PRISM: REPRESENTATIVE INSTANCE SELECTION FOR LLM-DRIVEN AUTOMATED ALGORITHM DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

A major bottleneck in LLM-based Automated Algorithm Design (AAD) lies in repeatedly evaluating candidate algorithms across a broad instance suite to guide search. However, evaluating candidate algorithms on a full, static suite often creates a major efficiency bottleneck: it incurs heavy computational costs while introducing evaluation noise and structural redundancy that can misguide search. In this paper, we show that full-suite evaluations contain substantial performance redundancy, which can be effectively uncovered by probing with a small set of reference algorithms. Inspired by this observation, we propose PRISM (Probe-guided Representative Instance Selection with Metric-aware feedback), a versatile plug-in for AAD. PRISM operates in two key steps: first, it uses initial algorithm responses as probes to evaluate instance representativeness and select a compact subset that generalizes to unseen algorithms. Second, it adapts the subset evaluations to downstream selection rules via metric-aware feedback, maintaining high selection resolution without requiring extra instance runs. Extensive experiments across diverse combinatorial optimization problems demonstrate that PRISM reduces physical algorithm-instance evaluations by a factor of up to 20 while preserving—and under scalar selection, often improving—search performance compared to full-suite evaluations.

1 INTRODUCTION

Automatic Algorithm Design (AAD) driven by Large Language Models (LLMs) has emerged as a promising paradigm for automated heuristic discovery in combinatorial optimization (Yang et al., 2024; Liu et al., 2024; Romera-Paredes et al., 2024; Ye et al., 2024). By framing algorithm design as evolutionary search over code space, LLM-driven AAD iteratively prompts foundation models to mutate and recombine algorithmic ideas based on empirical execution feedback.

Despite its success, scaling this search paradigm encounters a severe evaluation bottleneck. In complex problem domains, evaluating every LLM-generated candidate algorithm across an extensive instance suite incurs prohibitive computational overhead, limiting search depth and iteration count. To mitigate this cost, existing frameworks commonly resort to heuristic sub-sampling or static instance subsets. However, such ad-hoc practices introduce severe drawbacks: selecting redundant instances confines search to a narrow sub-distribution, whereas arbitrary subsets easily induce preference drift, misdirecting LLM mutations toward local shortcuts that fail to generalize. Crucially, existing approaches overlook a fundamental mechanics: *the choice of evaluation instances implicitly shapes the preference landscape over algorithmic behaviors and directly dictates the search trajectory*.

This challenge demands a distinct formulation beyond conventional benchmark compression or static objective reduction. In LLM-driven AAD, instance selection is particularly hard due to three interconnected challenges:

- **High-dimensional and biased code evolution:** LLM-driven AAD fundamentally differs from traditional optimization problems. Unlike standard solution spaces, the algorithm space is discrete and hard to characterize with explicit features; even minor code modifications can trigger drastic jumps in algorithm behavior. Moreover, as a search operator, the LLM exhibits inherent structural biases and is highly sensitive to input prompts. Due to this

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.

unpredictability, an instance subset selected for early algorithms rapidly loses its predictive accuracy for evolved candidates.

- **Trade-off between generality and specialization:** Because problem instances lack explicit structural features, we cannot straightforwardly identify which instances are broadly representative and which favor niche specialists. This ambiguity inherently forces a difficult trade-off between selecting domain-wide benchmark bases and overfitting instance-specific algorithms.
- **Dynamic distortion of the online fitness landscape:** Instance selection directly alters the input domain of downstream selection criteria, thereby distorting the selection pressure required to guide effective search trajectories.

To address these challenges, we propose **P**robe-guided **R**epresentative **I**nstance **S**election with **M**etric-aware feedback, a general, plug-and-play evaluation layer that seamlessly integrates into scalar- and set-based LLM-driven AAD pipelines. PRISM operates in two tightly coupled stages without altering the host's underlying mutation or crossover operators. First, to overcome behavioral drift and filter out instance-specific specialists without manual feature engineering, we utilize initial search traces as probing signals and formulate instance selection as a Leave-One-Algorithm-Out (LOAO) response matrix reconstruction problem. Minimizing this LOAO reconstruction error selects an instance basis that optimizes predictive generalization for unseen algorithms while filtering out redundant, non-representative preferences. Second, to counteract the selection pressure drop induced by dimension reduction, we introduce a metric-aware feedback adapter. By dynamically re-mapping compressed score vectors through preference-direction sampling and projection, the adapter restores full-suite selection dynamics back to the reduced evaluation space without incurring extra executions.

Our primary contributions are summarized as follows:

- **Formulation & Framework:** We identify the selection drift challenge in reduced-suite AAD and propose PRISM, a general evaluation framework that formulates instance selection as a Leave-One-Algorithm-Out matrix reconstruction problem. By extracting representative instance bases directly from execution dual-profiles, PRISM eliminates manual feature engineering while preserving full-suite trajectory fidelity.
- **Selection-Aware Metric Calibration:** We design a metric-aware feedback adapter that mitigates signal sparsity in dimension-reduced spaces, preserving full-suite selection dynamics and continuous search pressure for host AAD frameworks.
- **Empirical Validation:** Across three optimization tasks, PRISM achieves an average $22.8\times$ speedup in evaluation overhead while matching or exceeding full-suite search quality and demonstrating strong cross-distribution transferability. Crucially, under equal evaluation budgets, PRISM consistently outperforms full-suite baselines by enabling deeper algorithmic exploration.

2 Related Work

LLM-driven Automated Algorithm Design (AAD). LLM-driven AAD combines program generation with evolutionary frameworks to discover executable heuristics in combinatorial optimization (Liu et al., 2024; Romera-Paredes et al., 2024; Ye et al., 2024; Dat et al., 2025; Zheng et al., 2025; Hu & Zhang, 2026). These frameworks rely on repeated candidate evaluations across instance suites to guide code mutation. While effective, existing approaches typically evaluate candidates on fixed or unguided instance suites, creating a severe computational evaluation bottleneck.

Instance-aware Algorithm Design & Adaptive Environments. Traditional algorithm configuration uses handcrafted structural features or co-evolution to identify instance-specific algorithm strengths (Liu et al., 2019; Tang et al., 2021; Smith-Miles & Muñoz, 2023). Recently, LLM-driven AAD has integrated instance-level signals to evolve specialized heuristics (Zhang et al., 2025; Liu et al., 2026b; Hu et al., 2026) or construct adversarial curricula via co-evolving instance generators (Karimi et al., 2025; Duan et al., 2025; Gungordu et al., 2026; Liu et al., 2026a). However, these methods alter evaluation environments to force specialist trajectories or robustness, rather than preserving full-suite search dynamics on a compact evaluation basis.

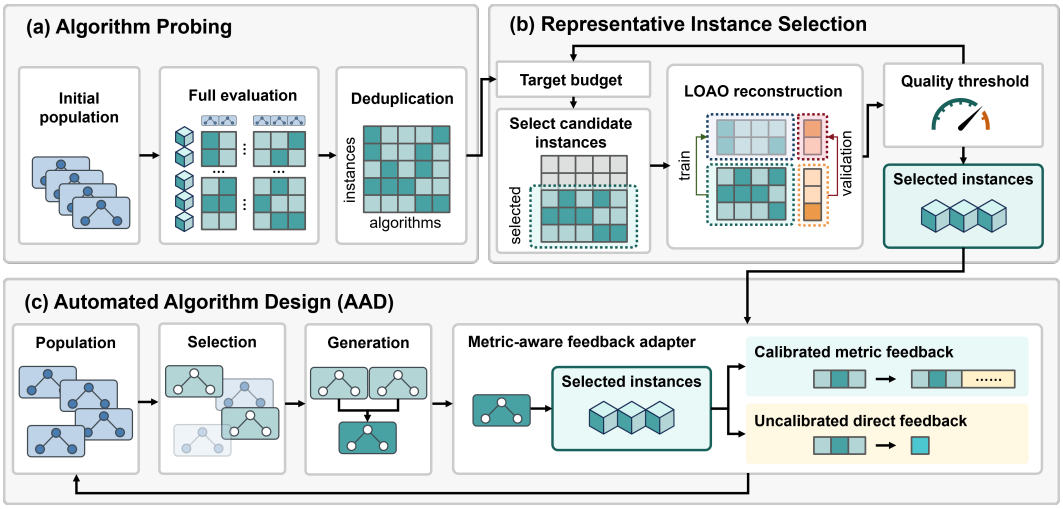


Figure 1: Overview of PRISM within automated algorithm design (AAD). (a) Initial algorithms are evaluated on the full instance suite to obtain a response matrix, with instances as rows and algorithms as columns. (b) Leave-one-algorithm-out reconstruction error guides representative instance selection; the retained subset is determined by an instance budget or a reconstruction-error tolerance. (c) During the subsequent AAD search, new algorithms are evaluated on the retained instances. Metric-aware feedback uses the framework's scalar aggregation rule or augments the preference directions used for CPI-based set selection. Both branches reuse the same observed scores; preference augmentation requires no additional instance evaluations.

Instance, Feature, and Objective Reduction. Evaluation compression connects to three reduction paradigms: (1) *instance selection* via racing or static benchmarking (López-Ibáñez et al., 2016; Polo et al., 2024; Zhang et al., 2026); (2) *feature selection* using domain-specific properties (Bowyer et al., 2026); and (3) *objective selection* preserving Pareto relations (Brockhoff & Zitzler, 2009). Unlike static dataset reduction or feature-based selection, AAD evaluation sits at the dynamic intersection of all three: it requires high algorithmic resolution without domain features to capture code drifts, alongside preference alignment to preserve continuous selection pressure.

3 Methodology

3.1 Overview

We propose **PRISM** (**P**robe-guided **R**epresentative **I**nstance **S**election with **M**etric-aware feedback), a plug-in that curates informative instance subsets and aligns their evaluation signals with downstream selection rules in automated algorithm design (AAD). By integrating instance reduction directly with the host framework's selection mechanics, PRISM cuts evaluation overhead while preserving the necessary guidance for search.

As illustrated in Figure 1, PRISM operates in three stages. **Algorithm probing** evaluates the initial algorithms on the full suite to construct a response matrix (panel a). **Representative instance selection** builds a nested greedy path using leave-one-algorithm-out (LOAO) reconstruction error, stopping at a prescribed instance budget or error tolerance (panel b). **Metric-aware feedback** converts the selected-instance scores into the host framework's selection signal (panel c). For scalar selection, this is an aggregate score; for set selection, we use the Complementary Performance Index (CPI) (Liu et al., 2026b), which averages the best performance available in an algorithm set across evaluation criteria. An algorithm's marginal contribution is the change in this set-level index when it is added.

3.2 Algorithm Probing: Constructing the Response Matrix

Assuming an instance pool $\mathcal{I} = \{x_i\}_{i=1}^{m}$ and an initial algorithm set $\mathcal{A}_0$ generated by the host AAD framework, PRISM views algorithms and instances through a dual lens: just as instances

evaluate algorithms, an algorithm's performance across instances reveals the structural properties of the instance suite. PRISM thus uses $\mathcal{A}_0$ to map this underlying space. Evaluating these initial algorithms across all instances yields their raw performance profiles. To eliminate redundant probing signals and ensure an informative representation, duplicate or non-informative performance profiles are deduplicated, leaving n distinct algorithms. This probing step yields the performance matrix $\mathbf{R} \in \mathbb{R}^{m \times n}$, where the element R_{ia} is the score of algorithm h_a on instance x_i. Each column $\mathbf{r}^a = \mathbf{R}_{:a}$ contains the scores of algorithm h_a across all instances, and each row $\mathbf{r}_i = \mathbf{R}_{i:}$ represents an instance as an n-dimensional vector in the algorithm space. High correlations between these row vectors indicate strong redundancy among instances. Rather than evaluating the entire suite, this structure suggests that a small subset of row vectors can form a "basis" that spans the full performance space—a target we formalize next.

