PARETOEVO: PARETO-GUIDED EVOLUTIONARY TREE SEARCH FOR AUTOMATIC HEURISTIC DESIGN WITH LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Combinatorial optimization (CO) generally relies on human-designed heuristics to obtain high-quality solutions when exact search is computationally prohibitive. Large language model (LLM)-based automatic heuristic design (AHD) reduces this reliance by iteratively generating, evaluating, and revising executable heuristics. However, evaluating heuristics primarily by average solution quality can overlook execution cost and difficult-instance performance. Moreover, attained objective values do not necessarily reveal a heuristic's potential for further evolution. To address these limitations, we propose PARETOEVO, a Pareto-guided evolutionary tree-search framework that combines objective trade-offs with evidence of search potential. We develop Pareto-dominance Monte Carlo tree search (PD-MCTS) to allocate evaluations through vector-valued subtree returns over average quality, execution time, and worst-instance quality. To complement descendant-based evidence with earlier indications of search potential, we propose adaptive productivity control (APC) to grant additional expansions when productive creation empirically predicts productive subsequent search. To adapt revisions to search experience, reflection-conditioned Actor and Reflector LLM agents use evaluated transitions to update guidance for subsequent heuristic generation. Experiments across diverse CO tasks demonstrate improvements in solution quality, execution efficiency, and worst-instance performance over LLM-based AHD baselines, alongside faster convergence and higher final Pareto-front quality.

1 INTRODUCTION

Manually designed heuristics can provide high-quality solutions to combinatorial optimization (CO) problems within limited runtime (Bengio et al., 2021). However, their development remains labor-intensive and dependent on problem-specific expertise. To alleviate this burden, automatic heuristic design (AHD) automates algorithm configuration (Hutter et al., 2009; López-Ibáñez et al., 2016) and the search over human-specified components and operators (Burke et al., 2013). More recently, large language model (LLM)-based methods, including FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), LLaMEA (van Stein & Bäck, 2025), and HSEvo (Dat et al., 2025), have broadened this search by generating executable programs. In this process of heuristic evolution, LLMs transform selected parent heuristics into new candidates, whose evaluations guide subsequent search.

However, heuristic evolution driven by average solution quality can favor computationally expensive heuristics while obscuring poor performance on difficult instances. To account for execution efficiency, MEoH (Yao et al., 2025) maintains a Pareto front over quality and runtime, while MPaGE (Ha et al., 2026) introduces Pareto-grid guidance to preserve diverse heuristics along these trade-offs. Although these methods retain multiple performance trade-offs, attained objective values alone do not establish a heuristic's value for further evolution. Consequently, guiding heuristic evolution under a fixed evaluation budget requires considering both current performance and further search potential.

To account for this potential, Monte Carlo tree search preserves parent–child relations and propagates descendant evaluations to guide exploration (Silver et al., 2016; Segler et al., 2018; Wang et al.,

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.

2020; Hao et al., 2023; Zheng et al., 2025). In heuristic evolution, MCTS-AHD uses this structure to refine temporarily underperforming heuristics and guide LLM generation along tree paths. Nevertheless, its branch selection remains driven by scalar solution quality. Moreover, evidence from a candidate's descendants becomes available only after that candidate has been expanded. The generation step itself, however, can already indicate progress when the candidate remains competitive with the population or improves upon its source. Our central insight is therefore to guide heuristic evolution with complementary evidence: multi-objective subtree outcomes and productivity observed when each candidate is created.

To address these limitations, we propose PARETOEVO, a Pareto-guided evolutionary tree-search framework that integrates objective trade-offs and generation productivity to design high-quality, efficient, and robust heuristics. Specifically, we develop Pareto-dominance Monte Carlo tree search (PD-MCTS) to integrate average quality, runtime, and worst-instance quality into source selection. Building on multi-objective tree search (Wang & Sebag, 2013; Perez et al., 2015; Painter et al., 2020), PD-MCTS uses vector-valued subtree returns to allocate evaluations while retaining these objective trade-offs. To exploit early evidence of search potential, we further introduce adaptive productivity control (APC). APC estimates whether productive creation steps predict productive first expansions and grants supported nodes one additional expansion opportunity. Finally, to adapt generation to search experience, we develop reflection-conditioned Actor and Reflector LLM agents. The Actor generates executable offspring through mutation or crossover of selected heuristics, guided by reflections for the corresponding evolutionary operator. The Reflector updates this guidance from prior reflections and evaluated transitions to condition subsequent Actor generations. Thus, evaluation evidence guides both source selection and revision throughout heuristic evolution.

Our main contributions are summarized as follows. **(1)** We formulate AHD over average quality, runtime, and worst-instance quality, and develop PD-MCTS to allocate evaluations through Pareto-aware comparisons of vector-valued subtree returns. **(2)** We introduce APC to assess the predictive value of productive generation steps and use this evidence to allocate additional expansions. **(3)** We develop reflection-conditioned Actor and Reflector LLM agents that continually adapt evolutionary operator guidance to evaluated heuristic changes, objective outcomes, and productivity. **(4)** Experiments across diverse CO tasks demonstrate improvements in solution quality, execution efficiency, and worst-instance performance over LLM-based AHD baselines, with faster Pareto-front convergence and higher final front quality.

2 PRELIMINARIES

2.1 LLM-BASED AUTOMATIC HEURISTIC DESIGN

AHD seeks executable heuristics for solving instances of a CO problem (Burke et al., 2013). Such a problem is specified by $P = (\mathcal{X}, \mathcal{Z}, \ell)$, where $\mathcal{X}$ is the instance space, $\mathcal{Z}(x)$ contains feasible solutions, and $\ell(x, z)$ is the cost of solution z. LLM-based AHD uses LLMs to generate candidates from a heuristic space $\mathcal{H}$ within a fixed solver framework Φ (Zheng et al., 2025; Gungordu et al., 2026). For each candidate h, this framework returns a feasible solution $\Phi[h](x) \in \mathcal{Z}(x)$ for every instance $x \in \mathcal{X}$. To compare candidates, we evaluate their solution quality $J(h, x)$, using $J(h, x) = -\ell(x, \Phi[h](x))$ for cost-minimization tasks. The scalar AHD objective maximizes expected quality over an evaluation distribution $\mathcal{D}$:

$$Q(h) := \mathbb{E}_{x\sim\mathcal{D}}[J(h, x)], \qquad h^\star \in \arg\max_{h\in\mathcal{H}} Q(h). \tag{1}$$

2.2 MULTI-OBJECTIVE OPTIMIZATION

We consider the multi-objective problem $\max_{s\in\mathcal{S}} \boldsymbol{y}(s)$, where $\mathcal{S}$ is a feasible candidate set and $\boldsymbol{y} : \mathcal{S} \to \mathbb{R}^K$ maps s to $\boldsymbol{y}(s) = (y_1(s), \ldots, y_K(s))^\top$. All objectives are maximized, with trade-offs characterized by Pareto optimality.

Definition 1 (Pareto dominance). For $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^K$, weak dominance $\boldsymbol{a} \succeq \boldsymbol{b}$ means $a_k \geq b_k$ for all k. Pareto dominance $\boldsymbol{a} \succ \boldsymbol{b}$ additionally requires $a_k > b_k$ for some k.

Definition 2 (Pareto optimality and Pareto front). A candidate is Pareto optimal if and only if it belongs to the non-dominated set $\mathrm{nd}_{\boldsymbol{y}}(\mathcal{S}) := \{s \in \mathcal{S} \mid \nexists\, s' \in \mathcal{S} : \boldsymbol{y}(s') \succ \boldsymbol{y}(s)\}$. The objective

vectors of candidates in this non-dominated set form the Pareto front, $\mathcal{F}_{\boldsymbol{y}}(\mathcal{S}) := \{\boldsymbol{y}(s) \mid s \in \mathrm{nd}_{\boldsymbol{y}}(\mathcal{S})\}$.

Definition 3 (Crowding distance). For a finite non-dominated set $\mathcal{S}$, let Δ_k be the range of objective k and let s^{-k}, s^{+k} be the neighbors of s in its sorted order. The crowding distance of an interior solution is $\mathrm{cd}_{\boldsymbol{y}}(s;\mathcal{S}) = \sum_{k:\Delta_k>0}[y_k(s^{+k}) - y_k(s^{-k})]/\Delta_k$, with infinite distance assigned to boundary solutions (Deb et al., 2002). Larger distances favor sparsely represented trade-offs.

2.3 Monte Carlo tree search

Monte Carlo tree search (MCTS) grows a search tree by selecting and expanding nodes, then using simulation returns to guide subsequent selections (Kocsis & Szepesvári, 2006; Browne et al., 2012). Backpropagation updates the visit count n_v and mean return $\bar{r}_v$ at each visited node v. Upper Confidence Bounds applied to Trees (UCT) balances estimated returns against exploration to select among the children $\mathcal{C}(p)$ of a fully expanded parent p:

$$\mathrm{Select}_{\mathrm{UCT}}(p) \in \arg\max_{v\in\mathcal{C}(p)} \left(\bar{r}_v + \lambda\sqrt{\frac{\log n_p}{n_v}}\right), \tag{2}$$

where $\lambda \geq 0$ controls exploration of less visited branches.

3 Methodology

3.1 Multi-objective automatic heuristic design

The scalar AHD objective in Equation 1 measures average quality but can overlook high computational cost and poor performance on difficult instances. We therefore formulate multi-objective AHD as the search for high-quality heuristics that remain efficient and robust under a fixed evaluation budget. Following Section 2.2, we set $K = 3$ and specify $\boldsymbol{y} : \mathcal{H} \to \mathbb{R}^3$. Let $T_{\mathrm{run}}(h, x)$ be the accumulated heuristic-call time on x. We define

$$E(h) = \mathbb{E}_{x\sim\mathcal{D}}[T_{\mathrm{run}}(h, x)], \qquad W(h) = \min_{x\in\mathrm{supp}(\mathcal{D})} J(h, x), \tag{3}$$

where $E(h)$ is expected running time and $W(h)$ is worst-instance quality. The objective vector is

$$\boldsymbol{y}(h) = \big(Q(h), -E(h), W(h)\big)^\top, \tag{4}$$

with all three coordinates maximized. Under a budget of B completed heuristic evaluations, our goal is to identify a diverse set of non-dominated heuristics that achieve high quality while remaining efficient and robust on hard instances. Appendix B.1 defines the empirical evaluation distribution, objective directions, runtime aggregation, and budget accounting.

3.2 Method overview

We organize heuristic evolution as tree search over evaluated heuristics and their source–offspring relations. To guide further evolution, PD-MCTS combines multi-objective subtree returns with count-based exploration, while APC uses productive creation steps to allocate additional expansions. As shown in Figure 1, the Actor generates offspring through mutation or crossover of selected heuristics. Evaluation then updates subtree returns and APC evidence. The Reflector uses these transitions to update evolutionary operator guidance for subsequent Actor generations. Algorithm 1 in Appendix B.2 gives the complete evolution loop.

3.3 Pareto-dominance MCTS

PD-MCTS guides heuristic evolution through a rooted tree, allocating evaluations using multi-objective subtree evidence and count-based exploration. The virtual root v_{r} has no associated heuristic and contains the initial heuristics as its children. Each non-root node v stores an executable heuristic h_v. Edges between heuristic nodes record generation steps from selected sources. For a parent p, let $\mathcal{C}(p)$ denote its children. At iteration t, PD-MCTS selects a source node v_t^{src} and attaches one child to it.

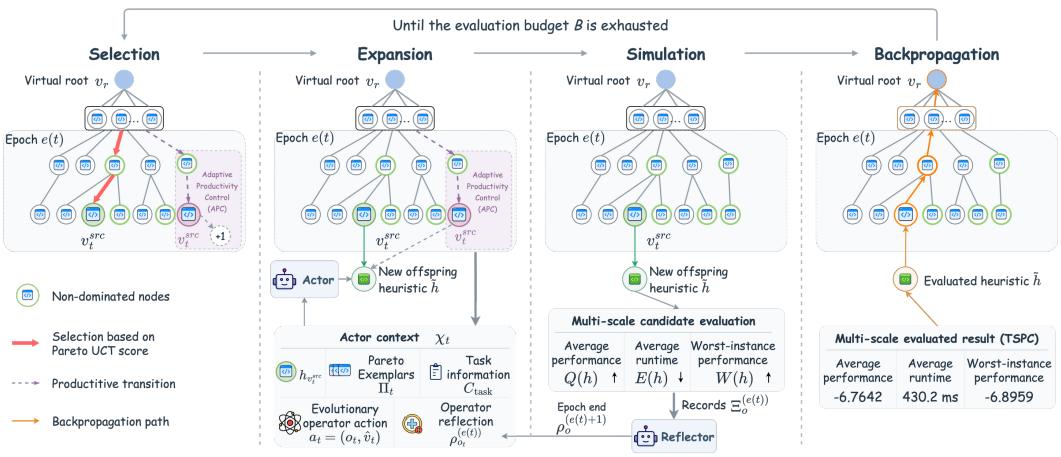

Figure 1: Heuristic evolution in PARETOEVO. The four columns show one iteration of selection, expansion, simulation, and backpropagation. Red arrows trace Pareto-UCT selection, while purple dashed links mark productive transitions used as APC evidence. The $+1$ marker denotes one additional expansion granted by an APC promotion. Orange arrows show objective vectors backpropagated to ancestral nodes. The lower feedback path returns reflections for each evolutionary operator to the Actor context for the next epoch, and the upper loop repeats the cycle until budget B is exhausted. Sections 3.4 and 3.5 define the epoch and agent-context notation.

Each node receives $b_0 \in \mathbb{N}_+$ expansion opportunities at creation. We restrict selection at parent p to the viable children $\mathcal{C}_t^+(p) \subseteq \mathcal{C}(p)$. A child is *viable* at iteration t if it or any of its descendants has a remaining expansion opportunity, allowing search to continue within that branch.

Pareto-UCT score. Building on multi-objective MCTS (Wang & Sebag, 2012; Perez et al., 2015), we construct a vector-valued UCT score that ranks sibling subtrees separately along each objective and adds count-based exploration. Let $\mathcal{T}(v)$ contain the non-root nodes in the subtree rooted at v, and let $N_v = |\mathcal{T}(v)|$ denote the number of evaluated heuristics in this subtree. PD-MCTS represents its branch-level evidence by the mean return vector

$$\bar{\boldsymbol{y}}_v := \frac{1}{N_v} \sum_{v' \in \mathcal{T}(v)} \boldsymbol{y}(h_{v'}). \tag{5}$$

PD-MCTS therefore evaluates each child by the returns observed throughout its subtree rather than by the heuristic stored at that child. To put objective values on a common scale with the exploration term, we first rescale subtree means by local ranks over $\mathcal{C}_t^+(p)$. The three percentile scores form $\mathcal{R}_{p,t}(\bar{\boldsymbol{y}}_v) \in [0,1]^3$, which serves as the exploitation term. Appendix B.1 gives the exact average-rank mapping and its boundary cases. We then define

$$\boldsymbol{u}_p(v) = \mathcal{R}_{p,t}(\bar{\boldsymbol{y}}_v) + \lambda \sqrt{\frac{\log(N_p+1)}{N_v+1}} \mathbf{1}, \qquad v \in \mathcal{C}_t^+(p), \tag{6}$$

where $\lambda \geq 0$ is the exploration constant in Equation 2 and $\mathbf{1} \in \mathbb{R}^3$ is the all-ones vector. Following UCT (Kocsis & Szepesvári, 2006), the exploration term assigns larger bonuses to less-explored subtrees and encourages their reconsideration as evaluations accumulate under the same parent.

Selection. When $\mathcal{C}_t^+(p) \neq \varnothing$, we form the non-dominated set $\mathcal{F}_{p,t} = \mathrm{nd}_{\boldsymbol{u}_p}(\mathcal{C}_t^+(p))$ under the Pareto-UCT scores. To retain diversity, we sample uniformly from the children in $\mathcal{F}_{p,t}$ with maximal crowding distance:

$$\mathrm{Select}_{\mathrm{PD}}(p) \sim \mathrm{Unif}\left(\arg\max_{v \in \mathcal{F}_{p,t}} \mathrm{cd}_{\boldsymbol{u}_p}(v; \mathcal{F}_{p,t}) \right). \tag{7}$$

Starting at v_{r}, we apply $p \leftarrow \mathrm{Select}_{\mathrm{PD}}(p)$ while $\mathcal{C}_t^+(p) \neq \varnothing$. The recursion stops at the first node with no viable child and at least one remaining opportunity. Thus, the source node v_t^{src} is the recursion endpoint, which may be internal or a leaf node.

Expansion. PD-MCTS expands the selected source v_t^{src} by preparing a candidate-generation action a_t. This action specifies an evolutionary operator and, when required, a second parent, as defined in Section 3.5. Conditioned on this expansion context, the Actor LLM agent proposes an executable candidate heuristic $\tilde{h}_t$.

