BALLAD: Bandit-Based LLM Routing for Automated Heuristic Discovery

Samidha Verma[1] **Ankit Anand**[2] **Sayan Ranu**[1]
[1]Indian Institute of Technology Delhi [2]Google DeepMind

Abstract

Modern automated heuristic discovery (AHD) systems increasingly rival hand-crafted heuristics by leveraging Large Language Models (LLMs) to iteratively mutate and refine solutions. A natural assumption is that more powerful and more expensive models consistently yield better heuristics. We challenge this empirically: across multiple NP-hard combinatorial optimization problems, we find no consistent advantage for larger, costlier models. This raises a deeper question: given a pool of LLMs spanning frontier and freely hosted or open-weight models, how should one adaptively select which model to invoke at each step of the search? The challenge is non-trivial. The optimal LLM may shift over the course of a run, and because one LLM's output seeds the context for subsequent calls, each model's reward signal is non-stationary and entangled across the heuristic population, thus violating the independence assumptions of classical online decision-making settings. We address this through BALLAD (**Ba**ndit-based **LL**M Routing for **A**utomated Heuristic **D**iscovery), an online bandit algorithm that learns to orchestrate a pool of LLMs into an ensemble that matches or exceeds what any soloist LLM achieves alone. This orchestration is powered by an LLM selection policy that favors rare, high-scoring heuristics over stable averages, a DAG-based credit propagation mechanism through the genealogy of the heuristic population, and discounting of stale observations to remain attuned to each model's current utility. Across five NP-hard combinatorial optimization problems and a pool of four models of varying capability and cost, BALLAD not only matches the best individual LLM, but in several cases surpasses every individual model, while reducing API cost by up to 65.6% relative to always using the strongest model.

1 Introduction and Related Works

Heuristics for hard combinatorial problems have traditionally been designed by human experts, a process that is slow, domain-specific, and difficult to scale. Recent works have shown that Large Language Models (LLMs) based systems such as FunSearch [18], EoH [15], and AlphaEvolve [16] iteratively prompt an LLM to mutate and refine heuristics expressed as code, achieving performance that rivals or exceeds hand-crafted solutions on a range of combinatorial problems.

A practical bottleneck is cost. Automated heuristic discovery (AHD) requires thousands of LLM calls per run, making the choice of model a first-order concern. A natural assumption is that more powerful and more expensive models consistently yield better heuristics, justifying the overhead. We challenge this assumption directly: across five combinatorial optimization benchmarks, we find **no consistent advantage for larger, costlier models**, and in several cases smaller or free models outperform their expensive counterparts. This finding has an immediate practical implication: rather than committing to a fixed model or a fixed ensemble that pays for every model on every round, one should *adaptively* select which LLM to invoke at each step, based on online feedback from the search itself.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

At its core, the problem maps itself to *LLM routing* [1, 17, 20, 10], where one learns to select the best LLM for a specific task and input at hand[1]. These routing approaches share two limiting assumptions: that routing decisions can be trained on offline labelled data, and that each query is routed independently with no notion of how that choice affects future calls. Neither holds in AHD. There is no offline supervision signal; the only feedback available is the evaluator score assigned after each heuristic is tested. Also, the choice of LLM at each step in AHD affects the future routing decisions. Beyond these two limitations, AHD exposes three further properties. First, the goal is not to route each query to the best model in isolation, but to discover a single best heuristic (one that maximizes the score) across the entire run (P1). Second, because each LLM's output seeds the shared program population that conditions future prompts, routing decisions are causally coupled across steps and models (P2). Third, as the population evolves, the relative utility of each model shifts, so the optimal routing policy is non-stationary over the course of the search (P3).

To address these challenges, we propose BALLAD (**Ba**ndit-based **LL**M Routing for **A**utomated Heuristic **D**iscovery), which frames adaptive LLM selection in AHD as an online bandit problem, learning which model to invoke from evaluator scores accumulated during the search itself. Classical bandit algorithms, however, assume independent arms and a cumulative reward objective, and are therefore not directly applicable under (P1) - (P3). BALLAD addresses all three through a single per-arm index combining a bootstrapped quantile-of-maxima estimator (P1), a DAG-lineage credit signal that attributes downstream improvements to the model whose output enabled them (P2), and a sliding-window normalisation that keeps the index sensitive to each model's current rather than historical fertility (P3). BALLAD is a drop-in replacement for the fixed-LLM generator in any FunSearch- or AlphaEvolve-style pipeline, requiring no offline supervision and no changes to prompt construction or evaluation.

Contributions: BALLAD makes following key contributions:

1. **Empirical finding.** We demonstrate that larger, more expensive LLMs do not consistently outperform smaller or open-weight alternatives in AHD, motivating adaptive model routing as a practical necessity rather than a mere optimization.
2. **Problem formalization.** We formalize LLM routing in AHD as a sequential decision problem and identify three properties: extreme-value objective (P1), cross-arm DAG coupling (P2), and non-stationary rewards (P3), that jointly distinguish it from existing bandit families.
3. **Algorithm design.** We propose BALLAD, which casts LLM routing in AHD as a bandit learning problem. BALLAD is distinguished by the amalgamation of three algorithmic ideas tailored to the unique challenges of AHD: a single per-arm index that (i) hunts for rare, high-scoring heuristics through extreme-value arm ranking rather than mean performance, (ii) propagates credit through a novel DAG-based model of the mutation genealogy, and (iii) tracks non-stationary rewards to remain attuned to each model's current rather than historical utility.
4. **Empirical validation.** BALLAD not only routes but orchestrates: across five NP-hard combinatorial optimization problems and a pool of four LLMs spanning open- and closed-weight models of varying capability and cost, it matches or surpasses the best individual LLM on several problems; a result that cannot be explained by selection alone, and demonstrates that BALLAD learns a collaborative dynamic among models that no single model can replicate. This comes at a reduced cost, cutting API expenditure by up to 65.6% relative to always using the strongest model.

2 Preliminaries and Problem Formulation

We begin by formalizing the automated heuristic discovery setting, then introduce the notation needed to characterize LLM selection within it, and finally state the three properties that make this selection problem distinct from existing routing and bandit formulations. All notations introduced in sections 2 and 3 are listed in Table 3 for reference.

2.1 Automated Heuristic Discovery

Given a problem $\mathcal{P}$, a set of instances $\mathcal{I}_\mathcal{P}$, and a scoring function $f_\mathcal{P}$, the goal of an AHD system is to discover a heuristic $\hat{h} \in \mathcal{H}$ that maximizes average performance over $\mathcal{I}_\mathcal{P}$:

$$\hat{h} \;=\; \arg\max_{h \in \mathcal{H}} \frac{1}{|\mathcal{I}_\mathcal{P}|} \sum_{I \in \mathcal{I}_\mathcal{P}} f_\mathcal{P}(h, I). \tag{1}$$

[1]Please refer to App. A for a more detailed related works discussion on AHD and LLM routing.

The system operates iteratively over a horizon of T steps. At each step t, a context-building algorithm $\mathcal{A}$ selects k heuristics from all heuristics generated so far, $H_t = \mathcal{A}(h_{<t})$, which are used to prompt an LLM $\ell_t \in \mathcal{L}$ to produce a new candidate h_t via mutation and crossover. The candidate is then evaluated by a problem-specific objective function, $f_{\mathcal{P}}$, yielding a scalar score r_t[2]. We assume $\mathcal{A}$ is fixed and depends only on the set of heuristics available at time t. Traditionally, the choice of LLM is also fixed or decided based on a pre-decided policy. Our objective, which we will define more formally later in the section, is to *learn* a policy that dynamically selects the best LLM from a given set of options at each stage of AHD.

2.2 Preliminary Definitions

We now define the concepts that characterize the AHD search process and that the routing algorithm will operate over.

LLM pool. Let $\mathcal{L} = \{\ell_1, \ldots, \ell_n\}$ denote the finite pool of candidate LLMs. When prompted with context H_t, each LLM ℓ acts as a stochastic heuristic generator: we write $h_t \sim \mu_\ell(\ell, H_t)$ to denote a heuristic sampled from ℓ's output distribution given H_t. Furthermore, we denote the authoring LLM of h_t as $\texttt{LLM}(h_t) \in \mathcal{L}$.

Per-LLM score buffer. For each LLM ℓ, we maintain a buffer $\mathcal{O}_\ell = \{\{f_{\mathcal{P}}(h)\}\}$ containing the evaluation scores of all heuristics generated by ℓ[3]. We denote the number of times LLM ℓ has been pulled as $n_\ell = |\mathcal{O}_\ell|$.

LLM **quality via upper-tail quantile.** Because the objective in Eq. (1) cares about the *best* heuristic found rather than average performance, an LLM's value is better characterized by its ability to produce rare, high-scoring outputs than by its mean score. We therefore characterize LLM ℓ by its upper-tail quantile $q_p(\ell)$, defined as the score threshold exceeded by only a p fraction of outputs from LLM ℓ:

$$q_p(\ell) \;=\; \min\{x \in \mathbb{R} : \Pr(f_{\mathcal{P}}(h) > x \mid \texttt{LLM}(h) = \ell) \le p\}, \tag{2}$$

where $p \in (0, 1)$ is the tail level. Smaller p corresponds to more extreme high-score outcomes; as $p \to 0$, $q_p(\ell)$ approaches the maximum score achievable by LLM ℓ, which is precisely the quantity our objective seeks to maximize.

Mutation Genealogy DAG The key insight in AHD is that each of the LLMs don't operate independently but the output of one LLM could seed the context of other LLM, leading to coupling of different models in the system. In these situations, a model deserves partial credit whenever its outputs appear as parents in the context of a high-scoring heuristic produced by a different model, even if that improvement arrives several rounds later. To operationalize this, we explicitly define the mutation genealogy of the entire heuristic population in the form of a a *directed acyclic graph* (DAG).

Definition 1 (Mutation Genealogy DAG). The genealogy of heuristics generated over the run till time t is captured in the form a directed acyclic graph $\mathcal{G}_t = (\mathcal{V}, \mathcal{E})$, where each node $h \in \mathcal{V}$ is a generated heuristic and a directed edge $(h', h) \in \mathcal{E}$ indicates that h' was included in the context when h was generated.

For each heuristic h, we denote its parent set $Pa(h) \subseteq h_{<t}$ as the k heuristics selected by $\mathcal{A}$ to construct the prompt from which h was generated. Now, the d-hop ancestry of h is defined recursively as the set of all heuristics reachable from h by following at most d parent edges:

$$\mathrm{Pa}^{(d)}(h) \;=\; \begin{cases} \emptyset & \text{if } d = 0, \\ Pa(h) \;\cup \bigcup\limits_{h' \in Pa(h)} \mathrm{Pa}^{(d-1)}(h') & \text{if } d \ge 1, \end{cases} \tag{3}$$

2.3 Problem Statement

Problem 1 (**Adaptive LLM Routing in AHD**). Given a problem $\mathcal{P}$ with instance set $\mathcal{I}_{\mathcal{P}}$ and scoring function $f_{\mathcal{P}}$, a finite pool of LLMs $\mathcal{L} = \{\ell_1, \ldots, \ell_n\}$, a context-building algorithm $\mathcal{A}$, a seed heuristic h_0, and a finite horizon T, find an adaptive LLM selection policy $\pi : h_{<t} \mapsto \mathcal{L}$ that maximizes the score of the single best heuristic discovered across the entire run:

$$\pi^* \;=\; \arg\max_{\pi} \max_{h \in h^{\pi}_{\le T}} \left\{ \frac{1}{|\mathcal{I}_{\mathcal{P}}|} \sum_{I \in \mathcal{I}_{\mathcal{P}}} f_{\mathcal{P}}(h, I) \right\}, \tag{4}$$

[2] w.l.o.g., we assume higher is better. If lower is better, we use the negative of the score.

[3] $\mathcal{O}_\ell$ is a multi-set since the same heuristic may be authored multiple times by an LLM

where $h_{<t} = \{h_0, h_1, \ldots, h_{t-1}\}$ denotes the set of all heuristics generated before round t, LLM $\ell_t = \pi(h_{<t})$ is selected by the policy, context $H_t = \mathcal{A}(h_{<t})$ is constructed by selecting k heuristics from $h_{<t}$, and a new heuristic $h_t \sim \mu_{\ell_t}(\ell_t, H_t)$ is generated and evaluated. The seed heuristic h_0 is fixed and π is applied for $t \geq 1$.

3 BALLAD: Proposed Methodology

The problem of selecting the best LLM at each step t can be naturally casted as an online bandit problem, where each LLM $\ell \in \mathcal{L}$ corresponds to an arm and the evaluator score $f_{\mathcal{P}}(h_t)$ serves as the reward signal. However, the AHD setting departs from standard bandit formulations in three fundamental ways, each of which invalidates a core assumption of classical bandit algorithms and necessitates a tailored solution.

1. **[P1] Extreme-value objective.** The goal is to maximize the score of the single best heuristic found across all T steps, not the expected score of any individual LLM call. Arms should therefore be ranked by their upper-tail quality $q_p(\ell)$ rather than their mean, and $q_p(\ell)$ must be estimable reliably from the small buffer $\mathcal{O}_\ell$ accumulated during the run.
2. **[P2] Cross-arm coupling.** AHD relies fundamentally on mutation: at each step, an LLM is prompted with a set of existing heuristics and asked to produce an improved variant. When a single LLM drives the entire search, attributing reward to that LLM is unambiguous. With a pool of LLMs, however, this attribution becomes non-trivial. When a strong heuristic is discovered, should the reward *only* go to the LLM that produced it, or also to those whose outputs contributed to its prompt? The programs in the prompt may have been authored by different LLMs in earlier rounds, making the discovery a combinatorial outcome of multiple models' contributions across multiple steps. Attributing reward solely to the calling LLM is therefore blind to this combinatorial nature of the discovery process: it ignores the possibility that a weaker LLM produced a heuristic that, when placed in the context of a stronger one, unlocked a breakthrough that neither model could have achieved alone. Any arm quality estimator that ignores this coupling will misevaluate the true contribution of each model to the search.
3. **[P3] Non-stationary reward distributions.** The score distribution $P_t(\cdot \mid \ell)$ of LLM ℓ drifts with t as the population of heuristics evolves. An estimator that pools all past observations of LLM ℓ uniformly will lag behind the current regime, over-weighting contributions from early rounds when the population was less refined.

No existing LLM router or bandit algorithm simultaneously satisfies all three requirements and hence, we design BALLAD.

3.1 Design Overview

BALLAD addresses the three properties (P1) - (P3) through a single per-arm scalar score $\mathrm{Score}_\ell(t)$ that is recomputed at every round from evaluator scores accumulated till the current timestep of the search process. Each LLM $\ell \in \mathcal{L}$ corresponds to an arm and at any round t, the arm with the highest *score* is pulled, i.e., $\pi^* = \arg\max_{\ell \in \mathcal{L}} Score_\ell(t)$. The score has three additive terms, each targeting a specific characteristic, to holistically characterize the current state of the search process:

$$\mathrm{Score}_\ell(t) \;=\; \underbrace{(1-\lambda)\,\hat{q}_\tau(\ell) + \lambda\,\max\{\mathcal{O}_\ell\}}_{\text{hybrid extreme-value (P1)}} \;+\; \underbrace{\alpha\,\bar{L}_\ell(t)}_{\text{Mutation genealogy + recency (P2, P3)}} \;+\; \underbrace{\beta\sqrt{\frac{\log T}{n_\ell}}}_{\text{exploration}} \tag{5}$$

The first term ranks arms by their ability to produce rare, high-scoring heuristics rather than by average performance, where $\hat{q}_\tau(\ell)$ is a bootstrap estimate of the upper-tail quantile $q_p(\ell)$ of §2.2 computed from the finite sample $\mathcal{O}_\ell$, with $\lambda \in [0, 1]$ controlling the blend between this robust tail estimator and the empirically recorded maximum $\max\{\mathcal{O}_\ell\}$. The second term rewards LLMs whose past outputs have seeded improvements by other LLMs in later rounds, capturing the combinatorial nature of multi-LLM discovery, expressed as a windowed fertility score $\bar{L}_\ell(t)$ (defined in §3.3), with $\alpha \geq 0$ controlling its contribution relative to the base score. The third is an exploration bonus weighted by $\beta \geq 0$ that prevents the score from prematurely concentrating pulls on a single arm, ensuring that every model accumulates enough observations for its score to be reliable before exploitation dominates. We next deep-dive into each of the individual terms in §3.2–§3.4.

3.2 Hybrid Extreme-Value Base Score (P1)

The first term replaces the empirical mean of $\mathcal{O}_\ell$ with an upper-tail estimator that is robust at small sample sizes. Two failure modes motivate the design. A direct plug-in estimator of a high quantile of $\mathcal{O}_\ell$ has high variance when n_ℓ is small, making it unreliable early in the run. Conversely, $\max \mathcal{O}_\ell$ is a single order statistic that can be dominated by one outlier draw, overstating an arm's true tail quality. Neither alone is sufficient; we therefore combine them through a bootstrap.

Given the score buffer $\mathcal{O}_\ell$ of size n_ℓ, we draw B bootstrap resamples of scores in $\mathcal{O}_\ell^{(b)} \sim \mathrm{Unif}(\mathcal{O}_\ell, n_\ell)$, take the per-resample maximum $M_\ell^{(b)} = \max \mathcal{O}_\ell^{(b)}$, and estimate

$$\hat{q}_\tau(\ell) \;=\; \mathrm{Quantile}_\tau\big(\{M_\ell^{(b)}\}_{b=1}^{B}\big), \tag{6}$$

where $\tau \in (0,1)$ is the bootstrap quantile level. $\hat{q}_\tau(\ell)$ estimates the upper-tail quantile $q_p(\ell)$ of §2.2 but with greater stability: it is the τ-quantile of the bootstrap distribution of the sample maximum, which remains well-behaved under heavy tails even at small n_ℓ [3]. Yet discarding $\max \mathcal{O}_\ell$ entirely would discard the single most informative observation, since the best heuristic found is exactly what `AHD` reports at the end of the run. We therefore form the hybrid estimator:

$$\mathrm{base}(\ell) \;=\; (1-\lambda)\,\hat{q}_\tau(\ell) \;+\; \lambda \max \mathcal{O}_\ell, \qquad \lambda \in [0,1], \tag{7}$$

which interpolates between a low-variance tail estimator ($\lambda = 0$) and a record-value estimator ($\lambda = 1$). The sensitivity of performance to λ is examined in the ablation study (§ 4.5).

3.3 Lineage Credit on the Heuristic DAG (P2)

The second term focuses on rewarding cross-arm coupling. Using heuristic genealogy graph $\mathcal{G}_t$ defined in § 2.2, whenever a heuristic h is discovered, we traverse its ancestry in $\mathcal{G}_t$ and attribute a credit c to every `LLM` that occurs in the ancestors. This transforms the routing problem from one of evaluating isolated `LLM` outputs to one of reasoning about the causal influence of each model across the entire search trajectory. However, since there are only a few LLMs which generate thousand of heuristics, a naïve credit assignment in time to all ancestors could lead to almost uniform credit assignment across all `LLM`s. To address this, we introduce restrictions across three different axes in credit assignment, across depth, time and absolute score respectively. Each of these restrictions help in introducing novel differentiation in `LLM` arms leading to significant performance improvements. Mathematically, let h_t denote the heuristic generated at round t, and let $\mathrm{Pa}^{(d)}(h_t)$ denote the set of ancestor heuristics within d-hop from h_t in $\mathcal{G}_t$. For each arm $\ell \in \mathcal{L}$, we maintain a (timestamped) credit history $\mathcal{C}_\ell$ as an ordered sequence of (t', c) pairs, where $t' \le t$ is the round at which credit c was earned due to lineage. Credits are stored with their birth timestamp t, which enables accounting for recency.

