DUAL-PROCESS TREE SEARCH WITH HEURISTIC MERIT AND EVOLUTIONARY CREDIT FOR AUTOMATIC HEURISTIC DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-based automatic heuristic design (AHD) has recently emerged as a promising paradigm for automating heuristic discovery with reduced reliance on manual engineering. Tree-structured search has demonstrated advantages in organizing and exploring LLM-generated heuristics. However, parent-path credit propagation during search is structurally misaligned with exemplar-guided evolution, causing improvements induced by auxiliary exemplars to be attributed predominantly to the parent lineage. Meanwhile, fitness-based exemplar selection conflates standalone heuristic merit with evolutionary utility, potentially overlooking heuristics with modest standalone performance but stronger capacity to guide subsequent refinement. Moreover, evaluating heuristics on a finite set of instances yields potentially biased fitness estimates, steering the search toward overfitted local optima. To address these limitations, we propose Dual-Process Tree Search (DPTS), a framework that disentangles evolutionary credit from heuristic merit within a shared search tree. DPTS maintains distinct estimates of parent lineage merit and exemplar-induced evolutionary credit, selects exemplars according to downstream evolutionary utility, and aggregates evidence across structurally related nodes to improve the robustness of frontier evaluation under finite-sample noise. Experiments on representative combinatorial optimization benchmarks demonstrate that DPTS consistently outperforms state-of-the-art AHD baselines across constructive and ACO-based settings.

1 INTRODUCTION

Automatic Heuristic Design (AHD) aims to automate the discovery of effective heuristics for complex optimization problems, shifting heuristic development from expert-driven manual engineering to systematic exploration of algorithmic design spaces Burke et al. (2013); Desale et al. (2015). Classical genetic programming provides an early principled route for heuristic generation, but remains constrained by manually specified operators, restrictive search grammars, and insufficient semantic awareness of algorithmic structure Duflo et al. (2019); Koza (1994). Recent advances in large language models (LLMs) have transformed AHD into a programmatic search process over open-ended heuristic spaces Novikov et al. (2025). Existing LLM-based methods synthesize executable heuristics and refine them through LLM-mediated evolutionary operators guided by instance-level evaluation feedback Romera-Paredes et al. (2024); Liu et al. (2024); Ye et al. (2024); Dat et al. (2025). MCTS-AHD Zheng et al. (2025) further reformulates LLM-based AHD as tree-structured heuristic search, using Monte Carlo Tree Search (MCTS) to preserve evolutionary histories and retain temporarily underperforming candidates for subsequent refinement.

However, tree-structured exploration remains constrained by a node-centric credit assignment mechanism, which is poorly aligned with exemplar-guided heuristic evolution. In LLM-mediated operators such as crossover with an elite exemplar, offspring heuristics are derived by modifying the current parent with algorithmic motifs extracted from auxiliary exemplars outside the parent lineage, thereby introducing non-local dependencies into heuristic generation. However, as illustrated in Figure 1, existing tree-search-based AHD methods such as MCTS-AHD Zheng et al. (2025) propagate offspring gains through the ancestral path of the selected parent, underrepresenting the contribution of auxiliary exemplars. Moreover, existing AHD methods often conflate standalone heuristic per-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

formance with evolutionary utility, despite their non-equivalent roles in heuristic search. During exemplar selection, existing performance-ranked strategies implicitly use empirical heuristic fitness as a proxy for evolutionary informativeness, although high standalone merit does not necessarily indicate stronger potential for downstream refinement. A top-scoring node, such as h_2 in Figure 1, may be structurally close to the current node h_*, introduce few novel algorithmic motifs, and lead to only marginal improvement under crossover-based generation. By contrast, heuristics with weaker standalone fitness may encode complementary algorithmic motifs that provide more informative signals for downstream offspring generation. Moreover, heuristic quality is typically approximated using a limited set of evaluation instances, such as a 64-instance Traveling Salesman Problem benchmark, which may compromise the reliability of node-level fitness estimation and bias the search toward evaluation-set-specific optima rather than robust heuristic improvements.