Simulation. PD-MCTS evaluates $\tilde{h}_t$ under the evaluation dataset $\mathcal{D}$ to obtain the objective vector $\boldsymbol{y}(h_{v_t})$. The evaluated heuristic is attached as a new child v_t of v_t^{src}, with $h_{v_t} \leftarrow \tilde{h}_t$, $N_{v_t} \leftarrow 1$, and $\bar{\boldsymbol{y}}_{v_t} \leftarrow \boldsymbol{y}(h_{v_t})$. PD-MCTS maintains the elite population of non-dominated heuristics by updating $\mathcal{A}_t = \mathrm{nd}_{\boldsymbol{y}}(\mathcal{A}_{t-1} \cup \{h_{v_t}\})$, with $\mathcal{A}_0$ the non-dominated set of the initial heuristics.

Backpropagation. To propagate the new candidate's objective vector, PD-MCTS updates every ancestor v from v_t^{src} to v_{r} by setting $N_v \leftarrow N_v + 1$ and then $\bar{\boldsymbol{y}}_v \leftarrow \bar{\boldsymbol{y}}_v + (\boldsymbol{y}(h_{v_t}) - \bar{\boldsymbol{y}}_v)/N_v$. This preserves the subtree mean in Equation 5.

3.4 Adaptive productivity control

PD-MCTS assesses search potential through evaluated descendants. To use evidence available at node creation, we introduce APC, which assesses whether productive creation predicts productive first expansion. When observed transitions support this relationship, APC grants an eligible node one additional expansion, termed a *promotion*.

Transition productivity. To keep productivity labels comparable, APC groups post-initialization iterations into epochs of length L and fixes a reference set within each epoch. At iteration t, epoch $e(t) = \lfloor (t-1)/L \rfloor$ spans $e(t)L + 1, \ldots, (e(t)+1)L$, with reference set $\mathcal{A}^{(e(t))} := \mathcal{A}_{e(t)L}$. The transition from v_t^{src} to its evaluated child v_t receives the label

$$Y_t = 1 \quad \Longleftrightarrow \quad \Big(\nexists\, h \in \mathcal{A}^{(e(t))} : \boldsymbol{y}(h) \succeq \boldsymbol{y}(h_{v_t})\Big) \\ \vee \left(\boldsymbol{y}(h_{v_t}) \succ \boldsymbol{y}(h_{v_t^{\mathrm{src}}})\right). \tag{8}$$

All other transitions receive $Y_t = 0$. The two conditions capture population competitiveness and improvement over the source, respectively. The resulting creation label is attached to the child as evidence for node valuation.

Node value. To assess whether productive creation predicts productive first expansion, APC pairs the creation label Y_m of each non-initial node v_m, created at iteration m, with the productivity Y_m^+ of its first expansion. For creation class $i \in \{0, 1\}$, let $\vartheta_i = \Pr(Y_m^+ = 1 \mid Y_m = i)$. We model uncertainty in these success probabilities using independent $\mathrm{Beta}(\alpha_0, \beta_0)$ priors. Under a Bernoulli likelihood, productive and non-productive evidence weights $S_i(t), F_i(t)$ give the posterior $\vartheta_i \mid S_i(t), F_i(t) \sim \mathrm{Beta}(\alpha_0 + S_i(t), \beta_0 + F_i(t))$. These weights accumulate first-expansion outcomes observed from initialization through iteration t, including those of nodes created in earlier epochs. At each epoch end, we downweight the accumulated evidence after node valuation and retain it for subsequent epochs. To compare the creation classes, APC evaluates the posterior support score under these two Beta distributions:

$$q_t = \Pr\big(\vartheta_1 > \vartheta_0 + \delta \,\big|\, \{S_i(t), F_i(t)\}_{i=0}^{1}\big). \tag{9}$$

Here $\delta \geq 0$ specifies the required productivity gap. We require $S_i(t) + F_i(t) \geq 3$ for both creation classes before assessing node value. A node v_t is then valuable if $Y_t = 1$ and $q_t \geq \eta$, where $\eta \in [0, 1]$ is the posterior support threshold. Appendix B.3 gives the posterior derivation and evidence collection procedure.

Promotion selection. At the start of iteration t, the pending set $\mathcal{P}_t$ contains unpromoted valuable nodes v_m from the current epoch, $e(t)L < m < t$, with fixed creation-time scores q_m. To preserve ordinary tree exploration, we cap cumulative APC expansions and require a minimum number of ordinary PD-MCTS expansions between them, as specified in Appendix B.3. When both limits are satisfied and $\mathcal{P}_t \neq \varnothing$, APC selects a node for one additional expansion:

$$v_t^{\mathrm{src}} = v_t^{\mathrm{APC}} \sim \mathrm{Unif}\left(\arg\max_{v_m \in \mathcal{P}_t} q_m\right). \tag{10}$$

Otherwise, v_t^{src} is the endpoint of the PD-MCTS traversal in Section 3.3. Promoted nodes leave $\mathcal{P}_t$, which is cleared at epoch boundaries.

3.5 REFLECTION-CONDITIONED LLM AGENTS

PD-MCTS and APC determine which heuristic to evolve further. To adapt its revision to search experience, we develop reflection-conditioned Actor and Reflector agents. For each evolutionary operator $o \in \mathcal{O} = \{\text{mut}, \text{cross}\}$, $\rho_o^{(e)}$ denotes the reflection held fixed throughout epoch e, with epochs defined in Section 3.4.

Actor. At iteration t, the Actor uses action $a_t = (o_t, \hat{v}_t)$, where $o_t \in \mathcal{O}$ is the selected operator and $\hat{v}_t$ is the optional second parent. Mutation revises the source alone with probability $\gamma \in [0, 1]$. Otherwise, crossover selects a second parent from two evaluated nodes with code distinct from the source, prioritizing lower Pareto rank and then larger crowding distance. The Actor uses $\rho_{o_t}^{(e(t))}$, the reflection for operator o_t in epoch $e(t)$, to construct its generation context:

$$\chi_t = \left(C_{\text{task}}, h_{v_t^{\text{src}}}, h_{\hat{v}_t}, \Pi_t, \rho_{o_t}^{(e(t))}\right), \qquad \tilde{h}_t \sim \pi_A(\cdot \mid \chi_t, a_t), \tag{11}$$

where C_{task} specifies the task interface, Π_t contains representative heuristics from $\mathcal{A}_{t-1}$, and π_A is the Actor's text-generation policy. Under mutation, $h_{\hat{v}_t} = \varnothing$. The source anchors the revision, Pareto exemplars provide competitive code, and reflection supplies guidance from previously evaluated changes.

Reflector. To accumulate experience across epochs, the Reflector updates its previous guidance using evaluated transitions. Let $\Xi_o^{(e)}$ collect the heuristic changes, source-relative changes in Q, E, and W, and productivity labels produced by evolutionary operator o during epoch e. At the end of epoch e, its text-generation policy π_R updates each operator's reflection for epoch $e + 1$ as

$$\rho_o^{(e+1)} \sim \pi_R\left(\cdot \mid \rho_o^{(e)}, \Xi_o^{(e)}\right), \qquad o \in \mathcal{O}. \tag{12}$$

Through Equation 11, the updated reflection guides subsequent generations, whose evaluations inform the next update. This loop adapts the guidance driving heuristic evolution while keeping the pretrained LLM parameters fixed. Appendix B.1 specifies the population used for parent ranking.

4 EXPERIMENTS

We evaluate PARETOEVO on the traveling salesman problem (TSP), 0–1 knapsack problem (KP), capacitated vehicle routing problem (CVRP), multidimensional knapsack problem (MKP), and online/offline bin packing problem (BPP). We use step-by-step construction, ant colony optimization (ACO), and guided local search (GLS) frameworks. Appendix C specifies tasks, datasets, and interfaces. Appendix D.2 reports additional GLS/online BPP results, significance tests, and TSPLIB comparisons with `GPT-4o-mini`, and Appendix D.3 gives `GPT-5-NANO` results.

Experimental Settings. In this section, PD-MCTS uses 20 initial nodes, $b_0 = 6$, and $\lambda = 0.1$. APC uses $L = 20$, $\alpha_0 = \beta_0 = 1$, $\delta = 0.05$, and $\eta = 0.8$. We set $\gamma = 0.6$ and use `GPT-4o-mini` at temperature 1.0 for both agents. We report results over averaging five independent runs with $B = 1{,}000$ heuristic evaluations per run.

Baselines. Manual baselines include Greedy Construct for TSP/KP, standard ACO (Dorigo et al., 2006), and Best Fit/First Fit for online BPP (Johnson et al., 1974). Neural baselines are POMO for constructive TSP (TSPC) and KP (Kwon et al., 2020) and DeepACO for ACO (Ye et al., 2023). LLM-based baselines share the task interfaces: EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), HSEvo (Dat et al., 2025), and MCTS-AHD (Zheng et al., 2025). For multi-objective comparisons on TSPC, we adapt MEoH (Yao et al., 2025) and MPaGE (Ha et al., 2026) to optimize the same Q, E, W objectives as PARETOEVO. Baseline sources and adaptations are described in Appendix E.

4.1 PERFORMANCE ANALYSIS

Tables 1–2 compare PARETOEVO with hand-designed, neural, and LLM-based baselines. Gap% measures average solution quality relative to the reference, while Time/s reports mean heuristic-call time per instance. Worst is the mean per-run worst-instance cost or profit in the task's original units.

Table 1: Comparison of constructive TSP and KP heuristics on 1,000 instances per test set, using LKH-3 (Helsgaun, 2017) and OR-Tools (Perron & Furnon, 2025) reference solutions, following MCTS-AHD (Zheng et al., 2025). LLM-based entries average five independent runs. Each pair shows Gap% (average reference gap) above Time/s (mean heuristic-call time per instance in seconds), alongside Worst (mean per-run maximum TSP cost/minimum KP profit). Green shading, blue bold text, and underlining mark the best LLM-based Gap%, Time/s, and Worst, respectively. Arrows indicate optimization directions, and dashes denote unavailable values.

Method	TSP constructive						KP constructive					
	N=50		N=100		N=200		N=50		N=100		N=200	
	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↑	Gap%↓ Time/s↓	Worst↑	Gap%↓ Time/s↓	Worst↑
Greedy Construct	+22.62% –	–	+24.94% –	–	+26.29% –	–	+0.26% –	–	+0.12% –	–	+0.09% –	–
POMO	+0.39% –	–	+3.01% –	–	+20.45% –	–	+2.12% –	–	+1.48% –	–	+0.09% –	–
EoH	+13.87% **0.0045**	7.752	+15.76% **0.026**	10.459	+17.72% **0.171**	14.329	+0.23% **0.000253**	14.666	+0.10% **0.000493**	32.848	+0.10% **0.000792**	47.596
ReEvo	+11.88% 0.364	7.663	+13.32% 5.00	10.241	+14.93% 76.5	13.584	+0.21% 0.0015	14.666	+0.10% 0.0038	32.842	+0.08% 0.0086	47.596
HSEvo	+11.08% 0.273	7.679	+12.94% 3.70	10.309	+15.25% 82.0	13.801	+0.22% 0.000591	14.666	+0.10% 0.0011	32.848	+0.09% 0.0016	47.596
MCTS-AHD	+11.52% 0.0095	7.720	+13.38% 0.068	10.256	+15.36% 0.520	13.904	+0.21% 0.0011	14.666	+0.10% 0.0043	32.848	+0.08% 0.013	47.596
ParetoEvo	**+11.05%** 0.0082	7.588	**+12.86%** 0.034	10.114	**+14.91%** 0.428	13.566	**+0.16%** 0.000530	14.649	**+0.08%** 0.0010	32.836	**+0.08%** 0.0014	47.596

Table 2: Comparison of ACO heuristics for TSP, CVRP, MKP, and offline BPP on eight test sets of 64 instances. Gap% uses the best listed mean objective as reference. Worst averages per-run maximum costs/bin counts or minimum MKP profits. Time/s, layout, and highlighting follow Table 1.

Method	TSP ACO				CVRP ACO				MKP ACO				Offline BPP			
	N=50		N=100		N=50		N=100		N=100		N=200		N=500		N=1000	
	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↑	Gap%↓ Time/s↓	Worst↑	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓
ACO	+4.47% –	–	+9.18% –	–	+27.77% –	–	+25.76% –	–	+2.25% –	–	+4.13% –	–	+2.81% –	–	+3.15% –	–
DeepACO	+1.87% –	–	+1.06% –	–	0.00% –	–	0.00% –	–	+0.72% –	–	+1.02% –	–	0.00% –	–	0.00% –	–
EoH	+1.76% **0.000315**	6.396	+0.48% **0.000514**	8.798	+4.75% **0.0022**	10.533	+4.94% 0.0075	17.325	+1.02% **0.0000627**	7.757	+1.89% **0.0000824**	10.434	+1.38% 0.389	213.667	+1.67% 1.58	423.333
ReEvo	+1.78% 0.142	6.415	+1.19% 0.267	8.991	+7.05% 0.353	10.843	+9.85% 0.677	18.132	+0.05% 0.286	7.743	+0.13% 0.077	10.607	+2.59% 2.79	216.667	+2.95% 10.2	429.667
HSEvo	+1.97% 0.0013	6.446	+1.07% 0.0043	8.887	+6.21% 0.020	10.542	+9.11% 0.086	17.917	+0.14% 0.000173	7.853	+0.11% 0.000192	10.615	+1.85% 0.856	215.000	+2.15% 3.45	426.000
MCTS-AHD	+1.12% 5.97	6.299	+0.53% 34.1	8.818	**+3.20%** 0.0056	10.309	+5.94% 0.021	17.672	+0.10% 0.143	7.677	0.00% 0.258	10.599	+0.22% 0.389	211.333	+0.26% 1.56	417.667
ParetoEvo	**0.00%** 0.521	6.239	**0.00%** 0.286	8.787	+4.66% 0.0054	10.471	**+4.58%** **0.0065**	17.056	**0.00%** 0.0000942	7.709	**0.00%** 0.0000936	10.592	**+0.02%** **0.0045**	211.000	**0.00%** **0.017**	415.000

Step-by-Step Construction Framework. As shown in Table 1, ParetoEvo achieves the lowest average gap among LLM-based methods on all six test sets. It also outperforms Greedy Construct throughout and POMO on KP and on TSP at $N = 200$. For TSP, the gains in average quality are accompanied by the lowest worst-instance cost among LLM-based methods at every size. Moreover, these quality improvements over ReEvo and HSEvo come with lower execution cost, with a 179–192-fold speedup at $N = 200$. On KP, efficiency remains a useful distinction even when solution quality is comparable. At $N = 200$, ParetoEvo matches ReEvo and MCTS-AHD at a 0.08% average gap and 47.596 worst-instance profit, while executing 6.1 and 9.3 times faster, respectively.

Ant Colony Optimization Framework. The gains extend to heuristic guidance within ACO. In Table 2, ParetoEvo achieves the lowest average gap among LLM-based methods on seven of the

eight test sets, outperforming standard ACO throughout and DeepACO on TSP and MKP. On offline BPP, it improves all three objectives over every LLM baseline at both sizes. At $N = 1{,}000$, it produces the lowest average bin count and a worst-instance count of 415, while executing approximately 92 times faster than the fastest LLM baseline.

The CVRP results further show that the relative advantage grows with instance size. From $N = 50$ to 100, PARETOEVO's average gap decreases from 4.66% to 4.58%, whereas all LLM baseline gaps increase. Its runtime also grows more slowly, by 1.2-fold compared with 1.9–4.3-fold for the baselines. Consequently, the quality–runtime trade-off with MCTS-AHD at $N = 50$ becomes an improvement in all three objectives over every LLM baseline at $N = 100$. Across all 14 construction and ACO test sets, no LLM baseline dominates PARETOEVO.

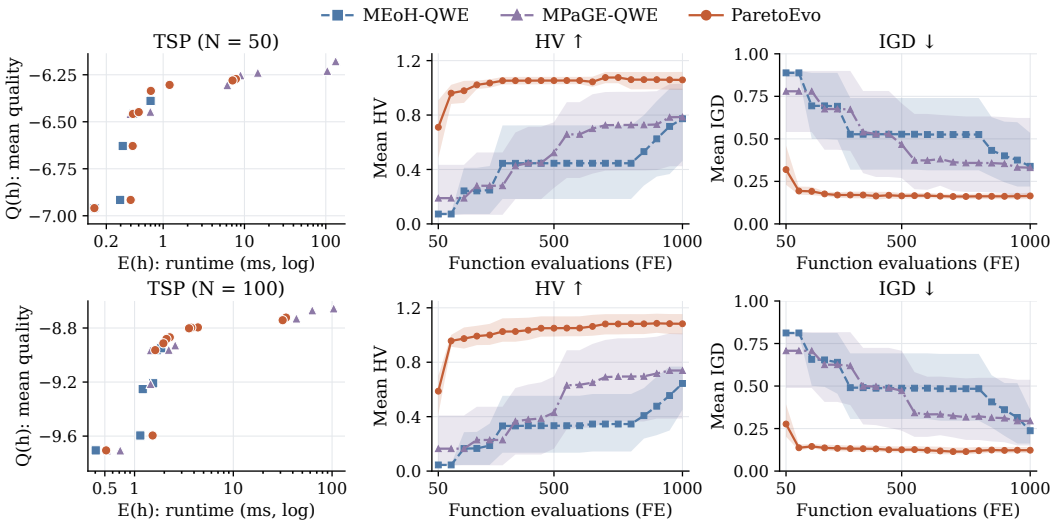

Figure 2: Three-objective heuristic evolution on constructive TSP ($N = 50$ top, $N = 100$ bottom). All methods optimize Q, E, W. Columns show Q–E fronts, three-objective HV, and IGD. Appendix D.1 defines the indicators.