3.3 Representative Instance Selection

Given the response matrix $\mathbf{R} \in \mathbb{R}^{m \times n}$ constructed from n reference algorithms across m initial instances, our goal is to select a compact subset of k instances ($k \ll m$) that captures the performance characteristics of the full suite. This process involves two essential questions: *how to measure the representativeness of a given subset*, and *how to determine the appropriate subset size* k.

Measuring Subset Quality via LOAO Reconstruction. A high-quality instance subset should allow the unselected instance evaluations to be accurately reconstructed from the selected ones, even for algorithms not seen during subset fitting. To evaluate a candidate instance subset $\mathcal{J} \subset \{1, \ldots, m\}$ of size $t = |\mathcal{J}|$, we compute a Leave-One-Algorithm-Out (LOAO) linear reconstruction error. Specifically, for each held-out algorithm $a \in \{1, \ldots, n\}$, we remove its column from $\mathbf{R}$ to form $\mathbf{R}^{-a} \in \mathbb{R}^{m \times (n-1)}$. Let $\bar{\mathcal{J}}$ denote the complement of $\mathcal{J}$. We fit a ridge regression mapping $\mathbf{W}_{\mathcal{J}}^{-a}$ from the selected rows $\mathbf{R}_{\mathcal{J},:}^{-a}$ to predict the unselected rows $\mathbf{R}_{\bar{\mathcal{J}},:}^{-a}$:

$$\mathbf{W}_{\mathcal{J}}^{-a} = \arg\min_{\mathbf{W} \in \mathbb{R}^{(m-t) \times t}} \left\| \mathbf{R}_{\bar{\mathcal{J}},:}^{-a} - \mathbf{W}\mathbf{R}_{\mathcal{J},:}^{-a} \right\|_F^2 + \lambda \|\mathbf{W}\|_F^2, \tag{1}$$

where $\lambda > 0$ is a ridge penalty coefficient determined via internal validation. Using the learned mapping $\mathbf{W}_{\mathcal{J}}^{-a}$, we predict the unselected scores of the held-out algorithm a as $\widehat{\mathbf{r}}_{\bar{\mathcal{J}}}^{a} = \mathbf{W}_{\mathcal{J}}^{-a} \mathbf{r}_{\mathcal{J}}^{a}$, while keeping the selected entries exact. Averaging over all held-out algorithms yields the full-suite reconstruction error:

$$E(\mathcal{J}) = \sqrt{\frac{1}{mn} \sum_{a=1}^{n} \left\| \mathbf{r}_{\bar{\mathcal{J}}}^{a} - \widehat{\mathbf{r}}_{\bar{\mathcal{J}}}^{a} \right\|_2^2}. \tag{2}$$

Intuitively, minimizing $E(\mathcal{J})$ encourages the selected subset $\mathcal{J}$ to faithfully reflect the algorithm performance gaps observed across the full suite, thereby preserving a reliable selection signal for downstream search. Consequently, $E(\mathcal{J})$ provides a principled objective for the greedy instance selection process described next.

Greedy Forward Selection Path. Finding the globally optimal subset of size t is NP-hard. We thus construct a sequence of nested subsets using greedy forward selection. Starting from an empty set $\mathcal{J}_0 = \varnothing$, at each step t, we iteratively append the instance that minimizes the LOAO reconstruction error:

$$j_t = \arg\min_{i \notin \mathcal{J}_{t-1}} E(\mathcal{J}_{t-1} \cup \{i\}), \qquad \mathcal{J}_t = \mathcal{J}_{t-1} \cup \{j_t\}. \tag{3}$$

This yields a solution path $\mathcal{J}_1 \subset \mathcal{J}_2 \subset \cdots \subset \mathcal{J}_m$, where each prefix $\mathcal{J}_t$ represents the candidate subset for budget t.

Determining the Subset Size k. The solution path provides a trade-off curve between evaluation budget t and reconstruction error $E(\mathcal{J}_t)$. PRISM supports two modes to determine the final subset size k: (a). Fixed Budget: If a rigid evaluation budget k is specified, PRISM directly returns $\mathcal{J}_k$. (b). Quality-Driven: If an error tolerance ϵ is specified, PRISM selects the smallest subset that meets the target reconstruction quality: $k = \min\{t \mid E(\mathcal{J}_t) \leq \epsilon\}$. This identifies the minimum number of instances required to retain sufficient evaluation signal, effectively establishing a principled lower bound on k.

Once $\mathcal{J}_k$ is determined, the probe-level reconstruction mappings $\mathbf{W}$ are discarded. During downstream search, we evaluate candidate algorithms directly using their actual execution scores on $\mathcal{J}_k$, rather than reconstructing full-suite scores via $\mathbf{W}$. This avoids the prediction bias caused by applying a static linear model to dynamically evolving algorithms. The problem of effectively transforming the observed subset scores into global guidance signals for algorithm search is subsequently addressed by our metric-aware feedback mechanism in section 3.4.

3.4 METRIC-AWARE FEEDBACK

The ultimate goal of instance selection is to compress the evaluation budget without distorting the selection preferences and selection pressure of the full-suite evaluation metrics. However, many downstream evaluation metrics (e.g., composite performance indicators or rank-based selections) are inherently sensitive to instance suite size; applying them directly to a reduced subset $\mathcal{J}_k$ often causes scale distortion or alters the intended selection bias. To address this, PRISM incorporates a *metric-aware feedback* mechanism that calibrates the observed subset performances back to the selection dynamics of the full suite.

Specifically, for each candidate algorithm h generated during search, PRISM executes it strictly on the selected instances in $\mathcal{J}_k$, obtaining an observed score vector $\mathbf{y}(h) \in \mathbb{R}^k$. As illustrated in fig. 1(c), PRISM then processes these same k observations through two metric-aware branches to handle scalar and set-based selection, respectively.

Uncalibrated Direct Feedback. When the downstream selection mechanism relies on metrics that are invariant to instance suite size (e.g., standard linear aggregations), PRISM directly applies the original evaluation rule over $\mathbf{y}(h)$:

$$\widehat{f}(h) = \boldsymbol{\omega}^\top \mathbf{y}(h), \qquad \boldsymbol{\omega} \geq \mathbf{0}, \qquad \mathbf{1}^\top \boldsymbol{\omega} = 1. \tag{4}$$

In this pathway, the aggregation operator remains unchanged; the evaluation budget is saved solely by restricting execution to $\mathcal{J}_k$. This corresponds to the "Direct Integration" branch in fig. 1(c).

Calibrated Metric Feedback. When algorithms are selected based on fitness profiles across instances—rather than simple scalar aggregations—reducing the instance suite can severely distort the underlying selection pressure. A prominent representative of this category is the Composite Performance Indicator (CPI) used in search frameworks like EoH-S, which evaluates the marginal contribution of an algorithm within an algorithm portfolio $\mathcal{S}$. On the full instance suite, CPI measures portfolio performance by taking the best instance score across $\mathcal{S}$:

$$\text{CPI}(\mathcal{S}) = \frac{1}{m} \sum_{j=1}^{m} \min_{h \in \mathcal{S}} \mathbf{r}_j(h). \tag{5}$$

If applied directly to the reduced subset $\mathcal{J}_k$, CPI degenerates into evaluating performance along only k coordinate directions $\Omega_0 = \{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$. Consequently, the number of algorithms that can yield non-zero marginal contributions to CPI is strictly capped by at most k. This artificial reduction in evaluation capacity severely compresses the selection pressure, leaving many high-potential trade-off algorithms unrewarded simply because there are insufficient coordinate dimensions to reflect their contributions.

To restore the full-suite selection dynamics under a reduced budget, PRISM introduces a metric-aware calibration for CPI. We retain the coordinate directions Ω_0 and augment them with a set of M interior preference directions $\Omega_{\text{int}} = \{\boldsymbol{\omega}_1, \ldots, \boldsymbol{\omega}_M\}$ sampled uniformly from the $(k-1)$-simplex Δ^{k-1}:

$$\Delta^{k-1} = \left\{ \boldsymbol{\omega} \in \mathbb{R}^k \;\middle|\; \boldsymbol{\omega} \geq \mathbf{0},\ \mathbf{1}^\top \boldsymbol{\omega} = 1 \right\}. \tag{6}$$

The calibrated score for candidate h within set $\mathcal{S}$ is then computed over the combined directional set $\Omega = \Omega_0 \cup \Omega_{\text{int}}$:

$$\text{CPI}_{\text{calibrated}}(\mathcal{S}) = \frac{1}{|\Omega|} \sum_{\boldsymbol{\omega} \in \Omega} \min_{h \in \mathcal{S}} \left(\boldsymbol{\omega}^\top \mathbf{y}(h) \right). \tag{7}$$

By assigning positive weights across multiple instance scores, these sampled preference directions expose interior trade-offs. This effectively prevents selection drift and preserves the intended full-suite selection pressure for vector- and set-based evaluation protocols.

3.5 Complete Procedure and Evaluation Cost

Algorithm 1 summarizes the complete integration of PRISM within an AAD framework. While the host framework manages candidate generation and population updates, PRISM exclusively optimizes instance evaluation and score feedback.

Algorithm 1 PRISM-Accelerated Automated Algorithm Design (AAD)

Require: Full instance suite $\mathcal{I}$ ($m = |\mathcal{I}|$); probe algorithm set $\mathcal{A}_{\text{probe}}$ ($n = |\mathcal{A}_{\text{probe}}|$); budget k or tolerance ϵ; search budget T.
Ensure: Optimized algorithm h^* or algorithm portfolio $\mathcal{S}^*$.

// Phase 1: Offline Subset Selection

$\mathbf{R} \leftarrow \text{Evaluate}(\mathcal{A}_{\text{probe}}, \mathcal{I})$ ▷ $\mathbf{R} \in \mathbb{R}^{m \times n}$ constructed from probe runs
$\mathcal{J}_0 \leftarrow \emptyset, \quad \bar{\mathcal{J}}_0 \leftarrow \{1, \ldots, m\}$
for $t = 1, 2, \ldots, m$ **do**
  $j^* \leftarrow \arg\min_{j \in \bar{\mathcal{J}}_{t-1}} E(\mathcal{J}_{t-1} \cup \{j\})$ ▷ Evaluate $E(\mathcal{J})$ via eq. (2)
  $\mathcal{J}_t \leftarrow \mathcal{J}_{t-1} \cup \{j^*\}, \quad \bar{\mathcal{J}}_t \leftarrow \bar{\mathcal{J}}_{t-1} \setminus \{j^*\}$
end for
$k \leftarrow \text{SelectSize}(\{E(\mathcal{J}_t)\}_{t=1}^m; k, \epsilon)$ ▷ Determine k via fixed budget or tolerance ϵ
$\mathcal{J}_k \leftarrow \text{ExtractSubset}(\{\mathcal{J}_t\}, k)$

// Phase 2: Online AAD Search Dynamics

$\mathcal{S} \leftarrow \text{InitializePopulation}()$
for $step = 1, 2, \ldots, T$ **do**
  $h \leftarrow \text{GenerateCandidate}(\mathcal{S})$ ▷ LLM/Search mutation operator
  $\mathbf{y}(h) \leftarrow \text{Evaluate}(h, \mathcal{J}_k)$ ▷ Execute strictly on reduced suite $\mathcal{J}_k \in \mathbb{R}^k$
  $\text{feedback}(h) \leftarrow \text{ComputeFeedback}(\mathbf{y}(h))$ ▷ Apply eq. (4) or eq. (7)
  $\mathcal{S} \leftarrow \text{UpdatePopulation}(\mathcal{S}, h, \text{feedback}(h))$
end for
return h^* or $\mathcal{S}^* \leftarrow \text{ExtractBest}(\mathcal{S})$

Evaluation Cost Analysis. Evaluating T candidate algorithms on the full suite requires $(n + T)m$ instance runs, whereas PRISM reduces this cost to $nm + Tk$, saving $T(m - k)$ evaluations. The one-time subset selection runs on cached probe matrices without extra instance evaluations, and calculating calibrated feedback during search adds negligible arithmetic overhead ($O(k|\Omega|)$ per candidate). Consequently, this evaluation reduction directly yields substantial wall-clock speedups whenever instance execution dominates computation.