Adaptive Clipped Reward: Firstly, we focus on absolute score. The reward of a newly generated heuristic h_t denoted as r_t could be set to simply $f_{\mathcal{P}}(h_t)$. However, this risks flooding the lineage signal with routine improvements from, wherein in reality, we are interested in improvements that pushes the frontier. To capture this, we clip each new score against a dynamic baseline. Specifically, the reward r_t is computed as $r_t \;=\; \max\big(0,\; f_{\mathcal{P}}(h_t) - b_t\big)$ where b_t is a time dependent dynamic baseline. Specifically, we use median elite baseline, $b_t = \mathrm{median}(\mathrm{top}_K(\bigcup_{\ell\in\mathcal{L}} \mathcal{O}_\ell))$ where top_K denote the top-K scores in the set.

Credit is propagated only when $r_t > 0$, i.e. when h_t genuinely advances beyond the current elite frontier. The median of the elite pool is a natural threshold: it rises adaptively as the search progresses, ensuring that only heuristics that genuinely advance the frontier propagate credit, while keeping the threshold robust to outliers. Setting b_t too low would flood ancestor arms with uninformative credit; setting it too high would starve the lineage signal. Using the elite median balances these two failure modes. This design is consistent with the elitist parent selection used in FunSearch [18], AlphaEvolve [16], and ReEvo [25].

Decaying credit with genealogical distance As the genealogy graph $\mathcal{G}_t$ grows to hundreds of nodes, the causal influence between different nodes in the same lineage decreases with each mutation. This is captured in credit assignment by assigning a credit $c = \gamma^d \cdot r_t$ equivalent to a geometrically-discounted share of the reward r_t obtained from h_t to every ancestor arm, with discount factor $\gamma \in (0,1)$. Simply, this ensures that direct parents receive more credit than distant ancestors. In addition, credits are bounded up to depth $D_{\max}$ from the newly generated heuristic.

$$\mathcal{C}_{\texttt{LLM}(h')} \;\leftarrow\; \mathcal{C}_{\texttt{LLM}(h')} \;\cup\; \big\{\big(t,\; \gamma^d \cdot r_t\big)\big\} \qquad \forall h' \in \mathrm{Pa}^d(h_t); 0 \le d \le D_{\max} \tag{8}$$

Recency bias (P3) With the credit attribution rule of Eq. (8) in place, a natural choice for the lineage score of arm ℓ at round t is simply the sum of all credits accumulated in $\mathcal{C}_l$, i.e., $L_\ell(t) = \sum_{(t',c')\in\mathcal{C}_\ell} c'$. This is intuitive but problematic: it accumulates credit indefinitely, so an arm that was highly productive early in the search, when the heuristic population was crude and easy to improve upon, would maintain a high score long after the population had moved on and its outputs were no longer competitive. To address this, we restrict credit assignment in time axis. Effectively, the Lineage credit $L_l(t)$ is restricted to credits earned within the last W rounds, i.e., $\mathcal{C}_\ell^W \leftarrow \{(t',c') \in \mathcal{C}_\ell : t' > t - W\}$, and define the windowed fertility score as:

$$\bar{L}_\ell(t) = \frac{\sum_{(t',c')\in\mathcal{C}_\ell^W} c'}{\log\big(\max(e,\, n_\ell)\big)}. \tag{9}$$

The logarithmic denominator in Eq. (9) corrects for exposure bias: an arm pulled more frequently has contributed more programs to the population and therefore had more opportunities to appear in parent sets; not necessarily because it is a better model, but simply because it has been used more. Dividing by $\log(n_\ell)$ corrects for this. The floor at $\log e = 1$, where e denotes the Euler's number, prevents arms with very few pulls from having their lineage score artificially inflated by division by a near-zero quantity.

3.4 Exploration Bonus and Selection Rule

The base score and lineage term are both point estimates derived from observed data. In the absence of continued testing of under-explored arms, the score would concentrate pulls on whichever `LLM` produced a high early score, leaving other `LLM`s starved and potentially missing a stronger one whose quality only becomes apparent after sufficient sampling. To overcome this, we add a fixed-horizon UCB bonus $\beta\sqrt{\frac{\log T}{n_\ell}}$ as the third term to Eq. 5.

We use the total horizon T rather than the current round t in the numerator. This is deliberate: because the budget is fixed and known in advance, the exploration bonus is calibrated to the full horizon rather than growing unboundedly with elapsed time. Concretely, this means the bonus shrinks faster than standard UCB by front-loading exploration when the budget is plentiful and concentrating exploitation in later rounds when the remaining budget is scarce.

3.5 Pipeline integration.

BALLAD is a drop-in replacement for the fixed-`LLM` generator inside any FunSearch- or AlphaEvolve-style loop [18, 16]. The host framework owns prompt construction, evaluation, and the program database; BALLAD is touched at exactly two points per round. Before prompt construction, the router selects which `LLM` to invoke based on $\mathrm{Score}_\ell(t)$ as described above. After evaluation, the router updates the selected arm's score buffer $\mathcal{O}_\ell$ with $f_\mathcal{P}(h_t)$, increments n_ℓ, and propagates lineage credits through $\mathcal{G}_t$ if $r_t > 0$. No information beyond the evaluator score $f_\mathcal{P}(h_t)$, the ancestry set $Pa^{Dmax}(h_t)$ and the identity of the selected `LLM` needs to cross the interface, making BALLAD lightweight and agnostic to the specifics of prompt construction and evaluation. Pseudocodes and time complexity analysis are provided in App. B.

4 Experiments

In this section, we benchmark BALLAD on NP-hard combinatorial optimization (CO) problems to answer the following concrete research questions:

- **RQ1. Does model size predict performance?** Is the common assumption that "larger, more expensive `LLM`s consistently yield better heuristics" empirically supported?
- **RQ2. Matching vs. surpassing.** Does BALLAD learn to orchestrate the pool of `LLM`s such that the resulting ensemble not only matches but *exceeds* the best individual `LLM` in hindsight?
- **RQ3. Cost effectiveness.** Does BALLAD achieve competitive or superior performance at a lower API cost than always using the strongest model?
- **RQ4. Ablation.** What are the marginal contributions of each component of BALLAD's index score (Eq. 5), and how sensitive is performance to the choice of hyperparameters?

Table 1: Test-set performance. We report the best program (selected on validation) evaluated on the test set. For each model, we show mean ± std (over 3 runs) for *Task-Specific Objective (Obj.)* (↓) and *Rank* (↓), where lower is better for both Obj, since we benchmark on minimization problems, and rank. Ranks are integer-valued; draws at full precision share the lowest rank, and the next rank is skipped. *Overall* is the mean of per-problem means; its std is the root-mean-square of per-problem stds.

Model	Metric	TSP	CVRP	Online BP	Offline BP	GED	Overall
gemma-3-12b-it	Obj.	6.396 ± 0.028	10.753 ± 0.031	2.510 ± 0.000	215.719 ± 0.000	4.953 ± 0.091	48.066 ± 0.045
	Rank	3.333 ± 0.471	**1.667 ± 0.471**	3.000 ± 0.000	4.000 ± 0.000	4.667 ± 0.471	3.333 ± 0.365
gemma-3-27b-it	Obj.	6.934 ± 0.003	15.735 ± 4.348	2.593 ± 0.059	215.719 ± 0.000	4.804 ± 0.040	49.157 ± 1.945
	Rank	4.667 ± 0.471	5.000 ± 0.000	4.333 ± 0.943	4.000 ± 0.000	2.333 ± 0.471	4.067 ± 0.516
gemini-2.5-flash-lite	Obj.	6.258 ± 0.064	10.776 ± 0.070	1.574 ± 0.217	204.750 ± 0.034	4.792 ± 0.032	45.630 ± 0.108
	Rank	**1.333 ± 0.471**	2.333 ± 0.943	1.667 ± 0.471	**1.667 ± 0.471**	**1.667 ± 0.943**	**1.733 ± 0.699**
gemini-2.5-flash	Obj.	6.720 ± 0.304	11.984 ± 0.743	2.510 ± 0.000	204.781 ± 0.058	4.941 ± 0.069	46.187 ± 0.361
	Rank	3.667 ± 1.247	4.000 ± 0.000	3.000 ± 0.000	2.333 ± 0.943	4.333 ± 0.471	3.467 ± 0.730
BALLAD	Obj.	**6.233 ± 0.056**	**10.685 ± 0.075**	**1.483 ± 0.276**	**204.693 ± 0.323**	**4.789 ± 0.022**	**45.577 ± 0.195**
	Rank	1.667 ± 0.471	2.000 ± 0.816	**1.333 ± 0.471**	1.667 ± 0.943	2.000 ± 0.816	1.733 ± 0.730

The codebase of BALLAD and the programs generated for the various NP-hard CO problems are available at `https://github.com/idea-iitd/Ballad`. Hardware details and hyperparameter settings are described in App. D.

4.1 Experiment Setup

Problems. We evaluate BALLAD across 3 classes of NP-hard combinatorial optimisation problems.

- **Routing** includes the Traveling Salesman Problem (TSP) and Capacitated Vehicle Routing Problem (CVRP), both on instances with 50 nodes. CVRP uses a vehicle capacity constraint of $W = 50$.
- **Bin packing** includes both the Online and Offline Bin Packing Problems. The online variant uses large-scale instances of 5,000 items drawn from a Weibull distribution [15, 26, 7], evaluated under two bin capacity settings ($C \in \{100, 500\}$); the offline variant uses 500-item instances with bin capacity $C = 150$.
- **Graph Edit Distance (GED)** is evaluated on the Linux dataset [24], comprising unlabelled program dependence graphs with an average of 8 nodes and 7 edges per graph.

Detailed problem descriptions are provided in App. C; dataset splits and instance counts are summarized in Table 4.

LLMs. We consider 4 models for this study, `gemini-2.5-flash-lite` [9] and `gemini-2.5-flash` [9] as the two closed-source paid models, and `gemma-3-12b-it` [12] and `gemma-3-27b-it` [12] as the two open-weight free models. Among these, `gemini-2.5-flash` is the largest and most expensive.

Metrics: We benchmark BALLAD along three dimensions: **(i)** the average **objective value** attained by each `LLM` and BALLAD, **(ii)** the average **rank** across runs for each `LLM` and BALLAD, and **(iii)** the inference (API) **cost** incurred by each `LLM` and BALLAD. All experiments are repeated three times; we report the mean and standard deviation. Both the objective value and the rank are evaluated on the held-out test set. For each method, the selected heuristic is the one that achieves the best objective value on the validation set. The API cost for `AHD` is measured on the training set. Details of the train/validation/test splits are provided in Table 4.

4.2 RQ1: Does model size predict performance?

Table 1 reports the objective values and ranks achieved by each `LLM` and BALLAD. We observe that `gemini-2.5-flash`, despite being the largest and most capable model, does not achieve the best objective value or rank on any of the problems. In contrast, `gemma-3-12b-it`, the smallest model, achieves the best performance on CVRP. Furthermore, `gemini-2.5-flash-lite` exhibits the most consistent performance across all combinatorial optimization tasks.

These results clearly indicate that larger models do not necessarily yield better performance, thereby motivating the need for `LLM` routing in `AHD`.

4.3 RQ2: Can BALLAD match or surpass the best `LLM`?

Revisiting Table 1, we find that BALLAD achieves the best objective value overall and is consistently ranked first or second across all tasks. Aggregated across problems, it matches or outperforms all individual `LLM`s in both average objective and rank.

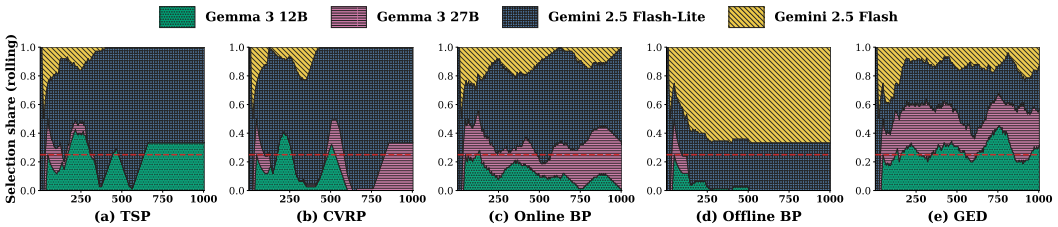

Figure 1: **LLM selection share across search steps (rolling window of 100 calls).** Each panel shows, for a combinatorial optimization problem, the fraction of LLM calls routed to each model by BALLAD over the course of the evolutionary search. The dotted horizontal line marks the uniform-allocation baseline (1/K=25% here), corresponding to a policy with no preference over arms. Deviations from this baseline reflect the learned routing policy by BALLAD favors models with selection share above 1/K due to higher estimated utility.

These results highlight two key observations. First, the strong aggregate performance of BALLAD indicates robust behavior across heterogeneous problem instances, even when the identity of the best-performing `LLM` varies by task. This suggests that BALLAD effectively learns to select an appropriate sequence of `LLM` calls conditioned on the problem. Second, and more notably, BALLAD occasionally surpasses the best single `LLM` on individual tasks. This indicates that it is not merely selecting the best model, but composing multiple `LLM` calls in a manner that yields complementary gains, i.e., the resulting performance exceeds that of any standalone model. In this sense, BALLAD goes beyond routing, learning to orchestrate an `AHD` process that leverages the collective strengths of multiple `LLM`s.

To further understand how BALLAD routes `LLM` calls, we analyze the sequence of selections over iterations $t \in [0, 1000]$ in Fig. 1. Across most tasks, BALLAD predominantly favors `gemini-2.5-flash-lite`, which is consistent with its strong average performance observed in Table 1. As iterations progress, the selection distribution becomes increasingly concentrated, with BALLAD effectively discarding two models and exploiting the remaining ones. This behavior aligns with the expected exploration–exploitation dynamics of the bandit phase, where later rounds prioritize exploitation.

In the offline-BP problem, the final phase is dominated by `gemini-2.5-flash` and `gemini-2.5-flash-lite`, which is consistent with their comparable performance in Table 1. An interesting deviation occurs for CVRP: although `gemma-3-12b-it` achieves the best standalone performance, BALLAD primarily selects `gemini-2.5-flash-lite` and `gemma-3-27b-it`. Despite not focusing on the individually best model, BALLAD still produces heuristics that achieve the best objective and rank in CVRP (Table 1). This shows that high-quality solutions can emerge from the interaction and mutation of outputs generated by non-optimal models, highlighting the importance of credit distribution through the mutational lineage in the `AHD` process.

In contrast, GED exhibits a different pattern, where all `LLM`s remain competitive throughout the iterations. This indicates that no single model is consistently dominated, and the `AHD` evolution maintains diversity along the performance frontier until the final stages.

We discuss the evolution of the best test objective over LLM calls, illustrating the progression and convergence of BALLAD in App. E.

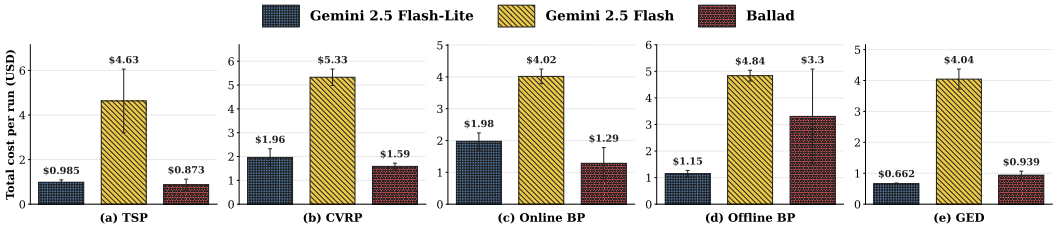

Figure 2: **LLM API cost per run.** Total monetary cost (USD) of one evolutionary search run on each task, averaged over three independent runs (mean $\pm$ std as error bars). Costs are computed from per-call input/output token counts using public Gemini 2.5 list prices: Flash-Lite at \$0.10/M input and \$0.40/M output tokens, Flash at \$0.30/M input and \$2.50/M output tokens. gemma-3 models incur no monetary cost, hence not shown. Average cost reduction relative to `gemini-2.5-flash` using BALLAD is 65.6%.

Table 2: **Ablation Study:** The first four rows are baseline arm-selection strategies. $\lambda = 0$ and $\lambda = 1$ denote quantile- and maximum-based variants, respectively; $\alpha = 0$ disables lineage-aware exploitation. τ controls the upper-tail level. *Exp. bonus* $\propto \sqrt{\log t}$ uses standard UCB exploration. *Cost-Penalized Exp.* accounts for `LLM` call costs and penalises exploration for expensive arms. **Bold** and underline denote the best and second-best, respectively. Due to computational constraints, we evaluate the ablations on three problems.

Model	Metric	TSP	CVRP	Online BP	Overall
RANDOM	Obj.	6.243 ± 0.037	10.851 ± 0.087	2.071 ± 0.621	6.378 ± 0.363
UCB	Obj.	$\mathbf{6.194 \pm 0.051}$	10.896 ± 0.088	1.640 ± 0.616	6.243 ± 0.360
SW-UCB	Obj.	6.306 ± 0.072	10.885 ± 0.118	1.752 ± 0.087	6.314 ± 0.094
THOMPSON SAMPLING	Obj.	6.294 ± 0.004	10.861 ± 0.023	2.398 ± 0.112	6.517 ± 0.066
BALLAD ($\lambda = 0$)	Obj.	6.255 ± 0.043	10.869 ± 0.086	1.603 ± 0.162	6.242 ± 0.108
BALLAD ($\lambda = 1$)	Obj.	6.250 ± 0.020	10.729 ± 0.017	1.988 ± 0.422	6.322 ± 0.244
BALLAD($\alpha = 0$)	Obj.	6.281 ± 0.130	10.714 ± 0.334	$\mathbf{1.483 \pm 0.276}$	$\underline{6.159 \pm 0.261}$
BALLAD($\tau = 0.7$)	Obj.	6.444 ± 0.231	$\mathbf{10.625 \pm 0.147}$	1.541 ± 0.447	6.203 ± 0.303
BALLAD($\tau = 0.99$)	Obj.	6.337 ± 0.013	10.695 ± 0.017	1.777 ± 0.733	6.270 ± 0.423
BALLAD (Exp. Bonus $\propto \sqrt{\log t}$)	Obj.	6.251 ± 0.024	$\underline{10.677 \pm 0.069}$	2.124 ± 0.509	6.351 ± 0.297
BALLAD(Cost-Penalized Exp.)	Obj.	6.266 ± 0.028	10.863 ± 0.135	2.079 ± 0.557	6.403 ± 0.331
BALLAD	Obj.	$\underline{6.233 \pm 0.056}$	10.685 ± 0.075	$\underline{1.483 \pm 0.276}$	$\mathbf{6.134 \pm 0.168}$

4.4 RQ3. Cost effectiveness.

Fig. 2 reports the inference cost incurred by BALLAD relative to `gemini-2.5-flash` and `gemini-2.5-flash-lite` across all five problems. Against `gemini-2.5-flash`, BALLAD is up to $5.30\times$ cheaper and on average $2.86\times$ cheaper across all problems. Against `gemini-2.5-flash-lite`, the cheaper of the two paid models, BALLAD is cheaper on three out of five problems, and only $1.19\times$ more expensive on average, despite exceeding `gemini-2.5-flash-lite`'s performance across the board. The exceptions are Offline BP and GED; as Fig. 1 shows, these are precisely the problems where `gemini-2.5-flash` remains consistently competitive throughout the full horizon, leading the routing policy to sustain higher allocation to the strongest model.

