectively serve exactly the customers scheduled for that day. The objective is to minimize the total Euclidean distance traveled by all vehicles over the complete planning period. We therefore set $g_\Phi(x)$ to this total distance, giving $f(\Phi, x) = d^{\text{ref}}/d(x)$ for a feasible solution.

D.3 Benchmark scale and experimental configuration.

The fixed training/test splits are as follows. Crew pairing contains 7/7 files; intra-operator parallelism 13/15 files; PDPTW 16/14 instances; and technology mapping 15/16 circuits. For the four classical optimization domains: CVRP contains 83/109 instances, flow-shop scheduling contains 60/60 Taillard instances, MDMKP contains 540/270 evaluated instances derived from 90/45 base cases, and PVRP contains 6/4 instances. Training and test instances generally differ in scale. In particular, CVRP uses 16–80 nodes for training and 101–1,001 for testing; flow-shop scheduling uses 20–50 jobs for training and 100–500 for testing; MDMKP uses 100–250 versus 500 items; and PVRP uses 50–75 versus 100 customers.

For the time limit, Crew pairing, intra-operator parallelism, PDPTW, and technology mapping use 60 seconds per training and test file. CVRP, flow-shop scheduling, MDMKP, and PVRP use 60 seconds per training file and 300 seconds per held-out test file.

Generator and Analyst calls used `us.openai.gpt-5.6-luna` through AWS Bedrock with reasoning effort `low`. We used non-streaming generation with a requested temperature of 1.0; unspecified sampling parameters used provider defaults.

GEAR begins with 10 candidate programs. Each of 10 subsequent iterations generates 10 candidates from each of the three operators $\mathrm{e1}$, $\mathrm{e2}$, and $\mathrm{m1}$, for 310 candidate programs per run. The population size is 10, with an elite ratio of 0.2 and tournament size three. Failed solver executions are not retried during evaluation and receive zero held-out fitness.

Penalty constants are domain-specific rather than governed by a single global M. PDPTW uses $M_{\mathrm{cap}} = M_{\mathrm{tw}} = 10^5$; intra-operator parallelism uses 10^{20} per illegal-cost unit; MDMKP uses $M_{\leq} = M_{\geq} = 10^6$; and PVRP uses 10^6 per structural violation and 10^3 per unit of capacity violation. Crew pairing instead includes a positioning cost of 10^4 as an objective term, while CVRP, flow-shop scheduling, and the primary technology-mapping evaluator treat invalid outputs as unsuccessful evaluations rather than applying a finite penalty.

E Experiment

E.1 Experiment Detail

In the experiments on all tested domains, we run the automatic heuristic design for 10 iterations and the population size is 10.

Search fitness and constraint handling. We distinguish between the fitness used during heuristic discovery and the strict fitness used for evaluation. The reported Average Fitness (AF) uses the definition in the main paper: an infeasible solution receives zero fitness, while a feasible solution is normalized against the fixed benchmark reference.

Using this strict feasibility gate during evolution produces a sparse search signal, because heuristics with substantially different degrees of constraint violation all receive zero fitness. We therefore use a standard penalty-based fitness shaping scheme during the offline discovery stage. Let $V(\Phi, x) \geq 0$ denote the aggregate constraint violation of solution x, where $V(\Phi, x) = 0$ if and only if x is feasible. We define the penalized cost

$$\tilde{g}(\Phi, x) = g(\Phi, x) + M_{\Phi} V(\Phi, x), \tag{13}$$

and the shaped search fitness

$$\tilde{f}(\Phi, x) = \frac{g(\Phi, x^{\mathrm{ref}})}{\tilde{g}(\Phi, x)}. \tag{14}$$

For feasible solutions, $\tilde{f}(\Phi, x) = f(\Phi, x)$. For infeasible solutions, $\tilde{f}$ provides a graded signal that favors candidates with smaller objective values and fewer or less severe violations.

The shaped fitness $\tilde{f}$ is used for parent selection, complementary-gain population management, and survival during evolution. The Analyst additionally receives the individual violation components so that it can distinguish different failure modes. In contrast, model selection and all AF values reported in the tables and figures use the strict fitness f, under which infeasible solutions receive zero.