4 Experiments

4.1 Experimental Settings

Tasks and benchmarks. We evaluate PRISM on three representative NP-hard combinatorial optimization problems: Online Bin Packing (OBP), the Traveling Salesman Problem (TSP), and the Capacitated Vehicle Routing Problem (CVRP). For OBP, methods design online packing heuristics, evaluated by the mean optimality gap relative to theoretical lower bounds. For TSP and CVRP, methods design constructive heuristics, evaluated by the mean optimality gap relative to the state-of-the-art exact/strong solver LKH-3 Helsgaun (2017). Lower values indicate superior performance.

OBP utilizes $m = 128$ training instances with sizes ranging from 200 to 2,000 items, nine standard held-out test sets, and six mixed-distribution stress-test sets. TSP uses 128 training instances, whereas CVRP uses 256 training instances. Both evaluate cross-scale generalization on unseen problem scales of 50, 100, 200, and 500 nodes, with 16 (TSP) and 32 (CVRP) generated test instances per scale. Detailed instance configurations and generation pipelines are provided in section A.

We adopt Cumulative Performance Improvement (CPI) as our primary metric because it simultaneously measures solution performance and portfolio diversity, thereby directly testing whether PRISM preserves algorithmic diversity under instance compression.

Baselines. We compare PRISM against two major categories of baselines: (1) **Host AAD frameworks**: EoH (scalar-fitness) and EoH-S (set-based fitness), which serve as our primary evaluation backbones to quantify the acceleration and performance gap of PRISM against standard full-suite training ($m = 128$ for OBP/TSP and $m = 256$ for CVRP). (2) **Standalone AAD baselines**: FunSearch and ReEvo, evaluated under standard full-suite training to provide external competitive references.

AAD framework settings and budget accounting. All search runs employ `gpt-5.6-luna` with temperature 1.0, top-p 1.0, and a total budget of $T = 2{,}000$ candidate algorithm samplings per run. The algorithm population size is standardized to 10 across all frameworks. All reported metrics are averaged over 3 independent random seeds unless explicitly stated otherwise.

Standard full-suite AAD frameworks require 256,000 instance evaluations for OBP/TSP and 512,000 for CVRP. By evaluating only 10 probing algorithms on the full training pool and the remaining candidates on a selected subset of $k = 5$ instances, PRISM reduces the total evaluation cost to 11,230 for the three tasks—achieving a $22.8\times$ and $40.9\times$ speedup, respectively.

4.2 Main Results

In-distribution and shifted OBP evaluation. Table 1 presents the comparative results on OBP across training, testing, and mixed-distribution stress tests. Using only 4.4% of the evaluation budget ($22.8\times$ speedup), PRISM achieves competitive or superior performance across both host frameworks. Specifically, PRISM+EoH significantly outperforms vanilla EoH on all test categories, reducing the average optimality gap from 0.0748 to 0.0525 (a 29.8% relative improvement). Meanwhile, PRISM+EoH-S retains 97.2% of full-suite EoH-S performance (0.0473 vs. 0.0461 mean gap), ranking top-two across all 15 benchmark suites.These results reflect distinct dynamics in the two frameworks. For scalar EoH, full-suite evaluation suffers from redundant instances that noise up fitness calculations; PRISM's instance selection acts as an implicit regularizer to enhance generalization. For set-based EoH-S, PRISM's subset preserves essential selection pressure along primary evolutionary directions, sacrificing only minor fringe-instance adaptations for massive efficiency gains.

Table 1: **Online Bin Packing (OBP) optimality gaps** across training-scale, held-out scale/capacity, and mixed-distribution stress tests. Results are reported as mean optimality gaps relative to lower bounds over 3 runs. The overall best results are highlighted in **bold**, and the second-best results are underlined. Baseline methods (EoH, ReEvo, and FunSearch) evaluate the single best-discovered algorithm, whereas their Top 10 variants evaluate the top 10 discovered algorithms, respectively. PRISM+EoH and PRISM+EoH-S denote our proposed framework integrated into scalar and set-based backbones AAD frameworks.

	Training			Standard held-out testing						Mixed-distribution testing					
Method	200/500	500/1k	1k/2k	1k/200	1k/500	5k/200	5k/500	10k/200	10k/500	Mix-1	Mix-2	Mix-3	Mix-4	Mix-5	Mix-6
FunSearch	.0623	.0298	.0367	.0213	**.0124**	.0159	.0070	.0152	.0051	.0602	.1379	.0806	.1182	.1340	.0594
FunSearch Top10	.0608	.0289	.0361	.0200	**.0124**	.0153	.0065	.0123	.0038	.0568	.1361	.0783	.1136	.1304	.0571
ReEvo	.0604	.0298	.0362	.0217	.0133	.0162	.0068	.0155	.0053	.0584	.1368	.0783	.1167	.1336	.0594
ReEvo Top10	.0580	.0285	.0349	.0203	**.0124**	.0125	.0063	.0108	.0049	.0569	.1352	.0769	.1134	.1306	.0561
EoH	.0619	.0307	.0371	.0240	.0133	.0182	.0075	.0170	.0059	.0590	.1382	.0816	.1217	.1350	.0597
EoH Top10	.0618	.0299	.0368	.0216	**.0124**	.0158	.0065	.0152	.0051	.0583	.1377	.0813	.1154	.1327	.0582
PRISM+EoH	.0568	.0278	.0338	.0180	**.0124**	.0143	.0060	.0134	.0047	.0569	.1335	.0735	.1120	.1324	.0531
EoH-S	**.0515**	**.0230**	**.0314**	**.0113**	**.0124**	.0033	**.0025**	**.0014**	**.0010**	**.0499**	**.1287**	**.0681**	**.1045**	**.1226**	**.0475**
PRISM+EoH-S	.0564	.0260	.0327	.0130	**.0124**	**.0028**	**.0025**	.0015	**.0010**	.0510	.1314	.0711	.1095	.1246	.0477

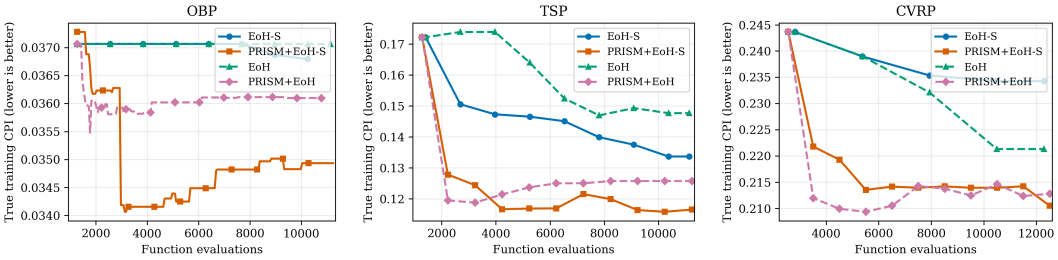
TSP and CVRP cross-scale generalization. Table 2 presents the generalization performance on graph-based routing problems (TSP and CVRP). Table 1 details the baselines and PRISM variants. Across all three combinatorial domains (OBP, TSP, and CVRP), integrating PRISM consistently preserves or even surpasses full-suite performance while consuming only a fraction of the computational budget. Specifically, PRISM+EoH-S remains highly competitive with full-suite EoH-S across all unseen problem scales, achieving superior generalization on two TSP test sets and the largest CVRP scale. While performance retention on CVRP decreases to 83.18%, we attribute this to the evaluation disparity: the full benchmark suite comprises 256 training instances, whereas PRISM operates under

Table 2: TSP and CVRP optimality gaps.

	TSP–Training	TSP–Testing				CVRP–Training	CVRP–Testing			
Method	10–200	50	100	200	500	20–200	50	100	200	500
FunSearch	.1563	.1319	.1503	.1726	.2069	.2940	.3182	.3421	.2467	.1911
FunSearch Top10	.1241	.0931	.1217	.1423	.1588	.2454	.2572	.2931	.2055	.1827
ReEvo	.1588	.1296	.1467	.1649	.1717	.2661	.3026	.3186	.2281	.1762
ReEvo Top10	.1190	.0836	.1066	.1315	.1498	.2428	.2743	.2903	.2056	.1607
EoH	.1525	.1328	.1481	.1663	.1939	.2765	.2838	.3101	.2426	.2054
EoH Top10	.1282	.0985	.1259	.1447	.1699	.2303	.2166	.2664	.2060	.1743
PRISM+EoH	.1259	.0845	.1207	.1238	.1511	.2128	.2078	.2573	.2033	.1789
EoH-S	**.1006**	**.0674**	.0932	**.1221**	.1479	**.1754**	**.1614**	**.2066**	**.1686**	.1535
PRISM+EoH-S	.1166	.0730	**.0916**	.1240	**.1406**	.2105	.2072	.2475	.1990	**.1525**

a fixed $k = 5$ budget for constrained demonstration. Overall, these results confirm that PRISM's instance selection effectively retains core heuristic evaluation signals, achieving robust cross-domain transferability without task-specific tuning.

4.3 Performance under Fixed Function Evaluations



(a) OBP (b) TSP (c) CVRP

Figure 2: Search convergence trajectories under a fixed function-evaluation budget.

To evaluate search efficiency when candidate generation is cheaper than evaluation, we compare search dynamics under a fixed budget of total function evaluations (FEs). As shown in fig. 2, PRISM lowers per-candidate evaluation costs, allowing more candidates to be explored within the same FE budget. Furthermore, PRISM-based variants rapidly optimize the true training CPI across OBP, TSP, and CVRP, confirming that the selected instance subset accurately represents the full training suite without optimization bias.

Table 3: Optimality gaps under the fixed function-evaluation budgets

	OBP–Testing						TSP–Testing				CVRP–Testing			
Method	1k/200	1k/500	5k/200	5k/500	10k/200	10k/500	50	100	200	500	50	100	200	500
EoH	.0200	**.0124**	.0117	.0070	.0115	.0050	.1047	.1252	.1507	.1518	**.2035**	.2741	.2054	.1713
PRISM+EoH	.0180	**.0124**	.0143	.0060	.0134	.0047	.0845	.1207	**.1238**	.1511	.2078	.2573	.2033	.1789
EoH-S	.0140	**.0124**	.0034	**.0025**	.0016	**.0010**	.0861	.1247	.1470	.1571	.2304	.2861	.2147	.1740
PRISM+EoH-S	**.0130**	**.0124**	**.0028**	**.0025**	**.0015**	**.0010**	**.0730**	**.0916**	.1240	**.1406**	.2072	**.2475**	**.1990**	**.1525**

This exploration efficiency leads to better test performance. As shown in table 3, PRISM+EoH-S outperforms or matches EoH-S across all test sets, achieving the best result in 12 of 14 cases. Similarly, PRISM+EoH improves upon baseline EoH on most test sets. These results indicate that PRISM effectively reduces evaluation overhead while maintaining a reliable search signal.