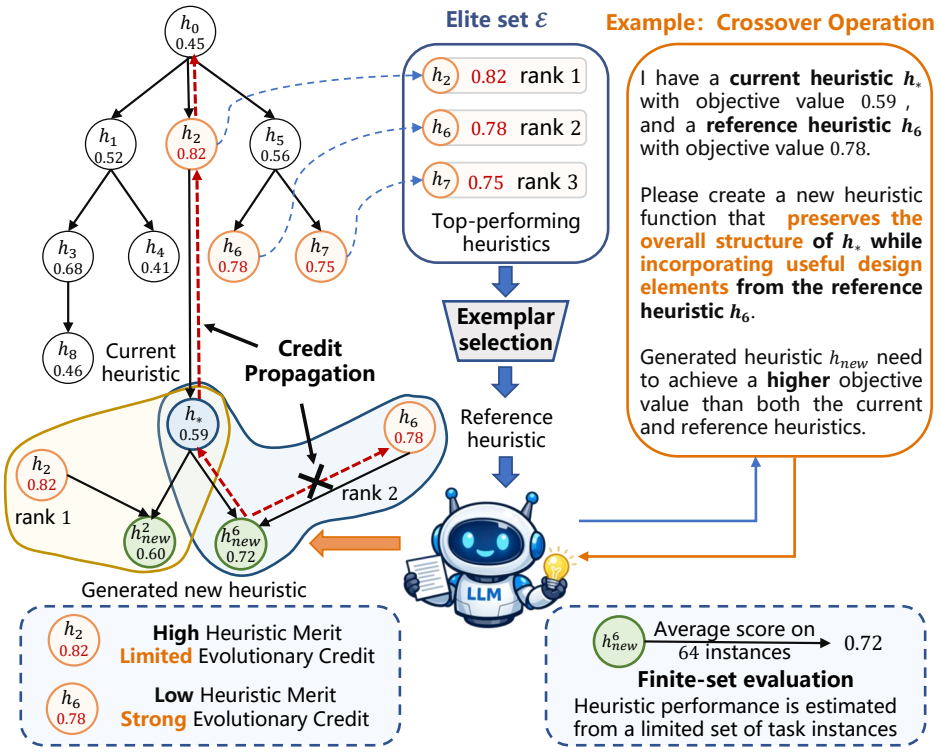


Figure 1: Structural limitations of node-level heuristic evolution: (1) Parent-path credit propagation underrepresents exemplar contributions, (2) fitness-driven exemplar selection conflates heuristic merit with evolutionary utility, (3) finite-set evaluation favors evaluation-set-specific optima.

To address these challenges, **D**ual-**P**rocess **T**ree **S**earch (DPTS) is proposed for LLM-based AHD, which explicitly disentangles heuristic merit from evolutionary credit within a shared search tree. By augmenting node-level fitness with exemplar-specific evolutionary credit, DPTS establishes a principled framework for faithful credit attribution, evolution-aware exemplar selection, and robust value estimation in heuristic evolution. The main contributions are summarized as follows [1]:

- DPTS establishes a dual-process search formulation that disentangles evolutionary credit from heuristic merit through two coordinated search-and-feedback processes within a shared tree, enabling separate attribution of parent-lineage merit and exemplar-induced evolutionary contribution.
- An evolution-aware exemplar selection mechanism is proposed to prioritize exemplars according to their downstream utility for heuristic improvement rather than empirical fitness alone, reducing the mismatch between standalone heuristic merit and evolutionary utility.

[1]The source code will be made publicly available upon acceptance.

- A topology-aware aggregation strategy is developed to integrate fitness evidence across structurally related nodes through a forward–backward propagation mechanism, reducing evaluation variance under finite-set evaluation and improving the reliability of frontier selection with theoretical guarantees.
- Experiments on diverse combinatorial optimization benchmarks demonstrate that DPTS outperforms existing LLM-based AHD methods under both constructive and ACO settings.