Adapting baselines to joint evolution. Most baseline methods were originally designed to evolve a single heuristic function. To ensure that all methods search over the same heuristic space, we adapt each baseline to represent an individual as a pair $\theta = (\theta_{\mathrm{cons}}, \theta_{\mathrm{ref}})$, consisting of a constructive heuristic and a refinement heuristic. We retain each baseline's original search procedure, including its parent selection, mutation or crossover operators, population update, and feedback mechanism. We modify only the generation interface: the baseline prompts are extended to request both functions jointly, and the solver interface is changed to accept the resulting pair of functions. Each paired candidate is then evaluated as a single individual using the same evaluation protocol. Thus, all methods have access to the same two-component search space, allowing the comparison to isolate the benefits of GEAR's population-management and analysis mechanisms rather than the benefit of joint evolution alone.

E.2 Additional Experiment - Convergence under Same Computation Budget

We further evaluate convergence performance based on total wall-clock time rather than generation steps. As shown in Figure 4, while GEAR incurs a slight computational overhead per iteration, its

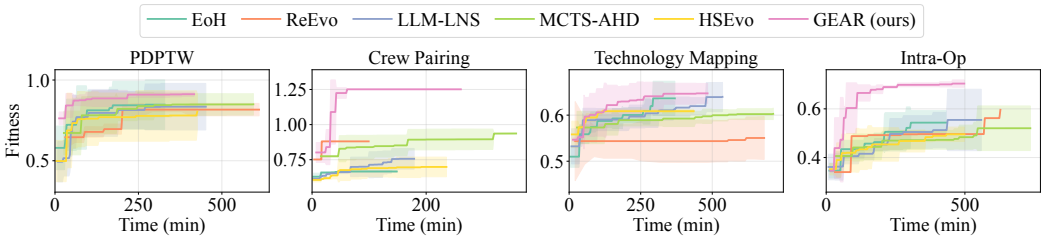

Figure 4: Convergence curves during training, plotted w.r.t. walltime.

Table 4: **Standard deviations for the comparison with LLM-based heuristic design method on less-explored domains**. Baseline methods evolve both initial solution and refinement heuristics at the same time.

Method	PDPTW		Crew Pairing		Techno Mapping		Intra-op	
	Train	Test	Train	Test	Train	Test	Train	Test
EoH	0.151	0.084	0.005	0.014	0.037	0.007	0.050	0.006
ReEvo	0.012	0.025	0.046	0.154	0.041	0.001	0.074	0.038
LLM-LNS	0.116	0.045	0.073	0.178	0.032	0.007	0.126	0.029
MCTS-AHD	0.122	0.152	0.002	0.116	0.012	0.001	0.092	0.013
HSEvo	0.089	0.072	0.070	0.072	0.004	0.018	0.030	0.009
GEAR	0.039	0.092	0.004	0.027	0.006	0.021	0.015	0.011

search efficiency remains superior. Under the same total time budget, GEAR still outperforms all baseline methods.

E.3 Additional Experiment: Variability Across Runs

Tables 4 and 5 report the sample standard deviations of AF over three independent runs for the less-studied and classic CO domains, respectively. These values complement the mean results reported in the main paper by showing the run-to-run variability for each method, domain, and train/test split.

E.4 Additional Experiment: Robustness to the Backbone LLM

The main results are obtained with `GPT-5.6-Luna`. To verify that the improvement is a property of the framework rather than of one particular backbone, we repeat the main-result protocol with `Gemini-3-Flash`, a model from a substantially different family and generation and report average over 3 runs. Given the weaker models, we extend the run to 20 iterations, which is the only difference other than backbone LLM.

As shown in Table 6, GEAR achieves the highest AF in all domains. We found that a weaker backbone degrades primarily through infeasible generalization, and the structure GEAR imposes on the search is what absorbs this failure mode.

E.5 Additional Experiment: Ablate Joint Evolution

To assess the importance of jointly evolving the initialization and refinement components, we compare our full method with variants that evolve only one component while keeping the other fixed. As shown in Table 7, single-component evolution generally underperforms joint evolution across the four domains. Overall, these results demonstrate that the two evolved components are complementary.