4.4 Ablation Study

To isolate the individual algorithmic contributions within PRISM, we evaluate three core components: (1) **Instance selection strategy**, (2) **Leave-One-Algorithm-Out (LOAO) subset scoring**, and (3)

Metric-aware feedback. To ensure strict and uniform benchmarking, all ablation variants in table 4 are evaluated post-search on a held-out OBP dataset.

Table 4: Ablation study of core PRISM components on OBP

Component	Variant	Description	Training ↓	Testing ↓
Instance selection	Random-5	5 randomly selected instances	0.02078	0.050582
	k-EMOSS Brockhoff & Zitzler (2009)	5 instances preserving Pareto ordering	0.01823	0.047243
	Resolution-aware	5 instances discriminating initial algorithms	0.01623	0.046576
LOAO scoring	w/o LOAO	Reconstructs initial response matrix directly	0.02046	0.049221
Metric feedback	w/o Metric Feedback	Uses fitness vector directly for CPI computation	0.01761	0.046605
Full Framework	PRISM+EoH-S (Ours)	Full PRISM framework with all components	**0.01592**	**0.044858**

Impact of instance selection strategy. Arbitrary instance selection severely degrades search-time selection pressure, compromising algorithm discovery. As shown in table 4, naive Random-5 yields the worst testing CPI (0.050582), while static heuristics like k-EMOSS (0.047243) and Resolution-aware (0.046576) improve performance but remain substantially inferior to our full framework (0.044858). We attribute this to an inherent search-time bias: static criteria overfit to initial algorithms and fail to maintain informative selection pressure as new candidate populations emerge. In contrast, our response-matrix formulation profiles intrinsic algorithm characterizations while continuously preserving sharp discriminatory pressure across evolutionary trajectories.

Impact of LOAO scoring. Direct subset scoring introduces strong instance-level overfitting that harms out-of-distribution generalization. Evaluating subset fitness directly without LOAO (w/o LOAO) results in a degraded testing CPI of 0.049221. Omitting LOAO causes the search to overfit to specialist instances that yield high initial reconstruction quality but lack broad representativeness.

Impact of metric-aware feedback. Metric-aware feedback is necessary for instance-sensitive selection criteria to maintain fine-grained selection pressure. As shown in table 4, evaluating raw fitness directly (w/o Metric Feedback) exhibits performance degradation on testing sets . Metric-aware feedback eliminates this sparsity by incorporating multi-objective preference vectors, allowing trade-off candidates to actively contribute to the aggregate CPI.

We also evaluate the sensitivity to the selected instance size k and the number of feedback directions D. Results indicate that performance remains consistent across different values of k beyond a minimal threshold, whereas increasing D monotonically improves performance before rapidly saturating. Detailed results are provided in section D.

5 Conclusions

In this work, we address the severe evaluation bottleneck in LLM-driven AAD by introducing PRISM, a flexible, plug-and-play evaluation layer. PRISM focuses on identifying a compact, representative instance basis that preserves the algorithm selection dynamics of the full evaluation suite. By unifying Leave-One-Algorithm-Out reconstruction with selection-metric-aware feedback adapters, PRISM accurately estimates candidate algorithm performance on a reduced instance set while minimizing distortion to selection pressure. Extensive experiments across OBP, TSP, and CVRP demonstrate that PRISM achieves an average $22.8\times$ speedup in evaluation overhead while matching or exceeding full-suite search quality and exhibiting strong cross-distribution transferability.

Future work includes exploring the high-dimensional instance–algorithm space and gaining deeper insights into the evolutionary dynamics of LLM-driven AAD. Beyond these, promising directions include automatically selecting instances tailored to different preferences to enhance algorithm diversity, dynamically adapting instances to steer search dynamics, and automated searching for optimal instance selection strategies.

AI USE STATEMENT

In this work, we used generative AI tools (OpenAI Codex and Gemini) for proposing or refining hypotheses, implementing methods and designing or providing feedback on research methodology. We have not used generative AI tools for generating synthetic datasets, and all other tasks with required disclosure (such as formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, assisting with translation, cleaning and reformating datasets, supporting qualitative/thematic data analysis, or interpreting results) are not applicable to this work.

Additionally, we used generative AI tools for brainstorming, creating or editing software code (utility and launch scripts), summarizing or analyzing existing literature, and editing the research paper to improve readability; we have not used generative AI tools for other tasks with recommended disclosure.

We have thoroughly reviewed all AI-assisted work. Specifically: (1) all primary research ideas, theoretical logic, and evolution frameworks were conceived and directed by human researchers, while AI was used purely to assist with brainstorming and method refinement; (2) all core search, evolution algorithms, and experimental frameworks were developed by the authors based on prior codebases, whereas AI-generated code was strictly limited to auxiliary execution interfaces and launch scripts, all of which were verified and tested for correctness by the authors; (3) all experimental raw data, CSV results, and plot figures were strictly collected from actual physical code executions rather than generated or modified by AI, with human authors conducting standard sanity checks on figure trends and summary statistics; and (4) all AI-assisted literature summaries and citations were cross-checked against original publications by the authors to prevent potential hallucinations. We take full responsibility for the final content of this work, including text, claims, code, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Experimental details including hyperparameters, instance generation procedures, and evaluation protocols are described in Section 4 and the appendix. The complete algorithm is provided as pseudocode in Section E. Our source code is anonymously available at `https://anonymous.4open.science/r/PRISM-6D80/`.

REFERENCES

Sam Bowyer, Acyr Locatelli, and Kris Cao. Efficient benchmarking is just feature selection and multiple regression. *arXiv preprint arXiv:2605.25773*, 2026.

Dimo Brockhoff and Eckart Zitzler. Objective reduction in evolutionary multiobjective optimization: Theory and applications. *Evolutionary computation*, 17(2):135–166, 2009.

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

Ruibo Duan, Yuxin Liu, Xinyao Dong, and Chenglin Fan. Ealg: Evolutionary adversarial generation of language model-guided generators for combinatorial optimization. *arXiv preprint arXiv:2506.02594*, 2025.

Oguzhan Gungordu, Siheng Xiong, and Faramarz Fekri. Mosaic: Adversarial co-evolution of specialist heuristics and problem instances for llm-based automated heuristic design. *arXiv preprint arXiv:2608.07544*, 2026.

Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 12(966-980):1, 2017.

Qinglong Hu and Qingfu Zhang. Partition to evolve: Niching-enhanced evolution with llms for automated algorithm discovery. *Advances in Neural Information Processing Systems*, 38:154771–154817, 2026.

Qinglong Hu, Qingfu Zhang, Fei Liu, Xialiang Tong, Kun Mao, and Mingxuan Yuan. Beyond average performance: Dynamic instance clustering and specialized algorithm design in llm-assisted evolutionary search. *arXiv preprint arXiv:2608.03129*, 2026.

Pantea Karimi, Dany Rouhana, Pooria Namyar, Siva Kesava Reddy Kakarla, Venkat Arun, and Behnaz Arzani. Robust heuristic algorithm design with llms. *arXiv preprint arXiv:2510.08755*, 2025.

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. *arXiv preprint arXiv:2401.02051*, 2024.

Fei Liu, Alessio Figalli, Patrick Owen, and Nicola Serra. Raise: Llm-based automated heuristic design with robust adversary instance search. *arXiv preprint arXiv:2606.31801*, 2026a.

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, 2026b. doi: 10.1609/aaai.v40i43.41038.

Shengcai Liu, Ke Tang, and Xin Yao. Automatic construction of parallel portfolios via explicit instance grouping. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 1560–1567, 2019.

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.

Felipe Maia Polo, Lucas Weber, Leshem Choshen, Yuekai Sun, Gongjun Xu, and Mikhail Yurochkin. tinybenchmarks: evaluating llms with fewer examples. *arXiv preprint arXiv:2402.14992*, 2024.

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

Kate Smith-Miles and Mario Andrés Muñoz. Instance space analysis for algorithm testing: Methodology and software tools. *ACM Computing Surveys*, 55(12):1–31, 2023.

Ke Tang, Shengcai Liu, Peng Yang, and Xin Yao. Few-shots parallel algorithm portfolio construction via co-evolution. *IEEE Transactions on Evolutionary Computation*, 25(3):595–607, 2021.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

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. *Advances in neural information processing systems*, 37:43571–43608, 2024.

Shaofeng Zhang, Shengcai Liu, Ning Lu, Jiahao Wu, Ji Liu, Yew-Soon Ong, and Ke Tang. Llm-driven instance-specific heuristic generation and selection. *arXiv preprint arXiv:2506.00490*, 2025.

Taolin Zhang, Hang Guo, Wang Lu, Tao Dai, Shu-Tao Xia, and Jindong Wang. Sparseeval: Efficient evaluation of large language models by sparse optimization. *arXiv preprint arXiv:2602.07909*, 2026.

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

Appendix

Table of Contents

Appendix A Problem Descriptions and Setups
Appendix B OBP Mixed-Distribution Test Sets
Appendix C Prompts for EoH and EoH-S
Appendix D Sensitivity Analysis
Appendix E Designed Heuristics

A Problem Descriptions and Setups

We consider three constructive heuristic design tasks: online bin packing, the traveling salesman problem, and capacitated vehicle routing. For each task, we describe the objective, heuristic interface, and datasets.

A.1 Online Bin Packing (OBP)

Problem definition. Items with sizes $s_1, \ldots, s_n \in (0, C]$ arrive sequentially and must immediately be assigned to bins of capacity C, without observing future items or moving previously packed items. A feasible packing partitions the items into bins $B_1, \ldots, B_K$ satisfying $\sum_{j \in B_k} s_j \leq C$. The objective is to minimize K.

Designed heuristic and evaluation. The function `priority(item, bins)` returns one score per feasible bin, where `bins` contains remaining capacities. The evaluator assigns the item to the bin with the largest score and updates its capacity. It initializes n empty bins, filters infeasible bins before each call, and counts nonempty bins at termination. We compute the relative gap as

$$g(h, i) = \frac{K(h, i) - b_i}{b_i}, \qquad b_i = \left\lfloor \frac{\sum_j s_{ij}}{C_i} + \frac{1}{2} \right\rfloor. \tag{8}$$

Here, b_i is the rounded total item size divided by capacity. Lower gaps indicate better performance; search fitness is the negative gap.

Data. The 128 training instances use capacity 100, item counts sampled uniformly from $\{200, \ldots, 1999\}$, Weibull shape $k \in \{1, 3, 5\}$, and scale $\lambda \in \{5, 10, 20, 40, 80\}$. The generator sets NumPy's seed to $2025 + i$ for zero-based instance index i, samples the count and parameters, draws $\lambda \,\mathrm{Weibull}(k)$ sizes, clips them to $[1, 100]$, and rounds to integers. The nine standard reporting groups comprise three training-scale groups (200–500, 500–1,000, and 1,000–2,000 items) and six held-out combinations of $n \in \{1{,}000, 5{,}000, 10{,}000\}$ and $C \in \{200, 500\}$, with five instances per held-out combination. The additional mixed-distribution groups are described in section B.