2 Related work

Automating heuristic design with computational intelligence is a long-standing research area spanning hyper-heuristics Burke et al. (2013), automated algorithm configuration Stützle & López-Ibáñez (2018), AutoML Feurer et al. (2015); Hutter et al. (2019), and genetic programming Koza (1994); Duflo et al. (2019). Recent progress in LLMs has enabled evaluator-guided program evolution for heuristic discovery over open-ended design spaces Achiam et al. (2023). Existing LLM-based AHD methods typically refine executable heuristic pools through task-level evaluation, fitness-guided selection, and LLM-mediated variation. FunSearch Romera-Paredes et al. (2024) pairs an LLM with an automatic evaluator to search over executable programs, while AlphaEvolve Novikov et al. (2025) generalizes evaluator-guided program evolution to larger codebases. EoH Liu et al. (2024) improves heuristic generation by co-evolving natural-language thoughts and executable code, and ReEvo Ye et al. (2024) introduces short- and long-term reflections to guide crossover and mutation. HSEvo Dat et al. (2025) promotes population diversity through harmony search and diversity metrics, MEoH Yao et al. (2025) formulates heuristic design as a multi-objective search problem with dominance-dissimilarity-based selection, and EoH-S Liu et al. (2026) extends single-heuristic optimization to complementary heuristic-set design through diversity-aware memetic search. CALM Huang et al. (2026) co-evolves heuristic algorithms and the underlying language model, while MoH Shi et al. (2026) introduces meta-optimization for generalizable heuristic generation. MCTS-AHD Zheng et al. (2025) introduces an MCTS-based search tree for LLM-based AHD, while both node selection and reward backpropagation remain governed by node-level empirical fitness. In contrast, DPTS disentangles evolutionary credit from heuristic merit and incorporates downstream evolutionary utility into exemplar selection and credit attribution.

In existing LLM-based AHD, generated heuristics are typically evaluated as independent candidates, with empirical fitness serving as the primary criterion for selection and retention. Search-based methods instead can exploit inter-node structural dependencies to improve node evaluation. MCTS refines ancestor values through trajectory-based backpropagation Kocsis & Szepesvári (2006), CNneim-A estimates heuristic values from scouted subtrees Xu et al. (1987), and Memory-Augmented MCTS retrieves estimates from previously observed similar nodes Xiao et al. (2018). B′ search maintains A* consistency Hart et al. (1968) through parent-child propagation, with Pathmax correcting forward underestimation and upward propagation mitigating backward inconsistency Mero (1984). Building on these perspectives, DPTS employs topology-aware forward–backward propagation to aggregate evidence across structurally related nodes in the heuristic evolution tree, yielding more reliable value estimates under noisy finite-set evaluation.

3 Preliminary on LLM-based AHD

AHD formulates heuristic discovery as a meta-optimization problem over a heuristic space $\mathcal{H}$, and each heuristic $h \in \mathcal{H}$ maps an instance $x \in \mathcal{I}$ to a feasible solution $h(x) \in \mathcal{S}$. Without loss of generality, AHD is formulated as score maximization, with minimization objectives converted accordingly. Let $f(x, h(x))$ denote the score achieved by heuristic h on instance x, where larger values indicate better performance. The objective is to identify a heuristic with maximal expected performance under the instance distribution $\mathcal{D}$:

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

Since $\mathcal{D}$ is generally inaccessible, $\mathcal{L}(h)$ is approximated on a finite evaluation set $\mathcal{D}_{\text{eval}} = \{x_i\}_{i=1}^N$:

$$\widehat{\mathcal{L}}(h) = \frac{1}{N} \sum_{i=1}^{N} f(x_i, h(x_i)), \tag{2}$$

which is susceptible to finite-set noise and does not encode parent-relative evolutionary credit. Evolutionary credit serves only to guide exemplar selection and does not alter the global objective of maximizing heuristic performance.

In LLM-based AHD, the heuristic space $\mathcal{H}$ is implicitly parameterized by executable programs generated and refined by LLMs. An LLM-mediated operator G synthesizes new candidates from existing heuristics, which may serve as parents or exemplars. Population-based methods maintain a finite candidate pool $\mathcal{P}_t \subset \mathcal{H}$ at iteration t, generate offspring through G, and truncate the expanded pool to a fixed-size population according to empirical fitness:

$$\widetilde{\mathcal{P}}_{t+1} = \mathcal{P}_t \cup \{G(h_{p_1}, \ldots, h_{p_k}) \mid h_{p_j} \in \mathcal{P}_t\}, \qquad \mathcal{P}_{t+1} = \mathrm{Select}_M(\widetilde{\mathcal{P}}_{t+1}; \widehat{\mathcal{L}}). \tag{3}$$

Empirical fitness therefore serves as the principal criterion for candidate selection and retention.