F Prompt Design

We present the prompt used in the framework for analyst LLM and generator LLM, respectively.

Table 5: **Standard deviations for the comparison with LLM-based heuristic design methods on classic CO domains.** Results are sample standard deviations of Average Fitness over three seeds. Test algorithms are evaluated with a 300-second time limit per instance.

Method	CVRP		FSSP		MDMKP		PVRP	
	Train	Test	Train	Test	Train	Test	Train	Test
EoH	0.003	0.150	0.017	0.215	0.004	0.443	0.027	0.013
ReEvo	0.007	0.162	0.012	0.023	0.000	0.000	0.004	0.012
LLM-LNS	0.017	0.185	0.002	0.152	0.063	0.250	0.015	0.013
MCTS-AHD	0.013	0.355	0.018	0.044	0.438	0.463	0.008	0.012
HSEvo	0.002	0.397	0.462	0.423	0.021	0.028	0.007	0.026
Ours	0.001	0.007	0.006	0.034	0.068	0.059	0.002	0.030

Table 6: **Transferred comparison of our framework GEAR with baselines on less-studied domains with a new LLM backbone**. Results are reported in terms of AF (higher is better) using `Gemini-3-Flash`.

Method	PDPTW		Crew Pairing		Techno Mapping		Intra-op	
	Train	Test	Train	Test	Train	Test	Train	Test
EoH	0.318	0.126	0.787	0.744	0.481	0.507	0.311	0.310
ReEvo	0.342	0.111	0.767	0.744	0.421	0.519	0.311	0.333
LLM-LNS	0.407	0.133	0.764	0.744	0.523	0.522	0.312	0.306
MCTS-AHD	0.377	0.253	0.767	0.744	0.472	0.499	0.314	0.333
HSEvo	0.422	0.123	0.767	0.744	0.462	0.497	0.315	0.333
GEAR	**0.784**	**0.664**	**0.910**	**0.848**	**0.562**	**0.539**	**0.628**	**0.435**

F.1 ANALYST

System Prompt

You are a world-class optimization expert specializing in metaheuristics and LNS. Your task is to design a suite of diagnostic features that characterize both the difficulty of a problem instance and the quality/bottlenecks of a current solution.

Objectives:

1. Discriminative Power: Features must help distinguish why certain instances are harder than others and why specific solutions are suboptimal. Please also be noted that different instances may have significant difference in scale and absolute cost values, which doesn't necessarily reflect the performance.
2. Feasibility Analysis: Capture violations, which are represented as soft constraint costs.
3. Diversity: Include spatial, temporal, and other structural metrics if applicable.

Implementation Requirements:

Table 7: **Ablation of joint evolution.** We compare joint evolution with variants that evolve only the initialization or only the refinement heuristic. Higher is better.

Method	PDPTW		Crew Pairing		Tech. Mapping		Intra-op	
	Train	Test	Train	Test	Train	Test	Train	Test
GEAR	**0.941**	**0.667**	**1.251**	**1.104**	**0.658**	**0.626**	**0.704**	**0.488**
w/o init evolution	0.896	0.605	0.934	0.895	0.601	0.609	0.340	0.393
w/o refinement evolution	0.895	0.613	1.213	0.804	0.510	0.576	0.535	0.468

- Input Argument: Every function must accept exactly the arguments as provided in the example.
- Output: Each function must return a single ‘float‘.
- Aggregation: You must provide a list named ‘feature_func_list‘ containing all your function objects.
- Environment: Python 3.12. Use ‘numpy‘ for efficiency. Include all necessary imports.
- Strictness: Output ONLY the Python code. No explanations, no markdown code blocks, no comments outside the code.

Problem Information: {problem_description}
Template analyst code: {template_analyst_code}

User Prompt

The following features were used in the previous iteration: {history_feature_code}

Your analysis for currently best individual is {current_best_heuristic_feature},
Whereas the solution costs are as follows: {current_best_heuristic_training_cost}

(if no improved in last iteration)
The previous iteration did NOT show improvement in heuristic performance. This suggests the current features might not be capturing the critical aspects of the problem needed for better decision-making. Please significantly modify or add new features that provide deeper insights into the solution state.