A.2 Traveling Salesman Problem (TSP)

Problem definition. Let $V = \{0, \ldots, n-1\}$ be cities with coordinates $x_v \in [0, 1]^2$ and Euclidean distances $d_{uv} = \|x_u - x_v\|_2$. A tour is a permutation π of V, with objective

$$\min_{\pi} \; L(\pi) = \sum_{t=1}^{n-1} d_{\pi_t, \pi_{t+1}} + d_{\pi_n, \pi_1}. \tag{9}$$

Each city is visited exactly once before returning to the start.

Designed heuristic and evaluation. The LLM designs `select_next_node` with the four arguments listed in section C.1. Starting at city 0, the evaluator supplies unvisited city IDs in increasing distance order from the current city. The function returns one unvisited city ID; the evaluator appends the final remaining city and closes the tour. The destination is always city 0. The function makes a single next-city decision at each call. For an LKH reference length L_i^{ref}, the reported relative gap is $(L(h, i) - L_i^{\mathrm{ref}})/L_i^{\mathrm{ref}}$.

Data. Training uses the 128 clustered instances: 32 for each combination of cluster count $q \in \{3, 10\}$ and coordinate standard deviation $\sigma \in \{0.03, 0.07\}$. Cluster centers lie in $[0.2, 0.8]^2$, with cities sampled around them. The instances contain 11–197 cities and include LKH reference lengths. We generate 16 test instances at each $n \in \{50, 100, 200, 500\}$ with coordinates sampled uniformly in $[0, 1)^2$. Generation uses a single NumPy random stream seeded at 2024, visiting sizes in ascending order. All methods use the same test instances.

A.3 CAPACITATED VEHICLE ROUTING PROBLEM (CVRP)

Problem definition. A depot 0 and customers V_c have Euclidean coordinates and distances d_{uv}. Customer v has demand $q_v > 0$ and each vehicle has capacity Q. Find routes $\mathcal{R}$ that each start and end at the depot, serve every customer exactly once, and satisfy $\sum_{v \in r \cap V_c} q_v \leq Q$ for every route r. The objective is

$$\min_{\mathcal{R}} \sum_{r \in \mathcal{R}} \sum_{(u,v) \in r} d_{uv}. \tag{10}$$

Designed heuristic and evaluation. The six-argument `select_next_node` function in section C.1 chooses one feasible unserved customer, or the depot to close the current route. The evaluator filters customers by remaining vehicle capacity, updates capacity after service, and returns to the depot automatically when no customer fits. It rejects infeasible outputs and depot-to-depot loops. Route length includes every depot return. As for TSP, performance is measured relative to the corresponding LKH reference length and lower gaps are better.

Data. Training uses 256 instances, with 20–200 nodes including the depot, capacities 10–150, uniform coordinates, and customer demands in $\{1, \ldots, 9\}$. Depot demand is zero. We generate 32 test instances for each node count $n \in \{50, 100, 200, 500\}$, including the depot, using a single NumPy random stream seeded at 2024 across ascending sizes. For each instance, capacity is sampled uniformly from $\{40, \ldots, 149\}$, followed by coordinates in $[0, 1)^2$ and customer demands in $\{1, \ldots, 9\}$. At $n = 500$, evaluation uses the common 30 instances with valid LKH references.

B OBP MIXED-DISTRIBUTION TEST SETS

The six mixed-distribution test sets, Mix-1 through Mix-6 in table 1, each contain five instances drawn from several distribution families. The sets are disjoint and cover a range of item counts, bin capacities, and item-size distributions.

B.1 CANDIDATE-BANK GENERATION

A bank of 300 instances is generated, with 50 instances per family in the order Weibull, normal, lognormal, exponential, beta, and bimodal normal. Parameters come from 300 rows of an eight-dimensional scrambled Latin hypercube, implemented by `scipy.stats.qmc.LatinHypercube` with seed 2026082701. For a row $u = (u_0, \ldots, u_7)$, the item count is

$$n = \text{round}\left(\exp\left[\log 200 + u_0(\log 10000 - \log 200)\right]\right). \tag{11}$$

Within each family, capacities cycle through $(100, 200, 500)$ for the 50 instances, yielding 17, 17, and 16 instances at those capacities. Parameters are linear maps of the designated coordinates to the intervals in table 5.

Table 5: Candidate item-size distributions before clipping and rounding. Size parameters are relative to bin capacity C. Coordinates are zero-indexed.

Family	Sampling rule and parameter ranges
Weibull	$X = \lambda W$, $W \sim \text{Weibull}(k)$; $k \in [0.7, 5.0]$ from u_1, $\lambda/C \in [0.05, 0.80]$ from u_2.
Normal	$X \sim \mathcal{N}(\mu, \sigma^2)$; $\mu/C \in [0.10, 0.90]$ from u_1, $\sigma/C \in [0.03, 0.35]$ from u_2.
Lognormal	$\log X \sim \mathcal{N}(\log(mC), \sigma_{\log}^2)$; median ratio $m \in [0.03, 0.60]$ from u_1, $\sigma_{\log} \in [0.20, 1.50]$ from u_2.
Exponential	$X \sim \text{Exp}(\text{scale} = \lambda)$; $\lambda/C \in [0.05, 0.60]$ from u_1.
Beta	$X = CZ$, $Z \sim \text{Beta}(\alpha, \beta)$; $\alpha, \beta \in [0.5, 8.0]$ from u_1, u_2, respectively.
Bimodal normal	$X \sim (1-w)\mathcal{N}(\mu_l, \sigma_l^2) + w\mathcal{N}(\mu_h, \sigma_h^2)$; $\mu_l/C \in [0.05, 0.45]$, $\mu_h/C \in [0.55, 0.95]$ from u_1, u_2; $\sigma_l/C, \sigma_h/C \in [0.02, 0.15]$ from u_3, u_4; $w \in [0.20, 0.80]$ from u_6.

For candidate index $j \in \{0, \ldots, 299\}$, items are sampled using NumPy's `default_rng` with seed $2026082701000 + j$. Samples are clipped to $[1, C]$, rounded to integers, and kept in their generated order. We retain instances with at least ten distinct item sizes, positive b_i, and at most half their items at the two boundaries combined. Duplicate item sequences are removed. Rejected samples are redrawn with seed $2026082701000 + j + 300r$ at retry r, up to 100 retries.

B.2 Response-based eligibility and grouping

We evaluate each candidate instance using three released EoH-S populations of ten heuristics, for a total of 9,000 algorithm–instance evaluations. Each evaluation uses the OBP construction and gap defined in section A, with Python and NumPy seeds set to zero. An instance is eligible if all 30 evaluations succeed with finite gaps and each population has a unique best heuristic:

$$W_r(i) = \left\{ h : g_{rih} \le \min_{h'} g_{rih'} + 10^{-12} \right\}, \qquad |W_r(i)| = 1 \quad (r = 1, 2, 3). \tag{12}$$

With selection seed 2026082702, we sample 100,000 candidate groupings. Each draws 30 eligible instances without replacement and randomly partitions them into six groups of five. A group must contain at least three distribution families, at least two capacities, and at least two distinct winning heuristics in each EoH-S population. We rank feasible groupings by four criteria, in order: minimum winner entropy across the 18 group–population pairs, mean winner entropy, minimum feature distance among the 30 instances, and mean within-group feature distance. Winner entropy is computed in bits from the five winner labels. This favors groups that expose complementary heuristic performance while retaining diverse instance characteristics.

Feature distances are Euclidean distances after standardization across all 300 candidates. The 18 features comprise $\log n$, $\log C$, normalized item-size mean and standard deviation, coefficient of variation, skewness, five normalized quantiles (10th, 25th, 50th, 75th, and 90th), fractions above $C/3$, $C/2$, and $2C/3$, fractions at sizes 1 and C, the number of distinct sizes divided by C, and $(\sum_j s_j - b_i C)/C$. The selected grouping has minimum and mean winner entropies of 1.922 and 2.122 bits, respectively.

Evaluation. The three released EoH-S populations guide test-set selection, so their results describe performance on the resulting calibration sets. The six sets remain fixed when evaluating all other methods.

For a final population P and a five-instance set S, the reported set gap is

$$G(P, S) = \frac{1}{5} \sum_{i \in S} \min_{h \in P} g(h, i). \tag{13}$$

We evaluate every population member on all five instances and report the mean best-member gap for each set.

B.3 Test-set composition

Table 6 lists the distribution family, item count, and capacity of each selected instance.

C Prompts for EoH and EoH-S

We provide the task descriptions, function templates, and search prompts used by EoH and EoH-S Liu et al. (2026b). Both methods share an initialization prompt. Each parent is represented by its description and code. The LLM completes the function body from the supplied signature and docstring.

C.1 Task descriptions and function templates

The OBP task description follows EoH-S. For TSP and CVRP, the task descriptions specify a single next-node decision at each call. The following templates define the required interfaces.

Table 6: Composition of the six mixed-distribution test sets.

Set	Distribution	Items n	Capacity C
Mix-1	Beta	3,050	500
	Beta	510	200
	Exponential	8,636	200
	Exponential	221	200
	Weibull	2,534	200
Mix-2	Beta	304	200
	Bimodal normal	1,393	100
	Bimodal normal	2,327	200
	Weibull	6,492	200
	Weibull	1,671	100
Mix-3	Beta	4,615	100
	Beta	8,519	500
	Bimodal normal	7,259	500
	Normal	2,044	100
	Weibull	3,783	100
Mix-4	Beta	1,439	100
	Bimodal normal	1,859	200
	Bimodal normal	4,265	500
	Weibull	3,343	200
	Weibull	1,922	500
Mix-5	Bimodal normal	5,205	200
	Bimodal normal	961	500
	Lognormal	5,506	200
	Normal	460	100
	Weibull	323	100
Mix-6	Bimodal normal	1,476	500
	Lognormal	3,743	100
	Normal	399	500
	Weibull	2,701	200
	Weibull	6,753	200

OBP task description

```
Implement a function that returns the priority with which we want to
    add an item to each bin.
```

OBP function template

```
def priority(item: float, bins: np.ndarray) -> np.ndarray:
    """Returns priority with which we want to add item to each bin.
    Args:
        item: Size of item to be added to the bin.
        bins: Array of capacities for each bin.
    Return:
        Array of same size as bins with priority score of each bin.
    """
```

TSP task description

```
Design a constructive heuristic for the travelling salesman problem.
    Start at city 0, visit every city exactly once, then return to
    city 0. At each step choose one integer node ID from
    unvisited_nodes. The unvisited array is ordered by increasing
    distance from the current node. distance_matrix is a read-only
    float64 Euclidean matrix. Minimize closed-tour length through
    repeated one-step decisions. The evaluator calls this function
    once at every tour step, so each call must choose only the
    immediate next city. Do not construct, simulate, search,
    optimize, or improve a complete or multi-step tour inside one
    call. In particular, do not use insertion-tour construction,
    2-opt, relocate, multi-start search, beam search, recursion, or a
    nested route planner. Compute bounded-cost per-node scores or
    features directly from the current arguments and return the best
    node. Do not access files, networks, processes, or external
    state. destination_node is always city 0: it has already been
    visited and is excluded from unvisited_nodes. Return an actual
    city ID, not its array position, a score vector, or a full tour.
    The evaluator appends the sole remaining city automatically and
    then closes the tour; do not return destination_node early. Do
    not modify the inputs. Use only the supplied arguments and NumPy;
    put all algorithm logic in this function. Output a brace-enclosed
    one-sentence description followed by exactly one Python code
    block defining select_next_node with the four named arguments in
    the shown order. NumPy is available as np. No extra top-level
    functions, global variables or decorators. A small nested helper
    is allowed only for the current one-step scoring computation, not
    for constructing or improving a future route. Candidate
    evaluation has a hard 120-second total timeout; a timeout makes
    the candidate invalid and still consumes one evaluated-sample
    budget unit. Single-line and multiline function signatures are
    both supported.
```