Tree-based LLM-AHD organizes generated heuristics into a search tree $\mathcal{T} = (\mathcal{V}, \mathcal{A})$, where each node $v \in \mathcal{V}$ stores a heuristic program h_v with empirical fitness $g_v = \widehat{\mathcal{L}}(h_v)$. In MCTS-AHD, parent selection balances g_v with UCB-based exploration, whereas reference-guided actions draw auxiliary exemplars from a fitness-ranked elite set E. Generated heuristics are attached to the selected parent, with their empirical fitness propagated along the corresponding parent trajectory. MCTS-AHD Zheng et al. (2025) is therefore primarily governed by node-level empirical fitness across parent selection, exemplar selection, and backpropagation. Such a fitness-centric formulation conflates standalone heuristic merit with evolutionary utility. Heuristic merit reflects the empirical solution quality achieved by a heuristic within the underlying solver, whereas evolutionary utility characterizes its contribution to offspring improvement. Under exemplar-guided generation, MCTS-AHD attributes gains to ancestral nodes along the parent path while under-crediting auxiliary exemplars. Meanwhile, fitness-ranked exemplar selection may overlook heuristics with weaker performance but greater utility for downstream refinement.

4 METHOD

To address biased credit attribution, fitness-based exemplar selection, and noisy finite-set evaluation in tree-based LLM-AHD, we propose **DPTS**, a **D**ual-**P**rocess **T**ree **S**earch framework, to disentangle evolutionary credit from heuristic merit within a shared search tree. The merit process evaluates parent nodes and algorithmic lineages, whereas the evolutionary-credit process quantifies exemplar contributions, thereby enabling faithful credit attribution, evolution-aware exemplar selection, and robust value estimation.

4.1 DUAL PROCESS SEARCH TREE

As illustrated in Figure 2, DPTS represents heuristic evolution as a rooted search tree $\mathcal{T} = (\mathcal{V}, \mathcal{A})$, where each node $v \in \mathcal{V}$ is associated with an executable heuristic program $h_v \in \mathcal{H}$. DPTS maintains two coupled processes on the shared tree, respectively estimating parent-lineage merit and auxiliary-exemplar credit. Accordingly, each node is associated with

$$\mathbf{s}_v = \big(g_v,\, W_v,\, N_v,\, Q_v,\, W_v^B,\, N_v^B,\, B_v\big), \tag{4}$$

where $g_v = \widehat{\mathcal{L}}(h)$ denotes the empirical fitness of h_v on the finite evaluation set, W_v and N_v denote the accumulated merit and visit count, with $Q_v = W_v/N_v$ estimating the *heuristic merit*. Unlike the local empirical fitness g_v, Q_v aggregates merit evidence propagated through the tree. The evolutionary-credit statistics W_v^B and N_v^B accumulate the credit and usage count of node v when selected as an auxiliary exemplar, with $B_v = W_v^B/N_v^B$ estimating its *evolutionary credit*.

Each root child of $\mathcal{T}$ is treated as the seed of distinct algorithm families, while descendants within each subtree represent successive intra-lineage refinements. DPTS expands the tree through two complementary LLM-mediated actions to balance exploitation and exploration.

Exploitation action refines an existing algorithm through an exemplar-guided crossover operation by generating an offspring u from a selected parent node p and an auxiliary exemplar node e:

$$h_u = \mathcal{G}_{\mathrm{exploit}}\big(h_p, h_e\big), \tag{5}$$

where h_p provides the inherited algorithmic backbone and h_e contributes transferable design motifs. The generated heuristic is attached as a child of p, yielding targeted intra-lineage refinement.