Heuristic evolution and convergence. To examine how quickly PARETOEVO discovers these trade-offs, we compare its evolving Pareto fronts with those of MEoH and MPaGE on constructive TSP. Figure 2 shows higher mean hypervolume (HV) and lower mean inverted generational distance (IGD) throughout search on both sizes. After only 100 evaluations, PARETOEVO's HV already exceeds both baselines' values after 1,000 evaluations. This early advantage persists to the final front, where HV is 34.7% and 46.5% higher than the strongest baseline for $N = 50$ and 100, respectively, and IGD is 50.2% and 48.3% lower than the lowest baseline IGD. The two-objective comparison in Appendix D.2.6 also shows higher final HV and lower IGD for PARETOEVO *w/o* Robustness Objective (W) against the original Q, E formulations of MEoH and MPaGE.

4.2 ABLATION STUDIES

We ablate the objectives, branch-selection rule, APC, and reflection on constructive TSP and ACO-based CVRP in Table 3, holding the other components and evaluation budget fixed. Appendix D.2.7 examines five hyperparameters on TSP, with several variants retaining similar average quality.

Efficiency and robustness objectives. Removing E increases runtime on both tasks, but the extra cost does not consistently improve quality. On TSP, the average gap falls by 0.07 percentage points and worst-instance cost from 10.114 to 10.032, while runtime grows 9.4-fold. On CVRP, runtime doubles while both quality measures deteriorate. In contrast, removing W produces faster heuristics but worsens average and worst-instance quality on both tasks. For example, CVRP runtime falls from 5.4 to 1.3 ms, while the average gap rises from 4.66% to 6.20% and worst-instance cost from 10.471 to 10.794. Thus, guiding evolution by difficult-instance performance also benefits average quality in these two tasks.

Table 3: Ablations on constructive TSP ($N = 100$) and ACO-based CVRP ($N = 50$). Gap% reports the average reference gap, Time/ms reports mean heuristic-call time per instance in milliseconds, and Worst reports the mean per-run maximum cost. Highlighting follows Table 1.

Method	*TSP* ($N = 100$)			*CVRP* ($N = 50$)		
	Gap%↓	Time/ms↓	Worst↓	Gap%↓	Time/ms↓	Worst↓
ParetoEvo (Full)	12.86%	34	10.114	**4.66%**	5.4	<u>10.471</u>
Scalar-UCT (Q, E, W)	15.55%	33	10.493	10.33%	7.9	11.106
w/o Efficiency Objective (E)	**12.79%**	319	<u>10.032</u>	5.01%	10.9	10.495
w/o Robustness Objective (W)	14.95%	**31**	10.827	6.20%	**1.3**	10.794
w/o APC	16.41%	109	10.682	6.22%	7.6	10.523
w/o Reflection	16.44%	64	10.851	6.66%	6.2	10.594

Pareto-based branch selection. We next replace Pareto branch selection with rank-normalized Scalar-UCT, which aggregates Q, E, W into an equally weighted score, as defined in Appendix B.1. Despite using all three objectives, Scalar-UCT substantially degrades solution quality: average gaps increase by 2.69 and 5.67 percentage points on TSP and CVRP, respectively. Worst-instance quality also deteriorates, with little runtime change on TSP and higher runtime on CVRP. Preserving objective trade-offs therefore improves heuristic quality over this fixed scalar ranking.

Productivity-guided expansion. Removing APC increases average gaps by 3.55 percentage points on TSP and 1.56 on CVRP, while also worsening runtime and worst-instance quality. These losses support using productive creation alongside descendant evaluations to identify heuristics for further evolution.

Reflection-conditioned generation. The benefits of search experience also extend to offspring generation. Removing reflection guidance from the Actor raises average gaps by 3.58 percentage points on TSP and 2.00 on CVRP. Worst-instance quality and runtime also worsen on both tasks, with TSP runtime rising from 34 to 64 ms. These results demonstrate that reflection guidance helps the Actor generate heuristics with better average and worst-instance quality and lower execution cost.

5 Discussion

Discussion. We provide additional experiments and analyses in the appendix. Appendix D.2 extends the evaluation to guided local search, online bin packing, and TSPLIB, and includes significance tests against EoH and heuristic evolution trajectories. The two-objective variant of ParetoEvo also achieves higher HV and lower IGD than MEoH and MPaGE on both TSP sizes. To assess performance across LLM backbones, we repeat the evaluation with `GPT-5-NANO` in Appendix D.3. ParetoEvo retains its advantage in average solution quality over EoH and MCTS-AHD across all four frameworks.

Limitations and future work. Although ParetoEvo designs effective, efficient, and robust heuristics, it evolves heuristic functions within a predefined solver framework and accumulates evidence separately in each search. In future work, we aim to evolve solver frameworks jointly with their heuristic functions, and to transfer this evidence across different tasks.

6 Conclusion

In this paper, we proposed ParetoEvo, a Pareto-guided tree-search framework for heuristic evolution that integrates objective trade-offs and generation history through Pareto-aware subtree selection, productivity-guided expansion, and reflection-conditioned generation. Experiments show improved solution quality, execution efficiency, worst-instance performance, and Pareto-front convergence compared to existing baselines. These findings support guiding heuristic evolution by both attained performance and further search potential.

REFERENCES

Florian Arnold and Kenneth Sörensen. Knowledge-guided local search for the vehicle routing problem. *Computers & Operations Research*, 105:32–46, 2019. doi: 10.1016/j.cor.2019.01.002.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2): 405–421, 2021. doi: 10.1016/j.ejor.2020.07.063. URL https://doi.org/10.1016/j.ejor.2020.07.063.

Leonardo C. T. Bezerra, Manuel López-Ibáñez, and Thomas Stützle. Automatically designing state-of-the-art multi- and many-objective evolutionary algorithms. *Evolutionary Computation*, 28(2): 195–226, 2020. doi: 10.1162/evco_a_00263. URL https://doi.org/10.1162/evco_a_00263.

Aymeric Blot, Marie-Éléonore Kessaci, Laetitia Jourdan, and Holger H. Hoos. Automatic configuration of multi-objective local search algorithms for permutation problems. *Evol. Comput.*, 27(1):147–171, March 2019. ISSN 1063-6560. doi: 10.1162/evco_a_00240. URL https://doi.org/10.1162/evco_a_00240.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810. URL https://doi.org/10.1109/TCIAIG.2012.2186810.

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.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/2460396f2d0d421885997dd1612ac56b-Abstract-Conference.html.

Augusto B. Corrêa, André G. Pereira, and Jendrik Seipp. Classical planning with LLM-generated heuristics: Challenging the state of the art with Python code. In *Advances in Neural Information Processing Systems*, volume 38, pp. 42070–42108. Curran Associates, Inc., 2025. doi: 10.52202/085713-1401. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/3bf4b55960aaa23553cd2a6bdc6e1b57-Abstract-Conference.html.

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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(25):26931–26938, 2025. doi: 10.1609/aaai.v39i25.34898. URL https://ojs.aaai.org/index.php/AAAI/article/view/34898.

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. doi: 10.1109/4235.996017.

Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006. doi: 10.1109/MCI.2006.329691. URL https://doi.org/10.1109/MCI.2006.329691.

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. doi: 10.1016/j.ejor.2019.07.073. URL https://doi.org/10.1016/j.ejor.2019.07.073.

A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999. doi: 10.1109/4235.771166.

Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 60(1–2): 25–64, 2010. doi: 10.1007/s10472-010-9213-y. URL https://link.springer.com/article/10.1007/s10472-010-9213-y.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/fef126561bbf9d4467dbb8d27334b8fe-Abstract-Conference.html.

Gabriel Grand, Lio Wong, Maddy Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. LILO: Learning interpretable libraries by compressing and documenting code. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/819cebb05f993840e8a52d7564c5c282-Abstract-Conference.html.

Oguzhan Gungordu, Siheng Xiong, and Faramarz Fekri. PathWise: Planning through world model for automated heuristic design via self-evolving LLMs. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=aiOv6NhF3N.

Minh Hieu Ha, Hung Phan, Tung Duy Doan, Tung Dao, Cong Dao Tran, and Huynh Thi Thanh Binh. Pareto-grid-guided large language models for fast and high-quality heuristics design in multi-objective combinatorial optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(43):36964–36972, 2026. doi: 10.1609/aaai.v40i43.41024. URL https://ojs.aaai.org/index.php/AAAI/article/view/41024.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.507. URL https://aclanthology.org/2023.emnlp-main.507/.

Conor F. Hayes, Mathieu Reymond, Diederik M. Roijers, Enda Howley, and Patrick Mannion. Monte carlo tree search algorithms for risk-aware and multi-objective reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 37(2):26, 2023. doi: 10.1007/s10458-022-09596-0. URL https://link.springer.com/article/10.1007/s10458-022-09596-0. Article 26.

Keld Helsgaun. An extension of the Lin–Kernighan–Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. Technical report, Roskilde University, Roskilde, Denmark, 2017. URL https://forskning.ruc.dk/da/publications/an-extension-of-the-lin-kernighan-helsgaun-tsp-solver-for-constra/.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/8b4add8b0aa8749d80a34ca5d941c355-Abstract-Conference.html.

Benjamin Hudson, Qingbiao Li, Matthew Malencia, and Amanda Prorok. Graph neural network guided local search for the traveling salesperson problem. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=ar92oEosBIg.

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. doi: 10.1613/jair.2861. URL https://doi.org/10.1613/jair.2861.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. LeDex: Training LLMs to better self-debug and explain code. In *Advances in Neural Information Processing Systems*, volume 37, pp. 35517–35543, 2024. doi: 10.52202/079017-1120. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/3ea832724870c700f0a03c665572e2a9-Abstract-Conference.html.

D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974. doi: 10.1137/0203025. URL https://doi.org/10.1137/0203025.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/hash/d9896106ca98d3d05b8cbdf4fd8b13a1-Abstract.html.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, volume 4212 of *Lecture Notes in Computer Science*, pp. 282–293, 2006. doi: 10.1007/11871842_29. URL https://link.springer.com/chapter/10.1007/11871842_29.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ByxBFsRqYm.

Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. POMO: Policy optimization with multiple optima for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 33, pp. 21188–21198, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/f231f2107df69eab0a3862d50018a9b2-Abstract.html.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328, 2022. doi: 10.52202/068431-1549. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html.

Haoyun Li, Ming Xiao, Kezhi Wang, Dong In Kim, and Merouane Debbah. Large language model based multi-objective optimization for integrated sensing and communications in UAV networks. *IEEE Wireless Communications Letters*, 14(4):979–983, 2025a. doi: 10.1109/LWC.2025.3529082. URL https://doi.org/10.1109/LWC.2025.3529082.

Haoyun Li, Ming Xiao, Kezhi Wang, Robert Schober, Dong In Kim, and Yong Liang Guan. Joint user association and beamforming design for ISAC networks with large language models. *IEEE Open Journal of the Communications Society*, 6:7620–7644, 2025b. doi: 10.1109/OJCOMS.2025.3606812. URL https://doi.org/10.1109/OJCOMS.2025.3606812.

Haoyun Li, Ming Xiao, Kezhi Wang, Robert Schober, Dong In Kim, and Yong Liang Guan. ComAgent: Multi-LLM based agentic AI empowered intelligent wireless networks. *IEEE Wireless Communications*, pp. 1–12, 2026. doi: 10.1109/MWC.2026.3711801. URL https://doi.org/10.1109/MWC.2026.3711801. Early access.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223. PMLR, 2024. URL https://proceedings.mlr.press/v235/liu24bs.html.

Fei Liu, Yilu Liu, Qingfu Zhang, Xialiang Tong, and Mingxuan Yuan. EoH-S: Evolution of heuristic set using LLMs for automated heuristic design. *Proceedings of the AAAI Conference on Artificial*

Intelligence, 40(43):37090–37098, 2026. doi: 10.1609/aaai.v40i43.41038. URL https://ojs.aaai.org/index.php/AAAI/article/view/41038.

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. doi: 10.1016/j.orp.2016.09.002. URL https://doi.org/10.1016/j.orp.2016.09.002.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/70c26937fbf3d4600b69a129031b66ec-Abstract-Conference.html.

Yining Ma, Jingwen Li, Zhiguang Cao, Wen Song, Le Zhang, Zhenghua Chen, and Jing Tang. Learning to iteratively solve routing problems with dual-aspect collaborative transformer. In *Advances in Neural Information Processing Systems*, volume 34, pp. 11096–11107, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/5c53292c032b6cb8510041c54274e65f-Abstract.html.

Yining Ma, Zhiguang Cao, and Yeow Meng Chee. Learning to search feasible and infeasible regions of routing problems with flexible neural k-opt. In *Advances in Neural Information Processing Systems*, volume 36, pp. 49555–49578, 2023. doi: 10.52202/075280-2157. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/9bae70d354793a95fa18751888cea07d-Abstract-Conference.html.

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

Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023. doi: 10.1038/s41586-023-06004-9. URL https://doi.org/10.1038/s41586-023-06004-9.

Su Nguyen, Yi Mei, and Mengjie Zhang. Genetic programming for production scheduling: A survey with a unified framework. *Complex & Intelligent Systems*, 3(1):41–66, 2017. doi: 10.1007/s40747-017-0036-x. URL https://doi.org/10.1007/s40747-017-0036-x.

OpenAI. GPT-4o mini: Advancing cost-efficient intelligence. OpenAI release announcement, July 2024.

OpenAI. Introducing GPT-5 for developers. OpenAI release announcement, August 2025.

Michael Painter, Bruno Lacerda, and Nick Hawes. Convex hull monte-carlo tree-search. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30(1):217–225, 2020. doi: 10.1609/icaps.v30i1.6664. URL https://ojs.aaai.org/index.php/ICAPS/article/view/6664.

Diego Perez, Sanaz Mostaghim, Spyridon Samothrakis, and Simon M. Lucas. Multiobjective monte carlo tree search for real-time games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):347–360, 2015. doi: 10.1109/TCIAIG.2014.2345842.

Laurent Perron and Vincent Furnon. OR-Tools, 2025. URL https://developers.google.com/optimization/.

Gerhard Reinelt. TSPLIB—a traveling salesman problem library. *ORSA Journal on Computing*, 3 (4):376–384, 1991. doi: 10.1287/ijoc.3.4.376. URL https://doi.org/10.1287/ijoc.3.4.376.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

Nasser R. Sabar and Graham Kendall. Population based monte carlo tree search hyper-heuristic for combinatorial optimization problems. *Information Sciences*, 314:225–239, 2015. doi: 10.1016/j.ins.2014.10.045. URL https://www.sciencedirect.com/science/article/pii/S0020025514010287.

Marwin H. S. Segler, Mike Preuss, and Mark P. Waller. Planning chemical syntheses with deep neural networks and symbolic AI. *Nature*, 555:604–610, 2018. doi: 10.1038/nature25978. URL https://doi.org/10.1038/nature25978.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/ce65173b994cf7c925c71b482ee14a8d-Abstract-Conference.html.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 2016. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961.

Jingyan Sui, Shizhe Ding, Boyang Xia, Ruizhi Liu, and Dongbo Bu. NeuralGLS: Learning to guide local search with graph convolutional network for the traveling salesman problem. *Neural Computing and Applications*, 36(17):9687–9706, 2024. doi: 10.1007/s00521-023-09042-6.

Niki van Stein and Thomas Bäck. LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29(2): 331–345, 2025. doi: 10.1109/TEVC.2024.3497793. URL https://doi.org/10.1109/TEVC.2024.3497793.

Linnan Wang, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. Neural architecture search using deep neural networks and monte carlo tree search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06):9983–9991, 2020. doi: 10.1609/aaai.v34i06.6554. URL https://doi.org/10.1609/aaai.v34i06.6554.

Weijia Wang and Michèle Sebag. Multi-objective Monte-Carlo tree search. In *Proceedings of the 4th Asian Conference on Machine Learning*, volume 25 of *Proceedings of Machine Learning Research*, pp. 507–522. PMLR, 2012. URL https://proceedings.mlr.press/v25/wang12b.html.

Weijia Wang and Michèle Sebag. Hypervolume indicator and dominance reward based multi-objective monte-carlo tree search. *Machine Learning*, 92(2–3):403–429, 2013. doi: 10.1007/s10994-013-5369-0. URL https://link.springer.com/article/10.1007/s10994-013-5369-0.

Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. Multi-objective evolution of heuristic using large language model. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(25):27144–27152, 2025. doi: 10.1609/aaai.v39i25.34922. URL https://ojs.aaai.org/index.php/AAAI/article/view/34922.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Helan Liang, and Yong Li. DeepACO: Neural-enhanced ant systems for combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 36, pp. 43706–43728, 2023. doi: 10.52202/075280-1895. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/883105b282fe15275991b411e6b200c5-Abstract-Conference.html.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, pp. 43571–43608. Curran Associates, Inc., 2024. doi: 10.52202/079017-1381. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/4ced59d480e07d290b6f29fc8798f195-Abstract-Conference.html.