(if improved in last iteration)
The previous iteration showed improvement in heuristic performance. You can choose to refine the existing features or add complementary ones to further capture the problem structure.
Please provide several feature functions that can characterize the state of a solution for this problem or the critical aspects of the problem instance. End your response with a list named ‘feature_func_list‘ containing the function objects.

User Prompt (Error Fixing)

Your previous feature code encountered errors during execution. Please identify and fix the issues in the feature functions to ensure they run correctly and provide meaningful analysis. Use same formatting as provided feature code and modify only the necessary parts to fix the errors.

Previous Feature Code:{past_feature_code}
The errors are as follows:{best_individual_analysis_traceback_errors}

F.2 GENERATOR

System Prompt

The task can be solved step-by-step by starting from a constructive initial solution and iteratively selecting a subset of decision variables to relax and re-optimize. In each step, most decision variables are fixed to their values in the current solution, and only a small subset is allowed to change. You need to score all the decision variables based on the information I give you, and I will choose the decision variables with high scores as neighborhood selection. To avoid getting stuck in local optima, the choice of the subset can incorporate a degree of randomness.

+ You need to come up with a heuristic function which score a state. A good practice is to analyze the cost function and current solution to find what contributes most to the cost. Think systematically and creatively. Also I suggest the score function to be weighted sum. Note that the cost function actually soft constraints with very heavy penalty. You should pay more attention to these than the normal cost.

+ You also need to provide an initial solution function, which should be a constructive heuristic. The initial solution itself does not necessarily perform well, but it needs to provide a warmstart for neighborhood search to get close to a good local minimum. You also need to pay attention to the synergy between the initial solution and heuristics.

+ You should be noted that the generated solution should be general for out-of-distribution data, which may vary in scale and characteristic of numbers. So don't just exploit what you know, try to find different strategies which all work good on training sets.

+ A problem template is provided below. You need to implement both the 'heuristic' and '_init_solution' function. Do NOT modify the function signature including the data types of the input arguments. Make sure you have imported necessary libraries and modules.

Template Heuristic Code: {template_heuristic}

User Prompt (Operator i1 - initialization)

"Your task is to initialize a new heuristic algorithm for scoring decision variables. " "Refer to the format of a trivial design provided. Be very creative."

"Step 1: Briefly describe your new algorithm and its main steps in one sentence. This description MUST be inside braces {} and commented by """ """ as in Python."

"Step 2: Implement the algorithm in Python. Ensure all necessary imports are included, and the function signature matches the template."

User Prompt (Operator e1 - crossover)

"I have two existing algorithms with their codes and performance features as follows:"

"Algorithm 1 (better):"

f"{Individual 1}"

"Algorithm 2 (worse):"

f"{Individual}"

"Please help me create a new algorithm that is different from the given ones but can be motivated from them." "Your task is to synthesize a new algorithm by merging the most effective components as hints and design patterns from both."

"Step 1: Briefly describe the new synthesized algorithm and its main steps in one sentence. This description MUST be inside braces and commented by """ """ as in python."

"Step 2: Implement the new algorithm in Python. Ensure all necessary imports are included and the function signature matches the template."

User Prompt (Operator e2 - crossover)

"I have two existing algorithms with their codes and performance features as follows:"
"Algorithm 1 (better):"

f"{Individual 1}"

"Algorithm 2 (worse):"

f"{Individual 2}"

"Your task is to evolve a new algorithm by identifying the differences in their performance and logic. " "Determine which algorithm performs better and why. Then, identify a limitation in the better algorithm that the other might address, " "or propose a novel mechanism inspired by their differences to overcome current bottlenecks."

"Step 1: Briefly describe the new evolved algorithm and its main steps in one sentence. This description MUST be inside braces and commented by """ """ as in python."

"Step 2: Implement the new algorithm in Python. Ensure all necessary imports are included and the function signature matches the template."