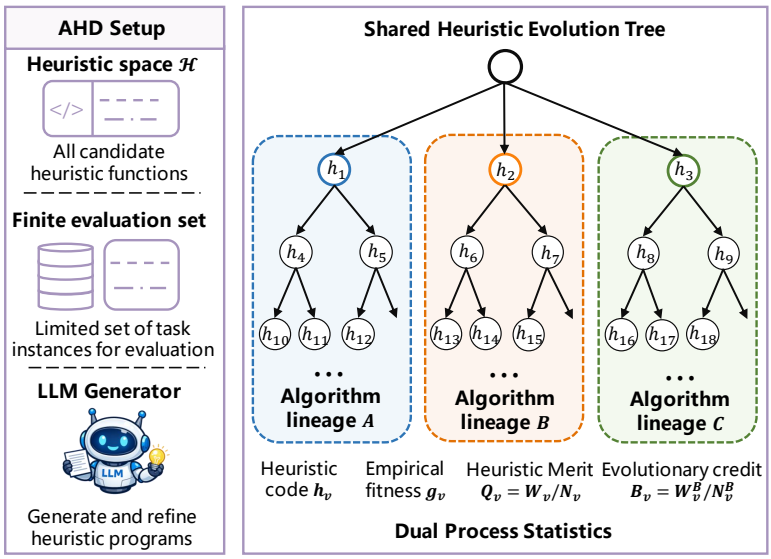


Figure 2: Shared heuristic evolution tree with dual-process statistics.

Since the offspring depends jointly on the parent and auxiliary exemplar, the resulting offspring performance can be expressed as

$$Y_{p,e} = m(p) + \tau(p, e) + \varepsilon, \tag{6}$$

where $m(p)$ denotes parent merit, $\tau(p, e)$ the exemplar contribution, and ε stochastic generation and evaluation noise, motivating disentangled modeling of heuristic merit and evolutionary credit.

Exploration action samples representative nodes $\mathcal{R} \subset \mathcal{V}$ from existing lineages and prompts the LLM to synthesize a structurally novel heuristic:

$$h_u = \mathcal{G}_{\text{explore}}\big(\{h_v \mid v \in \mathcal{R}\}\big). \tag{7}$$

The generated heuristic is attached directly to the root, thereby initiating a new algorithmic lineage.

4.2 Evolution-aware DPTS

As illustrated in Figure 3, DPTS performs a four-stage search procedure over the shared heuristic evolution tree: selection, expansion, evaluation, and back-propagation, followed by topology-aware merit aggregation for frontier evaluation. Heuristic merit and evolutionary credit are updated through separate feedback paths, respectively estimating parent lineage merit and auxiliary exemplar contribution.

Selection For exploitation, DPTS selects parent–exemplar pairs $\{(p_i, e_i)\}_{i=1}^{m}$, using heuristic merit for parent selection and UCB-guided evolutionary credit for exemplar selection. Let $\mathcal{O}$ denote the set of legal leaf nodes and $\mathcal{E}$ the elite set. At iteration i, a candidate subset $\mathcal{V}_i^C \subseteq \mathcal{O}$ is sampled to preserve search diversity Zhao et al. (2024):

$$p_i = \arg\max_{v \in \mathcal{V}_i^C} Q_v, \qquad e_i = \arg\max_{v \in \mathcal{E}} \left(B_v + c\sqrt{\frac{\log(N_{\text{ex}} + 1)}{N_v^B + 1}} \right), \tag{8}$$

where c controls exemplar exploration and N_{ex} is the total number of exemplar selections.

For exploration, DPTS selects representative root-level lineages according to heuristic merit and traverses their subtrees with an ϵ-greedy policy:

$$\mathcal{C} = \text{TopK}_{v \in \text{Ch}(v_0)}(Q_v, k), \qquad \mathcal{R} = \text{EGreedy}\left(\{\mathcal{T}_r \mid r \in \mathcal{C}\}, Q\right), \tag{9}$$

where v_0 is the root, $\mathcal{C}$ denotes the selected root-level lineage nodes, and $\mathcal{R} = \{h_{\mathcal{R}}^{(1)}, \ldots, h_{\mathcal{R}}^{(k)}\} \subseteq \mathcal{V}$ denotes the representative nodes used for exploration.