Qingfu Zhang and Hui Li. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007. doi: 10.1109/TEVC.2007.892759. URL https://doi.org/10.1109/TEVC.2007.892759.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 78338–78373. PMLR, 2025. URL https://proceedings.mlr.press/v267/zheng25o.html.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

A Related Work

A.1 Classical and learning-based heuristic design

AHD reduces manual effort by searching over algorithm parameters, heuristic choices, programs, or policies. ParamILS (Hutter et al., 2009) and irace (López-Ibáñez et al., 2016) configure existing algorithms by searching their parameter spaces. To automate decisions beyond parameter values, hyper-heuristics address the selection or generation of heuristics and their components (Burke et al., 2013). Selection hyper-heuristics coordinate low-level heuristics through selection and acceptance mechanisms that respond to the search process (Drake et al., 2020). Genetic programming instead constructs decision rules by evolving program expressions. In production scheduling, these expressions combine job, machine, and system attributes into reusable scheduling heuristics (Nguyen et al., 2017). These approaches expose different design decisions to search, from the settings of a prescribed algorithm to the structure of the heuristic program.

Learning-based methods can automate either solution construction or the discovery of the algorithm used to construct solutions. Graph-based reinforcement learning constructs solutions through a learned greedy policy (Khalil et al., 2017), while attention-based models learn sequential decisions for routing problems (Kool et al., 2019). AlphaDev (Mankowitz et al., 2023) uses reinforcement learning to discover assembly programs, assessing correctness and execution latency. The routing policies are applied directly to problem instances, while AlphaDev searches for executable

algorithms. LLM-based heuristic design also searches for programs, using pretrained models to generate candidates from task descriptions and existing code.

A.2 LLM-BASED HEURISTIC EVOLUTION AND PROGRAM SEARCH

LLM-based heuristic evolution alternates candidate generation, evaluation, and selection for further revision. FunSearch (Romera-Paredes et al., 2024) couples language-model generation with evolutionary selection for program discovery. EoH (Liu et al., 2024) evolves natural-language heuristic descriptions alongside their implementations. ReEvo (Ye et al., 2024) incorporates reflections into evolutionary search, whereas HSEvo (Dat et al., 2025) emphasizes population diversity. LLaMEA (van Stein & Bäck, 2025) generates and evaluates metaheuristic algorithms. These methods direct program search through two complementary choices: the information supplied to the generator and the rule used to select candidates for further refinement.

The role of the generated output determines whether an LLM-based method designs an algorithm or solves a particular instance. LLMs can, for example, serve as optimizers within established numerical methods. For integrated sensing and communication networks, Li et al. (2025b) combine LLM-based user association with convex-based beamforming optimization. This approach generates decisions for a given network, whereas AHD searches for executable heuristics that can be applied across instances.

Within program search, the output can extend beyond a single heuristic. EoH-S (Liu et al., 2026) evolves complementary heuristic sets for instances with different sizes and distributions. In classical planning, generated Python heuristics guide greedy best-first search within an established solver (Corrêa et al., 2025). LILO (Grand et al., 2024) alternates program synthesis, compression, and documentation to construct reusable libraries. These methods organize the output as complementary heuristics, solver guidance, or reusable abstractions. In PARETOEVO, we seek heuristics spanning quality, efficiency, and robustness trade-offs. To guide heuristic evolution across these objectives, we retain source–offspring relations and combine subtree outcomes with generation productivity in source selection.

A.3 MULTI-OBJECTIVE OPTIMIZATION AND ALGORITHM DESIGN

When objectives compete, the allocation of search effort depends on how their trade-offs are represented. NSGA-II (Deb et al., 2002) combines non-dominated sorting with diversity preservation, whereas MOEA/D (Zhang & Li, 2007) coordinates scalar subproblems obtained through decomposition. In wireless networks, LEDMA (Li et al., 2025a) uses LLMs as evolutionary operators in a decomposition-based evolutionary algorithm. It optimizes aerial vehicle deployment and power control to balance communication and sensing objectives. Multi-objective evaluation also appears in automatic algorithm configuration. Blot et al. (2019) jointly optimize complementary performance indicators when configuring multi-objective local search algorithms. AutoMOEA+ (Bezerra et al., 2020) constructs multi- and many-objective evolutionary algorithms from configurable components. Its analysis examines how component choices and evaluation metrics affect the resulting algorithm designs.

The objectives of the underlying problem and the criteria used to assess its solver therefore require separate specification. A multi-objective solver can be configured using one performance indicator, while a solver for a single-objective problem can be assessed by both quality and cost. In LLM-based AHD, MEoH (Yao et al., 2025) considers quality and execution efficiency through a dominance–dissimilarity mechanism for population management. MPaGE (Ha et al., 2026) uses Pareto-grid guidance for heuristic design in multi-objective CO. We evaluate generated heuristics through average quality, execution time, and worst-instance quality. To retain these criteria when allocating search effort, we compare vector-valued subtree returns during source selection.

A.4 TREE SEARCH AND ADAPTIVE SEARCH ALLOCATION

Tree search allocates evaluations through the outcomes associated with successive decisions. This structure has been used both to compose heuristics and to guide sequential decisions. Sabar & Kendall (2015) combine a population-based mechanism with MCTS to identify sequences of low-level heuristics. Multi-objective extensions include HV-based and dominance-based MCTS (Wang

& Sebag, 2013), convex-hull MCTS (Painter et al., 2020), and methods using nonlinear utility and return distributions (Hayes et al., 2023). In language agents, LATS (Zhou et al., 2024) combines tree search with LLM-based evaluation, reflection, and environmental feedback. MCTS-AHD (Zheng et al., 2025) applies tree search directly to heuristic evolution. Comparing these uses of tree search requires specifying what each node represents, how its return is evaluated, and which evidence is propagated to earlier decisions.

Search feedback can also adapt allocation without a tree. In evolutionary computation, parameter-control methods change algorithm settings during a run, including in response to search feedback (Eiben et al., 1999). Bandit-based selection assigns credit to evolutionary operators from observed fitness improvements and adjusts their subsequent use (Fialho et al., 2010). Here, credit concerns an evolutionary operator. In ParetoEvo, the allocation decision instead concerns a candidate and its subtree. We combine subtree returns with evidence from productive generation steps so that allocation can use evidence available at different stages of search. APC assesses whether productivity at node creation predicts productivity at first expansion, and grants bounded additional exploration when the accumulated evidence supports this relationship. This decision complements PD-MCTS while the mutation and crossover probabilities remain fixed.

A.5 Reflection and feedback for program improvement

Program refinement requires both evidence about the current output and a rule for using that evidence in the next revision. Self-Refine (Madaan et al., 2023) alternates feedback and refinement using the same model. Self-Debugging (Chen et al., 2024) combines code explanation with execution feedback to identify and repair errors, whereas CRITIC (Gou et al., 2024) uses external tools to validate and revise model outputs. External verification is relevant because a model-generated critique alone does not establish correctness. Huang et al. (2024) identify limitations of intrinsic self-correction without external feedback in the reasoning settings they examine.

Execution evidence can also guide training and candidate selection. CodeRL (Le et al., 2022) trains a code-generating actor with feedback from a correctness critic and uses unit tests and critic scores for program regeneration. LeDex (Jiang et al., 2024) trains models on code explanation and refinement trajectories filtered through execution verification. CodeT (Chen et al., 2023) instead generates tests and ranks candidate programs by execution agreement, without updating model parameters. Beyond functional correctness, Shypula et al. (2024) study prompting and fine-tuning for runtime optimization using performance-improving code edits and controlled evaluation. Thus, feedback can target correctness or execution performance, and can affect model training, candidate generation, or candidate selection. These choices determine what evidence a refinement procedure can reuse.

Feedback can also be retained beyond the current candidate. Reflexion (Shinn et al., 2023) stores verbal reflections for subsequent attempts, while ExpeL (Zhao et al., 2024) extracts and retrieves insights from experiences across training tasks. In optimization-related program generation, Eureka (Ma et al., 2024) uses reward feedback to guide reward-function evolution, and ReEvo (Ye et al., 2024) incorporates reflections into heuristic search. ComAgent (Li et al., 2026) coordinates planning, coding, and scoring agents to formulate wireless optimization problems and generate simulation code. Execution errors and feasibility assessments guide iterative code refinement. To guide heuristic evolution through both search allocation and code revision, ParetoEvo maintains mutation- and crossover-specific reflections linking heuristic changes to objective variation and transition productivity. The Actor conditions generation on these reflections, while the same evaluated transitions update subtree returns and APC evidence. Consequently, evaluation informs both how a heuristic is revised and which heuristic receives further search effort, with the LLM parameters held fixed.

B Method Details

B.1 Evaluation and selection conventions

Empirical evaluation. We evaluate all candidates on the same fixed, nonempty, finite search set $\mathcal{D}_{\text{search}}$, with equal weight assigned to each instance. Accordingly, we define $\mathcal{D}$ as the empirical evaluation distribution that assigns probability $1/|\mathcal{D}_{\text{search}}|$ to each search instance. We set J to negative solution cost for TSP, CVRP, and BPP, and to selected profit for KP and MKP.

The runtime $T_{\mathrm{run}}(h, x)$ sums the elapsed time of calls to h, excluding other solver work and LLM generation. For repeated solver runs, we first average outcomes and accumulated heuristic-call times within each instance, then compute Q, E, and W across instances. One function evaluation (FE) evaluates one candidate on the full search set, and budget B includes initialization.

Local percentile ranks. For $M_{p,t} = |\mathcal{C}_t^+(p)|$ viable children, the mapping used in Equation 6 is

$$[\mathcal{R}_{p,t}(\bar{\boldsymbol{y}}_v)]_k = \begin{cases} \dfrac{\sum_{u \in \mathcal{C}_t^+(p)} \mathbf{1}\{\bar{y}_{u,k} < \bar{y}_{v,k}\} + \frac{1}{2}\left(\sum_{u \in \mathcal{C}_t^+(p)} \mathbf{1}\{\bar{y}_{u,k} = \bar{y}_{v,k}\} - 1\right)}{M_{p,t} - 1}, & M_{p,t} > 1, \\ \frac{1}{2}, & M_{p,t} = 1. \end{cases} \tag{13}$$

Equal values receive their average rank, and a coordinate shared by all viable children receives $1/2$.

Crowding-distance conventions. In Definition 3, only objectives with positive range determine boundary solutions. Sets of one or two solutions receive infinite crowding distance, whereas larger sets of identical objective vectors receive zero distance.

Crossover parent ranking. The parent comparison in Section 3.5 uses Pareto ranks obtained by non-dominated sorting over all evaluated non-root nodes. Crowding distance is computed within each resulting front for this comparison.

Scalar-UCT ablation. Scalar-UCT (Q, E, W) replaces Pareto-based child selection with

$$S_p(v) = \frac{1}{3}\mathbf{1}^\top \mathcal{R}_{p,t}(\bar{\boldsymbol{y}}_v) + \lambda \sqrt{\frac{\log(N_p + 1)}{N_v + 1}}, \qquad v^* = \underset{v \in \mathcal{C}_t^+(p)}{\arg\max}\, S_p(v). \tag{14}$$

It uses equal objective weights and the same $\lambda = 0.1$, subtree returns, and generation procedure as Full.

B.2 Algorithmic summary

Algorithm 1 specifies the update order linking PD-MCTS, APC, and reflection. The index t labels post-initialization iterations. Source selection uses $\mathcal{P}_t$ at the start of iteration t, whereas node valuation uses evidence updated after that iteration's evaluation.

Algorithm 1 Search loop of ParetoEvo.

```
evaluate initial heuristics and initialize the tree, A_0, APC evidence, and evolutionary operator reflections
set t ← 1 and A^(0) ← A_0
while fewer than B completed heuristic evaluations, including initialization do
    if an APC candidate or calibration probe is eligible under Appendix B.3 then
        select v_t^src by Eq. 10, or a calibration probe if no certified candidate is pending
    else
        select v_t^src by PD-MCTS using Eq. 7
    end if
    generate h̃_t using Eq. 11, evaluate it, and attach the resulting node v_t to v_t^src
    assign Y_t using Eq. 8, backpropagate y(h_{v_t}), and update A_t
    if the source v_m is non-initial and expanded for the first time, record Y_m^+ ← Y_t and update its class evidence
    assess v_t for APC certification or calibration using the updated evidence
    add the evaluated transition to Ξ_{o_t}^{(e(t))}
    if the current epoch ends then
        update reflections using Eq. 12 and set A^{(e(t)+1)} ← A_t
        discount APC evidence and clear pending candidates and epoch transition records
    end if
    t ← t + 1
end while
return the final non-dominated set
```

B.3 APC EVIDENCE AND POSTERIOR DERIVATION

Evidence accumulation. APC initializes $S_i = F_i = 0$ and accumulates evidence throughout the search. A non-initial source v_m contributes once, at its first expansion, which may occur in a later epoch than its creation. Its outcome adds Y_m^+ to S_{Y_m} and $1 - Y_m^+$ to F_{Y_m}. Initial nodes have no creation label and do not contribute to this evidence. At each epoch boundary, we multiply accumulated evidence by $\zeta = 0.5$ while keeping the prior parameters fixed. For $\zeta < 1$, S_i and F_i are discounted evidence weights rather than sample counts. Algorithm 1 places this discount after the current iteration's certification decision.

Posterior derivation. Assuming conditionally independent first-expansion outcomes, combining the weighted Bernoulli likelihood with the Beta prior gives

$$\begin{aligned} \mathrm{p}(\vartheta_i \mid S_i(t), F_i(t)) &\propto \vartheta_i^{S_i(t)}(1-\vartheta_i)^{F_i(t)}\vartheta_i^{\alpha_0-1}(1-\vartheta_i)^{\beta_0-1} \\ &= \vartheta_i^{\alpha_0+S_i(t)-1}(1-\vartheta_i)^{\beta_0+F_i(t)-1}. \end{aligned} \tag{15}$$

Normalizing this kernel yields the Beta posterior used in Equation 9.

Calibration. Certification requires $S_i(t) + F_i(t) \geq 3$ for both creation classes. While the productive class lacks sufficient evidence, APC admits newly created productive nodes as calibration probes. Probes are prioritized by the productive-class posterior mean at creation and contribute their first-expansion outcomes to the same evidence.

Additional-expansion allowance. Certified promotions take priority over calibration probes, and both count towards budget B. Each requires at least two ordinary PD-MCTS expansions since the previous APC expansion. At iteration t, the cumulative number of completed APC expansions must be below $\lfloor 2\sqrt{t-1} \rfloor$.

B.4 PROMPT TEMPLATES

The following templates instantiate the generation and reflection interfaces in Section 3.5. To connect the search state to each LLM call, angle-bracketed fields are filled at runtime with the selected programs, action, and evaluation evidence. The Actor uses this context to generate a candidate, and the Reflector uses evaluated transitions to update guidance for each evolutionary operator.

Evolutionary operator selection. To specify the revision requested from the Actor, we instantiate the mutation and crossover classes in Section 3.5 with the five variants in Table 4. Their selection probabilities remain fixed throughout search. The mutation variants `m1`, `m2`, and `m3` sum to $\gamma = 0.6$, while the crossover variants `e1` and `e2` sum to $1 - \gamma = 0.4$.

Table 4: Evolutionary operators and their fixed selection probabilities for post-initialization expansions. Percentages are rounded to two decimal places.

Code	Mode	Revision type	Selection probability
`e1`	Crossover	Divergent revision	$6/35 \approx 17.14\%$
`e2`	Crossover	Compositional revision	$8/35 \approx 22.86\%$
`m1`	Mutation	Structural revision	$6/35 \approx 17.14\%$
`m2`	Mutation	Parameter revision	$10/35 \approx 28.57\%$
`m3`	Mutation	Robustness revision	$5/35 \approx 14.29\%$

Task and objective contract. The task contract specifies what the generated program must implement: the optimization problem, function interface, inputs, and outputs. In the TSPC example, the Actor implements a next-node selector used by a constructive solver. The evaluator executes the completed solver on the search instances and reports the three heuristic-level criteria used by PARETOEVO. The first two modules of the Actor prompt specify this task contract and the objective semantics needed to interpret the returned values.