User Prompt (Operator m1 - mutation)

"I have one existing algorithm with its code and performance features as follows:"

f"{Individual}"

"Your task is to 'mutate' this algorithm to create a new variant. " "Analyze the current scoring logic and identify one core component or assumption that can be improved, " "diversified, or replaced with a different mathematical approach to explore a new area of the search space."

"Step 1: Briefly describe the new mutated algorithm and its main steps in one sentence. This description MUST be inside braces and commented by """ """ as in python."

"Step 2: Implement the new algorithm in Python. Ensure all necessary imports are included and the function signature matches the template."

F.3 Format of Individual Heuristic

Individual

Response at iteration {iteration}
code={code}
text_description="""{text description}"""
avg_objective {avg objective}
error_msg='{error msg}'
Performance Summary:
- {feature 1}: Avg={avg feature 1 value}, Range=[minimum feature 1 value, maximum feature 1 value]
- {feature 2}: Avg={avg feature 2 value}, Range=[minimum feature 2 value, maximum feature 2 value]
...
- {feature N}: Avg={avg feature N value}, Range=[minimum feature N value, maximum feature N value]

- Worst Instance ({instance}): Fitness={instance fitness}, Features={instance feature values}"

- Best Instance ({instance}): Fitness={instance fitness}, Features={instance feature values}"