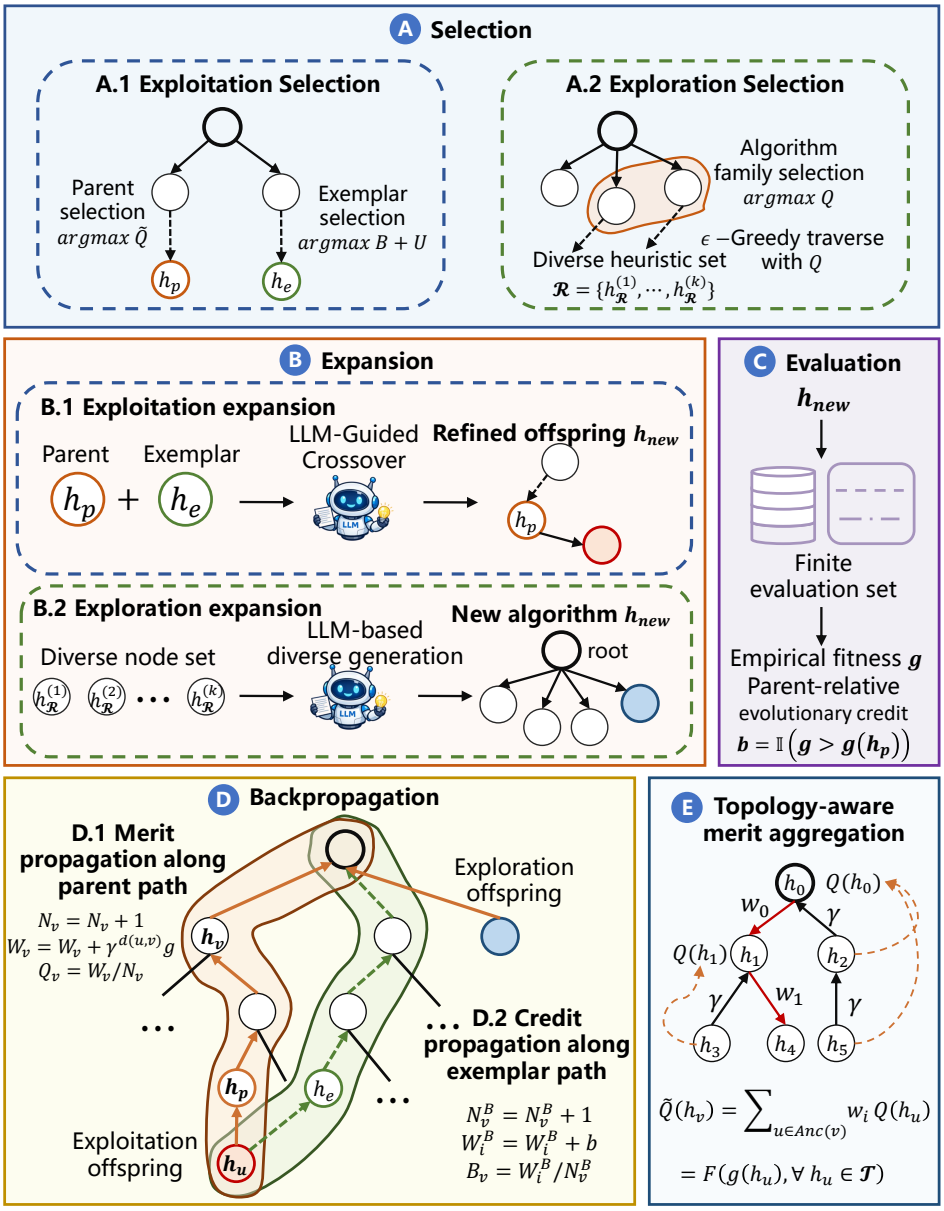


Figure 3: Overall workflow of DPTS for LLM-based AHD.

Expansion DPTS invokes action-specific LLM prompts to expand the search tree. Exploitation performs reference-guided refinement for each selected pair (p_i, e_i), while exploration synthesizes a structurally novel heuristic from the representative set $\mathcal{R}$:

$$h_{u_i}^{\text{exp}} = \mathcal{G}_{\text{exploit}}\big(h_{p_i}, h_{e_i}\big), \quad i = 1, \ldots, m, \qquad h_u^{\text{explore}} = \mathcal{G}_{\text{explore}}\big(\{h_v \mid v \in \mathcal{R}\}\big). \tag{10}$$

Exploitation offspring are attached to their corresponding parents, whereas the exploration offspring is attached to the root. Prompt templates are provided in Appendix A.