These results highlight the core dilemma that BALLAD resolves. Given a pool of `LLM`s, there is no a priori way to determine which model is best suited for a given task. Discarding expensive models is risky, as stronger models may exhibit superior reasoning in some settings; however, as RQ1 shows, smaller and cheaper models frequently outperform them. The only reliable naive strategy is to evaluate all candidates exhaustively, which multiplies cost by the pool size. BALLAD sidesteps this dilemma by automatically identifying an effective composition of `LLM`s without exhaustive evaluation, not only matching but often surpassing the best individual model, while incurring only a fraction of the cost. Notably, this cost efficiency emerges implicitly: the index score (Eq. 5) is entirely agnostic to model cost, and the savings arise naturally from the empirical finding that the most expensive model is not consistently optimal. If desired, the framework can be readily extended to explicitly incorporate inference cost into the arm scoring function (Eq. 5).

4.5 RQ4. Comparison with baselines and ablation of BALLAD components

The ablation study (Table 2) isolates the contributions of BALLAD 's core algorithmic components. By prioritizing upper-tail *breakthroughs* over mean performance (P1), BALLAD achieves a superior overall objective(6.134) to the standard UCB(6.243), SW-UCB(6.314) and Thompson Sampling(6.517) baselines (details in App. G). The hybrid mixing of bootstrap quantiles and empirical maxima provides a balanced *floor-and-ceiling* effect that outperforms both pure quantile-only and record-only configurations. Furthermore, disabling lineage-aware credit propagation ($\alpha = 0$) degrades performance on TSP and CVRP, validating the causal coupling of models across the mutation genealogy (P2). The tail-level parameter τ balances emphasis on upper-tail performance against estimation reliability: lower values dilute breakthrough signals, shifting the estimate toward average-case performance, whereas higher values rely on fewer order statistics, yielding noisier estimates. Calibrating the exploration bonus to the fixed horizon T rather than the current round t concentrates search on fertile models and consistently outperforms growing-bonus schedules. Furthermore, cost-penalized exploration underperforms by favoring cheaper, weaker models over potentially breakthrough-generating arms.

We examine the impact of expanding the arm pool to nine models, including frontier models such as GPT-5 and Claude Sonnet 5, alongside Gemini 2.5 Flash, on Offline BP and GED in Table 5. The expanded pool further improves objective values on both problems, while Table 6 reports cost savings averaging 46.35%, with a maximum of 68.3%, compared to always using GPT-5 under the assumption of comparable token usage. We provide a comprehensive analysis of the ablation results across all dimensions in App. F.

5 Conclusions, Limitations and Future Directions

We presented BALLAD, a bandit-based approach for adaptive LLM routing in automated heuristic discovery. We show that larger, more expensive models are not consistently superior, making exhaustive evaluation the only reliable but a prohibitively costly strategy. BALLAD addresses this by learning to orchestrate a pool of LLMs, leveraging extreme-value scoring, lineage-aware credit assignment, and non-stationary adaptation to identify effective model compositions. Across multiple combinatorial optimization problems, BALLAD not only matches but often surpasses the best individual LLM, while significantly reducing inference cost. Notably, our results show that the optimal strategy is not to select a single model, but to dynamically compose multiple models based on their evolving utility.

Limitations and Future Work Our current formulation treats each LLM as a black-box generator with a fixed prompting strategy. This overlooks that different LLMs may benefit from customized prompts, mutation operators, incorporation of lineage information or context representations.

Impact Statement Potential risks are limited, as the method introduces no new models or datasets, but inherits general considerations of LLMs, including environmental impact.

Acknowledgement

We thank the Yardi School of Artificial Intelligence at the Indian Institute of Technology Delhi for sponsoring Samidha Verma's expenses to attend NeurIPS 2026. We also thank NeurIPS for providing her with financial aid for accommodation and conference registration. We also acknowledge the Google PhD Fellowship awarded to Samidha Verma and the additional research funding provided by Incept Labs to support this work.

References

[1] Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, et al. Automix: Automatically mixing language models. *arXiv preprint arXiv:2310.12963*, 2023.

[2] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[3] Dorian Baudry, Yoan Russac, and Emilie Kaufmann. Efficient algorithms for extreme bandits. In *International Conference on Artificial Intelligence and Statistics*, pages 2210–2248. PMLR, 2022.

[4] Sujay Bhatt, Ping Li, and Gennady Samorodnitsky. Extreme bandits using robust statistics. *arXiv preprint arXiv:2109.04433*, 2021.

[5] 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.

[6] Alexandra Carpentier and Michal Valko. Extreme bandits. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[7] Ignacio Castiñeiras, Milan De Cauwer, and Barry O'Sullivan. Weibull-based benchmarks for bin packing. In *International Conference on Principles and Practice of Constraint Programming*, pages 207–222. Springer, 2012.

[8] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[9] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[10] Dujian Ding, Ankur Mallick, Shaokun Zhang, Chi Wang, Daniel Madrigal, Mirian Del Carmen Hipolito Garcia, Menglin Xia, Laks VS Lakshmanan, Qingyun Wu, and Victor Rühle. Best-route: Adaptive llm routing with test-time optimal compute. In *International Conference on Machine Learning*, pages 13870–13884. PMLR, 2025.

[11] Aurélien Garivier and Eric Moulines. On upper-confidence bound policies for switching bandit problems. In *International conference on algorithmic learning theory*, pages 174–188. Springer, 2011.

[12] Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ram'e, Morgane Rivière, Louis Rouillard, Thomas Mesnard, et al. Gemma 3 technical report. *ArXiv*, abs/2503.19786, 2025.

[13] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

[14] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[15] Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning*, pages 32201–32223. PMLR, 2024.

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

[17] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

[18] 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.

[19] Tal Shnitzer, Anthony Ou, Mírian Silva, Kate Soule, Yuekai Sun, Justin Solomon, Neil Thompson, and Mikhail Yurochkin. Large language model routing with benchmark datasets. In *First Conference on Language Modeling*.

[20] Wei Song, Zhenya Huang, Cheng Cheng, Weibo Gao, Bihan Xu, GuanHao Zhao, Fei Wang, and Runze Wu. Irt-router: Effective and interpretable multi-llm routing via item response theory. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15629–15644, 2025.

[21] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[22] Asterios Tsiourvas, Wei Sun, and Georgia Perakis. Causal llm routing: End-to-end regret minimization from observational data. *arXiv preprint arXiv:2505.16037*, 2025.

[23] Samidha Verma, Arushi Goyal, Ananya Mathur, Ankit Anand, and Sayan Ranu. Grail: Graph edit distance and node alignment using llm-generated code. In *International Conference on Machine Learning*, pages 61301–61322. PMLR, 2025.

[24] Xiaoli Wang, Xiaofeng Ding, Anthony KH Tung, Shanshan Ying, and Hai Jin. An efficient graph indexing method. In *2012 IEEE 28th International Conference on Data Engineering*, pages 210–221. IEEE, 2012.

[25] 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.

[26] Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. In *Forty-second International Conference on Machine Learning*.

Table 3: Notations and Hyperparameter Settings for BALLAD

Symbol	Description	Setting
Problem and Search Space		
$\mathcal{P}$	Problem specification	–
$\mathcal{I}_{\mathcal{P}}$	Set of problem instances	–
$f_{\mathcal{P}}$	Scoring/Objective function	–
h	Candidate heuristic (code)	–
h_0	Seed heuristic	–
T	Total budget of LLM calls (horizon)	1000
m	Batch size (heuristics per step)	4
$\mathcal{A}$	Context-building algorithm	–
k	Number of parents in a prompt	$k = 2$
H_t	Prompt context at step t	–
LLM Routing and Bandits		
$\mathcal{L}$	Pool of candidate LLMs	–
l_t	Selected model at step t	–
$Score_l(t)$	Index score for model l at step t	–
$\mathcal{O}_l$	Score buffer for model l	–
n_l	Pull count for model l	–
$q_p(l)$	True tail quantile at level p	–
$\hat{q}_\tau(l)$	Bootstrap tail quantile estimate	–
B	Number of bootstrap resamples	200
τ	Bootstrap quantile level	0.9
λ	Hybrid mixing weight	0.3
β	Exploration bonus coefficient	1.0
n_{min}	Warm-up threshold per LLM	3
Mutation Lineage and Credit		
$\mathcal{G}_t$	Mutation Genealogy DAG	–
$Pa(h)$	Direct parents of heuristic h	–
$Pa^{(d)}(h)$	d-hop ancestry of heuristic h	–
$\mathcal{C}_l$	Credit history for model l	–
c	Individual discounted lineage credit	–
r_t	Clipped reward at step t	–
b_t	Adaptive elite-median baseline	–
K	Top-K set size for b_t computation	5
γ	Credit discount factor	0.9
D_{max}	Maximum ancestry depth	4
α	Lineage-aware weight	1.0
W	Sliding window size (rounds)	25
$\overline{L}_l(t)$	Windowed fertility score for model l	–

A Related Works

Algorithm Discovery. Automating heuristic design predates LLMs, with extensive work in genetic programming [13] and hyper-heuristics [5] that search over program trees or rule combinations using hand-crafted mutation and crossover operators. These approaches do not exploit semantic structure and scale poorly. FunSearch [18] replaces such operators with a single LLM, while later work adds

reflections [25] and code logic generation in natural language or "thoughts" [15]. *AlphaEvolve* [16] uses two LLMs in a fixed-ratio ensemble—a smaller model for high-throughput generation and a larger model for occasional higher-quality proposals—incurring the cost of both at every round. Whether the best LLM varies across problems or over the course of a run remains unexplored. BALLAD addresses this by selecting a single LLM per round based on online feedback, avoiding unnecessary calls.

LLM **Routing.** Existing routers select among LLMs using preference-trained classifiers [17, 22], query-difficulty predictors [20, 10], benchmark-derived rankings [19], or learned cascades [8, 1], all of which require offline supervision. The signals they depend on are absent in AHD, as discussed above. This rules out the offline families and motivates an online, bandit-based approach.

Multi-Armed Bandits. The classical bandit [14] minimises cumulative regret with respect to the best mean reward via index policies such as UCB [2] and Thompson sampling [21]. *Extreme bandits* replace the mean by an upper-tail functional, directly addressing (P1): ExtremeHunter [6] bounds the regret of identifying the arm with the heaviest tail, Max-Median sampling [4] ranks arms by the median of subsample maxima, and QoMax [3] introduces a quantile-of-maxima estimator robust to single-sample outliers. *Non-stationary bandits*, including discounted UCB and sliding-window UCB [11], address (P3). Every method in these three families, however, assumes *i.i.d.* rewards within an arm and *independence* across arms, ruling out (P2). To our knowledge, no existing bandit formulation simultaneously addresses (P1) - (P3); BALLAD unifies them under a single index.

B Detailed Methodology

This section expands Section 3 into full pseudocode for the six sub-procedures of BALLAD (BASESCORE, ATTRIBUTE, FERTILITY, SCORE$_\ell$, SELECTARM, UPDATE), and gives the complete integration with a FunSearch-style [18] host pipeline. No new equations are introduced; the appendix only operationalises the equations already defined in Section 3 as algorithms.

B.1 Bootstrap Quantile-of-Maxima Base Score

Algorithm 1 implements the hybrid extreme-value estimator $\mathrm{base}(\ell)$ of Eq. (7). The warm-up rule returns $+\infty$ until $n_\ell \geq n_{\min}$, which forces round-robin coverage of every arm before the base score is trusted.

B.2 BFS Lineage Attribution

Algorithm 2 implements the credit-propagation rule of Eq. (8). Each visited parent enqueues *its* own parents, so the traversal expands the full DAG neighbourhood up to depth $D_{\max}$ rather than a single path. Credits are stored together with their birth timestamp, which is exploited by the windowing in App. B.3.

Algorithm 1 BASESCORE()

Require: LLM ℓ; score buffer $\mathcal{O}_\ell$; λ (quantile weight); τ (bootstrap quantile); B (bootstrap samples); $n_{\min}$ (warm-up calls)
Ensure: $\mathrm{base}(\ell) \in \mathbb{R} \cup \{+\infty\}$
$n_\ell \leftarrow |\mathcal{O}_\ell|$
if $n_\ell < n_{\min}$ **then**
  return $+\infty$ ▷ force round-robin warm-up
end if
for $b = 1, \ldots, B$ **do**
  $\mathcal{O}_\ell^{(b)} \leftarrow \mathrm{rng.choice}(\mathcal{O}_\ell,\, n_\ell,\, \mathrm{replace{=}True})$
  $M_\ell^{(b)} \leftarrow \max \mathcal{O}_\ell^{(b)}$
end for
$\hat{q}_\tau \leftarrow \mathrm{Quantile}_\tau\big(\{M_\ell^{(b)}\}_{b=1}^{B}\big)$
return $(1-\lambda)\,\hat{q}_\tau + \lambda \max \mathcal{O}_\ell$

We adopt a *path-based* credit assignment, where ancestors may receive multiple contributions if they are reachable via multiple paths in the genealogy DAG. Concretely, each path of length d from an ancestor to h contributes an additive discounted reward $\gamma^d r$. This design captures the intuition that heuristics reused across multiple intermediate generations exert a stronger influence on successful descendants. Consequently, ancestors that participate in multiple compositional chains are credited proportionally to their structural contribution, rather than being counted only once.

Algorithm 2 ATTRIBUTE()

Require: child h with parents $\mathrm{Pa}(h)$; clipped reward $r \geq 0$; discount factor γ; depth cap $D_{\max}$; round t; per-arm lineage credit history (deque) $\{\mathcal{C}_\ell\}$
Ensure: in-place update of $\{\mathrm{C}_\ell\}$

if $r \leq 0$ **or** $\mathrm{pa}(h) = \emptyset$ **then**
  return
end if
$\mathcal{F} \leftarrow \mathrm{Pa}(h)$; $d \leftarrow 1$
while $\mathcal{F} \neq \emptyset$ **and** $d \leq D_{\max}$ **do**
  $\mathcal{F}_{\mathrm{next}} \leftarrow \emptyset$
  for all $h' \in \mathcal{F}$ **do**
    $\ell' \leftarrow \texttt{LLM}(h')$
    $\mathrm{C}_{\ell'}.\mathrm{append}\big((t,\, \gamma^d \cdot r)\big)$
    for all $h'' \in \mathrm{pa}(h')$ **do**
      $\mathcal{F}_{\mathrm{next}} \leftarrow \mathcal{F}_{\mathrm{next}} \cup \{h''\}$
    end for
  end for
  $\mathcal{F} \leftarrow \mathcal{F}_{\mathrm{next}}$; $d \leftarrow d + 1$
end while

B.3 Recency Bias: Windowed Fertility Score

Algorithm 3 implements the sliding-window intensity of Equation (9). The pruning step is amortized $O(1)$ per credit because each (t', c) tuple is inserted into the history deque once and removed at most once.

Algorithm 3 FERTILITY()

Require: `LLM` ℓ; deque $\mathcal{C}_\ell$ of (t', c'); round t; window W; pull count n_ℓ
Ensure: $\bar{L}_\ell(t) \in \mathbb{R}_{\geq 0}$

if $\mathcal{C}_\ell = \emptyset$ **or** $n_\ell = 0$ **then**
  return 0
end if
$\mathrm{cutoff} \leftarrow t - W$
while $\mathcal{C}_\ell \neq \emptyset$ **and** $\mathcal{C}_\ell.\mathrm{front}().t' \leq \mathrm{cutoff}$ **do**
  $\mathcal{C}_\ell.\mathrm{popleft}()$
end while
$S \leftarrow \sum_{(t',c') \in \mathcal{C}_\ell^W} c'$
return $S \,/\, \log\big(\max(e,\, n_\ell)\big)$

B.4 Per-Arm Index

Algorithm 4 composes the three sub-procedures above into the index of Eq. (5).

B.5 Arm Selection

Algorithm 5 implements ROUTER.SELECTARM, the index policy that is queried once per round before prompt construction. Any arm that has not yet completed its warm-up receives score $+\infty$ and is returned immediately via round-robin tie-breaking; once all arms are warmed up, the arm with the highest $\mathrm{Score}_\ell(t)$ (Algorithm 4) is selected.

Algorithm 4 $\mathrm{Score}_\ell(t)$

Require: LLM ℓ; state $(\mathcal{O}_\ell, \mathcal{C}_\ell, n_\ell)$; hyper-parameters $\lambda, \tau, B, \alpha, \beta, W, n_{\min}$; horizon T; round t
Ensure: $\mathrm{Score}_\ell(t) \in \mathbb{R} \cup \{+\infty\}$

if $n_\ell < n_{\min}$ **then**
  return $+\infty$
end if
$\mathrm{base}(\ell) \leftarrow$ BASESCORE() ▷ Alg. 1
$\bar{L}_\ell(t) \leftarrow$ FERTILITY() ▷ Alg. 3
$exp_bonus \leftarrow \beta \sqrt{\log T \,/\, n_\ell}$
return $\mathrm{base}(\ell) + \alpha\, \bar{L}_\ell(t) + exp_bonus$

Algorithm 5 ROUTER.SELECTARM()

Require: LLMs $\mathcal{L}$; per-arm pull counts $\{n_\ell\}$; warm-up $n_{\min}$; state $\{(\mathcal{O}_\ell, \mathcal{C}_\ell, n_\ell)\}_{\ell \in \mathcal{L}}$; hyper-parameters $\lambda, \tau, B, \alpha, \beta, W$; horizon T; round t
Ensure: selected arm $\ell_t \in \mathcal{L}$

for all $\ell \in \mathcal{L}$ **do** ▷ warm-up pass
  if $n_\ell < n_{\min}$ **then**
    return ℓ ▷ round-robin forced exploration
  end if
end for
$\ell_t \leftarrow \arg\max_{\ell \in \mathcal{L}} \ \mathrm{Score}_\ell(t)$ ▷ Alg. 4
return ℓ_t

B.6 Router Update

Algorithm 6 implements ROUTER.UPDATE, the single entry point called by the host after each evaluation batch. It clips raw scores into non-negative rewards, appends them to the per-arm observation buffer $\mathcal{O}_\ell$, and delegates lineage credit propagation to ATTRIBUTE (Algorithm 2).