Evolutionary operator instructions. The selected evolutionary operator determines which instruction is added to the Actor context. Mutation revises the source alone, while crossover combines it with a second evaluated heuristic. Each instruction requires the same function interface and targets the raw objective values (Q, E, W). Because Q and W denote raw task outcomes in these templates, the objective semantics module specifies how they map to the maximized search vector in Equation 4.

e1 – Crossover: divergent revision

EVOLUTIONARY OPERATOR INSTRUCTION

```
You are the Actor for a crossover generation step.

Given the selected source heuristic and a second evaluated parent, create a
divergent crossover offspring that:
- draws useful ideas from both parents,
- changes the main decision rule rather than merely merging code,
- remains compatible with the required function signature and return format,
- aims to improve the objective vector (Q,E,W).

First describe your new algorithm briefly, then provide the code in a ```python```
    block.
```

e2 – Crossover: compositional revision

EVOLUTIONARY OPERATOR INSTRUCTION

```
You are the Actor for a crossover generation step.

Given the selected source heuristic and a second evaluated parent, create a
compositional crossover offspring that:
- preserves useful structural elements from both parents,
- combines complementary components into one coherent heuristic,
- remains compatible with the required function signature and return format,
- aims to improve the objective vector (Q,E,W).

First describe your combined algorithm briefly, then provide the code in a ```python```
    block.
```

m1 – Mutation: structural revision

EVOLUTIONARY OPERATOR INSTRUCTION

```
You are the Actor for a mutation generation step.

Given the selected source heuristic, create a structurally revised offspring
that:
- changes the main control flow, scoring decomposition, or candidate-selection strategy,
- keeps useful components when they support the new design,
- remains compatible with the required function signature and return format,
- aims to improve the objective vector (Q,E,W).

First describe your structural changes briefly, then provide the code in a ```python```
    block.
```

m2 – Mutation: parameter revision

EVOLUTIONARY OPERATOR INSTRUCTION

```
You are the Actor for a mutation generation step.

Given the selected source heuristic, create a parameter-revised offspring that:
- adjusts numerical constants, thresholds, weights, normalization terms, or loop bounds,
- may replace fixed values with adaptive values when this preserves the same structure,
- keeps the overall algorithmic structure unchanged,
- aims to improve the objective vector (Q,E,W).

First describe your parameter adjustments briefly, then provide the code in a
    ```python``` block.
```
```

m3 – Mutation: robustness revision

EVOLUTIONARY OPERATOR INSTRUCTION

```
You are the Actor for a mutation generation step.

Given the selected source heuristic, create a robustness-revised offspring that:
- improves behavior across diverse instances,
- adds edge-case handling or adaptive behavior when useful,
- simplifies overly specialized logic when it harms robustness,
- remains compatible with the required function signature and return format,
- aims to improve the objective vector (Q,E,W).

First describe your generalization changes briefly, then provide the code in a
    ```python``` block.
```

Actor prompt. The shared Actor prompt instantiates Equation 11 after the search procedure has selected the source, evolutionary operator, and optional second parent. These decisions enter as fixed context, allowing the Actor to focus on implementing the requested revision. Parent selection, evolutionary operator selection, and evaluation allocation remain with the search procedure.

Actor user prompt

TASK CONTRACT

```
=== TASK CONTRACT ===
task_desc: Given node coordinates for a TSP instance, construct a tour by
    repeatedly selecting the next unvisited node. The goal is to produce short
    tours across training instances without excessive heuristic-call runtime.
func_name: select_next_node
func_signature: select_next_node(current_node: int,
    destination_node: int, unvisited_nodes: set,
    distance_matrix: np.ndarray) -> int
func_desc: Return the next unvisited node appended to the current tour.
func_inputs: current_node, destination_node, unvisited_nodes, distance_matrix
func_outputs: next_node
inout_inf: current_node, destination_node, next_node, and unvisited_nodes are
    node IDs; distance_matrix stores pairwise node distances.
other_inf: unvisited_nodes varies during construction. The function must be
    deterministic, valid for every call, and compatible with NumPy arrays.
```

OBJECTIVE SEMANTICS

```
=== OBJECTIVE SEMANTICS ===
Q: average task quality over evaluated training instances. For TSPC, lower raw
    tour length is better.
E: average heuristic runtime per instance, measured only over calls to the
    evolved heuristic. Lower raw runtime is better.
W: worst-instance task quality on the same instances. For TSPC, this is the
    largest tour length.
search_vector: ParetoEvo multiplies Q and W by the task direction and uses -E.
    For TSPC, the maximized internal vector is (-Q, -E, -W).
goal: generate executable heuristics that are non-dominated across quality,
    efficiency, and worst-instance robustness.
```

SOURCE HEURISTIC

```
=== SOURCE HEURISTIC ===
ID: <source_id>
Description: <source_description>
Raw objectives: Q=<Q>, E=<E>s, W=<W>
```python
<source_code>
```
```

**SECOND PARENT FOR CROSSOVER**

```
=== SECOND PARENT FOR CROSSOVER ===
ID: <second_parent_id>
Description: <second_parent_description>
Raw objectives: Q=<Q>, E=<E>s, W=<W>
```python
<second_parent_code>
```
```
```

PARETO CONTEXT SUMMARY

```
=== PARETO CONTEXT SUMMARY ===
- <candidate_id>: Q=<Q>, E=<E>s, W=<W>; <description>
```

EVOLUTIONARY OPERATOR REFLECTION

```
=== EVOLUTIONARY OPERATOR REFLECTION ===
<reflection>
```

FIXED ACTION

```
=== FIXED ACTION ===
operator=<operator>
source_id=<source_id>
second_parent_ids=<second_parent_id_list>
Execute this action exactly. Do not choose another evolutionary operator or parent.
```

EVOLUTIONARY OPERATOR INSTRUCTION

```
=== EVOLUTIONARY OPERATOR INSTRUCTION ===
<operator_system_prompt>

Return one complete candidate only: a brief algorithm description followed by one
    ```python``` implementation.
```

Reflector prompt. To convert observed revisions into guidance for later generation, the Reflector prompt instantiates Equation 12. At each epoch boundary, it combines transition records for the same evolutionary operator with the previous reflection. The resulting update conditions subsequent Actor calls using that evolutionary operator.

Reflector template

SYSTEM MESSAGE

```
You are the Reflector for one fixed evolutionary operator (mutation or crossover).
```

USER MESSAGE

```
Update the reflection for evolutionary operator <operator>.
Use only the observed training transitions below.
State when this evolutionary operator worked, what harmed Q, E, or W, and what to keep
    or avoid next time.
Do not select parents, allocate search budget, or predict unobserved fitness.
Previous reflection: <previous_reflection_or_NONE>
POSITIVE TRANSITIONS:
<positive_transition_records_or_NONE>
NEGATIVE OR INVALID TRANSITIONS:
<negative_transition_records_or_NONE>
Return at most 400 tokens of actionable text.
```

TRANSITION RECORD

```
transition=<transition_id>; source=<source_id>; second_parent=<second_parent_id>;
    child=<child_id>; productive=<productive>; reasons=<reason_list>;
    delta_Q=<delta_Q>; delta_E=<delta_E>; delta_W=<delta_W>; error=<error_type>;
    change=<code_diff_summary>
```

B.5 Examples of LLM outputs

The example below illustrates how reflection and generation produce different outputs within the same revision loop. The Reflector links observed code changes to objective changes and productivity, returning guidance for later revisions. The Actor expresses a revision as a short design statement and an executable Python function.
```

Reflector output

EVOLUTIONARY OPERATOR REFLECTION

```
### Reflector output: evolutionary operator reflection

Keep parameter edits that sharpen the source scoring rule without changing its
main structure. Useful changes increase the contrast between direct travel cost
and lookahead terms while keeping candidate selection simple.

Avoid large weight changes that improve one transition but fail the
productivity label, and avoid edits that leave Q nearly unchanged while
worsening W. Later generations should prefer small adaptive normalizations that
do not increase E or W.
```

Actor output

DESIGN STATEMENT

```
### Actor output: design statement
This offspring keeps the parent's scoring structure but retunes the weights on direct
     distance and entropy-like lookahead, with adaptive normalization over the
     remaining unvisited nodes.
```

PYTHON IMPLEMENTATION

```python
import numpy as np

def select_next_node(current_node: int, destination_node: int, unvisited_nodes: set, distance_matrix:
     np.ndarray) -> int:
    """Select the next node to visit from the unvisited nodes using a refined scoring approach."""
    scores = {}

    # Calculate average distance to unvisited nodes with safety checks
    avg_distance_to_unvisited = {}
    for node in unvisited_nodes:
        distances = [distance_matrix[node][other] for other in unvisited_nodes if other != node]
        avg_distance_to_unvisited[node] = np.mean(distances) if distances else 0

    total_unvisited = len(unvisited_nodes)

    for node in unvisited_nodes:
        # Calculate direct distance from current node to considered node
        direct_distance = distance_matrix[current_node, node]

        # Calculate information gain as an entropy-like measure with adaptive normalization
        remaining_distances = [distance_matrix[node, next_node] for next_node in unvisited_nodes if
     next_node != node]
        if len(remaining_distances) > 1:
            avg_remaining_distance = np.mean(remaining_distances)
            info_gain = avg_remaining_distance / (1 + np.log(len(remaining_distances)))  # Adaptive
     normalization
        else:
            info_gain = 0

        # Scoring based on both distance and info gain with refined weights
        score = (1.5 * direct_distance - info_gain) + (1.2 * direct_distance -
     avg_distance_to_unvisited[node])