TSP function template

```
def select_next_node(current_node: int, destination_node: int,
                     unvisited_nodes: np.ndarray, distance_matrix:
    np.ndarray) -> int:
    """Return one unvisited node ID using a bounded-cost one-step
    decision rule."""
```

CVRP task description

```
Design a constructive heuristic for the capacitated vehicle routing
    problem (CVRP). Node 0 is the depot. At each call return exactly
    one integer: a customer ID from unvisited_nodes, or depot (0) to
    end the current route. unvisited_nodes contains only currently
    feasible unserved customers whose demand does not exceed
    rest_capacity. demands and the Euclidean distance_matrix are
    read-only. Minimize total distance while serving each customer
    once without exceeding capacity. Make only the immediate
    next-node decision. Do not construct, simulate, search, optimize,
    or improve a complete or multi-step route plan inside one call;
    no Clarke-Wright construction, local search, 2-opt, relocate,
    multi-start, beam search, recursion, or nested route planner. The
    evaluator calls this function once per route decision, so
    repeated work inside one call is multiplied across the route. Let
    k be len(unvisited_nodes) and n the total number of customers.
    O(k^2) candidate-pair scoring and O(n^2) distance-matrix
    statistics per call are allowed, matching released EoH-S
    heuristics. This includes feasible-by-feasible distance
    submatrices and neighborhood distance reductions; use at most
    O(n^2) temporary data. Prefer cheaper scoring where possible. Do
    not use O(n^3) or higher work inside a single call, triple
    customer loops, or iterative route optimization. Quadratic
    scoring repeated over route decisions can make overall
    construction O(n^3); that alone is not a forbidden operator, but
    the evaluation time budget still applies. Do not modify inputs or
    access files, networks, processes, or external state. Return a
    node ID, not an array index, score vector, or route. Use only
    NumPy and put all logic in select_next_node. Output a
    brace-enclosed one-sentence description followed by exactly one
    Python code block defining the six named arguments in the shown
    order. Candidate evaluation runs in a separate process with a
    hard 120-second total timeout across all requested training
    instances; exceeding it terminates evaluation, makes the
    candidate invalid, and still consumes one evaluated sample.
```

CVRP function template

```
def select_next_node(current_node: int, depot: int, unvisited_nodes:
    np.ndarray,
                     rest_capacity: int, demands: np.ndarray,
                     distance_matrix: np.ndarray) -> int:
    """Return depot or one feasible unvisited customer ID."""
```

C.2 Initialization and search operators

Each prompt requests a one-sentence description enclosed in braces, followed by Python code. For TSP and CVRP, `inside within boxed {}` is replaced by `inside literal curly braces, e.g. {Short algorithm description}, not a LaTeX box.` EoH-S

E1 encourages complementary heuristics across instance distributions; M1 refines an existing heuristic.

Shared initialization (I1)

```
{task description}
1. First, describe your new algorithm and main steps in one
    sentence. The description must be inside within boxed {}.
2. Next, implement the following Python function:
{task template}
Do not give additional explanations.
```

EoH E1

```
{task description}
I have {number of parents} existing algorithms with their codes as
    follows:
{parent descriptions and codes}
Please help me create a new algorithm that has a totally different
    form from the given ones.
1. First, describe your new algorithm and main steps in one
    sentence. The description must be inside within boxed {}.
2. Next, implement the following Python function:
{task template}
Do not give additional explanations.
```

EoH E2

```
{task description}
I have {number of parents} existing algorithms with their codes as
    follows:
{parent descriptions and codes}
Please help me create a new algorithm that has a totally different
    form from the given ones but can be motivated from them.
1. Firstly, identify the common backbone idea in the provided
    algorithms.
2. Secondly, based on the backbone idea describe your new algorithm
    in one sentence. The description must be inside within boxed {}.
3. Thirdly, implement the following Python function:
{task template}
Do not give additional explanations.
```

EoH M1

```
{task description}
I have one algorithm with its code as follows. Algorithm description:
{parent description}
Code:
{parent code}
Please assist me in creating a new algorithm that has a different
    form but can be a modified version of the algorithm provided.
1. First, describe your new algorithm and main steps in one
    sentence. The description must be inside within boxed {}.
2. Next, implement the following Python function:
{task template}
Do not give additional explanations.
```

EoH M2

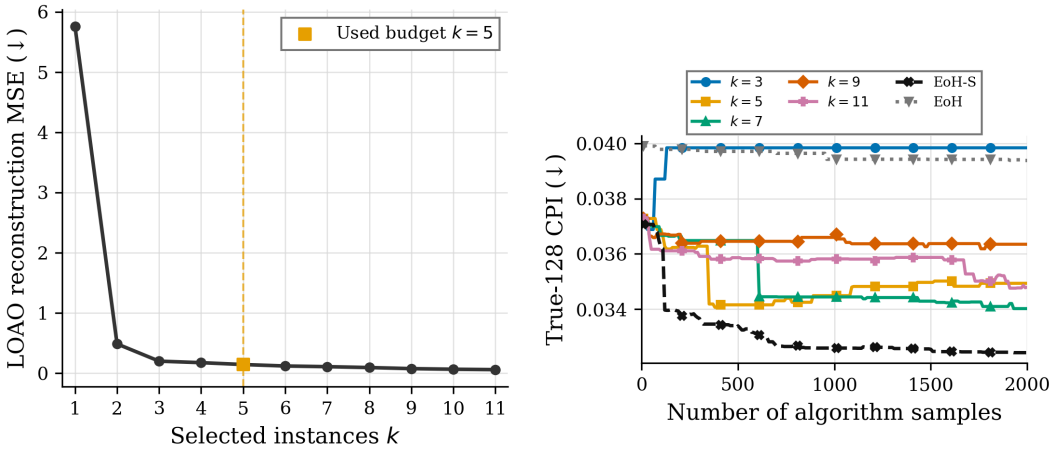
```
{task description}
I have one algorithm with its code as follows. Algorithm description:
{parent description}
Code:
{parent code}
Please identify the main algorithm parameters and assist me in
    creating a new algorithm that has a different parameter settings
    of the score function provided.
1. First, describe your new algorithm and main steps in one
    sentence. The description must be inside within boxed {}.
2. Next, implement the following Python function:
{task template}
Do not give additional explanations.
```

EoH-S E1

```
{task description}
I have {number of parents} existing algorithms with their codes as
    follows:
{parent descriptions and codes}
These algorithms are effective for solving different instance
    distributions. Please help me create a new algorithm that is
    different from the given ones.
1. First, describe your new algorithm and main steps in one
    sentence. The description must be inside within boxed {}.
2. Next, implement the following Python function:
{task template}
Do not give additional explanations.
```

EoH-S M1

```
{task description}
I have one algorithm with its code as follows. Algorithm description:
{parent description}
Code:
{parent code}
Please assist me in creating an improved version of the algorithm
    provided.
1. First, describe your new algorithm and main steps in one
    sentence. The description must be inside within boxed {}.
2. Next, implement the following Python function:
{task template}
Do not give additional explanations.
```



(a) LOAO reconstruction error along the instance selection process. The dashed line and square mark the selected budget $k = 5$.

(b) Population CPI evaluated on the same 128 training instances (↓). Black dashed and gray dotted curves denote EoH-S and EoH references, respectively.

Figure 3: **Sensitivity analysis on instance subset budget k.** The left panel measures reconstruction error as the budget k grows; the right panel measures search dynamics on a common evaluation suite.

D Sensitivity Analysis

To evaluate the parameter stability of PRISM, we analyze two critical hyperparameters: the size of selected instance subset k and the number of directions D added by metric-aware feedback.

Sensitivity to subset evaluation budget k. We evaluate the impact of the instance evaluation budget by varying $k \in \{3, 5, 7, 9, 11\}$ under a fixed sampling budget (2,000 algorithm candidates) within the PRISM+EoH-S framework. As depicted in fig. 3, the left panel illustrates the leave-one-algorithm-out (LOAO) reconstruction error along the nested selection path, while the right panel compares the resulting true CPI trajectories against full-suite baselines (EoH and EoH-S). The trajectories reveal that overly aggressive compression ($k = 3$) distorts population selection dynamics. In contrast, budgets with $k \geq 5$ effectively preserve evolutionary search progress. Specifically, $k = 7$ achieves the optimal terminal CPI, whereas $k = 5$ offers a highly competitive trade-off by drastically reducing per-candidate evaluation overhead.

We highlight two key observations regarding the choice of k:

- **Non-monotonic scaling with respect to k:** Increasing the subset size k does not yield strictly monotonic gains in final search performance. This indicates that expanding the evaluation subset reshapes the multi-objective selection landscape—introducing distinct trade-offs rather than simply serving as a higher-fidelity proxy.
- **Search trajectory stability over generations:** Ultra-compact budgets (e.g., $k = 3$) suffer from selection misalignment relative to the full 128-instance evaluation, causing temporary CPI degradation (visible as upwards spikes in the search curve over time). Moderate budgets ($k \geq 5$) eliminate these optimization fluctuations, restoring selection pressure fidelity and ensuring steady, reliable convergence.

Sensitivity to directions D in metric-aware. Fixing the instance budget at $k = 5$, we vary the total number of preference directions $D \in \{5, 50, 100, 150, 200, 250, 300\}$ to evaluate the required resolution for metric-aware feedback. Here, $D = 5$ corresponds to evaluating solely on the raw selected instance coordinates, whereas $D > 5$ preserves these primary coordinates while augmenting them with uniform simplex directions. Figure 4 plots the resulting CPI convergence trajectories across different values of D.

Two key insights emerge:

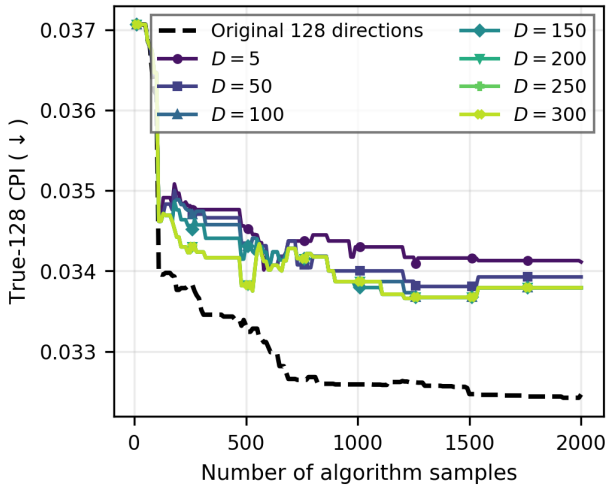


Figure 4: **True-CPI dynamics across preference direction counts D.** $D = 5$ uses the selected instances directly without preference augmentation; larger D values add more simplex directions.

- **Efficacy of metric-aware feedback:** Augmenting preference directions ($D > 5$) consistently improves terminal CPI over direct five-instance evaluation, confirming that directional expansion effectively mitigates evaluation signal sparsity introduced by instance selection.
- **Diminishing returns with scale:** The performance gains saturate rapidly beyond a moderate resolution (e.g., $D \geq 100$), with larger values of D yielding marginal benefits. This is because once the directional density reaches a certain threshold, the reward signals for the trade-off algorithm saturate, leaving the selection pressure unchanged.