Algorithm 6 ROUTER.UPDATE()

Require: LLM ℓ; generated functions $\{h_{t,i}\}_{i=1}^m$; raw scores $\{s_{t,i}\}_{i=1}^m$; clip threshold b_t; per-arm buffers $\mathcal{O}_\ell$, $\mathcal{C}_\ell$, n_ℓ; round t
Ensure: in-place update of $\mathcal{O}_\ell$, $\mathcal{C}_\ell$, n_ℓ

for all $i \in \{1, \ldots, m\}$ **do**
  $r_{t,i} \leftarrow \max(0,\ s_{t,i} - b_t)$ ▷ clip to non-negative reward
  $\mathcal{O}_\ell.\mathrm{append}(s_{t,i})$ ▷ raw score to observation buffer
  ATTRIBUTE$(h_{t,i},\ r_{t,i})$ ▷ Alg. 2; propagates credit up DAG
end for
$n_\ell \leftarrow n_\ell + m$

B.7 FunSearch Pipeline Integration

The router is invoked at exactly two points in the loop: ROUTER.SELECTARM() before generating candidates, and ROUTER.UPDATE() after their evaluation. All other components are handled by the host framework, including maintaining the heuristic set $\mathcal{H}$, constructing prompts $\mathcal{A}(h_{<t})$, sampling candidates via μ_ℓ, and evaluating them using $f_{\mathcal{P}}$. The genealogical DAG $\mathcal{G}_T$ serves as the shared state, where each heuristic is annotated with its generating arm and parent set. The router maintains all per-arm statistics, hyperparameters, and intermediate state internally and is updated each round with the generated heuristics, their scores, parent sets, and generating LLM. For simplicity, we omit explicit argument passing in ROUTER.SELECTARM() and ROUTER.UPDATE(). Algorithm 7 summarizes the overall procedure.

The separation of concerns is clean: the host framework owns $\mathcal{D}$, prompt formatting, and evaluation; the router owns $\mathcal{A}$ selection and the per-arm score buffers $\{\mathcal{O}_\ell\}$. No information beyond the scalar scores $\{s_{t,i}\}$ and the arm identity a_t needs to cross the interface.

Algorithm 7 FunSearch [18] with BALLAD router

Require: seed program h_0; horizon T; batch size m; router hyper-parameters $\{\Phi = (\lambda,\ \tau,\ B,\ \beta,\ n_{\min},\ \alpha,\ W,\ \gamma,\ D_{\max})$; evaluator $f_{\mathcal{P}}$; LLM output distribution $\{\mu_\ell\}_{l\in\mathcal{L}}$

Ensure: best heuristic $\hat{h} = \arg\max_{h\in\mathcal{H}} f_{\mathcal{P}}(h)$

$\mathcal{H} \leftarrow$ INITDATABASE(h_0)
ROUTER $\leftarrow$ INITROUTER$(\mathcal{A}, T, \Phi)$
for $t = 1, \ldots, T$ **do**
  $\ell_t \leftarrow$ ROUTER.SELECTARM() ▷ **router in**; see Alg. 5
  $H_t \leftarrow \mathcal{A}(h_{<t})$
  $\{h_{t,i}\}_{i=1}^{m} \leftarrow \mu_{\ell_t}(\ell_t,\ H_t)$
  for all $i \in \{1, \ldots, m\}$ **do**
    LLM$(h_{t,i}) \leftarrow \ell_t$;
    $s_{t,i} \leftarrow f_{\mathcal{P}}(h_{t,i})$
    $\mathcal{H} \leftarrow \mathcal{H} \cup (h_{t,i},\ s_{t,i})$
  end for
  ROUTER.UPDATE() ▷ **router out**; see Alg. 6
end for
return $\hat{h}$

B.8 Time Complexity

We analyse the per-round cost of each sub-procedure of BALLAD, then sum over T rounds to obtain the total router overhead.

BaseScore. Algorithm 1 draws B bootstrap resamples of size $|\mathcal{O}_\ell|$ for each arm ℓ. At round t the total buffer size across all $|\mathcal{L}|$ arms is $\sum_\ell |\mathcal{O}_\ell| = m \cdot t$, so the selection step costs $O(B \cdot m \cdot t)$ per round.

BFS Lineage Attribution. Algorithm 2 is called once per generated function. The BFS expands the ancestry DAG level by level: the direct parents of a new function form a frontier of size at most P' (the parent-set size); each of those nodes contributes up to P' grandparents at depth 2, yielding a frontier of size $(P')^2$; and so on. The total number of ancestor nodes visited across all depths is therefore

$$\sum_{d=1}^{D_{\max}} (P')^d \;=\; P' \,\frac{(P')^{D_{\max}} - 1}{P' - 1} \;=\; O\big((P')^{D_{\max}}\big)\,.$$

Since $D_{\max}$ and P' are small constants (e.g. $D_{\max} = 4$, $P' = 2$ in our setting), this is $O(1)$ per function. Applying it to all m functions generated in a round gives an update cost of $O(m)$ per round.

Fertility Score. Algorithm 3 prunes and sums the sliding-window deque for each of the $|\mathcal{L}|$ arms. Each credit tuple is appended once and removed once (amortised $O(1)$), and the live window contains at most W entries per arm, giving $O(|\mathcal{L}| \cdot W) = O(|\mathcal{L}|)$ per round for constant window size W.

Arm Selection. Algorithm 5 iterates over all $|\mathcal{L}|$ arms. For each warmed-up arm it calls $\mathrm{Score}_\ell(t)$ (Algorithm 4), which invokes BASESCORE at cost $O(B \cdot |\mathcal{O}_\ell|)$ and FERTILITY at amortised $O(1)$. Summing over arms, the per-round cost is $O(|\mathcal{L}| \cdot B \cdot m \cdot t)$. Since $|\mathcal{L}|$ is a small constant, this is $O(B \cdot m \cdot t)$, the same class as BASESCORE alone.

Router Update. Algorithm 6 loops over the m generated functions. Each iteration performs an $O(1)$ buffer append and one call to ATTRIBUTE, whose cost is $O\big((P')^{D_{\max}}\big) = O(1)$ for constant $D_{\max}$ and P'. The total per-round cost is therefore $O(m)$, already subsumed by the ATTRIBUTE term.

Overall. Summing over T rounds, the total router overhead is

$$\underbrace{O(B \cdot m \cdot T^2)}_{\text{BASESCORE+SELECTARM}} \;+\; \underbrace{O\big(m \cdot T \cdot (P')^{D_{\max}}\big)}_{\text{ATTRIBUTE+UPDATE}} \;+\; \underbrace{O(|\mathcal{L}| \cdot W \cdot T)}_{\text{FERTILITY}} \;=\; O(B \cdot m \cdot T^2),$$

since $D_{\max}$, P', W, and $|\mathcal{L}|$ are all small constants. The lineage components add only lower-order $O(m \cdot T)$ and $O(|\mathcal{L}| \cdot T)$ terms and do not change the asymptotic complexity relative to the base score computation.

In practice, this is negligible relative to the cost of LLM inference and sandboxed evaluation, which dominate each round by several orders of magnitude.

B.9 A Note on Regret Guarantees

Regret guarantees in the bandit literature rest on two assumptions: that rewards are stationary within an arm, and that arms are independent. Both fail here by construction. The genealogy couples the arms, since one model's output seeds the context another is prompted with (P2), and every arm's score distribution drifts as the population conditioning it improves (P3).

A bound can be recovered by positing a parametric form for an LLM's score distribution and for how that distribution shifts across iterations. We, however, deliberately avoid this since no such characterisation exists to the best of our knowledge.

C Benchmark CO Problems and Datasets

In this section, we describe the NP-hard combinatorial optimization problems used to benchmark the efficacy of BALLAD. Further, the dataset-specific splits and configurations are presented in Table 4.

Table 4: **Benchmark dataset configurations.** Problems are grouped by type. *Problem Size* denotes the average instance size: number of cities/nodes for routing problems, number of items for packing problems, and $(\overline{|V|}, \overline{|E|})$ for graph problems. Problem-specific constraints, such as vehicle capacity W (CVRP) and bin capacity C (BP), are listed in the *Constraints* column. Instance counts are given per split.

Problem	Problem Size	Constraints	Train	Val	Test
Routing					
TSP	50	-	64	64	1,000
CVRP	50	$W = 50$	10	64	64
Bin Packing					
Online BP	5,000	$C \in \{100, 500\}$	2	5	5
Offline BP	500	$C = 150$	5	64	64
Graph Matching					
GED (Linux)	$(8, 7)$	-	1,000	1,000	1,000

Travelling Salesman Problem . The Traveling Salesman Problem (TSP) is defined on a vertex set $V = \{v_1, v_2, \ldots, v_N\}$ with distance matrix $D = [d_{ij}] \in \mathbb{R}^{N \times N}$. A feasible solution is a Hamiltonian cycle (tour) $\mathcal{T} = (v_1, \ldots, v_N)$ visiting each vertex exactly once, with cost

$$\text{minimise } f_{\mathcal{P}}(\mathcal{T}) = \sum_{t=1}^{N-1} d_{\mathcal{T}_t, \mathcal{T}_{t+1}} + d_{\mathcal{T}_N, \mathcal{T}_1}. \tag{10}$$

In our setting, a heuristic h maps an instance $I \in \mathcal{I}_{\mathcal{P}}$ to a tour $\mathcal{T} = h(I)$, which is evaluated by $f_{\mathcal{P}}(\mathcal{T})$. Given a partial tour, the generated heuristic selects the next node to visit. We consider 64 problem instances with $N = 50$ nodes [26].

Capacitated Vehicle Routing Problem . The Capacitated Vehicle Routing Problem (CVRP) is defined on a vertex set $V = \{v_0, v_1, \ldots, v_N\}$, where v_0 denotes the depot and $\{v_1, \ldots, v_N\}$ are customer nodes, together with a distance matrix $D = [d_{ij}] \in \mathbb{R}^{(N+1)\times(N+1)}$, customer demands $\{\delta_i\}_{i=1}^{N}$, and vehicle capacity C. A feasible solution is a set of routes $\mathcal{R} = \{\mathcal{T}^1, \ldots, \mathcal{T}^q\}$, where each route $\mathcal{T}^j = (v_0, \mathcal{T}_1^j, \ldots, \mathcal{T}_{n_j}^j, v_0)$ starts and ends at the depot and visits a subset of customers.

The objective is to minimize the total travel cost,

$$f_{\mathcal{P}}(\mathcal{R}) = \sum_{j=1}^{q} \sum_{t=0}^{n_j} d_{\mathcal{T}_t^j,\, \mathcal{T}_{t+1}^j}, \tag{11}$$

subject to the capacity constraints

$$\sum_{v_i \in \mathcal{T}^j} \delta_i \leq C, \quad \forall j \in \{1, \ldots, q\}, \tag{12}$$

and the requirement that each customer is visited exactly once. In our setting, a heuristic h parameterizes an Ant Colony Optimization (ACO) procedure, which constructs a solution $\mathcal{R} \sim \mathrm{ACO}(I; h)$ for an instance $I \in \mathcal{I}_{\mathcal{P}}$, and is evaluated via $f_{\mathcal{P}}(\mathcal{R})$. We utilize the dataset introduced in [26] where the CVRP instances have the depot fixed at $(0.5, 0.5)$.

Bin Packing Problems . The Bin Packing Problem (BPP) is a classical NP-hard combinatorial optimization problem defined over a set of items with sizes $\{s_i\}_{i=1}^{N}$ and bins of fixed capacity W. The objective is to assign each item to a bin such that the total size of items in any bin does not exceed W, while minimizing the total number of bins used.

In the online setting, items arrive sequentially, and each assignment decision must be made irrevocably upon arrival, whereas in the offline setting, the full item set is available in advance. In our framework, a heuristic h maps an instance $I \in \mathcal{I}_{\mathcal{P}}$ to a packing solution, which is evaluated via $f_{\mathcal{P}}(h(I))$, where $f_{\mathcal{P}}$ denotes the number of bins used.

For the online setting, the heuristic is designed to extend a partial assignment step by step, selecting each next action based on the current state. In contrast, for offline bin packing, the heuristic is designed to construct a complete packing using an Ant Colony Optimization (ACO)–style procedure: items are processed sequentially, and each item is assigned to a bin by sampling from a probability distribution guided by pheromone values and feasibility information such as remaining bin capacity.

We utilize the dataset introduced in [26] for our study. This setup is consistent with prior work in the literature, including [18, 15].

Graph Edit Distance . Graph Edit Distance (GED) between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is defined as the minimum cost of transforming G_1 into G_2 through a sequence of edit operations, such as node and edge insertions, deletions, and substitutions. Formally, letting $\mathcal{M}$ denote the set of all possible node mappings between the two graphs, GED can be written as

$$\mathrm{GED}(G_1, G_2) = \min_{\pi \in \mathcal{M}} \mathrm{GED}_{\pi}(G_1, G_2), \tag{13}$$

where $\mathrm{GED}_{\pi}(G_1, G_2)$ is the edit cost induced by a particular node mapping π. Since the number of possible mappings grows factorially with graph size, exact computation is NP-hard.

We follow GRAIL's [23] formulation for GED approximation, where the prompt includes the task specification and the top-k programs discovered so far. Given a graph pair (G_1, G_2), a heuristic h generates a weight matrix W defining a weighted bipartite graph; maximum-weight bipartite matching then produces a node mapping π, which is used to compute the resulting edit cost.

The cost induced by a mapping π is given by

$$f_{\mathcal{P}}(h(G_1, G_2)) = \sum_{v_1 \in V_1} \mathbb{I}(L_1(v_1) \neq L_2(\pi(v_1))) + \frac{1}{2} \sum_{v_1 \in V_1} \sum_{v_2 \in V_1} \mathbb{I}(e_1(v_1, v_2) \neq e_2(\pi(v_1), \pi(v_2))), \tag{14}$$

where $L(\cdot)$ denotes node labels and $e(\cdot, \cdot)$ indicates edge existence. The objective is to

$$\min_{h} \; f_{\mathcal{P}}(h(G_1, G_2)). \tag{15}$$

While the original formulation targets a diverse set of complementary heuristics via a submodular objective, we instead use the same pipeline to learn a single heuristic that minimizes GED for each graph pair.

We evaluate on the Linux benchmark from [23], which consists of 1,000 unlabeled program dependence graphs extracted from the Linux kernel, with nodes representing statements and edges

capturing dependencies. The graphs are small (on average 8 nodes and 7 edges), making the dataset a standard testbed for GED heuristics.

Problem-specific seed programs and the best programs on the validation set, discovered across all runs of BALLAD have been provided in App. H and App. I, respectively.

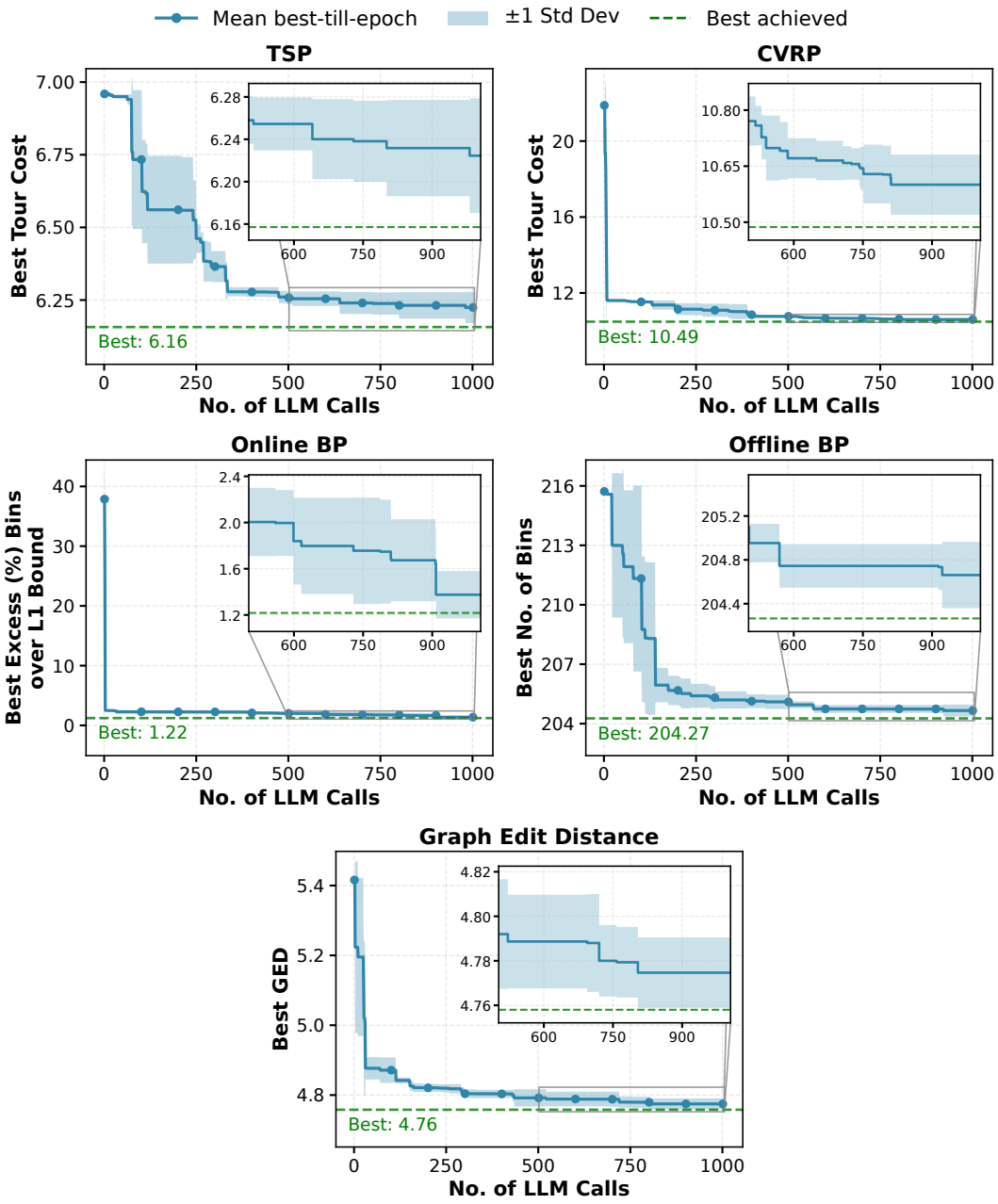

Figure 3: **Best test objective vs. number of LLM calls for BALLAD (lower is better).** After each call, all programs generated so far are evaluated on the test set, and the best-performing program is retained. The blue line shows the mean across three runs, with the shaded region indicating ± 1 standard deviation. The green dashed line marks the best objective achieved across all runs. Insets zoom in on the second half of the search to highlight smaller improvements. Since the best program is selected using test performance at each step, these curves provide an optimistic estimate of the results in Table 1, where selection is based on validation performance. The final mean is comparable to or better than the corresponding table value.

D Experimental Setup: Details

D.1 Hyperparameters.