        # Store score for the node
        scores[node] = score

    # Select the node with the best (minimum) score
    next_node = min(scores, key=scores.get)
    return next_node
```

C Task Definitions, Dataset Construction, and General Frameworks

Each benchmark pairs an optimization problem with a fixed solver framework Φ that calls the evolved heuristic function h. To specify what the search can modify and how candidates are evaluated, we describe the search and test datasets, define the five underlying problems, and identify the heuristic interface within each of the three solver frameworks. The problem objectives below assess individual solutions. Section 3.1 defines the quality, efficiency, and robustness criteria used to compare heuristics across instances.

C.1 DATASET CONSTRUCTION

Search and test sets are generated before heuristic evolution and held fixed across methods and independent runs. Candidate evaluation uses $\mathcal{D}_{\text{search}}$, and the final comparisons in Section 4 use disjoint test sets.

For TSP, we sample city coordinates uniformly from the two-dimensional unit square and compute their Euclidean distances. TSPC, TSP ACO, and TSP GLS share this geometric instance model. Their differences arise from the solver framework and the interface of the evolved function.

For CVRP, instances combine Euclidean node coordinates with integer customer demands. We fix the depot at $(0.5, 0.5)$ with zero demand, sample each customer demand from $\{1, \ldots, 9\}$, and set vehicle capacity to $\kappa = 50$.

For KP, we sample item profits and weights independently from the continuous uniform distribution on $[0, 1]$. The knapsack capacity is $\kappa = 12.5$ for $N = 50$ and $\kappa = 25$ for $N = 100$ and $N = 200$.

For MKP, each item has a profit and a vector of resource consumptions. We sample profits and unnormalized consumptions independently from continuous uniform distributions on $[0, 1]$. For each resource, we sample its capacity between the largest single-item consumption and the total consumption of all items. Dividing each resource's consumptions by its capacity produces the normalized matrix supplied to the solver, with unit capacity for every resource.

For BPP, the online and offline settings use different instance generators. Online item sizes are sampled from a Weibull distribution with shape parameter 3.0, multiplied by 0.45κ, clipped to $[1, \kappa]$, and rounded to integers. Offline instances provide all items before packing, using integer sizes from 20 to 100 and bin capacity $\kappa = 150$.

Table 5: Search and held-out test datasets for the eight benchmark tasks. Here N is the instance size, denoted by n in the task definitions below, and κ is the knapsack or bin capacity.

Problem	Task–framework	Search dataset $\mathcal{D}_{\text{search}}$	Held-out test datasets
Constructive decision tasks			
TSP	TSPC	64 TSP instances with $N = 50$	1,000 instances for each $N \in \{50, 100, 200\}$
KP	KP constructive	64 KP instances with $N = 100$ and $\kappa = 25$	1,000 instances for each $(N, \kappa) \in \{(50, 12.5), (100, 25), (200, 25)\}$
BPP	Online BPP	One Weibull sequence for $N = 1{,}000$ with $\kappa = 100$ or $\kappa = 500$. One Weibull sequence for $N = 5{,}000$ with $\kappa = 100$ or $\kappa = 500$	Five Weibull sequences for each $N \in \{1{,}000, 5{,}000, 10{,}000\}$. Each size is tested with $\kappa = 100$ and $\kappa = 500$
Ant colony optimization tasks			
TSP	TSP ACO	5 TSP instances with $N = 50$	64 instances for each $N \in \{50, 100\}$
CVRP	CVRP ACO	10 CVRP instances with $N = 50$	64 instances for each $N \in \{50, 100\}$
MKP	MKP ACO	5 MKP instances with $N = 100$ and $d = 5$ resources	64 instances for each $N \in \{100, 200\}$ with $d = 5$
BPP	Offline BPP ACO	5 BPP instances with $N = 500$ and $\kappa = 150$	64 instances for each $N \in \{500, 1{,}000\}$ with $\kappa = 150$
Guided local search task			
TSP	TSP GLS	10 TSP instances with $N = 200$	1,000 instances for each $N \in \{100, 200\}$. 64 instances for each $N \in \{500, 1{,}000\}$

C.2 TASK DEFINITIONS

C.2.1 TRAVELING SALESMAN PROBLEM

TSP seeks a shortest tour that visits every city exactly once and returns to its starting city. Let $\mathcal{V} = \{1, \ldots, n\}$ denote the cities, with coordinates $\boldsymbol{r}_i \in \mathbb{R}^2$ and distances $c_{ij} = \|\boldsymbol{r}_i - \boldsymbol{r}_j\|_2$. Representing a tour by a permutation σ of $\mathcal{V}$ and setting $\sigma(n+1) = \sigma(1)$ gives

$$\min_{\sigma \in \text{Perm}(\mathcal{V})} \sum_{k=1}^{n} c_{\sigma(k),\sigma(k+1)}. \tag{16}$$

TSPC, TSP ACO, and TSP GLS share this objective and feasible set. Their solver frameworks expose different functions to heuristic search, as specified below.

C.2.2 CAPACITATED VEHICLE ROUTING PROBLEM

CVRP minimizes total travel distance while serving every customer within vehicle capacity. Let $\mathcal{V} = \{1, \dots, n\}$ denote the customers, 0 the depot, $0 < q_i \leq \kappa$ the customer demands, and κ the vehicle capacity. The distance c_{ij} is Euclidean, including for connections to the depot.

A feasible solution partitions the customers into n_{routes} nonempty ordered sequences $R_k = (i_{k,1}, \dots, i_{k,m_k})$, where m_k is the number of customers on route k and $i_{k,j}$ its jth customer. Setting $i_{k,0} = i_{k,m_k+1} = 0$ makes each route start and end at the depot. The optimization problem is

$$\min_{\substack{1 \leq n_{\text{routes}} \leq n \\ R_1, \dots, R_{n_{\text{routes}}}}} \sum_{k=1}^{n_{\text{routes}}} \sum_{j=0}^{m_k} c_{i_{k,j}, i_{k,j+1}} \quad \text{subject to} \quad \sum_{i \in R_k} q_i \leq \kappa, \qquad k = 1, \dots, n_{\text{routes}}. \tag{17}$$

The partition excludes split deliveries and ensures that every customer is served once. The number of routes is a decision variable rather than a fixed benchmark parameter.

C.2.3 0–1 KNAPSACK PROBLEM

KP selects items to maximize total profit under a single capacity constraint. For n items with profits p_i, weights a_i, and knapsack capacity κ, let $z_i \in \{0, 1\}$ indicate selection. Then

$$\max_{\boldsymbol{z} \in \{0,1\}^n} \sum_{i=1}^{n} p_i z_i \quad \text{subject to} \quad \sum_{i=1}^{n} a_i z_i \leq \kappa. \tag{18}$$

C.2.4 MULTIDIMENSIONAL KNAPSACK PROBLEM

MKP extends KP by requiring each selected item to satisfy several resource constraints simultaneously. In this benchmark, MKP refers to this shared selection problem, not assignment to multiple independent knapsacks. Let p_i denote item i's profit, a_{ir} its consumption of resource r, $b_r > 0$ the resource capacity, and $d \in \mathbb{N}_+$ the number of resources. With $z_i \in \{0, 1\}$ indicating selection, the problem is

$$\max_{\boldsymbol{z} \in \{0,1\}^n} \sum_{i=1}^{n} p_i z_i \quad \text{subject to} \quad \sum_{i=1}^{n} a_{ir} z_i \leq b_r, \qquad r = 1, \dots, d. \tag{19}$$

We use $d = 5$ resources and supply the solver with normalized consumptions $\bar{a}_{ir} = a_{ir}/b_r$. This representation expresses all capacity constraints relative to a unit resource budget.

C.2.5 BIN PACKING PROBLEM

BPP seeks the smallest number of bins needed to pack all items. Given item sizes $a_1, \dots, a_n$ and capacity κ, represent a packing by a partition $\mathcal{B}$ of the item indices into nonempty bins. The problem is

$$\min_{\mathcal{B}} |\mathcal{B}| \quad \text{subject to} \quad \sum_{i \in S} a_i \leq \kappa, \qquad S \in \mathcal{B}. \tag{20}$$

The online and offline variants share this objective and capacity constraint. Their distinction is the information available when a placement is made: online BPP assigns each arriving item irrevocably without observing future sizes, while offline BPP provides all sizes before packing begins.

C.3 GENERAL FRAMEWORKS

To identify the part of each solver exposed to heuristic search, we separate the fixed solution procedure from the evolved function. The framework constructs or improves solutions, while h supplies decisions or numerical preferences within that procedure. We use step-by-step construction, ACO, and GLS.

C.3.1 STEP-BY-STEP CONSTRUCTION

A constructive solver calls h on the current state to choose each successive step. For TSPC, h selects an unvisited city, which the solver appends to the partial tour. The solver closes the tour after all

cities have been visited. For KP, h selects an item from the remaining list subject to the capacity check, after which the solver removes the item and updates the remaining capacity. For online BPP, h scores bins that can accommodate the arriving item, and the solver chooses a highest-scoring bin. In each case, earlier decisions remain fixed as construction proceeds.

C.3.2 Ant colony optimization

ACO constructs multiple solutions per iteration by combining pheromone values with task-specific heuristic preferences. To provide the latter, h is called once per instance and its output is preprocessed into positive values ψ. The preferences remain fixed throughout the run, while pheromone values τ are updated after each iteration.

For TSP and CVRP, an ant at node i samples its next node from the currently feasible set $\mathcal{F}$ according to

$$\Pr(j \mid i, \mathcal{F}) = \frac{\tau_{ij}^{\alpha} \psi_{ij}^{\beta}}{\sum_{k \in \mathcal{F}} \tau_{ik}^{\alpha} \psi_{ik}^{\beta}}, \qquad j \in \mathcal{F}, \tag{21}$$

where $\alpha, \beta > 0$ control the influence of pheromone and heuristic preferences. The feasible set enforces task constraints: TSP excludes visited cities, and CVRP additionally checks remaining capacity while permitting depot returns to start new routes. MKP applies the same normalized-product rule to item-indexed vectors. Its feasible set excludes selected items and any choice that would violate a resource constraint.

For offline BPP, selection depends on the items already placed in the current bin. The solver combines pairwise item preferences with pairwise pheromone values, averages the resulting weights over the items in a nonempty bin, and adds a small positive offset. It then samples among remaining items that fit. The first item in each new bin is sampled uniformly, and a new bin is opened when no remaining item fits in the current bin.

After construction, pheromone evaporation and reinforcement update the sampling distribution for the next iteration. Reinforcement uses inverse solution cost for TSP and CVRP, total profit for MKP, and a bin-utilization score for offline BPP. The best solution is retained according to the task objective. For offline BPP, this means minimizing bin count, even though reinforcement uses bin utilization. These updates are part of the fixed solver, so heuristic evolution changes the preferences supplied by h.

C.3.3 Guided local search

For TSP GLS, the evolved function controls where the solver applies edge penalties. The function h is called once on the distance matrix to produce a guidance matrix G. Starting from a nearest-neighbor tour, the solver applies 2-opt and relocate moves, then alternates guided perturbation with local search. It maintains both the current tour and the best tour under the original distances.

During perturbation, edge priorities depend on $G_{ij}/(1 + P_{ij})$, where P_{ij} counts previous penalties on edge (i, j). The solver increments the selected edge's penalty and applies local moves with modified costs $c_{ij} + \mu_{\mathrm{GLS}} P_{ij}$ and fixed strength $\mu_{\mathrm{GLS}} > 0$. Subsequent local search returns to the original distances. Thus, G affects the perturbation trajectory through penalty selection, while the TSP objective, local-search operators, penalty updates, and best-tour tracking remain fixed.

C.4 Heuristic interfaces for the eight tasks

The interfaces in Table 6 determine both the information available to h and how often it is called. Constructive functions act repeatedly on changing partial solutions, whereas ACO and GLS functions produce a vector or matrix once per instance for use throughout the run.

Table 6: Task–framework pairs and evolved function interfaces. Here n denotes the number of cities, customers, or items in the corresponding problem. CVRP includes one additional depot node.

Task	Framework	Inputs to h	Output of h
TSPC	Constructive	Current city, start city, unvisited set, distance matrix	Index of the next unvisited city
KP	Constructive	Remaining capacity, weights and profits of remaining items	Index of the next item in the remaining list
Online BPP	Constructive	Arriving item size, residual capacities of feasible bins	One priority score per feasible bin
TSP ACO	ACO	Distance matrix	$n \times n$ edge-preference matrix
CVRP ACO	ACO	Distance matrix, coordinates, demands, vehicle capacity	$(n+1) \times (n+1)$ edge-preference matrix
MKP ACO	ACO	Profits and $n \times d$ normalized resource-consumption matrix	n item-preference values
Offline BPP	ACO	Item sizes and bin capacity	$n \times n$ pairwise item-preference matrix
TSP GLS	GLS	Distance matrix	$n \times n$ guidance matrix for edge penalties

The evaluator applies several additional interface conventions. CVRP also accepts a two-input form containing the distance matrix and capacity-normalized demands. Before calling h for TSP ACO or CVRP ACO, the evaluator sets diagonal distance entries to one and retains Euclidean off-diagonal entries. For online BPP, the feasible-bin input includes unused bins as well as bins that already contain items. These conventions define the function inputs while preserving the problem objectives and constraints.

D Experimental Details

We first specify the Pareto-front evaluation metrics used in Section 4. Appendices D.2 and D.3 then report extended results with `GPT-4o-mini` and `GPT-5-NANO`, respectively. For `GPT-5-NANO`, we use temperature 1.0 and reasoning effort `medium`. Except for the significance tests in Appendix D.2.3, results are averaged over five independent runs per method and task.

D.1 Pareto-front metrics

LLM-based heuristic design produces heuristics with different trade-offs among solution quality, execution efficiency, and worst-instance performance. Each generated heuristic $h \in \mathcal{H}$ is represented by the performance vector $\boldsymbol{y}(h)$ in Equation 4. For the metric definitions, we use the equivalent minimization vector $\boldsymbol{f}(h) = -\boldsymbol{y}(h) = (-Q(h), E(h), -W(h))^\top$. The non-dominated performance vectors produced by each method form an approximate Pareto front of heuristics. We compare these fronts using hypervolume (HV) and inverted generational distance (IGD) (Yao et al., 2025; Ha et al., 2026).

Objective normalization. To compare LLM-generated heuristic fronts on a common scale, let $\mathcal{P} \subset \mathbb{R}^K$ collect the performance vectors of the heuristics included across the compared methods for a given problem, with $K = 3$ in our formulation. From this finite, nonempty set, we obtain empirical ideal and nadir estimates for each objective:

$$z_i^{\text{ideal}} = \min_{\boldsymbol{v} \in \mathcal{P}} v_i, \qquad z_i^{\text{nadir}} = \max_{\boldsymbol{v} \in \mathcal{P}} v_i, \qquad i = 1, \ldots, K. \tag{22}$$

These estimates provide shared bounds for normalizing the performance of each heuristic. For $z_i^{\text{nadir}} > z_i^{\text{ideal}}$, define

$$f_i'(h) = \frac{f_i(h) - z_i^{\text{ideal}}}{z_i^{\text{nadir}} - z_i^{\text{ideal}}}. \tag{23}$$

If the two bounds coincide, we set $f_i'(h) = 0$. The same transformation applies to all compared methods, mapping heuristic performance to $[0, 1]^K$ without dependence on the original measurement units. Let $A \subset [0, 1]^K$ denote the finite, nonempty set of non-dominated normalized vectors $\boldsymbol{f}'(h)$ for the heuristic set being evaluated.

Hypervolume. HV measures the volume of performance space dominated by the heuristic front A and bounded by a reference point $\boldsymbol{r} \in \mathbb{R}^K$. For a reference point satisfying $r_i \geq v_i$ for all $\boldsymbol{v} \in A$ and $i = 1, \ldots, K$, the hypervolume is

$$\mathrm{HV}(A; \boldsymbol{r}) = \mathrm{vol}_K \left(\bigcup_{\boldsymbol{v} \in A} \prod_{i=1}^{K} [v_i, r_i] \right), \tag{24}$$

where vol_K denotes K-dimensional Lebesgue measure. Each hyperrectangle $\prod_{i=1}^{K} [v_i, r_i]$ extends from one heuristic's normalized performance vector to $\boldsymbol{r}$. Their union captures both improvement in heuristic performance and coverage of different trade-offs, without requiring the true Pareto front. We use $\boldsymbol{r} = (1.1, 1.1, 1.1)^\top$ for the three-objective comparison. Larger HV values indicate better performance when the normalization and reference point are shared.

Inverted generational distance. IGD assesses how closely a method's heuristic front approximates a reference front. Since the true Pareto front of the heuristic space $\mathcal{H}$ is unavailable, we construct a shared empirical reference set $R \subset [0, 1]^K$ from the non-dominated normalized performance vectors of the heuristics represented in $\mathcal{P}$. For this finite, nonempty set, IGD averages the distance from each reference vector to its nearest vector in A:

$$\mathrm{IGD}(A; R) = \frac{1}{|R|} \sum_{\boldsymbol{u} \in R} \min_{\boldsymbol{v} \in A} \|\boldsymbol{u} - \boldsymbol{v}\|_2, \tag{25}$$