E DESIGNED HEURISTICS

We report one representative heuristic for each task–method combination, chosen as the final member with the lowest mean per-instance rank across the test sets. Ranking instances separately gives equal weight to different problem scales. For methods that produce a heuristic set, we show its best-ranked member.

E.1 OBP

EoH on OBP

```python
import numpy as np


def priority(item: float, bins: np.ndarray) -> np.ndarray:
    """Returns priority with which we want to add item to each bin.
    Args:
        item: Size of item to be added to the bin.
        bins: Array of capacities for each bin.
    Return:
        Array of same size as bins with priority score of each bin.
    """
    remaining = bins - item
    mask = remaining >= 0
    priorities = np.zeros_like(bins)
    if np.any(mask):
        ratio = bins[mask] / (remaining[mask] + 1e-10)
        priorities[mask] = bins[mask] * (1 / (1 + np.exp(remaining[
            mask]))) * np.log(ratio + 1)
    return priorities
```

EoH-S on OBP

```python
import numpy as np


def priority(item: float, bins: np.ndarray) -> np.ndarray:
    capacity_ratio = np.percentile(bins, 80) / np.percentile(bins,
        20)
    dynamic_target = 0.55 + 0.25 * (1 / (1 + np.exp(-3.0 * (
        capacity_ratio - 1.1))))

    remaining = bins - item
    valid = remaining >= 0
    filled = (bins - remaining) / bins

    # Enhanced multi-tier capacity scoring
    tier_weights = 0.85 + 0.5 * np.tanh(2.5 * (bins / np.percentile(
        bins, 65) - 1.2))
    optimal_zone = dynamic_target * (1 + 0.18 * np.sin(1.5 * np.pi *
         tier_weights))

    # Dual-phase attraction-repulsion
    attraction = np.exp(-5 * np.abs(filled - optimal_zone))
    repulsion = 1 - 0.35 * np.power(np.maximum(0, filled - (
        optimal_zone + 0.15)), 2.0)
    reward = 3.5 * attraction * repulsion * tier_weights

    # Density-adaptive dispersion
```

```
    density_gradient = np.abs(filled - np.median(filled[valid])) if
        np.any(valid) else 0
    dispersion = 1.4 - 0.4 * np.tanh(6 * density_gradient) * (1 -
        0.25 * np.cos(3 * np.pi * filled))

    # Resonant harmonic modulation
    harmonic_factor = 1 + 0.2 * np.sin(3 * np.pi * (filled -
        dynamic_target + 0.05))
    priority_scores = np.where(valid,
                               -np.log1p(remaining) * reward *
                                   dispersion * harmonic_factor,
                               np.inf)
    return priority_scores
```

PRISM+EoH on OBP

```
def priority(item: float, bins: np.ndarray) -> np.ndarray:
    bins = np.asarray(bins, dtype=float)
    remaining = bins - item
    feasible = remaining >= 0
    scores = np.full(bins.shape, -np.inf, dtype=float)

    if item == 0:
        scores[feasible] = -remaining[feasible]
        return scores

    scale = max(abs(item), np.finfo(float).eps)
    normalized_remaining = remaining[feasible] / scale

    nearest_multiple = np.rint(normalized_remaining)
    distance = np.abs(normalized_remaining - nearest_multiple)

    alignment_score = 0.35 + 0.65 * np.cos(2.0 * np.pi * distance)
    residual_penalty = 0.04 * normalized_remaining**2

    scores[feasible] = alignment_score - residual_penalty
    return scores
```

PRISM+EoH-S on OBP

```
def priority(item: float, bins: np.ndarray) -> np.ndarray:
    bins = np.asarray(bins, dtype=float)
    scores = np.zeros_like(bins, dtype=float)

    try:
        item = float(item)
    except (TypeError, ValueError):
        return scores

    if not np.isfinite(item) or item < 0:
        return scores

    valid = np.isfinite(bins) & (bins >= 0)

    if not np.any(valid):
        return scores

    scale = max(1.0, abs(item), float(np.max(bins[valid])))
    tolerance = np.finfo(float).eps * scale * 8.0

```

```
    remaining = bins - item
    feasible = valid & (remaining >= -tolerance)

    if not np.any(feasible):
        return scores

    residual = np.maximum(remaining[feasible], 0.0)
    capacities = bins[feasible]

    order = np.argsort(residual, kind="stable")
    ranks = np.empty(order.size, dtype=float)
    ranks[order] = np.arange(order.size, dtype=float)

    rank_score = 1.0 / (1.0 + ranks)

    item_scale = max(item, np.finfo(float).eps)
    fit_score = 1.0 / (1.0 + residual / item_scale)

    utilization = np.divide(
        item,
        capacities,
        out=np.zeros_like(capacities),
        where=capacities > 0,
    )
    utilization = np.clip(utilization, 0.0, 1.0)

    fragmentation_penalty = np.where(
        (residual > tolerance) & (residual < item),
        0.35,
        1.0,
    )

    exact_fit = residual <= tolerance
    exact_fit_bonus = np.where(exact_fit, 3.0, 1.0)

    feasible_scores = (
        (0.50 * rank_score + 0.30 * fit_score + 0.20 * utilization)
        * fragmentation_penalty
        * exact_fit_bonus
    )

    scores[feasible] = np.nan_to_num(
        feasible_scores,
        nan=0.0,
        posinf=0.0,
        neginf=0.0,
    )

    return scores
```

E.2 TSP

EoH on TSP

```
import numpy as np


def select_next_node(current_node: int, destination_node: int,
    unvisited_nodes: np.ndarray, distance_matrix: np.ndarray) -> int:
    """
    Design a novel algorithm to select the next node in each step.
```

```

    Args:
    current_node: ID of the current node.
    destination_node: ID of the destination node.
    unvisited_nodes: Array of IDs of unvisited nodes.
    distance_matrix: Distance matrix of nodes.

    Return:
    ID of the next node to visit.
    """
    current_dist = distance_matrix[current_node, unvisited_nodes]
    dest_dist = distance_matrix[destination_node, unvisited_nodes]
    remaining_nodes = len(unvisited_nodes)
    sigmoid = 1 / (1 + np.exp(-remaining_nodes / len(distance_matrix)))
    proximity_weight = 0.6 * sigmoid
    progress_weight = 0.4 * (1 - sigmoid)
    diversity_weight = 0.2 * (1 - sigmoid)
    progress_term = distance_matrix[current_node, destination_node] - dest_dist
    diversity_term = np.mean(distance_matrix[unvisited_nodes][:, unvisited_nodes], axis=1)
    score = (proximity_weight * current_dist) + (progress_weight * progress_term) - (diversity_weight * diversity_term)
    return unvisited_nodes[np.argmin(score)]
```

EoH-S on TSP

```
import numpy as np


def select_next_node(current_node: int, destination_node: int, unvisited_nodes: np.ndarray, distance_matrix: np.ndarray) -> int:
    current_dist = distance_matrix[current_node, unvisited_nodes]
    dest_dist = distance_matrix[destination_node, unvisited_nodes]
    progress = distance_matrix[current_node, destination_node] - dest_dist
    exploration_factor = np.sqrt(len(unvisited_nodes))
    centrality = np.median(distance_matrix[unvisited_nodes][:, unvisited_nodes], axis=1)
    penalty = np.maximum(0, dest_dist - np.percentile(dest_dist, 75))
    k = min(4, len(unvisited_nodes) - 1)
    if k > 0:
        sub_matrix = distance_matrix[np.ix_(unvisited_nodes, unvisited_nodes)]
        cluster_novelty = -np.mean(np.partition(sub_matrix, k, axis=1)[:, :k], axis=1)
    else:
        cluster_novelty = np.zeros(len(unvisited_nodes))
    phase = len(unvisited_nodes) / len(distance_matrix)
    w_dist = 0.3 + (0.1 * phase)
    w_progress = 0.2 - (0.05 * phase)
    w_explore = 0.2 - (0.1 * phase)
    w_centrality = 0.1 + (0.05 * phase)
    w_penalty = 0.1 - (0.05 * phase)
    w_novelty = 0.1 + (0.05 * phase)
    momentum = np.mean(distance_matrix[unvisited_nodes] - distance_matrix[current_node], axis=1) if len(unvisited_nodes) > 1 else np.zeros(len(unvisited_nodes))
    w_momentum = 0.05 * (1 - phase)
```

```
    score = (w_dist * current_dist) + (w_progress * progress) + (
        w_explore * exploration_factor) - (w_centrality * centrality)
         + (w_penalty * penalty) + (w_novelty * cluster_novelty) + (
        w_momentum * momentum)
    return unvisited_nodes[np.argmin(score)]
```

PRISM+EoH on TSP

```
import numpy as np
def select_next_node(current_node: int, destination_node: int,
    unvisited_nodes: np.ndarray, distance_matrix: np.ndarray) -> int:
    """Return one unvisited node ID using a bounded-cost one-step
        decision rule."""
    nodes = np.asarray(unvisited_nodes, dtype=np.intp)
    if nodes.size == 1:
        return int(nodes[0])
    immediate = distance_matrix[current_node, nodes]
    pool = np.concatenate((nodes, np.asarray([destination_node],
        dtype=np.intp)))
    pairwise = distance_matrix[nodes[:, None], pool[None, :]]
    accessibility = np.mean(pairwise, axis=1)
    eps = np.finfo(np.float64).eps
    edge_scale = float(np.percentile(immediate, 50)) + eps
    access_scale = float(np.percentile(accessibility, 85)) + eps
    accessibility_term = np.clip(accessibility / access_scale, 0.0,
        2.5)
    scores = immediate / edge_scale - 0.65 * accessibility_term
    return int(nodes[int(np.argmin(scores))])
```

PRISM+EoH-S on TSP

```
import numpy as np
def select_next_node(current_node: int, destination_node: int,
    unvisited_nodes: np.ndarray, distance_matrix: np.ndarray) -> int:
    """Return one unvisited node ID using a bounded-cost one-step
        decision rule."""
    candidates = np.asarray(unvisited_nodes, dtype=np.intp)
    count = candidates.size
    if count == 1:
        return int(candidates[0])
    travel = np.asarray(distance_matrix[current_node, candidates],
        dtype=np.float64)
    depot = np.asarray(distance_matrix[candidates, destination_node
        ], dtype=np.float64)
    sample_count = min(count, 96)
    if sample_count == count:
        sample_indices = np.arange(count, dtype=np.intp)
    else:
        sample_indices = np.unique(np.linspace(0, count - 1,
            sample_count, dtype=np.intp))
    probes = candidates[sample_indices]
    pairwise = np.asarray(distance_matrix[np.ix_(candidates, probes)
        ], dtype=np.float64)
    pairwise[candidates[:, None] == probes[None, :]] = np.inf
    neighbor_count = min(7, probes.size - 1)
    nearest = np.partition(pairwise, neighbor_count - 1, axis=1)[:,
        :neighbor_count]
    connectivity = np.mean(nearest, axis=1)

    def robust_scale(values: np.ndarray) -> float:
```

```
        center = float(np.median(values))
        spread = float(np.median(np.abs(values - center)))
        return max(center, spread, 1e-12)
    travel_norm = travel / robust_scale(travel)
    depot_norm = depot / robust_scale(depot)
    connectivity_norm = connectivity / robust_scale(connectivity)
    remaining = float(max(count - 1, 1))
    early = 1.0 / np.sqrt(remaining)
    depot_weight = 0.2 + 0.62 * early
    isolation_weight = 0.24 + 0.48 * early
    interaction_weight = 0.08 + 0.2 * (1.0 - early)
    scores = travel_norm - depot_weight * depot_norm -
        isolation_weight * connectivity_norm + interaction_weight *
        travel_norm * connectivity_norm
    return int(candidates[int(np.argmin(scores))])
```