In our experiments, we set $k = 2$ (number of heuristics in the prompt) and follow the sampling strategy of [18], with a total budget of $T = 1000$ evaluations. Hyperparameters are selected based on validation performance, with tuning restricted to the most sensitive ones (λ, α, and $D_{\max}$), while others are fixed to reasonable defaults from preliminary experiments. The index score (Eq. 5) uses $\lambda = 0.3$, $\alpha = 1.0$, and $\beta = 1.0$. The lineage term is computed with a sliding window $W = 25$ rounds, discount factor $\gamma = 0.9$, and maximum depth $D_{\max} = 4$. The bootstrap estimator uses $B = 200$ resamples at quantile $\tau = 0.9$, and a warm-up threshold of $n_{\min} = 3$ observations per `LLM`. For getting diverse outputs, the temperature hyperparameter of `LLMs` is set to 0.99. A complete list of hyperparameters is provided in Table 3. Further, we ran the evolutionary search, i.e., each run for a maximum of 24 hours, during which a majority of the time was spent evaluating the generated heuristics. To handle throttling due to API rate limits, a 10-minute backoff with a maximum of 5 retries was also applied.

D.2 Hardware Details.

Hardware details All experiments were performed on Intel(R) Xeon(R) Gold 6426Y: 64 cores, 126 GB RAM, 2 NVIDIA-L40S GPUs, 45 GiB each running on Ubuntu 20.04.6 LTS. The Gemini API was used to access `LLMs` `gemini-2.5-flash-lite` and `gemini-2.5-flash` under the paid tier, and `gemma-3-12b-it` and `gemma-3-27b-it` under the free tier. The evaluations of TSP, Online BP, and GED were CPU-bound, whereas ACO-based CVRP and Offline BP were GPU-based.

E LLM Calls vs. Test Performance.

Figure 3 shows how the best test objective of BALLAD evolves with the number of LLM calls. Across all problems, most improvements occur early in the search, with the initial calls yielding substantial gains. As the best program becomes stronger, finding better candidates becomes increasingly difficult, and progress slows to occasional, smaller improvements (insets). Nevertheless, these late improvements remain meaningful, as they refine an already strong program.

These curves provide an optimistic estimate because they report the best program on the test set, whereas Tables 1 and 2 report the program selected using validation data. The final means of the curves are therefore at or below the table values (curve vs. table: TSP 6.225 vs. 6.233; CVRP 10.600 vs. 10.685; Online BP 1.375 vs. 1.483; Offline BP 204.661 vs. 204.693; GED 4.775 vs. 4.789). These differences reflect the generalization gap between selecting programs on validation data and evaluating them on the test set. Since all models in Tables 1 and 2 follow the same validation-based selection procedure, the comparison between them remains consistent and fair.

F Discussion: Ablation Study

We here expand upon the ablation study results from Table 2, offering a granular discussion of how individual design choices influence performance across various problem dimensions. This analysis clarifies the mechanistic contribution of each algorithmic component to the overall discovery process.

Impact of Extreme Value Objective. Aligning with property [P1](Refer section 3), BALLAD optimizes for the upper-tail quality of the heuristic population rather than mean performance. This *tail-hunting* strategy enables it to outperform the UCB baseline on the majority of tasks, achieving a statistically robust lead on CVRP(10.685 ± 0.075 vs. 10.896 ± 0.088) where performance intervals show no overlap. While UCB exhibits a marginal edge on TSP(6.194 ± 0.051 vs. 6.233 ± 0.056) and BALLAD shows a lower mean on Online BP(1.483 ± 0.276 vs. 1.640 ± 0.616), these specific differences are not statistically significant due to overlapping standard deviation intervals. Ultimately, BALLAD delivers a superior aggregate objective value of 6.134 compared to UCB's 6.243, establishing extreme-value orchestration as a more potent approach for `AHD`.

The advantage also extends to more advanced mean-reward-based strategies. Both SW-UCB and THOMPSON SAMPLING yield worst objective values than BALLAD across all three tasks, with aggregate values of 6.314 and 6.517, respectively, compared to 6.134. This indicates that adapting reward estimates to recent observations or incorporating posterior uncertainty alone does not capture the benefits of explicitly prioritizing the upper tail of heuristic quality. The detailed implementation of all baselines has been provided in App. G.

Impact of λ (Max-Quantile Mixing). The parameter λ balances the stability of the bootstrap quantile ($\lambda = 0$) with the aggressive pursuit of empirical maximums ($\lambda = 1$). Table 2 confirms that neither extreme is optimal on its own. While the record-based approach ($\lambda = 1$) outperforms the conservative quantile-only version ($\lambda = 0$) on both TSP (6.250 ± 0.020 vs. 6.255 ± 0.043) and CVRP (10.729 ± 0.017 vs. 10.869 ± 0.086), it remains vulnerable to being "blinded" by non-replicable `LLM` outliers. The hybrid configuration used in BALLAD yields the best overall objective (6.134 ± 0.168) compared to the quantile-only baseline (6.242 ± 0.108), demonstrating that the quantile provides a reliable quality "floor", while the empirical maximum provides the necessary "ceiling" to drive frontier discovery.

Impact of Tail level τ. τ sets how far into the upper tail the base score reads in Eq. 5. BALLAD uses $\tau = 0.90$ by default. Both extremes degrade (Refer Table 2): at $\tau = 0.70$ the estimator reverts toward average-case behaviour, at $\tau = 0.99$ it rests on too few order statistics and becomes unstable (Online BP, ±0.733).

Impact of Lineage. Setting $\alpha = 0$ disables the propagation of rewards through the mutation genealogy, effectively treating each `LLM` as an isolated generator. This configuration results in a performance degradation across almost all metrics, including TSP (6.281 ± 0.130 vs. 6.233 ± 0.056) and CVRP (10.714 ± 0.334 vs. 10.685 ± 0.075). These results validate property [P2]: automated discovery is a combinatorial process where a "weaker" model may provide the foundational code necessary for a "stronger" model to achieve a breakthrough. While Online BP remains unchanged (1.483), the superior Overall objective of BALLAD (6.134) compared to the lineage-blind version (6.159) confirms that rewarding the causal influence of ancestor models is essential for orchestrating an effective multi-model search.

To further examine the robustness of the impact of lineage, we evaluate Offline BP and GED using the nine-model pool variant, BALLAD (Scaled, $\alpha = 0$) in Table 5. Lineage improves the objective on four of the five problems and ties on the fifth (Online BP, 1.483 in both settings). The comparison with the four-model configuration is more telling. With only four models, BALLAD matches the nine-model pool without lineage on Offline BP (204.693 vs. 204.734) and surpasses it on GED (4.789 vs. 5.178, with non-overlapping intervals). Adding five frontier models therefore does not compensate for removing lineage credit.

Impact of Fixed Horizon Exploration Bonus. Standard UCB algorithms utilize an exploration bonus that grows with the current round t, which can lead to excessive exploration in the final stages of a fixed-budget search. By instead calibrating the bonus to the total horizon T, BALLAD front-loads exploration when the budget is plentiful and shifts toward exploitation as the remaining budget depletes. As shown in Table 2, replacing this fixed-horizon schedule with the standard growing-bonus exploration ($\propto \sqrt{\log t}$) degrades overall performance, particularly on Online BP. This confirms that a budget-aware exploration schedule more effectively concentrates the final search effort on the most fertile models identified earlier in the run.

Impact of Cost-Penalized Exploration. Cost-penalised exploration probes the exploration term further, discounting each arm's bonus by its mean observed call cost via a softmax of the negative mean cost, i.e., $\rho(\ell) = e^{-\bar{c}_\ell} / \sum_{\ell' \in \text{paid}} e^{-\bar{c}_{\ell'}}$. Thus, the paid arms are discounted relative to one another in proportion to cost while free models (no cost observations) satisfy $\rho \equiv 1$.

Down-weighting exploration by price is worse on all three problems(Refer Table 2): it biases the search toward cheaper but weaker models, whereas in a discovery pipeline an arm's value lies in its capacity for rare breakthroughs. This substantiates section 4.4; the 65.6% saving is emergent, since an expensive model that underperforms simply accumulates few pulls.

Table 5: **Scaling to Frontier Models:** Objective values on Offline BP and GED for four- and nine-model pools, with and without lineage-aware exploitation. *Scaled* denotes the nine-model pool. **Bold** and underline denote the best and runner-up values, respectively.

Model	Offline BP	GED
BALLAD (Scaled, $\alpha = 0$)	204.734 ± 0.109	5.178 ± 0.238
BALLAD (Scaled)	$\mathbf{204.539 \pm 0.242}$	$\mathbf{4.706 \pm 0.026}$
BALLAD (Table 1, 4 models)	$\underline{204.693 \pm 0.323}$	$\underline{4.789 \pm 0.022}$

Table 6: **Cost Comparison:** Average cost per run (USD) of BALLAD (Scaled) versus pricing the same token volume entirely at GPT-5's rate. Relative savings are shown in parentheses.

Dataset	BALLAD (Scaled)	GPT-5 only	Savings (USD)
Offline BP	22.629	29.948	7.319 (24.4%)
GED	9.745	30.739	20.994 (68.3%)

Impact of Scaling to Frontier Models. We expand the arm pool from four to nine models: GPT-5, GPT-5 Mini, GPT-5 Nano, Claude Sonnet 5, Claude Haiku 4.5, Gemini 2.5 Flash, Gemini 2.5 Flash Lite, Gemma 3 27B, and Gemma 3 12B. We denote this variant BALLAD (Scaled). Due to the prohibitive cost of running frontier models, we restrict this study to Offline BP and GED. As shown in Table 5, the expanded pool improves quality on both problems. On GED, the standard-deviation intervals do not overlap, indicating a stronger improvement.

We analyze the cost of BALLAD (Scaled) by computing the total input and output tokens over the runs of BALLAD, pricing every call at the rate of the model that served it and averaging across runs. We compare this against pricing the identical token volume entirely at GPT-5's rate. This comparison assumes that GPT-5 alone would consume a comparable number of tokens, an assumption we do not verify empirically. Table 6 reports the cost per run. The saving is larger on GED because the router concentrates on cheaper arms there, whereas Offline BP sustains a higher allocation to strong models, consistent with the pattern reported in Section 4.4.

G `LLM` Routing Baselines

G.1 Random

The random router selects l_t uniformly at random from $\mathcal{L}$ at every round, independent of past outcomes.

G.2 UCB

We describe the UCB-based `LLM` routing baseline. At each round t, an arm $l_t \in \mathcal{L}$ is selected by maximizing the UCB index, and m heuristics are sampled and evaluated on $\mathcal{I}_{\mathcal{P}}$. To define the reward baseline, we maintain an elite pool of the top-S scoring programs observed so far. During the warm-up phase, when the pool contains fewer than S elements, the baseline B is set to the running maximum score. Once the pool is full, B is defined as the median score of the elite set. The utility signal for arm l_t is the mean relative improvement over this baseline,

$$R_t^{(j)} \;=\; \max\Big(0,\; S_t^{(j)} - B\Big)\,,$$

which credits an arm only when it advances the current elite frontier rather than rewarding absolute score levels. Using raw scores $S_t^{(j)}$ directly would penalize high-variance arms whose outputs are inconsistent but capable of rare, large improvements, precisely the arms most valuable in `AHD`. The baseline-relative reward may become sparse in practice, because once the search plateaus, most samples yield $R_t^{(j)} = 0$. This makes it difficult for any mean-based estimator to discriminate between arms in later rounds; however, exploration helps escape such plateaus.

For exploration, UCB uses a bonus $\beta\sqrt{\log t \,/\, (n_l + 1)}$ that grows with the elapsed round t, reflecting cumulative-regret guarantees in the classical bandit setting. In `AHD`, however, the horizon T is fixed, and the objective is a single best outcome rather than a cumulative sum, so a growing bonus over-explores in later rounds when the remaining budget is scarce. The algorithm used has been provided in Alg. 8.

Algorithm 8 UCB-based LLM Router

$\texttt{LLMs}\mathcal{L} = \{l_1, \ldots, l_n\}$, instances $\mathcal{I}_\mathcal{P}$,
Require: horizon T, batch size m, exploration β,
elite pool size S

Initialize $n_l \leftarrow 0,\ \bar{U}_l \leftarrow 0$ for all $l \in \mathcal{L}$
Initialize elite pool $\mathcal{E} \leftarrow \emptyset$
$B \leftarrow -\infty$
for $t = 1, 2, \ldots, T$ **do**
  Router selection:
  $l_t \leftarrow \arg\max_{l \in \mathcal{L}} \big(\bar{U}_l + \beta\sqrt{\frac{\log t}{n_l+1}}\big)$
  $\mathcal{U} \leftarrow \emptyset$
  for $j = 1$ to m **do**
    Sample $h_t^{(j)} \sim \mu(l_t, H_t)$
    $S_t^{(j)} \leftarrow \frac{1}{|\mathcal{I}_\mathcal{P}|}\sum_{I \in \mathcal{I}_\mathcal{P}} f_\mathcal{P}(h_t^{(j)}, I)$
    Update $\mathcal{E}$ with $(h_t^{(j)}, S_t^{(j)})$ (keep top-S)
    if $|\mathcal{E}| < S$ **then**
      $B \leftarrow \max(B, S_t^{(j)})$
    else
      $B \leftarrow \text{median}\big(\{s : (h, s) \in \mathcal{E}\}\big)$
    end if
    $R_t^{(j)} \leftarrow \max(0,\ S_t^{(j)} - B)$
    Append $R_t^{(j)}$ to $\mathcal{U}$
  end for
  Aggregate round utility:

$$U_t \ \leftarrow \ \frac{1}{m}\sum_{u \in \mathcal{U}} u$$

  Update arm statistics:

$$n_{l_t} \ \leftarrow \ n_{l_t} + 1, \qquad \bar{U}_{l_t} \ \leftarrow \ \bar{U}_{l_t} + \frac{U_t - \bar{U}_{l_t}}{n_{l_t}}$$

end for

G.3 SW-UCB

Sliding-Window UCB [11] targets non-stationarity. The usefulness of an LLM may drift as the population evolves; for example, a model that contributes early can stop producing improvements once the search moves past the regime it is good at. Plain UCB responds to such drift slowly, because it averages over all time. SW-UCB is identical to UCB except that arm statistics are computed only over the most recent W rounds.

Let $n_l^W(t)$ be the number of rounds among the last W in which l was selected, and let $\bar{U}_l^W(t)$ be the mean round utility U_τ over those rounds. The router then selects

$$l_t \ = \ \arg\max_{l \in \mathcal{L}} \left(\bar{U}_l^W(t) + \beta\sqrt{\frac{\log \min(t, W)}{n_l^W(t)}} \right).$$

Any arm with $n_l^W(t) = 0$ receives an infinite index, so an arm that has not been pulled within the window is forced to be re-tried. Rewards, the elite-pool baseline, and the round utility U_t are computed exactly as in Alg. 8. Capping the logarithm at $\log W$ also keeps the exploration bonus bounded. This partially mitigates the over-exploration of a growing bonus in late rounds noted above.

Algorithm 9 Thompson Sampling `LLM` Router

Require: `LLMs` $\mathcal{L}$, instances $\mathcal{I}_{\mathcal{P}}$, horizon T, batch size m, elite pool size S, prior $(\mu_0, \kappa_0, a_0, b_0)$, warm-up pulls $n_{\min}$

$(\mu_l, \kappa_l, a_l, b_l) \leftarrow (\mu_0, \kappa_0, a_0, b_0)$, $n_l \leftarrow 0$ for all $l \in \mathcal{L}$
Initialize elite pool $\mathcal{E} \leftarrow \emptyset$, $B \leftarrow -\infty$
for $t = 1, \ldots, T$ **do**
  if $t \le n_{\min}|\mathcal{L}|$ **then**
    $l_t \leftarrow l_{\lceil t/n_{\min} \rceil}$ ▷ warm-up
  else
    for each $l \in \mathcal{L}$ **do**
      $\tilde{\sigma}_l^2 \sim \mathrm{InvGamma}(a_l, b_l), \quad \tilde{\theta}_l \sim \mathcal{N}(\mu_l, \tilde{\sigma}_l^2/\kappa_l)$
    end for
    $l_t \leftarrow \arg\max_l \tilde{\theta}_l$
  end if
  Sample m heuristics from l_t, evaluate on $\mathcal{I}_{\mathcal{P}}$, and compute $R_t^{(j)} = \max(0, S_t^{(j)} - B)$ as in Alg. 8
  Update $\mathcal{E}$ and B
  $\bar{R}_t \leftarrow \frac{1}{m}\sum_{j=1}^m R_t^{(j)}, \quad Q_t \leftarrow \sum_{j=1}^m \big(R_t^{(j)} - \bar{R}_t\big)^2$
  $b_{l_t} \leftarrow b_{l_t} + \frac{1}{2}Q_t + \frac{\kappa_{l_t} m\,(\bar{R}_t - \mu_{l_t})^2}{2(\kappa_{l_t}+m)}$
  $\mu_{l_t} \leftarrow \frac{\kappa_{l_t}\mu_{l_t} + m\bar{R}_t}{\kappa_{l_t}+m}$
  $\kappa_{l_t} \leftarrow \kappa_{l_t} + m, \quad a_{l_t} \leftarrow a_{l_t} + \frac{m}{2}$
  $n_{l_t} \leftarrow n_{l_t} + 1$
end for

G.4 Thompson Sampling

Thompson Sampling [21] replaces the explicit exploration bonus with posterior sampling. Each individual reward from arm l is modelled as $R \sim \mathcal{N}(\theta_l, \sigma_l^2)$, where both the mean and the variance are unknown. Each arm carries a conjugate Normal-Inverse-Gamma posterior $\mathrm{NIG}(\mu_l, \kappa_l, a_l, b_l)$, initialised at a shared prior $(\mu_0, \kappa_0, a_0, b_0)$. Since rewards are clipped at zero, the Gaussian likelihood is a working approximation. Its purpose is to provide tractable conjugate updates, not to be an exact model of the reward distribution.

At each round, one draw of the mean is taken from every arm's posterior, and the arm with the largest draw is pulled. Arms with wide posteriors are thus sampled optimistically (exploration), while arms with tight, high-mean posteriors are pulled more often (exploitation). After the round, the posterior of l_t is updated in closed form with its m rewards. Unlike UCB, which aggregates each round into a single utility U_t, the posterior is updated with all m individual rewards. Confidence in an arm therefore grows with the number of evaluated heuristics rather than with the number of rounds.

Before sampling drives selection, a warm-up phase pulls each arm $n_{\min}$ times in sequence, so that every posterior has been updated at least once. We use $(\mu_0, \kappa_0, a_0, b_0) = (0, 1, 2, 1)$ and $n_{\min} = 1$. The full procedure, including the posterior update, is given in Alg. 9.

H Seed Heuristics:

For each combinatorial optimization problem, FunSearch [18] begins with a seed program annotated by `@funsearch.evolve`. The function below, `priority_v0`, is the initial, deliberately simple, heuristic that the `LLM` is asked to evolve. Its docstring defines the *contract* (inputs, outputs, problem semantics) that every evolved variant must respect.