where $\|\cdot\|_2$ is the Euclidean norm. Lower IGD values indicate closer approximation to the common reference front. Reference regions poorly represented by the generated heuristics contribute larger distances, so IGD reflects both convergence and coverage of the observed heuristic trade-offs.

D.2 Extended Results with GPT-4o-mini

The additional `GPT-4o-mini` experiments examine heuristic performance and search behavior. Appendices D.2.1 and D.2.2 test different solver and information settings through GLS and online BPP. Appendix D.2.3 reports significance tests for solution-quality differences, while Appendix D.2.4 presents comparisons on TSPLIB. Appendix D.2.5 relates changes in heuristic performance to the expansion and reflection events observed during search. Appendix D.2.6 evaluates two-objective evolution, while Appendix D.2.7 examines sensitivity to parameter changes and the rationale for the default settings.

D.2.1 Guided Local Search Framework

To evaluate heuristic design within an improvement-based solver, we apply ParetoEvo to GLS, where edge penalties guide exploration beyond local optima. The evolved function produces the penalty-guidance matrix, while local-search operators and penalty updates remain fixed. Table 7 compares this guidance with hand-designed, neural, and LLM-based alternatives on four TSP test sets.

ParetoEvo achieved the lowest average gap among LLM-based methods at $N = 200$, 500, and 1,000. It also achieved lower gaps than KGLS at these sizes and all four neural baselines at $N = 100$ and 200. Across the four sizes, it attained the lowest worst-instance tour cost on three, including a tie at $N = 100$, and executed faster than ReEvo, HSEvo, and MCTS-AHD throughout.

These improvements yielded dominance over MCTS-AHD in all three reported objectives at $N =$ 100, 200, and 1,000. On the largest test set, ParetoEvo also dominated EoH, ReEvo, and HSEvo, achieving an average gap of 1.2902% and worst-instance tour cost of 23.87648 in 0.295 s of heuristic runtime. This runtime was 26.4% lower than EoH's, with approximately 4.7-fold, 301-fold, and 29-fold speedups over MCTS-AHD, ReEvo, and HSEvo, respectively. The joint improvement in quality and execution cost shows that accurate GLS guidance need not require more expensive heuristic evaluations.

Table 7: Comparison of GLS heuristics for TSP on four test sets. N denotes the number of cities. The sets contain 1,000 instances each at $N = 100$ and 200, and 64 instances each at $N = 500$ and 1,000. Gap% measures the percentage excess of mean tour cost over the mean LKH reference cost. Gap% values for KGLS and neural methods are taken from Table 8 of Zheng et al. (2025), where starred entries originate from Ye et al. (2024). All three metrics are minimized. Time/s, Worst, layout, and highlighting follow Table 1.

Method	TSP–GLS							
	$N = 100$		$N = 200$		$N = 500$		$N = 1000$	
	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓
Reference	0.0000% –	–	0.0000% –	–	0.0000% –	–	0.0000% –	–
KGLS	+0.0034% –	–	+0.2270% –	–	+0.9578% –	–	+1.5348% –	–
Neural methods with the GLS framework								
VRP-DACT	+1.7943% –	–	+91.9267% –	–	– –	–	– –	–
NeuOpt	+0.2950% –	–	+0.9152% –	–	– –	–	– –	–
NeuralGLS*	+0.4700%* –	–	+3.6220%* –	–	– –	–	– –	–
GNNGLS*	+0.7050%* –	–	+3.5220%* –	–	– –	–	– –	–
LLM-based AHD methods								
EoH	+0.0085% 0.00397	8.45786	+0.2232% 0.0158	11.35122	+0.9310% 0.0988	17.08433	+1.2910% 0.401	23.89309
FunSearch	+0.0059% **0.000197**	8.45504	+0.2348% **0.000592**	11.36269	+1.0358% **0.00347**	17.09797	+1.4306% **0.0167**	23.97530
ReEvo	+0.0119% 0.117	8.45828	+0.2199% 0.565	11.36421	+0.9139% 7.42	17.08897	+1.2937% 88.8	23.92996
HSEvo	**+0.0054%** 0.0800	8.45504	+0.2109% 0.321	11.37148	+0.9419% 2.03	17.11556	+1.3374% 8.46	23.93475
MCTS-AHD	+0.0078% 0.1082	8.45712	+0.2074% 0.825	11.36893	+0.8915% 0.337	17.07567	+1.2961% 1.375	23.9301
PARETOEVO	+0.0058% 0.000554	8.45504	**+0.1984%** 0.0928	11.33800	**+0.8888%** 0.0633	17.10264	**+1.2902%** 0.295	23.87648

D.2.2 ONLINE BIN PACKING

Online BPP tests decisions made without access to future items. Across the three item counts and two capacities in Table 8, PARETOEVO matched or improved the average quality of Best Fit and First Fit on all six sets. It also attained the best LLM-based average and worst-instance quality on three sets. At 10,000 items and capacity 500, its average gap was 0.15% and its worst-instance bin count was 807, while its 1.18 s runtime matched the fastest LLM-based result. Thus, the quality improvement on this set was achieved without increasing execution time relative to the fastest LLM-based baseline.

The change in runtime across item counts also supports the efficiency comparison. From 1,000 to 10,000 items, PARETOEVO's runtime grew 22.5-fold at capacity 100 and 30.3-fold at capacity 500. The corresponding factors ranged from 35.5–100.5 and 40.7–98.3 for the four LLM-based baselines. Consequently, PARETOEVO achieved the lowest runtime at both capacities on the largest sets. This is an observed scaling advantage over the tested size range, alongside the packing-quality results above.

D.2.3 P-VALUES FOR SIGNIFICANCE

We assess the statistical evidence for differences in solution quality between PARETOEVO and EoH on TSPC at $N = 100$ and ACO-based offline BPP at $N = 500$. Table 9 reports the p-values for these comparisons, together with seven run-level outcomes per method and task.

Table 8: Comparison of online BPP heuristics on six test sets with five instances each. Each label gives the item count and bin capacity. Gap% reports the average percentage gap to the L_1 lower bound, Time/s reports mean heuristic-call time per instance in seconds, and Worst reports the mean per-run maximum bin count. All three metrics are minimized. Layout and highlighting follow Table 1.

Method	Online BPP											
	1k-100		1k-500		5k-100		5k-500		10k-100		10k-500	
	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓
Best Fit	+4.77% –	–	+0.25% –	–	+4.31% –	–	+0.55% –	–	+4.05% –	–	+0.47% –	–
First Fit	+5.02% –	–	+0.25% –	–	+4.65% –	–	+0.55% –	–	+4.36% –	–	+0.50% –	–
EoH	+3.76% 0.042	422.000	+1.90% 0.046	83.333	**+0.86%** 0.728	2054.667	+0.66% 0.857	408.333	**+0.39%** 2.88	4051.333	+0.42% 3.29	809.333
ReEvo	+3.99% 0.050	425.000	**+0.17%** 0.055	81.667	+3.15% 0.644	2102.000	+0.40% 0.748	406.333	+3.09% 2.32	4153.667	+0.35% 2.78	808.667
HSEvo	+4.71% **0.028**	428.667	+0.25% **0.029**	82.000	+4.12% **0.286**	2125.000	+0.55% **0.329**	407.000	+3.97% 0.995	4187.667	+0.49% 1.18	810.000
MCTS-AHD	+3.26% 1.59	423.333	+0.58% 1.92	82.000	+2.68% 39.7	2091.333	+0.50% 47.7	406.667	+3.10% 159.8	4155.333	+0.44% 188.7	809.667
ParetoEvo	**+2.24%** 0.038	419.000	+0.25% 0.039	82.000	+0.90% 0.294	2058.000	**+0.10%** 0.383	405.000	+0.65% **0.854**	4058.000	**+0.15%** **1.18**	807.000

Table 9: Solution-quality comparison of EoH and ParetoEvo on constructive TSP and ACO-based offline BPP across seven runs per method and task. The columns run1–run7 give per-run mean tour costs or bin counts, both minimized. The columns avg and std report the mean and sample standard deviation across runs, and p-value reports the between-method test result.

CO Problem	Methods	run1	run2	run3	run4	run5	run6	run7	avg	std	p-value
General Framework: Step-by-step Construction, LLM: *GPT-4o-mini*											
TSP ($N = 100$)	EoH	8.964	9.049	8.963	8.993	8.950	8.963	9.252	9.019	0.108	
	ParetoEvo	8.822	8.836	8.950	8.831	8.767	8.964	8.796	8.852	0.075	0.006674534
General Framework: ACO, LLM: *GPT-4o-mini*											
Offline BPP ($N = 500$)	EoH	204.609	206.938	206.250	206.641	207.031	203.172	208.375	206.145	1.724	
	ParetoEvo	203.188	203.188	203.188	203.172	203.156	203.172	203.188	203.179	0.012	0.003884663

ParetoEvo reduced mean tour cost from 9.019 to 8.852 and mean bin count from 206.145 to 203.179. The corresponding p-values, 0.00667 for TSPC and 0.00388 for offline BPP, are both below 0.01. These comparisons support a quality advantage over EoH on the two evaluated tasks, spanning the construction and ACO frameworks.

D.2.4 Results on TSPLIB

To extend the comparison beyond the generated Euclidean test sets, we evaluate LLM-designed constructive heuristics on TSPLIB, a benchmark library for TSP (Reinelt, 1991).

In Table 10, ParetoEvo achieved the lowest average and worst-instance gaps on 13 of the 14 instances. Averaged across instances, the gap was 12.84%, compared with 15.87%–25.65% for the four LLM-based baselines. The difference between the average and worst-instance gaps was 1.29 percentage points, compared with 2.23–2.69 for the baselines. Thus, the lower average gaps were accompanied by lower worst-instance gaps and a smaller separation between the two measures.

ParetoEvo heuristics also executed faster than ReEvo and HSEvo on every instance, yielding approximately 208-fold and 176-fold speedups in mean runtime. They improved all three metrics relative to MCTS-AHD on 10 instances and achieved the lowest runtime on `pr264` and `pr226`. On `pr264`, the average and worst-instance gaps of 16.39% and 17.05% were lower than those of every baseline. Its runtime of 0.0357 s was also the lowest, including a 10.3-fold speedup over EoH.

Table 10: Comparison of constructive TSP heuristics on TSPLIB. Gap% and Worst report average quality Q and worst-instance quality W as percentage gaps to the published optimal tour costs. Time/s reports mean heuristic-call time per instance in seconds. All three metrics are minimized. Layout and highlighting follow Table 1.

Instance	EoH		ReEvo		HSEvo		MCTS-AHD		PARETOEVO	
	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓
ts225.tsp	7.34% **0.234**	8.45%	26.40% 122	27.02%	8.65% 118	9.09%	10.91% 0.607	11.46%	**3.96%** 0.51	4.08%
rat99.tsp	21.74% **0.025**	27.97%	18.17% 4.88	19.15%	18.06% 3.61	20.06%	19.52% 0.054	20.50%	**17.19%** 0.0463	18.57%
bier127.tsp	14.47% **0.0512**	16.17%	11.43% 12.8	14.77%	16.50% 9.34	19.55%	14.70% 0.114	18.08%	**11.24%** 0.136	12.02%
lin318.tsp	16.66% **0.623**	18.34%	25.05% 503	26.19%	18.44% 464	20.09%	16.87% 1.7	17.09%	**11.84%** 1.63	12.04%
eil51.tsp	9.52% **0.00452**	11.37%	12.31% 0.5	16.08%	9.10% 0.316	10.22%	8.10% 0.00811	11.02%	**5.41%** 0.00747	8.35%
d493.tsp	18.14% **2.2**	19.62%	18.49% 2840	19.74%	14.44% 1990	16.22%	18.90% 6.28	20.86%	**12.55%** 19.4	13.03%
kroB100.tsp	18.29% **0.0257**	20.97%	34.57% 5.18	37.42%	17.24% 6.5	20.71%	19.44% 0.0563	26.40%	**14.11%** 0.0554	15.67%
kroC100.tsp	18.26% **0.0254**	20.83%	34.05% 5.09	39.86%	14.53% 4.71	17.36%	17.40% 0.056	21.72%	**12.95%** 0.202	17.35%
ch130.tsp	12.77% **0.0522**	13.95%	13.52% 14	15.57%	12.52% 10.2	16.82%	11.77% 0.121	12.76%	**8.80%** 0.119	10.01%
pr299.tsp	21.58% **0.521**	22.64%	31.54% 393	34.88%	21.15% 404	21.59%	20.23% 1.41	20.85%	**17.50%** 1.35	17.52%
kroA150.tsp	19.95% **0.0758**	25.08%	48.66% 25.4	51.68%	14.84% 25.9	20.02%	14.28% 0.183	17.55%	**13.03%** 0.096	16.26%
pr264.tsp	25.45% 0.366	28.72%	35.17% 240	36.70%	20.46% 241	23.02%	20.45% 0.972	22.57%	**16.39%** **0.0357**	17.05%
pr226.tsp	19.46% 0.241	25.25%	22.39% 128	26.23%	17.19% 139	18.85%	21.72% 0.61	22.33%	**14.92%** **0.236**	15.79%
pr439.tsp	23.20% **1.58**	25.14%	27.37% 1770	29.29%	**19.08%** 1720	19.81%	22.42% 4.44	25.77%	19.88% 5.32	20.12%
Average	17.63% **0.43**	20.32%	25.65% 433	28.18%	15.87% 367	18.10%	16.91% 1.19	19.21%	**12.84%** 2.08	14.13%

D.2.5 HEURISTIC EVOLUTION TRAJECTORIES

To examine how heuristic evolution proceeds, we trace TSPC and ACO-based CVRP over 1,000 FEs. Figures 3 and 4 place changes in the best mean objective alongside representative generation steps, APC expansions, and reflection updates. These trajectories describe observed search events, while the ablations assess component contributions.

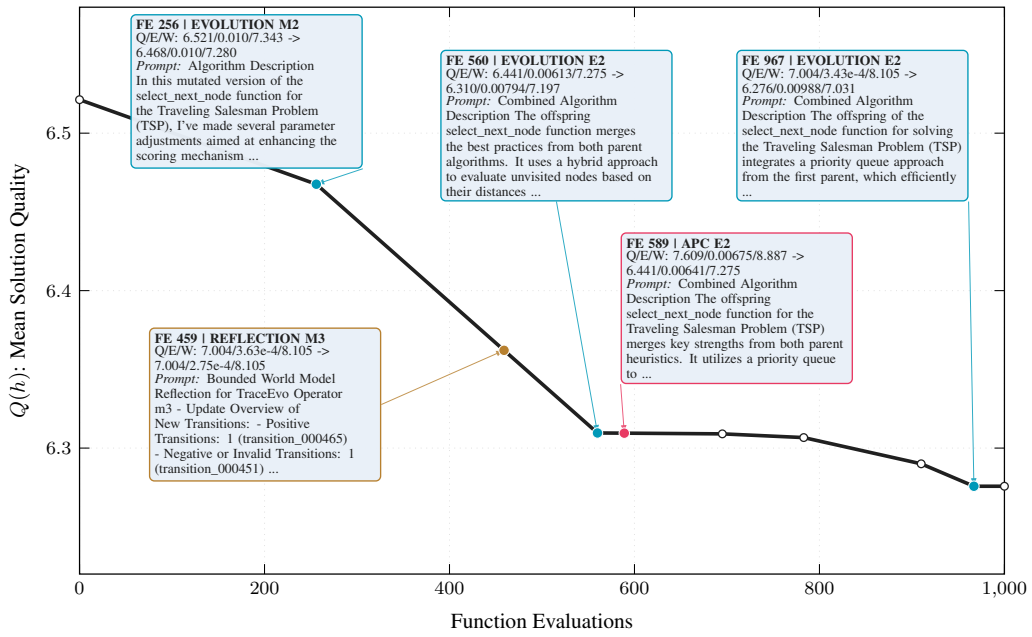

Figure 3: Evolution of constructive TSP heuristics with PARETOEVO. The black line shows the best mean tour cost at the plotted evaluation counts. Blue, pink, and orange callouts identify ordinary evolution, APC, and reflection events, respectively. FE denotes function evaluations. Each callout reports source and candidate values for mean tour cost Q, heuristic runtime E, and worst-instance tour cost W, all minimized. Negating the raw costs Q and W gives the quality coordinates in Equation 4. Reflection callouts identify the transitions used to construct the guidance.

For TSPC, the best mean tour cost fell from 6.521 to 6.276, a 3.76% improvement. Ordinary evolution reached 6.310 at FE 560 and the final value at FE 967, with a worst-instance cost of 7.031. The recorded transitions also show improvements beyond the best mean cost. At FE 459, the Reflector summarized a revision that reduced runtime by 24.3% while preserving both mean and worst-instance tour costs. At FE 589, an APC expansion improved all three objectives relative to its source: mean cost fell from 7.609 to 6.441, worst-instance cost from 8.887 to 7.275, and runtime by 5.0%. These events illustrate why source-relative productivity and incumbent quality capture different aspects of progress during heuristic evolution.

For ACO-based CVRP, the best mean route cost decreased from 10.124 to 9.189 by FE 223. At FE 268, APC expanded a source with mean cost 9.277 and produced a new best candidate with cost 9.159. The source was worse than the incumbent in average quality, yet its expansion improved the incumbent. This event illustrates the distinction between a candidate's attained quality and the value of further evolution.

Subsequent search improved the other objectives as well. At FE 997, a candidate improved all three objectives relative to the FE 268 incumbent: mean route cost decreased from 9.159 to 9.073, runtime decreased by 38.5% to 0.00152 s, and worst-instance cost decreased from 9.904 to 9.876. The final mean cost was 10.38% below the initial value. Thus, the trajectory includes both an early improvement from a non-incumbent source and a later joint gain in quality, efficiency, and worst-instance performance.

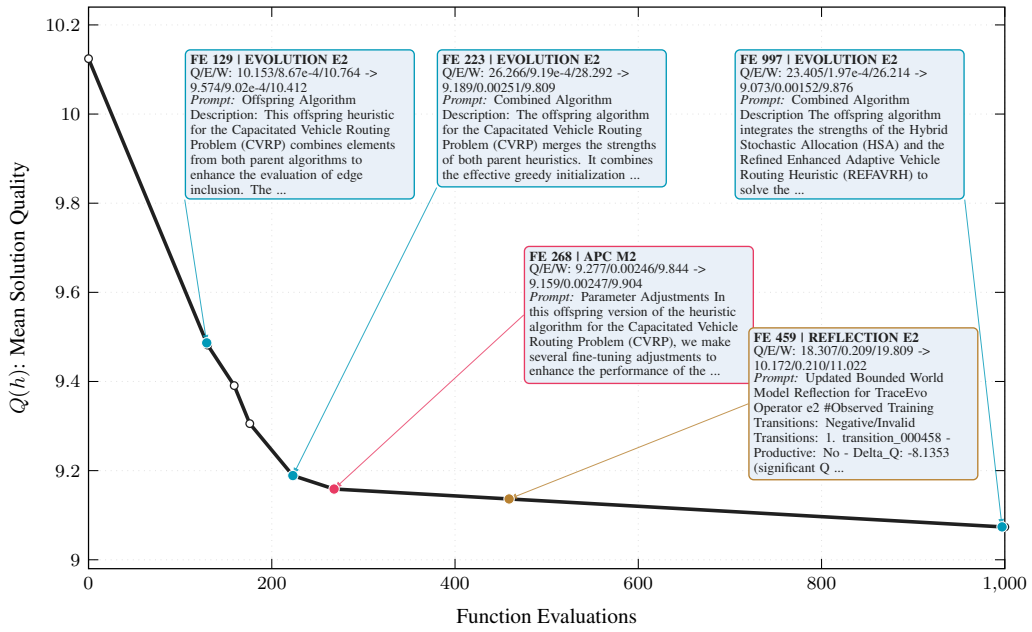

Figure 4: Evolution of ACO-based CVRP heuristics with PARETOEVO. The black line shows the best mean route cost at the plotted evaluation counts. Callouts report source and candidate mean route cost Q, heuristic runtime E, and worst-instance route cost W, all minimized. Negating the raw costs Q and W gives the quality coordinates in Equation 4. Event colors and reflection annotations follow Figure 3.

D.2.6 TWO-OBJECTIVE HEURISTIC EVOLUTION

To assess heuristic evolution with the original Q, E objectives of MEoH and MPaGE, Table 11 compares both methods with PARETOEVO *w/o* W using two-dimensional HV and IGD. PARETOEVO *w/o* W achieves higher HV and lower IGD at both sizes. Relative to the lowest baseline IGD, IGD decreases by 29.8% and 16.2% for $N = 50$ and 100, respectively, supporting improved quality–efficiency fronts under the same search objectives.

Table 11: Comparison of two-objective heuristic fronts on constructive TSP. All methods optimize Q, E. Hypervolume (HV) is maximized and inverted generational distance (IGD) is minimized. Bold values mark the best result in each column.

	TSP ($N = 50$)		*TSP* ($N = 100$)	
Method	HV↑	IGD↓	HV↑	IGD↓
MEoH	1.188672	0.131375	1.195209	0.087582
MPaGE	1.187500	0.134859	1.194658	0.082768
PARETOEVO *w/o* W	**1.198011**	**0.092194**	**1.202376**	**0.069348**

D.2.7 HYPERPARAMETER SENSITIVITY

We assess sensitivity to five hyperparameters: the exploration constant λ, epoch length L, promotion thresholds δ and η, and mutation probability γ. We vary one parameter at a time from the defaults in Section 4. Table 12 reports average quality, runtime, and worst-instance performance on constructive TSP at $N = 50$ and 100.

Table 12: Hyperparameter ablations of PARETOEVO on constructive TSP ($N = 50$ and 100). Only the indicated parameter differs from the default configuration. Definitions of Gap%, Time/ms, and Worst follow Table 3. All three metrics are minimized. Highlighting follows Table 1.

Setting	*TSP* ($N = 50$)			*TSP* ($N = 100$)		
	Gap%↓	Time/ms↓	Worst↓	Gap%↓	Time/ms↓	Worst↓
PARETOEVO (Default)	**11.05%**	8.2	7.588	**12.86%**	34	10.114
$\lambda = 0.03$	12.64%	11.7	7.610	14.67%	63.5	10.253
$\lambda = 0.3$	11.41%	7.9	7.967	13.65%	43.1	10.334
$L = 10$	14.27%	5.9	8.025	16.98%	42.1	10.905
$L = 40$	12.10%	22.2	7.757	13.50%	140.2	10.158
$\delta = 0$	14.42%	12.5	8.433	16.45%	75.7	10.748
$\delta = 0.1$	11.90%	7.4	8.048	13.60%	55.6	10.120
$\eta = 0.6$	13.90%	**4.0**	7.991	15.81%	**17.3**	10.722
$\eta = 0.95$	12.26%	9.7	7.772	13.67%	51.4	10.197
$\gamma = 0.3$	13.89%	34.7	7.883	15.51%	99.6	10.388
$\gamma = 0.9$	12.85%	21.9	7.744	15.18%	136.4	10.444

Increasing λ from 0.1 to 0.3 raised average gaps by only 0.36 and 0.79 percentage points at $N = 50$ and 100, respectively. Reducing λ to 0.03 produced larger average-gap increases but kept worst-instance costs closer to the defaults. These different responses support evaluating both quality measures when choosing exploration strength.

Shortening epochs to $L = 10$ produced larger losses in both quality measures than extending them to $L = 40$. A plausible explanation is that the shorter interval leaves too few opportunities to collect first-expansion evidence before the next reference update and evidence discount. Earlier observations also lose weight more rapidly, potentially leaving insufficient evidence for APC to assess whether productive creation predicts productive expansion. With $L = 40$, average gaps increased by 1.05 and 0.64 percentage points, but runtimes reached 2.7 and 4.1 times the defaults at $N = 50$ and 100, respectively. These results support $L = 20$ as a balance between accumulating evidence and updating the comparison reference and reflection guidance promptly.