G EXAMPLE OUTPUT

```
"""{
Construct a fast urgency-and-regret warm start with bounded pair insertion, then
select destroy neighborhoods using penalty severity, backward slack propagation,
route congestion, pair fragility, and an adaptive removal-impact estimate.
}"""


def _init_solution(
    num_vehicles: int,
    depot: int,
    pickup_to_delivery,
    distance_mat,
    node_earliests,
    node_latests,
    node_service_times,
    instance_capacity,
    node_demands,
    delivery_siblings,
    is_pickup,
) -> List[List[int]]:
    """
    Fast constructive heuristic.

    Unlike full regret insertion, this implementation uses a bounded number of
    candidate arcs and evaluates only the affected route.  This keeps growth
    close to linear for long routes while retaining feasibility pressure.
    """
    depot = int(depot)
    pairs = [
        (int(x[0]), int(x[1]))
        for x in pickup_to_delivery
        if len(x) >= 2
    ]
    if not pairs:
        return [[depot, depot]]

    D = np.asarray(distance_mat, dtype=float)
    E = np.asarray(node_earliests, dtype=float)
    L = np.asarray(node_latests, dtype=float)
    S = np.asarray(node_service_times, dtype=float)
    W = np.asarray(node_demands, dtype=float)

    nv = max(1, int(num_vehicles))
    Q = max(1e-9, abs(float(instance_capacity)))

    finite = D[np.isfinite(D) & (D > 0)]
    scale = max(1e-9, float(np.median(finite)) if finite.size else 1.0)

    def evaluate(route):
        t = 0.0
        load = 0.0
        penalty = 0.0
        distance = 0.0

        for a, b in zip(route[:-1], route[1:]):
            distance += D[a, b]
            if b == depot:
                continue

            arrival = t + D[a, b]
            late = max(0.0, arrival - L[b])
            if late > 0:
                penalty += 30000.0 + 800.0 * late

            t = max(arrival, E[b]) + S[b]
            load += W[b]

            if load > Q:
                penalty += 28000.0 + 1500.0 * (load - Q)
            if load < 0:
                penalty += 24000.0 + 1500.0 * (-load)

        return penalty + 0.01 * distance

    def positions(route, node):
        m = len(route) - 1
        if m <= 8:
            return range(1, m + 1)

        vals = []
        for i in range(1, m + 1):
            a, b = route[i - 1], route[i]
            extra = D[a, node] + D[node, b] - D[a, b]
            vals.append((extra, i))
```

```
        # Geometric candidates, route ends, and one temporal anchor.
        chosen = {1, m}
        chosen.update(i for _, i in sorted(vals)[:4])

        for i in range(1, m + 1):
            if E[route[i]] >= E[node]:
                chosen.add(i)
                break
        return sorted(chosen)

    def insert_options(route, p, d, max_options=2):
        base = evaluate(route)
        out = []

        for i in positions(route, p):
            r1 = route[:i] + [p] + route[i:]
            for j in positions(r1, d):
                if j <= i:
                    continue
                r2 = r1[:j] + [d] + r1[j:]
                delta = evaluate(r2) - base

                # Prefer compact pair placement only when penalty is comparable.
                delta += 0.002 * max(0, j - i - 1) * D[p, d] / scale
                out.append((delta, r2))

        out.sort(key=lambda z: z[0])
        return out[:max_options]

    def urgency(pair):
        p, d = pair
        wp = max(1e-6, L[p] - E[p])
        wd = max(1e-6, L[d] - E[d])
        span = max(1e-6, L[d] - E[p])
        return (
            1.0 / wp
            + 1.0 / wd
            + 0.04 * (D[p, d] + S[p] + S[d]) / span
        )

    remaining = sorted(pairs, key=urgency, reverse=True)
    routes = []

    while remaining:
        chosen = None
        chosen_route = None
        chosen_value = float("inf")
        chosen_regret = -float("inf")

        # Limit pair scans on very large instances while always inspecting the
        # most urgent requests.
        scan = remaining if len(remaining) <= 160 else (
            remaining[:120] +
            list(np.random.choice(
                remaining[120:],
                size=min(40, len(remaining) - 120),
                replace=False,
            ))
        )

        for p, d in scan:
            alternatives = []

            for rid, route in enumerate(routes):
                opts = insert_options(route, p, d)
                if opts:
                    first = opts[0]
                    second = opts[1][0] if len(opts) > 1 else first[0] * 1.08 + 1.0
                    alternatives.append((first[0], rid, first[1], second))

            if len(routes) < nv:
                fresh = [depot, p, d, depot]
                c = evaluate(fresh)
                alternatives.append((c, len(routes), fresh, c + max(1.0, abs(c)) * 0.12))

            if not alternatives:
                continue

            alternatives.sort(key=lambda x: x[0])
            best = alternatives[0]
            regret = alternatives[1][0] - best[0] if len(alternatives) > 1 else (
                max(1.0, abs(best[0])) * 0.12
            )

            # Urgent requests with few viable alternatives are inserted first.
            priority = regret + 0.03 * urgency((p, d))
            priority += 1e-7 * np.random.random()

            if chosen is None or priority > chosen_regret:
                chosen = (p, d)
                chosen_route = best
                chosen_regret = priority
                chosen_value = best[0]
```

```
        if chosen is None:
            chosen = remaining[0]
            p, d = chosen
            if len(routes) < nv:
                chosen_route = (0.0, len(routes), [depot, p, d, depot], 0.0)
            else:
                rid = min(range(len(routes)), key=lambda r: evaluate(routes[r]))
                chosen_route = (
                    0.0, rid,
                    routes[rid][:-1] + [p, d, depot],
                    0.0,
                )

        _, rid, route, _ = chosen_route
        if rid == len(routes):
            routes.append(route)
        else:
            routes[rid] = route
        remaining.remove(chosen)

    return routes if routes else [[depot, depot]]
```