H.1 Travelling Salesman Problem (TSP, constructive)

Idea. Greedy nearest-neighbour: among unvisited nodes, pick the one closest to the current node.

```python
@funsearch.evolve
def priority(current_node: int, destination_node: int,
             unvisited_nodes: set, distance_matrix: np.ndarray) → int:
  """
    Selects the next node to visit in the Travelling Salesman Problem
    (TSP) tour.

    The TSP aims to find a minimum-cost tour that visits each node exactly
    once and returns to the starting node. This 'priority' function
    defines a heuristic for choosing the next node during the construction
    of the tour.

    Args:
        current_node : int
            The index of the node currently being visited.
        destination_node : int
            The index of the tour's starting node (used for returning at
                the end of the tour).
        unvisited_nodes : set[int]
            The set of node indices that have not yet been visited.
        distance_matrix : np.ndarray
            A symmetric 2D array of shape (n, n) where entry [i, j]
                represents
            the Euclidean distance between node `i` and node `j`.

    Returns:
        int
            The index of the next node to visit, selected from `
                unvisited_nodes`
            according to the heuristic used in this function.
  """
  scores = {}
  for node in unvisited_nodes:
      scores[node] = distance_matrix[current_node, node]
  next_node = min(scores, key=scores.get)
  return next_node
```

Listing 1: TSP — `priority_v0`

H.2 Capacitated Vehicle Routing Problem (CVRP (ACO))

Idea. Inverse-distance heuristic: nearby edges are more attractive. Demand and capacity are ignored at v0.

```python
@funsearch.evolve
def priority(distance_matrix: np.ndarray, coordinates: np.ndarray,
             demands: np.ndarray, capacity: int) → np.ndarray:
  """
  Computes prior indicators of how promising it is to include each edge
  in a solution for the Capacitated Vehicle Routing Problem (CVRP).

  The CVRP extends TSP by using multiple vehicles with limited capacity
  to serve customers with known demands, minimizing total travel distance
  while respecting capacity constraints. The depot node is indexed by 0
  and serves as the starting and ending point for all vehicle routes.

  Args:
      distance_matrix : np.ndarray
          2D array of distances between all node pairs (shape: n by n).
      coordinates : np.ndarray
          2D array of Euclidean coordinates of nodes (shape: n by 2).
      demands : list or np.ndarray
```

```
        Vector of customer demand requirements (shape: n). The depot
            node
        (index 0) has demand = 0.
    capacity : int or float
        Integer capacity constraint representing the maximum capacity
        of each vehicle.

Returns:
    np.ndarray
        Heuristic matrix of the same shape as the distance_matrix, where
        entry [i, j] represents prior indicators of how promising it is
        to move from node i to node j, considering distance, demand,
        and capacity constraints.
"""
return 1 / distance_matrix
```

Listing 2: CVRP — `priority_v0`

H.3 Online Bin Packing

Idea. A handcrafted polynomial of remaining capacity and item size, with a sign flip for bins that can fit the item and a neighbour-difference smoothing pass.

```
@funsearch.evolve
def priority(item: float, bins_remain_cap: np.ndarray) → np.ndarray:
  """
    Computes a priority score for each bin in an Online Bin Packing
        problem.

    In the Online Bin Packing problem, items arrive sequentially and
    must be placed into bins with limited capacity. This ‘priority‘
    function assigns a heuristic score to each bin, indicating how
    suitable it is for accommodating the current item. The bin with the
    highest priority score is selected for placement.

    Args:
        item : float
            The size of the current item to be packed.
        bins_remain_cap : np.ndarray
            A 1D array where each element represents the remaining
                capacity
            of a corresponding bin.

    Returns:
        np.ndarray
            A 1D array of priority scores for all bins. Each score
                reflects the
            relative suitability of placing the current item in that bin.
            The bin with the highest score is considered the best choice.
  """
  max_bin_cap = max(bins_remain_cap)
  score = (bins_remain_cap - max_bin_cap)**2 / item + bins_remain_cap**2 /
      (item**2)
  score += bins_remain_cap**2 / item**3
  score[bins_remain_cap > item] = -score[bins_remain_cap > item]
  score[1:] -= score[:-1]
  return score
```

Listing 3: Online BPP — `priority_v0`

H.4 Offline Bin Packing (ACO)

Idea. Pairwise heuristic matrix: each row is the demand vector normalised by its maximum; capacity feasibility is left for evolution to discover.

```python
@funsearch.evolve
def priority(demand: np.ndarray, capacity: int) → np.ndarray:
  """
  Computes a heuristic matrix that represents the relative suitability of
  grouping items together in the same bin for a bin packing problem.

  The output is a square matrix where each entry [i, j] indicates how
  promising it is to place item i and item j in the same bin, considering
  their demand values. Pairs (i, j) whose combined demand exceeds the bin
  capacity should be assigned a lower heuristic score, as they cannot
  feasibly be placed in the same bin.

  Args:
      demand : np.ndarray
          1D array of demand values for n items (shape: (n,)).
      capacity : int
          Integer specifying the maximum capacity of each bin. This
              provides
          an upper limit that should guide which item pairs are
              practically
          feasible to be grouped together.

  Returns:
      np.ndarray
          2D heuristic matrix of shape (n, n).
          Entry [i, j] signifies the prior indicator of how suitable it is
          to assign item i and item j to the same bin, taking demand
          compatibility and capacity feasibility into account.
  """
  return np.tile(demand / demand.max(), (demand.shape[0], 1))
```

Listing 4: Offline BPP — `priority_v0`

H.5 Graph Edit Distance (GED)

Idea. A trivial seed: discard the input weights and return a zero matrix. Evolution is expected to discover how to refine the node-mapping probabilities using both adjacency matrices.

```python
@funsearch.evolve
def priority(graph1: list[list[int]], graph2: list[list[int]],
             weights: list[list[int]]) → list[list[float]]:
  """
    Computes the Graph Edit Distance (GED), a measure of the dissimilarity
    between two graphs.

    GED is defined as the minimum number of operations required to
    transform one graph into another. The primary operations considered
    in GED calculations include:

    - **Node Insertion/Deletion:** Adding or removing a node incurs a
      cost of +1.
    - **Edge Insertion/Deletion:** Adding or removing an edge between two
        nodes incurs a cost of +1.
    - **Node Relabeling:** Modifying the label of a node (if labels are
        present) adds a cost of +1 for each mismatch.

    Args:
        graph1: The adjacency matrix of the first graph.
        graph2: The adjacency matrix of the second graph.
        weights: A weight matrix representing the initial probabilities of
             mapping nodes           between ‘graph1‘ and ‘graph2‘.
                 Each entry is a probability value, where a higher value
                     indicates a higher likelihood and similarity of
                     mapping nodes. The size of the weight matrix is
                     determined by the maximum number of nodes in both
                     graphs squared.
```

```
    Returns:
        A refined weight matrix (float) using the initial input matrix and
             the adjacency matrices of graphs where each entry represents
            the probability of a node in `graph1` being mapped to a node in `graph2` in a way that minimizes the
            overall graph edit distance.
    """
    max_node = len(graph1)
    weights = [[0.0] * max_node for _ in range(max_node)]
    return weights
```

Listing 5: GED — `priority_v0`

I Best Discovered Heuristics

I.1 Travelling Salesman Problem (TSP)

Idea. Next-node selection via a weighted sum of immediate edge cost and an estimated future path cost computed by a greedy nearest-neighbour lookahead; the weight on immediate cost increases and the weight on future cost decreases as the tour progresses, transitioning from a lookahead regime at the start to a myopic nearest-neighbour regime near completion.

```
@funsearch.evolve
def priority_v2(current_node: int, destination_node: int, unvisited_nodes:
     set, distance_matrix: np.ndarray) → int:
    """
    Selects the next node to visit in the Travelling Salesman Problem (TSP)
        tour,
    prioritizing nodes that lead to a shorter estimated remaining tour, with
         a
    bias towards the immediate nearest neighbor when few nodes remain.

    This version is an improvement on `priority_v1` by more explicitly
    incorporating the total tour cost estimation into the priority
        calculation.
    It also uses a simpler, more robust dynamic weighting scheme.

    The logic:
    1. If only one node is unvisited, return it.
    2. For each `potential_next_node` in `unvisited_nodes`:
       a. Calculate the direct cost from `current_node` to `
            potential_next_node` (immediate cost).
       b. Estimate the "future path cost" from `potential_next_node` to the
            `destination_node`.
          This estimation is done by constructing a *temporary* greedy tour
              from
          `potential_next_node` through all other `unvisited_nodes - {
              potential_next_node}`,
          and finally back to `destination_node`.
       c. Dynamically determine weights for immediate and future costs:
          - As the number of unvisited nodes decreases (i.e., the tour
              progresses),
            the `w_immediate_cost` increases, and `w_future_path_cost`
                decreases.
          - This aims to make the heuristic behave more like a simple
              Nearest Neighbor
            at the end of the tour, and more like a lookahead-based
                heuristic at the beginning.
       d. Calculate the priority score as a weighted sum of immediate and
            future costs.
       e. Select the `potential_next_node` with the minimum priority score.

    Args:
        current_node : int
```

```
            The index of the node currently being visited.
        destination_node : int
            The index of the tour's starting node (used for returning at the
                 end of the tour).
        unvisited_nodes : set[int]
            The set of node indices that have not yet been visited.
        distance_matrix : np.ndarray
            A symmetric 2D array of shape (n, n) where entry [i, j]
                represents
            the Euclidean distance between node `i` and node `j`.

    Returns:
        int
            The index of the next node to visit, selected from `
                unvisited_nodes`
            according to the heuristic used in this function.
    """
    num_unvisited = len(unvisited_nodes)
    total_nodes = distance_matrix.shape[0]

    if num_unvisited == 1:
      return list(unvisited_nodes)[0]

    best_next_node = -1
    min_priority_score = float('inf')

    # --- Dynamic Weighting Strategy ---
    # The goal is to shift from a broader lookahead to a more myopic (
        nearest neighbor)
    # focus as the tour progresses.
    #
    # Let's define a "progress" metric. A simple way is to consider the
        fraction of
    # nodes remaining to be visited *after* the next step.
    # If num_unvisited = N-1 (start), progress is near 0.
    # If num_unvisited = 1 (end), progress is near 1.
    #
    # We can use this progress to adjust weights.
    # `progress_factor` will range from ~0 (at start) to ~1 (at end).
    # Let's use `(total_nodes - 1 - num_unvisited) / (total_nodes - 1)` as
        the progress factor.
    # However, if `total_nodes` is 1 or 2, the denominator can be zero.
    # A simpler and more robust way: use `num_unvisited` directly.
    #
    # When `num_unvisited` is large, `w_future` should be higher and `
        w_immediate` lower.
    # When `num_unvisited` is small, `w_immediate` should be higher and `
        w_future` lower.

    # Define base weights
    base_immediate_weight = 1.0
    base_future_weight = 0.8

    # Calculate an adjustment factor based on the number of unvisited nodes.
    # This factor will be higher when `num_unvisited` is small.
    # Let's scale `num_unvisited` relative to `total_nodes`.
    # A factor from 0 to 1, where 1 means `num_unvisited` is 1, and 0 means
        `num_unvisited` is `total_nodes - 1`.
    # This makes `adjustment` closer to 1 at the end of the tour, and closer
         to 0 at the start.

    if total_nodes <= 2: # No dynamic adjustment needed for trivial cases
        adjustment = 0.0
    else:
        # `num_unvisited` goes from `total_nodes - 1` down to `1`.
        # We want `adjustment` to be close to 1 when `num_unvisited` is 1.
        # We want `adjustment` to be close to 0 when `num_unvisited` is `
            total_nodes - 1`.
```

```
        # Linear mapping: `adjustment = (num_unvisited - 1) / (total_nodes - 2)`
        adjustment = (num_unvisited - 1) / (total_nodes - 2)

    # Adjust weights:
    # As `adjustment` increases (tour progresses), increase `w_immediate` and decrease `w_future`.
    w_immediate_cost = base_immediate_weight + adjustment * 0.8  # e.g., increases from 1.0 to 1.8
    w_future_path_cost = base_future_weight * (1 - adjustment * 0.6) # e.g., decreases from 0.8 to 0.8 * (1-0.6) = 0.32

    for potential_next_node in unvisited_nodes:
      # a. Calculate the direct cost from `current_node` to `potential_next_node`.
      immediate_cost = distance_matrix[current_node, potential_next_node]

      # b. Estimate the "future path cost" from `potential_next_node` to the `destination_node`.
      # This is a full greedy Nearest Neighbor tour starting from `potential_next_node`.
      future_path_cost = 0

      remaining_unvisited_for_future = unvisited_nodes - {potential_next_node}

      if not remaining_unvisited_for_future:
          # If this is the last unvisited node, the future path is just to the destination.
          future_path_cost = distance_matrix[potential_next_node, destination_node]
      else:
          # Perform a greedy Nearest Neighbor tour from `potential_next_node` through all remaining nodes.
          temp_current_for_future = potential_next_node
          temp_unvisited_for_future = copy(remaining_unvisited_for_future)

          while temp_unvisited_for_future:
              closest_remaining_node_in_future = -1
              min_dist_to_remaining_in_future = float('inf')

              for node in temp_unvisited_for_future:
                  dist = distance_matrix[temp_current_for_future, node]
                  if dist < min_dist_to_remaining_in_future:
                      min_dist_to_remaining_in_future = dist
                      closest_remaining_node_in_future = node

              future_path_cost += min_dist_to_remaining_in_future
              temp_current_for_future = closest_remaining_node_in_future
              temp_unvisited_for_future.remove(
                  closest_remaining_node_in_future)

          # Add the distance from the last visited node in the greedy path to the destination.
          future_path_cost += distance_matrix[temp_current_for_future, destination_node]

      # c. Calculate the priority score using dynamic weights.
      priority_score = (w_immediate_cost * immediate_cost) + \
                       (w_future_path_cost * future_path_cost)

      # d. Select the `potential_next_node` with the minimum priority score.
      if priority_score < min_priority_score:
        min_priority_score = priority_score
        best_next_node = potential_next_node

    return best_next_node
```

Listing 6: TSP (Constructive) — `priority_v2`

I.2 Capacitated Vehicle Routing Problem (ACO)

Idea. Edge heuristic matrix formed as a product of powered factors: inverse distance, destination-demand penalty, source-demand preference, destination peripheralness, source centrality, depot-return incentive, and a customer reachability score; edges involving customers whose demand exceeds vehicle capacity are zeroed and a small floor prevents stagnation.

```
@funsearch.evolve
def priority_v1(distance_matrix: np.ndarray, coordinates: np.ndarray,
    demands: np.ndarray, capacity: int) → np.ndarray:
  """
  An improved heuristic for CVRP edge selection, building upon ‘
      priority_v0‘.

  This version aims to refine the heuristic by:
  1.  **More Sophisticated Demand Handling:** Consider not just individual
       demand,
      but the *cumulative* demand on a potential route segment, relative
          to capacity.
      This is challenging to do purely statically. We’ll approximate by
          considering
      the demand of the *source* node more strongly and how it contributes
           to a partial route load.
  2.  **Dynamic Cost Adjustment (Static Approximation):** Make edge costs
      more sensitive to how they fit into a potential route. For example,
      edges that lead to a high concentration of remaining demand might be
      less preferred unless they are close to the depot.
  3.  **Customer "Reachability" Score:** Nodes that are "harder" to reach
      (further
      from clusters, higher demand) might be prioritized earlier in a
          route
      to ensure they are not missed.
  4.  **Balancing Inter-vehicle Load:** While a static heuristic can’t
      directly
      balance loads between vehicles, it can encourage choices that *tend*
           to
      lead to more balanced routes (e.g., by favoring shorter routes or
      routes that visit nodes with intermediate demands).
  5.  **Refined Peripheralness/Centrality:** Differentiate between
      peripheralness
      for the source and destination nodes.

  Args:
      distance_matrix : np.ndarray
          2D array of distances between all node pairs (shape: n by n).
      coordinates : np.ndarray
          2D array of Euclidean coordinates of nodes (shape: n by 2).
      demands : list or np.ndarray
          Vector of customer demand requirements (shape: n). The depot
              node
          (index 0) has demand = 0.
      capacity : int or float
          Integer capacity constraint representing the maximum capacity
          of each vehicle.

  Returns:
      np.ndarray
          Heuristic matrix of the same shape as the distance_matrix, where
          entry [i, j] represents prior indicators of how promising it is
          to move from node i to node j, considering distance, demand,
          and capacity constraints.
  """
```

```python
    num_nodes = distance_matrix.shape[0]
    safe_distance_matrix = np.where(distance_matrix == 0, 1e-9, distance_matrix)
    demands = np.asarray(demands)

    if capacity <= 0:
      return np.zeros_like(distance_matrix)

    # --- Factor 1: Inverse Distance ---
    # Always prefer shorter edges.
    inverse_distance = 1.0 / safe_distance_matrix

    # --- Factor 2: Destination Demand Considerations ---
    # Higher demand at destination is less preferred for the *current* route segment,
    # as it consumes capacity. But essential customers must be visited.
    # We penalize high demand destinations. A good approach is to consider demand relative to capacity.
    # Using 1 / (1 + (demand/capacity)^2) makes higher demands much less attractive.
    demand_destination_score = 1.0 / (1.0 + (demands / capacity)**2)
    demand_destination_matrix = np.tile(demand_destination_score, (num_nodes, 1))

    # --- Factor 3: Source Node Demand and Load Consideration ---
    # If node `i` has high demand *and* it's likely to be part of a route already,
    # departing from it might be less ideal if it leads to exceeding capacity.
    # This is a static approximation. We'll favor departing from nodes with lower *remaining* demand.
    # This is subtly different from v0: we want to penalize nodes that *already have high demand fulfilled*.
    # For a static heuristic, we'll use the current demand as a proxy for load.
    demand_source_score = np.zeros(num_nodes)
    # Prefer departing from nodes with lower demand.
    demand_source_score[1:] = 1.0 / (1.0 + demands[1:] / capacity)
    demand_source_score[0] = 1.0 # Depot is always a good source to start from.
    demand_source_matrix = np.tile(demand_source_score, (num_nodes, 1))

    # --- Factor 4: Peripheralness of Destination ---
    # This is about where a node is located geographically.
    # Peripheral nodes might be better visited towards the end of routes.
    # We'll use average distance to all other nodes as a proxy for peripheralness.
    avg_distances_from_node = np.mean(safe_distance_matrix, axis=1)
    max_avg_distance = np.max(avg_distances_from_node)
    # Higher score means more peripheral.
    peripheralness_score = (avg_distances_from_node / (max_avg_distance + 1e-9)) if max_avg_distance > 1e-9 else np.zeros(num_nodes)
    peripheralness_score[0] = 0.0 # Depot is not peripheral.
    peripheralness_destination_matrix = np.tile(peripheralness_score, (num_nodes, 1))

    # --- Factor 5: Centrality of Source ---
    # Conversely, starting from a more central node might be beneficial for initiating routes.
    # Let's define centrality as the inverse of peripheralness (or average distance to nearest neighbors).
    # A node close to many other nodes is central.
    # For simplicity, we'll use the inverse of peripheralness (but handle division by zero).
    centrality_score = np.zeros(num_nodes)
    # Centrality is higher for nodes with lower peripheralness score.
    centrality_score[1:] = 1.0 / (peripheralness_score[1:] + 1e-9)
```

```
    centrality_score[0] = 1.0 # Depot can be considered a highly central reference point.
    # Normalize centrality for consistent scaling.
    max_centrality = np.max(centrality_score)
    normalized_centrality = (centrality_score / (max_centrality + 1e-9)) if max_centrality > 1e-9 else np.ones(num_nodes)
    centrality_source_matrix = np.tile(normalized_centrality, (num_nodes, 1))

    # --- Factor 6: Depot Return Incentive ---
    # Strongly encourage returning to the depot from nodes that are far away or have significant demand.
    distance_to_depot = safe_distance_matrix[:, 0]
    max_dist_to_depot = np.max(distance_to_depot[distance_to_depot > 1e-9]) if np.max(distance_to_depot[distance_to_depot > 1e-9]) > 1e-9 else 1.0
    normalized_dist_to_depot = distance_to_depot / max_dist_to_depot
    normalized_demand = demands / capacity

    # Combine distance and demand to create a return incentive. Higher values mean stronger incentive.
    depot_return_incentive_from_node = normalized_dist_to_depot * 1.2 + normalized_demand * 1.8
    depot_return_incentive_from_node = np.clip(
        depot_return_incentive_from_node, 0.0, 5.0) # Cap at a higher value

    depot_return_matrix = np.tile(depot_return_incentive_from_node, (num_nodes, 1))

    # --- Factor 7: Customer "Reachability" Score (Approximation) ---
    # This is to ensure that "difficult" customers (e.g., far away, high demand) are not
    # left until the very end in a way that makes routes infeasible or inefficient.
    # Higher score means "harder to reach" or more critical to visit.
    reachability_score = np.zeros(num_nodes)
    # Combine normalized distance from depot and normalized demand.
    reachability_score[1:] = normalized_dist_to_depot[1:] + (demands[1:] / capacity)
    # Normalize reachability score
    max_reachability = np.max(reachability_score) if np.max(reachability_score) > 1e-9 else 1.0
    normalized_reachability = reachability_score / max_reachability
    normalized_reachability[0] = 0.0 # Depot is trivially reachable.

    # This score is applied to the destination node `j` of an edge `(i, j)`.
    reachability_destination_matrix = np.tile(normalized_reachability, (num_nodes, 1))


    # --- Combine factors ---
    # Adjusted weights for v1, emphasizing distance, demand handling, and reachability.
    # These weights are crucial for tuning the heuristic's behavior.
    weight_distance = 4.0              # Strong preference for shorter edges.
    weight_demand_dest = 2.5           # Penalize high demand destinations significantly.
    weight_demand_src = 1.2            # Slightly favor departing from low-demand sources.
    weight_peripheralness_dest = 1.5   # Favor peripheral destinations (for later in route).
    weight_centrality_src = 1.0        # Favor central sources (for earlier in route).
    weight_depot_return = 1.5          # Strong incentive to return to depot.
    weight_reachability = 1.8          # Prioritize "harder to reach" customers.
```

```
    # The heuristic is a product of powered factors.
    heuristic_matrix = (
        np.power(inverse_distance, weight_distance)
        * np.power(demand_destination_matrix, weight_demand_dest)
        * np.power(demand_source_matrix, weight_demand_src)
        * np.power(peripheralness_destination_matrix,
            weight_peripheralness_dest)
        * np.power(centrality_source_matrix, weight_centrality_src)
        * np.power(depot_return_matrix, weight_depot_return)
        * np.power(reachability_destination_matrix, weight_reachability)
    )

    # --- Post-processing ---

    # 1. Hard constraint: If a customer's demand exceeds vehicle capacity,
        no edge involving that customer is valid.
    # This is a critical CVRP constraint that cannot be purely handled by
        heuristics.
    invalid_demand_mask = demands > capacity
    # Mark edges where the destination node has demand exceeding capacity as
         invalid (heuristic = 0)
    heuristic_matrix[:, invalid_demand_mask] = 0.0
    # Mark edges where the source node has demand exceeding capacity as
        invalid.
    heuristic_matrix[invalid_demand_mask, :] = 0.0

    # 2. Ensure a minimum heuristic value for exploration.
    # Any edge that hasn't been explicitly set to 0 should have a small
        positive value.
    min_heuristic_value = 1e-6
    heuristic_matrix[heuristic_matrix < min_heuristic_value] =
        min_heuristic_value

    # Handle potential NaNs or Infs from calculations
    heuristic_matrix = np.nan_to_num(heuristic_matrix, nan=
        min_heuristic_value, posinf=1e9, neginf=-1e9)

    return heuristic_matrix
```