For the APC gate, increasing δ to 0.1 or η to 0.95 kept average gaps within 1.21 percentage points of the defaults at both sizes. Lowering δ to 0 or η to 0.6 increased gaps by 2.85–3.59 percentage points and worsened worst-instance quality. Because lower thresholds admit weaker predictive evidence, these losses are consistent with a benefit from maintaining a positive productivity margin and sufficient posterior support. The faster heuristics at $\eta = 0.6$, with approximately halved runtimes, also reveal a quality–efficiency trade-off.

Both $\gamma = 0.3$ and 0.9 worsened all three objectives at both sizes. These settings shift generation toward crossover or mutation, respectively, suggesting that the default $\gamma = 0.6$ balances combining distinct parents with refining the source heuristic.

Overall, several tested settings preserved average quality closely, but stability varied by objective. The defaults combined the best average and worst-instance quality with lower execution costs than most alternatives, supporting their use as a balanced configuration.

D.3 RESULTS WITH GPT-5-NANO

To examine the trade-offs with another LLM backbone, Tables 13–16 compare PARETOEVO, EoH, and MCTS-AHD using `GPT-5-NANO` across construction, ACO, GLS, and online BPP. Hand-designed and neural baselines retain their published results. LLM-based entries are means over five runs.

Step-by-Step Construction Framework. PARETOEVO heuristics achieved the lowest LLM-based average gaps on all six sets in Table 13. They also outperformed Greedy Construct throughout and POMO on KP and TSP at $N = 200$. For TSP at $N = 100$ and 200, PARETOEVO improved both quality measures relative to MCTS-AHD and executed 4.6 and 8.6 times faster, respectively. EoH executed faster on TSP, with gaps exceeding those of PARETOEVO by 5.60–7.19 percentage

Table 13: Comparison of constructive heuristics for TSP and KP using `GPT-5-NANO` on 1,000 instances per test set. N denotes the number of cities for TSP and items for KP. Reference solutions, definitions of Gap%, Time/s, and Worst, layout, and highlighting follow Table 1.

Method	TSP constructive						KP constructive					
	$N = 50$		$N = 100$		$N = 200$		$N = 50$		$N = 100$		$N = 200$	
	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↑	Gap%↓ Time/s↓	Worst↑	Gap%↓ Time/s↓	Worst↑
Greedy Construct	+22.62% –	–	+24.94% –	–	+26.29% –	–	+0.26% –	–	+0.12% –	–	+0.09% –	–
POMO	+0.39% –	–	+3.01% –	–	+20.45% –	–	+2.12% –	–	+1.48% –	–	+0.09% –	–
EoH	+13.90% **0.0244**	7.690	+16.16% **0.334**	10.352	+18.43% **4.96**	13.943	+0.21% **0.00479**	14.687	+0.10% 0.0354	32.843	+0.09% 0.223	47.624
MCTS-AHD	+8.31% 0.499	7.350	+10.01% 7.04	9.983	+11.52% 109	13.103	+0.12% 0.0237	14.715	+0.06% 0.0807	32.867	0.08% 0.326	47.615
PARETOEVO	**+8.30%** 0.187	7.502	**+9.78%** 1.52	9.759	**+11.24%** 12.7	13.000	**+0.04%** 0.00969	14.738	**+0.02%** **0.0320**	32.865	**+0.03%** **0.0702**	47.684

points. Thus, PARETOEVO achieved higher average quality than both baselines, with execution time between that of EoH and MCTS-AHD.

On KP, PARETOEVO's gaps were 0.02%–0.04%, with the lowest LLM-based runtimes at $N = 100$ and 200. On the largest set, it reduced MCTS-AHD's gap from 0.08% to 0.03% and increased worst-instance profit from 47.615 to 47.684. The 0.0702 s runtime provided 3.2-fold and 4.6-fold speedups over EoH and MCTS-AHD. Thus, all three metrics improved over both LLM-based baselines on the largest KP set.

Relative to the `GPT-4o-mini` results in Table 1, PARETOEVO heuristics improved average and worst-instance quality on all six sets. TSP average gaps were approximately 24%–25% lower. At $N = 200$, the gap decreased from 14.91% to 11.24% and worst-instance tour cost from 13.566 to 13.000, while runtime increased from 0.428 to 12.7 s. This comparison shows that the higher quality of the `GPT-5-NANO` heuristics came with greater execution cost, making runtime relevant when choosing between the resulting heuristics.

Table 14: Comparison of ACO heuristics using `GPT-5-NANO` for TSP, CVRP, MKP, and offline BPP on eight test sets with 64 instances each. For each set, the best mean objective among all listed methods, including standard ACO and DeepACO, defines the reference for Gap%. Definitions of Time/s and Worst follow Table 2. Layout and highlighting follow Table 1.

Method	TSP ACO				CVRP ACO				MKP ACO				Offline BPP			
	$N = 50$		$N = 100$		$N = 50$		$N = 100$		$N = 100$		$N = 200$		$N = 500$		$N = 1000$	
	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↑	Gap%↓ Time/s↓	Worst↑	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓
ACO	+3.29% –	–	+8.81% –	–	+27.76% –	–	+25.76% –	–	+2.26% –	–	+4.58% –	–	+3.64% –	–	+4.04% –	–
DeepACO	+0.71% –	–	+0.71% –	–	0.00% –	–	0.00% –	–	+0.74% –	–	+1.49% –	–	+0.81% –	–	+0.86% –	–
EoH	+3.19% **0.000220**	6.635	+7.82% **0.000414**	9.615	+24.16% **0.000701**	12.508	+28.69% **0.00160**	21.230	+1.43% **0.000113**	7.832	+4.85% **0.000114**	10.482	+1.17% **0.0377**	211.600	+1.16% **0.183**	417.200
MCTS-AHD	+0.09% 0.123	6.343	+0.12% 1.05	10.217	+7.18% 0.0130	10.715	+9.69% 0.0505	17.899	+0.06% 0.0483	7.832	+0.48% 0.0926	10.601	+1.13% 1.29	211.600	+1.22% 5.17	419.600
PARETOEVO	**0.00%** 0.208	6.341	**0.00%** 1.52	8.901	**+6.85%** 0.00379	11.127	**+8.29%** 0.00799	18.169	**0.00%** 0.0125	7.741	**0.00%** 0.0336	10.569	**0.00%** 0.0713	209.000	**0.00%** 0.307	413.800

Ant Colony Optimization Framework. Across the eight sets in Table 14, PARETOEVO heuristics improved average quality over standard ACO, EoH, and MCTS-AHD, and over DeepACO on TSP, MKP, and offline BPP. On offline BPP, this gain extended to worst-instance quality and was accompanied by 17–18-fold faster execution than MCTS-AHD. On CVRP, PARETOEVO achieved average gaps of 6.85% and 8.29% while executing 3.4 and 6.3 times faster than MCTS-AHD, respectively. MCTS-AHD retained better worst-instance performance on CVRP and MKP, and EoH had the lowest runtime throughout. Thus, the average-quality advantage extends across all eight

sets, while the joint gains in quality, worst-instance performance, and runtime depend on the task and comparator.

The comparison with `GPT-4o-mini` in Table 2 also depends on the task. On offline BPP, the `GPT-5-NANO` heuristics reduced worst-instance bin counts from 211 to 209 and from 415 to 413.8 at the two sizes, with approximately 16–18 times the runtime. On CVRP at $N = 50$, runtime decreased from 0.0054 to 0.00379 s, while the average gap increased from 4.66% to 6.85% and worst-instance cost also rose. The resulting trade-off therefore varies by task: lower execution cost and higher solution quality do not move together across the backbone comparisons.

Table 15: Comparison of GLS heuristics for TSP using `GPT-5-NANO` on four test sets. The sets contain 1,000 instances each at $N = 100$ and 200, and 64 instances each at $N = 500$ and 1,000. Gap% for LLM-based methods uses the LKH references in Table 7. KGLS and neural entries are taken from Zheng et al. (2025), with starred values originating from Ye et al. (2024). Definitions of Time/s and Worst follow Table 7. Layout and highlighting follow Table 1.

Method	TSP–GLS							
	$N = 100$		$N = 200$		$N = 500$		$N = 1000$	
	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓
Reference	0.0000% –	–	0.0000% –	–	0.0000% –	–	0.0000% –	–
KGLS	+0.0034% –	–	+0.2270% –	–	+0.9578% –	–	+1.5348% –	–
VRP-DACT	+1.7943% –	–	+91.9267% –	–	– –	–	– –	–
NeuOpt	+0.2950% –	–	+0.9152% –	–	– –	–	– –	–
NeuralGLS*	+0.4700%* –	–	+3.6220%* –	–	– –	–	– –	–
GNNGLS*	+0.7050%* –	–	+3.5220%* –	–	– –	–	– –	–
EoH	+0.0073% **0.000335**	8.45673	+0.2250% **0.000786**	11.33981	+0.9283% **0.00517**	17.10926	+1.2866% **0.0191**	23.93826
MCTS-AHD	+0.0066% 0.0252	8.45504	+0.2121% 0.106	11.35169	+0.9114% 0.791	17.08252	+1.2979% 4.15	23.91072
PARETOEVO	**+0.0063%** 0.118	8.45504	**+0.2120%** 0.309	11.34405	**+0.8934%** 0.0933	17.09645	**+1.2834%** 0.339	23.92304

Guided Local Search Framework. PARETOEVO heuristics achieved lower average gaps than EoH and MCTS-AHD at all four sizes in Table 15. They also outperformed KGLS from $N = 200$ onward and all neural baselines in the table at $N = 100$ and 200. At $N = 500$ and 1,000, the average gaps were 0.8934% and 1.2834%, with 8.5-fold and 12.2-fold faster execution than MCTS-AHD. This comparison supports a joint gain in average quality and runtime on the larger sets. MCTS-AHD retained slightly better worst-instance quality at these sizes, and EoH remained the fastest method.

Compared with the `GPT-4o-mini` results in Table 7, PARETOEVO's average gaps changed little across all four sizes. At $N = 1{,}000$, the gap decreased from 1.2902% to 1.2834%, while worst-instance cost increased from 23.87648 to 23.92304 and runtime from 0.295 to 0.339 s. The average-quality change in GLS was therefore smaller than in TSPC and did not coincide with an improvement in all three metrics.

Online Bin Packing. PARETOEVO heuristics improved average quality over both LLM-based baselines on all six sets in Table 16 and matched or improved Best Fit and First Fit throughout. Relative to MCTS-AHD, they also matched or improved worst-instance quality while executing 38–126 times faster. At 10,000 items and capacity 500, the gap decreased from 0.47% to 0.46% and runtime from 262 to 2.07 s. This case shows that a small difference in average packing quality can accompany a large difference in execution cost. EoH was faster on all six sets, with higher average gaps and worst-instance bin counts.

Relative to `GPT-4o-mini` in Table 8, the `GPT-5-NANO` heuristics matched average and worst-instance quality at 1,000 items and capacity 500, and had worse values on the other five sets. The `GPT-4o-mini` heuristics were also faster on every set. For example, at 10,000 items and ca-

Table 16: Comparison of online BPP heuristics using GPT-5-NANO on six test sets with five instances each. Each label gives the item count and bin capacity. Definitions of Gap%, Time/s, and Worst, including the L_1 gap reference, follow Table 8. Layout and highlighting follow Table 1.

Method	Online BPP											
	1k-100		1k-500		5k-100		5k-500		10k-100		10k-500	
	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓	Gap%↓ Time/s↓	Worst↓
Best Fit	+4.77% –	–	+0.25% –	–	+4.31% –	–	+0.55% –	–	+4.05% –	–	+0.47% –	–
First Fit	+5.02% –	–	+0.25% –	–	+4.65% –	–	+0.55% –	–	+4.36% –	–	+0.50% –	–
EoH	+4.51% **0.0181**	427.800	+0.74% **0.0195**	82.400	+4.03% **0.197**	2123.400	+0.62% **0.224**	407.400	+3.82% **0.678**	4181.400	+0.53% **0.812**	810.400
MCTS-AHD	+3.49% 2.15	423.400	+0.30% 2.62	82.000	+2.13% 53.3	2082.200	+0.53% 64.8	407.000	+1.67% 215	4099.400	+0.47% 262	809.600
PARETOEVO	**+3.33%** 0.0565	422.400	**+0.25%** 0.0574	82.000	**+2.04%** 0.658	2077.200	**+0.52%** 0.716	407.000	**+1.52%** 2.73	4093.000	**+0.46%** 2.07	809.400

pacity 100, GPT-4o-mini achieved a 0.65% gap in 0.854 s, compared with 1.52% in 2.73 s for GPT-5-NANO.

Effect of the LLM backbone. Comparing non-reasoning GPT-4o-mini (OpenAI, 2024) with reasoning GPT-5-NANO (OpenAI, 2025) reveals a trade-off between heuristic quality and execution cost. For PARETOEVO, the GPT-5-NANO heuristics take longer to execute on 22 of the 24 test sets, while improving both average and worst-instance quality on all six TSP/KP construction sets. These runtimes measure the generated code, so quality gains can incur additional execution costs on every new instance, beyond the LLM computation used during design.

The benefit of this additional computation depends on the solver framework. GLS shows little change in average gaps. In online BPP, the GPT-4o-mini heuristics match or improve both quality measures and execute faster on all six sets, thereby dominating their GPT-5-NANO counterparts in the three reported objectives. One plausible interpretation is that the benefit of a reasoning backbone depends on the decisions and information exposed to the generated heuristic. TSP/KP construction delegates successive solution-building decisions to the heuristic, whereas GLS already supplies a fixed local-search procedure and uses the heuristic to guide edge penalties. Online BPP further restricts each decision to the arriving item and current residual capacities.

These differences make backbone selection part of the multi-objective design problem: the backbone must be assessed against the target solver's requirements for quality, execution time, and worst-instance performance. Explicit efficiency and robustness objectives remain necessary with either backbone. These cross-model results do not isolate the effect of reasoning itself.

E BASELINES AND LICENSES

To make the comparisons traceable, we distinguish reference solutions from baseline methods and specify the source of each reported result. The license summary then records the terms associated with the upstream resources.

Reference solutions. To compute constructive gaps in Table 1, we use LKH-3 tours for TSP (Helsgaun, 2017) and OR-Tools solutions for KP (Perron & Furnon, 2025), following Zheng et al. (2025). GLS gaps in Table 7 also use LKH reference costs. For TSPLIB, Table 10 instead uses the published optimal tour costs (Reinelt, 1991).

Manually designed heuristics. The hand-designed baselines match the solver framework: Greedy Construct for TSPC and KP, standard ACO (Dorigo et al., 2006) for the four ACO tasks, and Best Fit and First Fit (Johnson et al., 1974) for online BPP. KGLS (Arnold & Sörensen, 2019) provides the GLS comparison, using TSP results reported by Zheng et al. (2025).

Neural combinatorial optimization baselines. For neural comparisons, POMO (Kwon et al., 2020) supplies the construction baseline and DeepACO (Ye et al., 2023) the ACO baseline. The GLS results in Table 7 include VRP-DACT (Ma et al., 2021), NeuOpt (Ma et al., 2023), NeuralGLS (Sui et al., 2024), and GNNGLS (Hudson et al., 2022). Their gaps are taken from Table 8 of Zheng et al. (2025), whose NeuralGLS and GNNGLS entries originate from Ye et al. (2024).

LLM-based AHD baselines. The LLM-based comparisons use EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), HSEvo (Dat et al., 2025), and MCTS-AHD (Zheng et al., 2025) with the task interfaces in Appendix C. FunSearch (Romera-Paredes et al., 2024) is also included for GLS.

Multi-objective AHD baselines. To compare search under the same three objectives, we extend the released MEoH (Yao et al., 2025) and MPaGE (Ha et al., 2026) implementations to include worst-instance quality alongside average quality and runtime. These variants, denoted MEoH-QWE and MPaGE-QWE in Figure 2, use population size 20. MEoH retains its dominance–dissimilarity selection, and MPaGE retains its Pareto-grid selection and semantic clustering. Table 11 additionally compares their original Q, E formulations with PARETOEVO *w/o* W.

E.1 LICENSES

Table 17 records upstream sources and the terms identified on 11 September 2026. The entry "Not specified" means that no explicit license was found at the linked source during that check. MEoH is distributed through LLM4AD, which uses BSD-2-Clause and includes Apache-2.0 components.

Table 17: Upstream sources and licenses for reference solvers, baseline implementations, and TSPLIB. NeuralGLS code can be requested from its authors through the linked publication.

Resource	Type	License / usage terms	URL
LKH-3	Code	Academic and non-commercial use	https://webhotel4.ruc.dk/~keld/research/LKH-3/
OR-Tools	Code	Apache-2.0	https://github.com/google/or-tools
POMO	Code	Not specified	https://github.com/yd-kwon/POMO
DeepACO	Code	MIT	https://github.com/henry-yeh/DeepACO
VRP-DACT	Code	MIT	https://github.com/yining043/VRP-DACT
NeuOpt	Code	MIT	https://github.com/yining043/NeuOpt
NeuralGLS	Code	Not specified	https://doi.org/10.1007/s00521-023-09042-6
GNNGLS	Code	Not specified	https://github.com/proroklab/gnngls
FunSearch	Code	Apache-2.0	https://github.com/google-deepmind/funsearch
EoH	Code	MIT	https://github.com/FeiLiu36/EoH
ReEvo	Code	MIT	https://github.com/ai4co/reevo
HSEvo	Code	MIT	https://github.com/datphamvn/HSEvo
MCTS-AHD	Code	MIT	https://github.com/zz1358m/MCTS-AHD-master
MEoH	Code	BSD-2-Clause	https://github.com/Optima-CityU/LLM4AD
MPaGE	Code	Not specified	https://github.com/langkhachhoha/MPaGE
TSPLIB	Dataset	Not specified	https://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

F AI USE STATEMENT

We did not use generative AI tools for tasks classified as requiring disclosure under the ICLR 2027 AI Policy for Authors. For tasks for which disclosure is recommended, we used these tools to assist with creating or modifying scientific figures and images, including revisions to Figure 1, and with formatting references. We also used generative AI for drafting and language editing, including spelling and grammar checks. Drafting assistance covered portions of the task definitions and general framework descriptions in Appendix C.

We manually reviewed all AI-generated or AI-modified text and visual content for accuracy and consistency with the research, and checked the references for bibliographic accuracy and formatting. We take responsibility for the final text, claims, and artifacts, including content produced with generative AI assistance.