Listing 1: Example of found θ_{cons} - constructive function on PDPTW, the one found in case study paragraph.

```
def heuristic_function(
    current_solution: List[List[int]],
    distance_mat: np.ndarray,
    early: np.ndarray,
    late: np.ndarray,
    service: np.ndarray,
    capacity: float,
    demand: np.ndarray,
    pickup_to_delivery: List[List[int]],
) -> np.ndarray:
    """
    High score means that the pair is a promising destroy/reinsert candidate.

    The new mechanism is an adaptive removal-impact term: expensive local
    bridges are weighted more strongly when they are followed by a fragile,
    low-slack suffix, but the weight is normalized by route-wide penalty.
    """
    pairs = [
        (int(x[0]), int(x[1]))
        for x in pickup_to_delivery
        if len(x) >= 2
    ]
    scores = np.zeros(len(pairs), dtype=float)
    if not pairs:
        return scores

    D = np.asarray(distance_mat, dtype=float)
    E = np.asarray(early, dtype=float)
    L = np.asarray(late, dtype=float)
    S = np.asarray(service, dtype=float)
    W = np.asarray(demand, dtype=float)
    Q = max(1e-9, abs(float(capacity)))

    finite = D[np.isfinite(D) & (D > 0)]
    scale = max(1e-9, float(np.median(finite)) if finite.size else 1.0)

    info = {}
    route_penalty = {}
    route_length = {}

    for rid, route in enumerate(current_solution):
        if len(route) < 3:
            continue

        t = 0.0
        load = 0.0
        bad = 0.0
        length = 0.0
        events = []

        for pos in range(1, len(route) - 1):
            a, b, nxt = map(int, (route[pos - 1], route[pos], route[pos + 1]))
            edge = D[a, b]
            length += edge

            arrival = t + edge
            late_amt = max(0.0, arrival - L[b])
            start = max(arrival, E[b])
            slack = max(1e-6, L[b] - start)
            t = start + S[b]
```

```
            load += W[b]
            overload = max(0.0, load - Q)
            underload = max(0.0, -load)

            bad += (
                28000.0 * float(late_amt > 0)
                + 800.0 * late_amt
                + 26000.0 * float(overload > 0)
                + 1400.0 * overload
                + 2200.0 * underload
            )

            bypass = max(0.0, D[a, b] + D[b, nxt] - D[a, nxt])
            events.append((b, pos, late_amt, slack, load, bypass, a, nxt))

        ripple = 0.0
        for b, pos, late_amt, slack, load, bypass, a, nxt in reversed(events):
            ripple = 0.78 * ripple + 1.0 / slack + 5.0 * late_amt
            info[b] = (rid, pos, late_amt, slack, load, bypass, ripple, a, nxt)

        route_penalty[rid] = bad
        route_length[rid] = length

    max_bad = max(1.0, max(route_penalty.values(), default=0.0))
    med_len = max(scale, float(np.median(list(route_length.values()))))
                if route_length else scale)

    for k, (p, d) in enumerate(pairs):
        ip, id_ = info.get(p), info.get(d)

        if ip is None or id_ is None:
            scores[k] = 1e6
            continue

        value = 0.0
        route_ids = set()

        for x in (ip, id_):
            rid, pos, late_amt, slack, load, bypass, ripple, a, nxt = x
            route_ids.add(rid)

            pressure = max(0.0, abs(load) / Q - 0.4)
            route_bad = route_penalty[rid] / max_bad
            bridge = bypass / scale

            # Novel adaptive term: bridge cost is amplified only when the
            # downstream route is fragile, avoiding excessive geometric bias.
            removal_impact = bridge * (1.0 + min(10.0, ripple)) * (
                1.0 + 0.8 * route_bad
            )

            value += (
                28000.0 * float(late_amt > 0)
                + 800.0 * late_amt
                + 25.0 / slack
                + 15.0 * ripple
                + 900.0 * pressure
                + 42.0 * bridge
                + 700.0 * route_bad
                + 7.0 * route_length[rid] / med_len
                + 20.0 * removal_impact
            )

        if len(route_ids) > 1:
            value += 700.0
        if ip[1] >= id_[1]:
            value += 1600.0

        gap = abs(ip[1] - id_[1])
        value += 16.0 / (1.0 + gap)

        wp = max(1e-6, L[p] - E[p])
        wd = max(1e-6, L[d] - E[d])
        span = max(1e-6, L[d] - E[p])

        value += 32.0 / wp + 32.0 / wd
        value += 600.0 * max(
            0.0,
            (D[p, d] + S[p] + S[d] - span) / scale,
        )

        scores[k] = max(0.0, value)

    # Diversify primarily among similarly ranked neighborhoods.
    noise = max(1.0, float(np.mean(scores)))
    scores += 0.0013 * noise * np.random.random(len(scores))
```