Listing 7: CVRP — `priority_v1`

I.3 Online Bin Packing

Idea. Per-bin priority score combining a tight-fit component, an ideal-fullness score, a future-capacity-potential reward, an item-to-bin harmony term, and an item-similarity proxy; scores for bins that cannot fit the current item are set to $-\infty$, and a small tie-breaking bias favours earlier bins.

```
@funsearch.evolve
def priority_v2(item: float, bins_remain_cap: np.ndarray) → np.ndarray:
  """Improved version of `priority_v1`."""
  """
    A refined priority function for Online Bin Packing, incorporating
        several advanced heuristics.

    This version aims to make more informed decisions by considering:
    1.  **Tight Fit:** Prioritizes bins where the item fits snugly,
        reducing wasted space.
    2.  **Ideal Fullness:** Encourages bins to be neither too full nor too
         empty, promoting
        a balanced distribution of items and minimizing fragmentation.
    3.  **Future Capacity Potential:** Rewards bins that will still have
        substantial
        remaining capacity after placing the item, making them more
            versatile for
        subsequent, potentially larger, items.
```

```
    4.  **Item Size Relation to Bin State:** Analyzes how the current item's size relates
        to the bin's current remaining capacity. It favors bins where the item occupies
        a "reasonable" proportion of the available space, avoiding situations where
        an item is too small for a bin or too large for the remaining space.
    5.  **Item Similarity to Existing Items:** (Introduced in v2) Considers the
        similarity of the current item to items already packed into a bin. This heuristic
        aims to group similar items together, which can sometimes lead to better packing
        solutions and potentially easier management or retrieval of items. Similarity
        is measured by the absolute difference between the item size and the sizes of
        items in the bin. Lower difference means higher similarity.
    6.  **Deterministic Tie-breaking:** Applies a small bias to favor bins that appear
        earlier in the `bins_remain_cap` array when scores are very close, ensuring
        consistent behavior.

    Args:
        item : float
            The size of the current item to be packed.
        bins_remain_cap : np.ndarray
            A 1D array where each element represents the remaining capacity
            of a corresponding bin.

    Returns:
        np.ndarray
            A 1D array of priority scores for all bins. Higher scores indicate
            a higher priority for placing the item in that bin.
    """
    num_bins = len(bins_remain_cap)
    if num_bins == 0:
      return np.array([], dtype=float)

    # Initialize scores to negative infinity for bins that cannot fit the item.
    scores = np.full(num_bins, -np.inf, dtype=float)

    # Identify bins that have enough remaining capacity to fit the current item.
    can_fit_mask = bins_remain_cap >= item
    can_fit_indices = np.where(can_fit_mask)[0]

    if len(can_fit_indices) > 0:
      current_remaining_capacities = bins_remain_cap[can_fit_indices]
      remaining_capacities_after_fit = current_remaining_capacities - item

      # --- Core Parameters ---
      BIN_CAPACITY = 100.0  # Assumed bin capacity
      ideal_remaining_capacity = BIN_CAPACITY / 2.0
      epsilon = 1e-9

      # --- Heuristic Components ---

      # 1. Tight Fit Score: Rewards bins where the item leaves minimal slack.
      slack = remaining_capacities_after_fit
      tight_fit_score = np.exp(-slack)

```

```
    # 2. Ideal Fullness Score: Rewards bins closer to being half-full.
    distance_from_ideal = np.abs(current_remaining_capacities -
        ideal_remaining_capacity)
    fullness_score = np.exp(-distance_from_ideal)

    # 3. Future Capacity Potential: Rewards bins that retain significant
        capacity.
    min_future_capacity_threshold = BIN_CAPACITY * 0.25  # e.g., at least
        25% capacity remaining
    ample_capacity_reward = np.zeros_like(current_remaining_capacities)
    ample_capacity_mask = remaining_capacities_after_fit >=
        min_future_capacity_threshold
    if np.any(ample_capacity_mask):
        capacity_above_threshold = remaining_capacities_after_fit[
            ample_capacity_mask] - min_future_capacity_threshold
        reward_scale = 0.4
        # Reward increases with capacity above the threshold.
        ample_capacity_reward[ample_capacity_mask] = 1.0 - np.exp(-
            capacity_above_threshold * reward_scale)

    # 4. Item Size Relation to Bin State: Rewards moderate item-to-
        capacity ratios.
    # This heuristic tries to avoid placing very small items in large
        empty bins
    # or trying to force large items into bins with little space.
    current_ratio = item / current_remaining_capacities
    target_ratio = 0.6  # Target for the item to fill of the current
        remaining capacity
    ratio_deviation = np.abs(current_ratio - target_ratio)
    # Score is higher when deviation is smaller. Penalize extreme ratios.
    item_size_harmony_score = np.exp(-ratio_deviation * 2.5)

    # 5. Item Similarity to Existing Items: (Requires knowledge of bin
        contents)
    # This is a placeholder as we only have remaining capacity. In a real
        scenario,
    # we would need to know the sizes of items already in each bin.
    # For this implementation, we'll simulate a similarity score based on
        how
    # 'close' the current item is to filling the bin to its 'ideal' state.
    # A more direct implementation would involve passing bin contents.
    # Here, we use the inverse of the slack as a proxy for similarity to
        what
    # would make the bin 'ideal' if it were filled perfectly by this item.
    # This is a simplification due to lack of direct bin content info.
    # If `item` is close to `current_remaining_capacities`, it means the
        item
    # is similar to the space that needs to be filled.
    # A high similarity score is desired.
    # Let's consider the difference between `current_remaining_capacities`
         and `item`.
    # If this difference is small, it's 'similar' in terms of filling.
    # This is effectively captured by `tight_fit_score` to some extent.
    # To make it distinct, let's focus on the absolute difference between
        the item
    # and what would make the bin *exactly* `ideal_remaining_capacity`.
    # This is implicitly handled by `fullness_score`.

    # Let's redefine this for v2: Similarity to *potential* future states.
    # A bin that is currently almost full and gets a small item is "
        similar" in that it's
    # close to being full. A bin that is very empty and gets a medium item
         is also
    # "similar" to a state where it's becoming moderately full.
    # A more direct interpretation: how close is `item` to `
        current_remaining_capacities`?
    # If `item` is very close to `current_remaining_capacities`, it means
        the bin has
```

```
    # a capacity that is "similar" to the item's size.
    # Let's use a score that rewards when the item size is "compatible"
        with the remaining capacity.
    # For example, if `current_remaining_capacities` is much larger than `
        item`, it's not very compatible.
    # If `current_remaining_capacities` is slightly larger than `item`, it
        's more compatible.
    # This can be modeled as a Gaussian around `item` = `
        current_remaining_capacities`.
    similarity_score = np.exp(-np.abs(current_remaining_capacities - item)
         * 0.5) # Tune scaling

    # --- Combine All Components ---
    # Weights for each heuristic component. These are crucial for tuning.
    w_fit = 1.5
    w_fullness = 0.8
    w_ample_capacity = 0.7
    w_item_size_harmony = 0.6
    w_similarity = 0.9  # Weight for the new similarity heuristic

    # Calculate the total score for the bins that can fit the item.
    current_scores = (
        w_fit * tight_fit_score +
        w_fullness * fullness_score +
        w_ample_capacity * ample_capacity_reward +
        w_item_size_harmony * item_size_harmony_score +
        w_similarity * similarity_score
    )
    scores[can_fit_indices] = current_scores

    # --- Tie-breaking Bias ---
    # Add a small bias to favor earlier bins in the array if scores are
        very close.
    bias_strength = 0.03
    tie_breaking_bias = bias_strength * (num_bins - can_fit_indices) /
        num_bins
    scores[can_fit_indices] += tie_breaking_bias

  return scores
```

Listing 8: Online BP — `priority_v2`

I.4 Offline Bin Packing (ACO)

Idea. Multi-faceted heuristic matrix combining a Gaussian tight-fit score, item importance, and additive bonuses for synergistic, complementary, and dominant-item pairings, all scaled by a multiplicative waste-minimisation factor; infeasible pairs are zeroed and the matrix is normalised to $[0, 1]$.

```
@funsearch.evolve
def priority_v2(demand: np.ndarray, capacity: int) -> np.ndarray:
  """
  A highly refined heuristic matrix generation function for the Bin
      Packing Problem,
  designed to provide superior guidance for Ant Colony Optimization (ACO).
  This version incorporates several advanced features to capture complex
      item interactions
  and packing efficiencies.

  Key features and improvements in v2:
  1.  **Multi-faceted Fit Scoring:**
      *   **Tight Fit (Gaussian):** Still a core component, rewarding
          pairs that leave
          minimal waste. `sigma` is tuned for sensitivity.
      *   **"Near-Perfect" Fit Bonus:** Explicitly rewards pairs that fit
          almost perfectly,
```

```
        adding a boost for very efficient packings beyond what the general Gaussian
        might emphasize.
    *   **"Dominant Item Gap" Bonus:** Focuses on scenarios where a large item is paired
        with a smaller item, ensuring the remaining gap is suitable for packing other
        items efficiently. This is distinct from general complementary gap scoring.

2.  **Advanced Item Pairing Heuristics:**
    *   **"Synergistic Pairing" Bonus:** Specifically rewards pairs of large items
        that fit together very snugly, aiming to 'pre-pack' difficult large items.
    *   **"Complementary Pairing" Bonus:** Rewards pairs of dissimilar sized items that
        leave a "strategic" gap, encouraging diversity in packing. The target gap range
        is more finely tuned.
    *   **Item Size Ratio Logic:** Incorporates the ratio of item sizes to dynamically
        adjust the value of certain pairings. For instance, a good complementary gap
        is more valuable if the items are very different in size.

3.  **Waste Management and Efficiency Focus:**
    *   **"Waste Minimization" Factor (Multiplicative):** A crucial factor that
        multiplicatively penalizes pairs with significant absolute waste. This
        ensures that even if a pair has good component scores, high waste will
        drastically reduce its heuristic value. It's applied only to non-perfect fits.
    *   **Feasibility Enforcement:** Strict enforcement of capacity constraints, setting
        infeasible pairs to zero.

4.  **Dynamic Thresholding and Normalization:**
    *   **Dynamic "Dominant Item" Thresholds:** Uses percentiles of demand to adapt
        the definition of "dominant" and "small" items to the specific instance's
        item size distribution.
    *   **Normalized Demand and Capacity Ratios:** Uses normalized demands and ratios
        of remaining capacity to capacity for scoring, making the heuristics more
        robust across different scales of demand and bin capacity.

5.  **Integrated Scoring and Weighting:**
    *   **Additive and Multiplicative Combinations:** Uses a combination of additive
        bonuses (like synergistic, complementary) on top of a core multiplicative
        heuristic (tight fit * item importance), all then modified by the waste
        minimization factor.
    *   **Tunable Weights:** Weights for different bonus components are provided,
        allowing for fine-tuning of the heuristic's behavior.

Args:
    demand : np.ndarray
        A 1D NumPy array representing the demand (size) of each item.
        Shape: (n,), where n is the number of items.
    capacity : int
```

```
        The maximum capacity of each bin.

    Returns:
        np.ndarray
            A 2D NumPy array representing the heuristic matrix of shape (n, n).
            Each entry `[i, j]` is a score between 0 and 1, indicating the
            suitability of placing item `i` and item `j` into the same bin.
    """
    n = demand.shape[0]
    heuristic_matrix = np.zeros((n, n))

    if n == 0:
      return heuristic_matrix

    # --- Normalization and Basic Calculations ---
    # Normalize demands to [0, 1] for consistent scoring.
    max_demand = np.max(demand)
    if max_demand == 0:
        normalized_demand = np.zeros_like(demand)
    else:
        normalized_demand = demand / max_demand

    # Calculate combined demand and remaining capacity for all pairs.
    combined_demand = demand[:, np.newaxis] + demand[np.newaxis, :]
    remaining_capacity = capacity - combined_demand

    # Create a mask for pairs that exceed bin capacity (infeasible).
    infeasible_mask = combined_demand > capacity

    # --- Component 1: "Tight Fit" Score (Gaussian Decay) ---
    # Rewards pairs that leave minimal space. A smaller sigma leads to a steeper penalty for waste.
    # Tuned sigma for sensitivity to waste. Smaller sigma means stricter requirement for small waste.
    sigma_tight_fit = max(1, capacity * 0.015) # Tighter sigma for even more sensitivity
    tight_fit_score = np.exp(- (remaining_capacity**2) / (2 * sigma_tight_fit**2))
    tight_fit_score[infeasible_mask] = 0 # Infeasible pairs get zero score for this component.

    # --- Component 2: "Item Importance" Score ---
    # Average normalized demand of the pair. Larger items contribute more.
    item_importance_score = (normalized_demand[:, np.newaxis] + normalized_demand[np.newaxis, :]) / 2.0

    # --- Core Heuristic (Multiplicative) ---
    # Combines tight fit and item importance. High score requires both efficiency and significant item sizes.
    core_heuristic = tight_fit_score * item_importance_score

    # --- Bonus Components (Additive) ---

    # Weighting factors for different bonus types. These are tunable.
    w_synergistic = 0.25
    w_complementary = 0.30
    w_dominant_gap = 0.35
    w_near_perfect = 0.10 # New: Bonus for near-perfect fits

    # --- Component 3: "Synergistic Pairing" Bonus ---
    # Rewards pairs of large items that fit together very snugly.
    large_item_threshold = 0.75  # Tunable: defines what is considered a "large" item.

    both_items_large_mask = (normalized_demand[:, np.newaxis] >= large_item_threshold) & \
```

```
                           (normalized_demand[np.newaxis, :] >=
                               large_item_threshold)

    # Bonus is applied when both items are large and fit with very little
        remaining capacity.
    synergy_capacity_limit = capacity * 0.05 # Tunable: maximum allowed
        waste for this bonus.
    synergy_capacity_mask = (remaining_capacity >= 0) & (remaining_capacity
        <= synergy_capacity_limit)

    # The bonus is proportional to how tightly they fit.
    synergistic_pairing_bonus = np.where(both_items_large_mask &
        synergy_capacity_mask, tight_fit_score, 0)

    # --- Component 4: "Complementary Pairing" Bonus ---
    # Rewards pairs of items with different sizes that leave a strategic gap
        .
    demand_diff_norm = np.abs(normalized_demand[:, np.newaxis] -
        normalized_demand[np.newaxis, :])

    # Ideal gap range for packing smaller items. Tunable.
    complementary_gap_lower_bound = capacity * 0.15
    complementary_gap_upper_bound = capacity * 0.40

    # Ensure the bounds are valid.
    if complementary_gap_upper_bound <= complementary_gap_lower_bound:
        complementary_gap_upper_bound = complementary_gap_lower_bound + 1.0

    # Score for being within the ideal gap range. Decays linearly.
    gap_range_width = complementary_gap_upper_bound -
        complementary_gap_lower_bound
    gap_penalty_factor = 1.0 / gap_range_width if gap_range_width > 0 else
        1.0

    # Score is 1 at the lower bound and decreases to 0 at the upper bound.
    complementary_gap_score = np.maximum(0.0, 1.0 - gap_penalty_factor * np.
        maximum(0.0, remaining_capacity - complementary_gap_lower_bound))

    # The bonus is proportional to the demand difference and how well the
        gap fits the 'strategic' range.
    # Higher demand difference (dissimilar items) paired with a good gap
        yields a higher bonus.
    complementary_pairing_indicator = demand_diff_norm *
        complementary_gap_score

    # Apply bonus only to feasible pairs within the specified gap range.
    complementary_mask = (remaining_capacity >=
        complementary_gap_lower_bound) & \
                         (remaining_capacity <=
                             complementary_gap_upper_bound)
    complementary_gap_bonus = np.where(complementary_mask & ~infeasible_mask
        , complementary_pairing_indicator, 0)


    # --- Component 5: "Dominant Item Gap" Bonus ---
    # Specifically rewards pairing a large item with a smaller item,
    # ensuring the remaining gap is reasonably sized for packing other items
        .
    # Dynamic thresholds for "dominant" and "small" items based on demand
        distribution.
    sorted_demand_norm = np.sort(normalized_demand)
    # If all demands are equal, use fixed thresholds.
    if np.all(normalized_demand == normalized_demand[0]):
        dominant_item_threshold_dynamic = 0.7
        small_item_threshold_dynamic = 0.2
    else:
        dominant_item_threshold_dynamic = np.percentile(sorted_demand_norm,
            80) # Top 20%
```

```
        small_item_threshold_dynamic = np.percentile(sorted_demand_norm, 30)
            # Bottom 30%

    # Ensure thresholds are ordered correctly, or fallback if they overlap significantly.
    if dominant_item_threshold_dynamic < small_item_threshold_dynamic:
        dominant_item_threshold_dynamic, small_item_threshold_dynamic =
            small_item_threshold_dynamic, dominant_item_threshold_dynamic
    if dominant_item_threshold_dynamic - small_item_threshold_dynamic <
        0.15: # Small margin for distinction
        dominant_item_threshold_dynamic = 0.7
        small_item_threshold_dynamic = 0.2

    # Mask for pairs where one item is dominant and the other is small.
    dominant_i_small_j_mask = (normalized_demand[:, np.newaxis] >=
        dominant_item_threshold_dynamic) & \
                              (normalized_demand[np.newaxis, :] <=
                                  small_item_threshold_dynamic)
    dominant_j_small_i_mask = (normalized_demand[np.newaxis, :] >=
        dominant_item_threshold_dynamic) & \
                              (normalized_demand[:, np.newaxis] <=
                                  small_item_threshold_dynamic)
    dominant_pairing_mask = dominant_i_small_j_mask |
        dominant_j_small_i_mask

    # The bonus is proportional to the demand difference and a normalized remaining capacity.
    # We reward moderate gaps for these pairings. `np.clip(
        remaining_capacity / capacity, 0, 0.5)`
    # assigns a score from 0 to 0.5 based on how much space is left relative to capacity.
    # A moderate gap is more valuable than a tiny or extremely large one for this bonus type.
    dominant_gap_bonus_score = np.where(dominant_pairing_mask & ~
        infeasible_mask,
                                        (demand_diff_norm / 2.0) * np.clip(
                                            remaining_capacity / capacity,
                                            0.0, 0.5),
                                        0.0)
    dominant_item_gap_bonus = dominant_gap_bonus_score


    # --- Component 6: "Near-Perfect Fit" Bonus ---
    # Rewards pairs with very little remaining capacity, independent of item size.
    near_perfect_capacity_limit = capacity * 0.02 # Max waste for this bonus
    near_perfect_mask = (remaining_capacity >= 0) & (remaining_capacity <=
        near_perfect_capacity_limit)
    # Bonus is proportional to the tightness of fit. Using tight_fit_score directly here.
    near_perfect_fit_bonus = np.where(near_perfect_mask & ~infeasible_mask,
        tight_fit_score, 0)


    # --- Multiplicative Waste Minimization Factor ---
    # This factor penalizes pairs that leave significant *absolute* wasted space,
    # unless they are perfectly filled. It's applied *after* additive bonuses.
    absolute_waste = np.maximum(0.0, capacity - combined_demand)
    max_possible_waste = capacity # Max waste for a single item.

    # Normalize waste to a 0-1 scale (0 = no waste, 1 = max waste).
    if max_possible_waste == 0:
        normalized_waste = np.zeros_like(absolute_waste)
    else:
        normalized_waste = absolute_waste / max_possible_waste