E.3 CVRP

EoH on CVRP

```
import numpy as np


def select_next_node(current_node: int, depot: int, unvisited_nodes:
     np.ndarray, rest_capacity: np.ndarray, demands: np.ndarray,
    distance_matrix: np.ndarray) -> int:
    """Design a novel algorithm to select the next node in each step
        .
    Args:
        current_node: ID of the current node.
        depot: ID of the depot.
        unvisited_nodes: Array of IDs of unvisited nodes.
        rest_capacity: rest capacity of vehicle
        demands: demands of nodes
        distance_matrix: Distance matrix of nodes.
    Return:
        ID of the next node to visit.
    """
    feasible_nodes = [n for n in unvisited_nodes if demands[n] <=
        rest_capacity]
    if not feasible_nodes:
        return depot

    current_dists = distance_matrix[current_node, feasible_nodes]
    depot_dists = distance_matrix[feasible_nodes, depot]
    urgency = 1 - np.minimum(rest_capacity / (demands[feasible_nodes
        ] + 1e-10), 1)

    high_urgency = urgency > 0.7
    medium_urgency = (urgency > 0.3) & ~high_urgency
    low_urgency = ~high_urgency & ~medium_urgency

    capacity_ratio = rest_capacity / np.max(demands[feasible_nodes])

    if any(high_urgency):
        tier_nodes = np.array(feasible_nodes)[high_urgency]
        scores = 0.6 * current_dists[high_urgency] + 0.2 *
            depot_dists[high_urgency] + 0.2 * urgency[high_urgency]
    elif any(medium_urgency):
        tier_nodes = np.array(feasible_nodes)[medium_urgency]
```

```
        scores = (0.4 + 0.2 * capacity_ratio) * current_dists[
            medium_urgency] + (0.4 - 0.2 * capacity_ratio) *
            depot_dists[medium_urgency] + 0.2 * urgency[
            medium_urgency]
    else:
        tier_nodes = np.array(feasible_nodes)[low_urgency]
        scores = (0.2 + 0.3 * capacity_ratio) * current_dists[
            low_urgency] + (0.5 - 0.3 * capacity_ratio) * depot_dists
            [low_urgency] + 0.3 * urgency[low_urgency]

    return tier_nodes[np.argmin(scores)]
```

EoH-S on CVRP

```
import numpy as np


def select_next_node(current_node: int, depot: int, unvisited_nodes:
     np.ndarray, rest_capacity: np.ndarray, demands: np.ndarray,
    distance_matrix: np.ndarray) -> int:
    feasible_nodes = unvisited_nodes[demands[unvisited_nodes] <=
        rest_capacity]
    if len(feasible_nodes) == 0:
        return depot

    distances = distance_matrix[current_node, feasible_nodes]
    depot_distances = distance_matrix[feasible_nodes, depot]
    capacity_ratio = demands[feasible_nodes] / (rest_capacity + 1e
        -6)

    quantum_super = np.exp(-3.1 * capacity_ratio**0.85) * np.sin
        (0.92 * np.pi * capacity_ratio**0.55)
    gravitational = np.exp(-np.abs(rest_capacity - demands[
        feasible_nodes]) / (np.percentile(demands[feasible_nodes],
        73) + 1e-6))

    bat_echo = (depot_distances - distance_matrix[current_node,
        depot]) / (distances**0.27 + 1e-6)
    imperialist = np.mean(distance_matrix[feasible_nodes][:,
        feasible_nodes], axis=1) / (distances**0.19 + 1e-6)

    diff_evolution = np.where(demands[feasible_nodes] > np.
        percentile(demands, 82),
                              1.65 * (1 - np.exp(-3.5 * capacity_ratio)
                                  ),
                              0.85 * np.exp(-4.3 * (1 - capacity_ratio)
                                  ))

    flower_poll = 0.72 * (1 + np.tanh(8.5 * (np.mean(demands[
        feasible_nodes]) / rest_capacity - 0.42)))
    colony_force = np.exp(-(distances + depot_distances) / (3.9 * np
        .mean(distance_matrix[feasible_nodes]) + 1e-6))

    quantum_phase = np.exp(-np.mean(np.abs(distance_matrix[
        feasible_nodes][:, unvisited_nodes] - np.percentile(
        distance_matrix, 74)), axis=1))
    grav_balance = 0.58 * (1 - distances / (np.max(distances) + 1e
        -6)) + 0.42 * (1 - depot_distances / (np.max(depot_distances)
         + 1e-6))

```

```
    bat_dynamics = np.mean(np.min(distance_matrix[feasible_nodes][:,
        unvisited_nodes], axis=1)) / (distances**0.16 + 1e-6)
    imperial_adapt = 1 - np.exp(-3.8 * (depot_distances / (np.mean(
        distance_matrix) + 1e-6)))

    scores = (flower_poll * (0.48 * quantum_super + 0.35 *
        gravitational + 0.14 * bat_dynamics + 0.03 * diff_evolution)
        +
              (1 - flower_poll) * (0.41 * bat_echo + 0.28 *
                  imperial_adapt + 0.23 * imperialist + 0.08 *
                  colony_force) +
              0.33 * quantum_phase +
              0.33 * grav_balance) / (distances**0.11 + 1e-6)

    return feasible_nodes[np.argmax(scores)]
```

PRISM+EoH on CVRP

```
import numpy as np
def select_next_node(current_node: int, depot: int, unvisited_nodes:
    np.ndarray, rest_capacity: int, demands: np.ndarray,
    distance_matrix: np.ndarray) -> int:
    """Return depot or one feasible unvisited customer ID."""
    import numpy as np
    nodes = np.asarray(unvisited_nodes, dtype=np.int64)
    if nodes.size == 0:
        return int(depot)
    demand_values = np.asarray(demands)
    nodes = nodes[nodes != int(depot)]
    if nodes.size == 0:
        return int(depot)
    feasible = demand_values[nodes] <= rest_capacity
    nodes = nodes[feasible]
    if nodes.size == 0:
        return int(depot)
    if nodes.size == 1:
        return int(nodes[0])
    matrix = np.asarray(distance_matrix, dtype=float)
    current_dist = np.nan_to_num(matrix[int(current_node), nodes],
        nan=np.inf, posinf=np.inf, neginf=np.inf)
    depot_dist = np.nan_to_num(matrix[int(depot), nodes], nan=np.inf
        , posinf=np.inf, neginf=np.inf)
    pairwise = np.nan_to_num(matrix[np.ix_(nodes, nodes)], nan=np.
        inf, posinf=np.inf, neginf=np.inf)
    np.fill_diagonal(pairwise, np.inf)
    neighbor_dist = np.min(pairwise, axis=1)
    finite_values = np.concatenate((current_dist[np.isfinite(
        current_dist)], depot_dist[np.isfinite(depot_dist)],
        neighbor_dist[np.isfinite(neighbor_dist)]))
    scale = float(np.median(finite_values)) if finite_values.size
        else 1.0
    scale = max(scale, 1e-12)
    node_demands = np.maximum(demand_values[nodes].astype(float),
        0.0)
    demand_scale = max(float(rest_capacity), float(np.max(
        node_demands)), 1.0)
    score = 0.82 * current_dist / scale - 0.18 * depot_dist / scale
        + 0.06 * neighbor_dist / scale - 0.16 * node_demands /
        demand_scale
    order = np.lexsort((nodes, score))
    return int(nodes[int(order[0])])
```

PRISM+EoH-S on CVRP

```python
import numpy as np
def select_next_node(current_node: int, depot: int, unvisited_nodes: np.ndarray, rest_capacity: int, demands: np.ndarray, distance_matrix: np.ndarray) -> int:
    """Return depot or one feasible unvisited customer ID."""
    import numpy as np
    current_node = int(current_node)
    depot = int(depot)
    candidates = np.asarray(unvisited_nodes, dtype=np.int64).reshape(-1)
    if candidates.size == 0:
        return depot
    candidate_demands = np.asarray(demands[candidates], dtype=float)
    feasible = (candidates != depot) & np.isfinite(candidate_demands) & (candidate_demands <= float(rest_capacity))
    candidates = candidates[feasible]
    candidate_demands = candidate_demands[feasible]
    if candidates.size == 0:
        return depot
    if candidates.size == 1:
        return int(candidates[0])
    huge = 1000000000000.0

    def clean(values):
        values = np.asarray(values, dtype=float)
        values = np.nan_to_num(values, nan=huge, posinf=huge, neginf=huge)
        return np.maximum(values, 0.0)
    to_candidate = clean(distance_matrix[current_node, candidates])
    from_depot = clean(distance_matrix[depot, candidates])
    to_depot = clean(distance_matrix[candidates, depot])
    current_to_depot = float(clean(distance_matrix[current_node, depot]))
    pairwise = clean(distance_matrix[np.ix_(candidates, candidates)])
    np.fill_diagonal(pairwise, huge)
    nearest_two = np.partition(pairwise, kth=1, axis=1)[:, :2]
    nearest = nearest_two[:, 0]
    second_nearest = nearest_two[:, 1]
    observed = np.concatenate((to_candidate, from_depot, to_depot, nearest, second_nearest))
    usable = observed[(observed > 0.0) & (observed < huge * 0.5)]
    if usable.size:
        median = float(np.median(usable))
        deviation = float(np.median(np.abs(usable - median)))
        scale = max(median, deviation * 1.4826, 1e-09)
    else:
        scale = 1.0
    capacity = max(float(rest_capacity), 1.0)
    demand_ratio = np.clip(candidate_demands / capacity, 0.0, 1.0)
    residual_capacity = float(rest_capacity) - candidate_demands
    future_feasible_count = np.sum(np.asarray(demands[candidates], dtype=float)[None, :] <= residual_capacity[:, None], axis=1) - 1.0
    future_fraction = np.clip(future_feasible_count / max(float(candidates.size - 1), 1.0), 0.0, 1.0)
    urgency = demand_ratio * (0.7 + 0.3 * (1.0 - future_fraction))
    travel = to_candidate / scale
    closing = np.minimum(to_depot / scale, 6.0)
    cohesion = np.minimum(nearest / scale, 5.0)
    secondary_cohesion = np.minimum(second_nearest / scale, 5.0)
```

```
    savings = np.maximum(0.0, current_to_depot + from_depot -
        to_candidate) / scale
    pressure = float(np.mean(demand_ratio))
    score = travel + (0.13 + 0.08 * pressure) * closing - (0.22 +
        0.08 * (1.0 - pressure)) * cohesion - 0.055 *
        secondary_cohesion - (0.12 + 0.08 * pressure) * urgency -
        (0.11 + 0.06 * (1.0 - pressure)) * np.minimum(savings, 6.0)
    best_index = int(np.argmin(score))
    best_node = int(candidates[best_index])
    if current_node != depot:
        continue_cost = float(to_candidate[best_index] + to_depot[
            best_index])
        restart_cost = float(current_to_depot + from_depot[
            best_index])
        restart_factor = 0.92 + 0.035 * pressure
        if restart_cost < continue_cost * restart_factor:
            return depot
    return best_node
```