Listing 2: Example of found θ_{ref} - refinement function on PDPTW, the one found in case study paragraph.

```python
import numpy as np
from typing import Dict, List, Tuple
def _f(x):
    try:
        y = float(x)
        return y if np.isfinite(y) else 0.0
    except Exception:
        return 0.0

def _arr(x, dtype=float):
    try:
        return np.asarray(x, dtype=dtype)
    except Exception:
        return np.asarray([], dtype=dtype)

def _log(x):
    return float(np.log1p(max(0.0, _f(x))))

def _ratio(a, b):
    return float(_f(a) / max(abs(_f(b)), 1e-12))

def _routes(sol, n):
    out = []
    for route in sol:
        try:
            r = np.asarray(route, dtype=int).reshape(-1)
        except Exception:
            continue
        r = r[(r >= 0) & (r < n)]
        if len(r):
            out.append(r)
    return out

def _edges(sol, coords):
    c = _arr(coords)
    vals = []
    for r in _routes(sol, len(c)):
        if len(r) > 1:
            vals.extend(np.linalg.norm(c[r[1:]] - c[r[:-1]], axis=1).tolist())
    return np.asarray(vals, dtype=float)

def _timing(sol, coords, early, late, service):
    c = _arr(coords)
    e = _arr(early)
    l = _arr(late)
    s = np.maximum(_arr(service), 0.0)
    n = min(len(c), len(e), len(l), len(s))
    waits, late_vals, slack_vals = [], [], []
    for r in _routes(sol, n):
        t = 0.0
        for u, v in zip(r[:-1], r[1:]):
            t += float(np.linalg.norm(c[v] - c[u]))
            arrival = t
            start = max(arrival, e[v])
            waits.append(max(0.0, e[v] - arrival))
            late_vals.append(max(0.0, start - l[v]))
            slack_vals.append(max(0.0, l[v] - start))
            t = start + s[v]
    return np.asarray(waits), np.asarray(late_vals), np.asarray(slack_vals)

def _penalties(tw, cp):
    a = max(0.0, _f(tw))
    b = max(0.0, _f(cp))
    return a, b, a + b

def feature_time_window_penalty_log(current_solution, coords, early, late, service, capacity, demand,
↪  pickup_to_delivery, total_cost, travel_distance, time_window_penalty, capacity_penalty):
    return _log(time_window_penalty)

def feature_capacity_penalty_log(current_solution, coords, early, late, service, capacity, demand,
↪  pickup_to_delivery, total_cost, travel_distance, time_window_penalty, capacity_penalty):
    return _log(capacity_penalty)

def feature_penalty_share(current_solution, coords, early, late, service, capacity, demand,
↪  pickup_to_delivery, total_cost, travel_distance, time_window_penalty, capacity_penalty):
    tw, cp, p = _penalties(time_window_penalty, capacity_penalty)
    return _ratio(p, abs(_f(travel_distance)) + p)

def feature_edge_tail_ratio(current_solution, coords, early, late, service, capacity, demand,
↪  pickup_to_delivery, total_cost, travel_distance, time_window_penalty, capacity_penalty):
    x = _edges(current_solution, coords)
    x = x[x > 1e-12]
    return _ratio(np.percentile(x, 90), np.median(x)) if len(x) else 0.0

def feature_lateness_concentration(current_solution, coords, early, late, service, capacity, demand,
↪  pickup_to_delivery, total_cost, travel_distance, time_window_penalty, capacity_penalty):
    _, late_vals, _ = _timing(current_solution, coords, early, late, service)
    return _ratio(np.max(late_vals) if len(late_vals) else 0.0, np.sum(late_vals))

def feature_slack_cv(current_solution, coords, early, late, service, capacity, demand,
↪  pickup_to_delivery, total_cost, travel_distance, time_window_penalty, capacity_penalty):
    _, _, slack = _timing(current_solution, coords, early, late, service)
```

```
    return _ratio(np.std(slack), np.mean(slack)) if len(slack) else 0.0

feature_func_list = [
    feature_penalty_share,
    feature_time_window_penalty_log,
    feature_capacity_penalty_log,
    feature_lateness_concentration,
    feature_slack_cv,
    feature_edge_tail_ratio,
]
```

Listing 3: Example of feature functions generated by analyst for PDPTW