```

```
    # Exponential decay: exp(-k * normalized_waste). Higher k penalizes
        waste more.
    waste_penalty_factor = 8.0 # Tunable parameter.
    waste_minimization_factor = np.exp(-waste_penalty_factor *
        normalized_waste)

    # Perfect fills (remaining capacity close to 0) should not be penalized
        for waste.
    epsilon = 1e-9 # For floating point comparisons.
    perfect_fill_mask = np.abs(remaining_capacity) < epsilon
    waste_minimization_factor[perfect_fill_mask] = 1.0
    waste_minimization_factor[infeasible_mask] = 0.0 # Infeasible pairs get
        no score.


    # --- Final Combination of Scores ---

    # Combine core heuristic with weighted additive bonuses.
    combined_additive_bonuses = (w_synergistic * synergistic_pairing_bonus +
                                 w_complementary * complementary_gap_bonus +
                                 w_dominant_gap * dominant_item_gap_bonus +
                                 w_near_perfect * near_perfect_fit_bonus)

    # The final heuristic value is (core + bonuses) * waste_factor.
    # This ensures that efficiency (core) and strategic packing (bonuses)
        are
    # valued, but significantly penalized if they result in too much waste.
    heuristic_matrix = (core_heuristic + combined_additive_bonuses) *
        waste_minimization_factor

    # --- Final Processing ---
    # Re-apply infeasibility mask to ensure it's zero.
    heuristic_matrix[infeasible_mask] = 0.0

    # Normalize the entire matrix to the [0, 1] range.
    max_heuristic_value = np.max(heuristic_matrix)
    if max_heuristic_value > 0:
        heuristic_matrix = heuristic_matrix / max_heuristic_value
    else:
        # If all scores are zero (e.g., no feasible pairs), return a zero
            matrix.
        heuristic_matrix = np.zeros((n, n))

    return heuristic_matrix
```

Listing 9: Offline BP — `priority_v2`

I.5 Graph Edit Distance

Idea. Iterative weight-matrix refinement through six structural factors applied multiplicatively: node-degree similarity, local neighbourhood-degree similarity, global density similarity, edge-existence consistency, common-neighbour degree similarity, and path-length-2 consistency; each row is finally re-normalised to sum to one.

```
@funsearch.evolve
def priority_v2(graph1: list[list[int]], graph2: list[list[int]], weights:
     list[list[int]]) -> list[list[float]]:
  """
    An improved version of `priority_v1` that incorporates more
        sophisticated
    graph structural features and inter-node relationships for weight
        refinement.

    This version builds upon the heuristics in `priority_v1` and adds:

```

```
    1.  **Node Label Similarity (if available):** If nodes had labels (represented
        by feature vectors `nf1` and `nf2` in the `GEDInstance`), this version
        would consider their similarity. For simplicity in this standalone function,
        this part is omitted but conceptually important.
    2.  **Common Neighbors Degree Similarity:** For a mapping of node `i` to `j`,
        this considers pairs of common neighbors (`k` from `graph1`, `l` from `graph2`)
        and boosts the `(k, l)` mapping probability if `k` and `l` have similar degrees.
        This reinforces structural consistency at the local neighborhood level.
    3.  **Iterative Refinement:** While not a full iterative process, this function
        applies a multi-stage refinement, allowing earlier adjustments to influence
        later ones.

    Args:
        graph1: The adjacency matrix of the first graph.
        graph2: The adjacency matrix of the second graph.
        weights: An initial weight matrix, where `weights[i][j]` represents the
                 initial likelihood of mapping node `i` from `graph1` to node `j`
                 from `graph2`.

    Returns:
        A refined weight matrix (list of lists of floats) where each entry
        `refined_weights[i][j]` represents an adjusted probability of mapping
        node `i` to node `j`, informed by more advanced graph structural
        similarities. The weights are normalized such that each row sums to 1.
    """
    n1 = len(graph1)
    n2 = len(graph2)
    max_node = max(n1, n2)

    # Initialize refined_weights with input weights, padding if necessary
    refined_weights = [[0.0] * max_node for _ in range(max_node)]
    for i in range(min(n1, len(weights))):
      for j in range(min(n2, len(weights[0]))):
        refined_weights[i][j] = weights[i][j]

    # Pre-calculate degrees for efficiency
    degrees1 = [sum(row) for row in graph1]
    degrees2 = [sum(row) for row in graph2]

    # --- Heuristics from priority_v1 (kept as they are foundational) ---

    # 1. Node Degree Similarity
    for i in range(n1):
      for j in range(n2):
        max_degree_val = max(n1, n2) - 1
        if max_degree_val <= 0:
            degree_sim = 1.0
        else:
            degree_diff_normalized = abs(degrees1[i] - degrees2[j]) / max_degree_val
            degree_sim = 1.0 - degree_diff_normalized
        # Increase weight based on similarity, adding a small bias
        refined_weights[i][j] *= (degree_sim + 0.05)

    # 2. Local Neighborhood Similarity (based on neighbor degrees)
```

```
  for i in range(n1):
    for j in range(n2):
      neighbors_i = [k for k in range(n1) if graph1[i][k] == 1]
      neighbors_j = [l for l in range(n2) if graph2[j][l] == 1]

      neighbor_degrees_i = sorted([degrees1[ni] for ni in neighbors_i])
      neighbor_degrees_j = sorted([degrees2[nj] for nj in neighbors_j])

      len_i, len_j = len(neighbor_degrees_i), len(neighbor_degrees_j)
      max_len = max(len_i, len_j)

      if max_len == 0:
        neighborhood_sim = 1.0
      else:
        padded_i = neighbor_degrees_i + [0] * (max_len - len_i)
        padded_j = neighbor_degrees_j + [0] * (max_len - len_j)

        max_possible_degree = max(n1, n2)
        normalized_i = [d / (max_possible_degree if max_possible_degree > 0 else 1.0) for d in padded_i]
        normalized_j = [d / (max_possible_degree if max_possible_degree > 0 else 1.0) for d in padded_j]

        total_diff = sum(abs(ni - nj) for ni, nj in zip(normalized_i, normalized_j))
        avg_diff = total_diff / max_len
        neighborhood_sim = 1.0 - avg_diff

      # Boost weight based on neighborhood similarity
      refined_weights[i][j] *= (1 + neighborhood_sim * 0.4)

  # 3. Global Graph Structure Similarity (Density)
  density1 = sum(sum(row) for row in graph1) / (n1 * (n1 - 1) if n1 > 1 else 1e-6)
  density2 = sum(sum(row) for row in graph2) / (n2 * (n2 - 1) if n2 > 1 else 1e-6)
  density_sim = 1.0 - abs(density1 - density2)

  for i in range(n1):
    for j in range(n2):
      refined_weights[i][j] *= density_sim

  # 4. Edge Existence Similarity (based on neighbor mapping probabilities)
  for i in range(n1):
    for j in range(n2):
      edge_similarity_score = 0.0
      count_edges = 0
      for k in range(n1):
        if graph1[i][k] == 1:
          for l in range(n2):
            if graph2[j][l] == 1:
              edge_similarity_score += refined_weights[k][l]
              count_edges += 1

      if count_edges > 0:
        avg_edge_neighbor_prob = edge_similarity_score / count_edges
        refined_weights[i][j] *= (1 + avg_edge_neighbor_prob * 0.2)

  # --- New Heuristics for v2 ---

  # 5. Common Neighbors Degree Similarity
  # For a mapping (i, j), consider pairs of common neighbors (k, l) where k is a neighbor of i
  # and l is a neighbor of j. Boost the mapping (k, l) if k and l have similar degrees.
  for i in range(n1):
    for j in range(n2):
      for k in range(n1):
```

```
        if graph1[i][k] == 1:  # k is a neighbor of i
          for l in range(n2):
            if graph2[j][l] == 1:  # l is a neighbor of j
              # Calculate similarity of degrees for these neighbors k and l
              max_degree_val = max(n1, n2) - 1
              if max_degree_val <= 0:
                  degree_sim_k_l = 1.0
              else:
                  degree_diff_k_l = abs(degrees1[k] - degrees2[l])
                  degree_sim_k_l = 1.0 - (degree_diff_k_l / max_degree_val)
              # Boost the probability of mapping k to l if their degrees are similar
              # This reinforces structural consistency around the i-j mapping
              refined_weights[k][l] *= (1 + degree_sim_k_l * 0.1) # Weight factor for this heuristic

  # 6. Path Consistency (considering paths of length 2)
  # For a mapping (i, j), if there's a path i → k → m in graph1,
  # and we map i→j, k→p, m→q, we expect a higher probability for mapping
  # p to q if p and q have similar structural properties.
  for i in range(n1):
    for j in range(n2):
      path_consistency_score = 0.0
      path_count = 0
      for k in range(n1):
        if graph1[i][k] == 1: # Intermediate node k
          for m in range(n1):
            if graph1[k][m] == 1 and m != i: # Path i → k → m
              path_count += 1
              # Find potential mappings for k and m
              for p in range(n2):
                if refined_weights[k][p] > 1e-6: # If k can map to p
                  for q in range(n2):
                    if refined_weights[m][q] > 1e-6 and q != j: # If m can map to q (and q is not j)
                      # We want to see if the mapping p → q is consistent with their properties
                      # This can be measured by degree similarity, neighborhood similarity etc.
                      max_degree_val = max(n1, n2) - 1
                      if max_degree_val <= 0:
                          degree_sim_p_q = 1.0
                      else:
                          degree_diff_p_q = abs(degrees1[k] - degrees2[p]) # Mistake: should be degrees1[k] vs degrees2[p]
                          # Correcting: it should be degrees of original nodes k and m mapped to p and q
                          degree_diff_k_p = abs(degrees1[k] - degrees2[p])
                          degree_diff_m_q = abs(degrees1[m] - degrees2[q])
                          avg_degree_sim_pq = (
                              (1.0 - (degree_diff_k_p / max_degree_val if max_degree_val > 0 else 0)) +
                              (1.0 - (degree_diff_m_q / max_degree_val if max_degree_val > 0 else 0))
                          ) / 2.0
                          degree_sim_p_q = avg_degree_sim_pq # Using average similarity of mapped neighbors' degrees

                      # Boost the probability of mapping i→j if its direct neighbors can map to
                      # nodes that are structurally similar to the mapped intermediate nodes.
```

```
                        # This is a complex interaction. A simpler approach is to boost based on
                        # the product of probabilities of intermediate mappings and their structural similarity.
                        path_consistency_score += (
                            refined_weights[k][p] * refined_weights[m][q] * (degree_sim_p_q + 0.1) # Add a small bias
                        )

            if path_count > 0:
                # This heuristic is tricky. We are trying to influence the i->j mapping
                # based on the consistency of mappings of its neighbors' paths.
                # A direct boost to refined_weights[i][j] might be too simplistic.
                # It's more about ensuring that if a path exists in G1, similar paths are likely in G2.
                # For now, let's simplify: if a node has many paths, and its potential mappings
                # lead to consistent downstream mappings, it reinforces the initial mapping.
                # A simpler heuristic here could be to just consider the number of common neighbors
                # of mapped neighbors, which is already captured in heuristic 5.
                # Let's try a simpler boost for now.
                # If there are many paths originating from i, and the downstream mappings are consistent,
                # it means i is structurally important.
                # The contribution of path_consistency_score might be better used to adjust
                # the initial weights of the direct mappings (i,j).
                # For now, let's use a simplified version: if the node has many paths and the average
                # downstream consistency is high, boost the node's initial mapping probability.
                avg_path_consistency = path_consistency_score / (path_count * n2 * n2) if (path_count * n2 * n2) > 0 else 0 # Normalize
                refined_weights[i][j] *= (1 + avg_path_consistency * 0.3) # Weight factor

    # --- Normalization ---
    for i in range(n1):
        row_sum = sum(refined_weights[i][:n2])
        if row_sum > 0:
            for j in range(n2):
                refined_weights[i][j] /= row_sum
        else:
            # If row_sum is 0, distribute probability equally among valid targets
            if n2 > 0:
                for j in range(n2):
                    refined_weights[i][j] = 1.0 / n2
            else:
                for j in range(n2):
                    refined_weights[i][j] = 0.0

    # Zero out weights for mappings to non-existent nodes.
    for i in range(n1):
        for j in range(n2, max_node):
            refined_weights[i][j] = 0.0
    for i in range(n1, max_node):
        for j in range(max_node):
            refined_weights[i][j] = 0.0

    return refined_weights
```

Listing 10: GED — priority_v2

J Declaration of LLM Usage

LLMs were used to assist with auxiliary aspects of this work. Specifically, they were employed to rephrase text and reduce verbosity in the manuscript, as well as to generate scripts for standard procedures, including data processing, plotting, and statistical analysis of results.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract makes two claims: (i) BALLAD performs similarly or outperforms the best LLM in hindsight for every benchmarked problem. Showcased by Table 1. (ii) BALLAD incurs significantly less cost than the strongest, most expensive model in a run, which is showcased in Fig. 2.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We have listed the limitations of our proposed method in section 5.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The algorithms corresponding to our methodology have been stated in App. B. We have provided the hyperparameters in App. D.1. However, discovery pipelines use `LLM`s with high temperature since diversity of generations is key to finding breakthrough heuristics. Therefore for reproductibility, we provide the best discovered programs for each problem in App. I.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We provide the link to an anonymous repository containing our datasets and code in section 4.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The dataset details have been shared in App. C. The hyperparameters are mentioned in App. D.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: All experiments were conducted across multiple runs. The tables and plots report the average and standard deviation of objectives/rankings/cost across runs.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The hardware details have been specified in App. D.2.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS Code of Ethics and confirm that this work complies with its principles, with no involvement of human subjects or sensitive data.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The impacts of the proposed work have been discussed in section 5.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.

- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: This work does not involve the release of new datasets, models, or systems with potential for misuse; it focuses on a methodological contribution for LLM routing using existing tools.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All external assets (datasets, code, and pre-trained models) are properly cited with attribution to their original sources. Their corresponding licenses and usage terms are explicitly followed, and only assets permitting academic/research use are included in the experiments. No license restrictions are violated.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

Justification: The pseudocodes for the methodology(Refer App. B), the hyperparameters(Refer App. D.1) and the anyonymous repository containing the code(Refer App. 4) have been provided in the paper.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: No human subjects were involved in this research work.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: No human subjects were involved in this research work.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: A declaration of LLM usage in this work has been made in App. J.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.