Evaluation. Each generated heuristic is evaluated on the finite evaluation set to obtain its empirical fitness and update the elite set $\mathcal{E}$. For an exploitation offspring u_i generated from parent p_i and exemplar e_i, DPTS assigns parent-relative evolutionary credit to the exemplar:

$$b_{e_i} = \mathbb{I}\left[g_{u_i} > g_{p_i}\right]. \tag{11}$$

The binary signal captures beneficial exemplar guidance while reducing sensitivity to noisy fitness estimates under finite-set evaluation.

Back-propagation DPTS propagates heuristic merit and evolutionary credit along distinct paths. For each newly generated node u, its empirical fitness is propagated along the parent path $\mathrm{Path}(u)$:

$$N_v \leftarrow N_v + 1, \qquad W_v \leftarrow W_v + \gamma^{d(v,u)} g_u, \qquad Q_v \leftarrow W_v / N_v, \quad \forall v \in \mathrm{Path}(u), \tag{12}$$

where $d(v, u)$ denotes the tree distance from v to u, and $\gamma \in (0, 1]$ controls the decay of merit propagation. In parallel, the exemplar credit b_e is propagated along the exemplar trajectory $\mathrm{Path}(e)$:

$$N_v^B \leftarrow N_v^B + 1, \qquad W_v^B \leftarrow W_v^B + b_e, \qquad B_v \leftarrow W_v^B / N_v^B, \quad \forall v \in \mathrm{Path}(e). \tag{13}$$

The updated statistics guide subsequent iterations, enabling DPTS to exploit high-merit lineages while prioritizing exemplars with demonstrated downstream evolutionary utility. Detailed algorithms are provided in Appendix B.

4.3 Topology-aware merit aggregation

Finite-set evaluation introduces inherent noise into node-level fitness estimates. Since descendants in the heuristic evolution tree are generated through parent-conditioned refinement, lineage-level evidence provides a more reliable basis for distinguishing robust algorithmic directions from finite-evaluation-specific optima. DPTS therefore aggregates propagated merit across structurally related nodes in the heuristic evolution tree. As summarized in Appendix C, for a leaf node v_l, let $\mathcal{L}(v_l) = \{v_0, v_1, \ldots, v_l\}$ denote its root-to-leaf path. Each ancestor v_i provides a back-propagated merit estimate Q_{v_i}, which is further aggregated forward to define

$$\widetilde{Q}_{v_l} = \sum_{i=0}^{l} \bar{w}_i Q_{v_i}, \qquad \bar{w}_i = \frac{w_i}{\sum_{j=0}^{l} w_j}, \qquad w_i = \max\{0,\, 1 - \alpha|l - i|\}, \quad \alpha > 0. \tag{14}$$

The truncated linear weights assign greater emphasis to structurally proximal ancestors. Such forward–backward estimation $\widetilde{Q}_v$ is employed to replace Q_v for more robust evaluation. Theoretical analysis further establishes that topology-aware aggregation increases the probability of correctly ranking the optimal frontier node. Detailed proof is provided in Appendix D.

Theorem 1. *Assume that each empirical node fitness is corrupted by independent Gaussian noise:*

$$g_v = g_v^* + \epsilon_v, \qquad \epsilon_v \sim \mathcal{N}(0, \sigma_b^2),$$

where g_v^* denotes the underlying expected fitness. Then, $Q_{v_i} \sim \mathcal{N}\big(\mu_{v_i}, \sigma_b^2 / N_{v_i}\big)$, where N_{v_i} is the number of evaluations propagated to v_i. For a frontier node v,

$$\mathrm{Var}(\widetilde{Q}_v) = \sigma_b^2 \sum_i \bar{w}_i^2 / N_{v_i} < \sigma_b^2 = \mathrm{Var}(Q_v), \tag{15}$$

where $Q_v = g_v$ for a frontier leaf. Let $v^\star$ and v_t denote the optimal frontier node and any competing node, respectively. For the expected merit gap $\Delta_t > 0$,

$$\Pr\Big(\widetilde{Q}_{v^\star} > \widetilde{Q}_{v_t}\Big) = \Phi\left(\frac{\Delta_t}{\sqrt{\mathrm{Var}(\widetilde{Q}_{v^\star}) + \mathrm{Var}(\widetilde{Q}_{v_t})}}\right) > \Phi\left(\frac{\Delta_t}{\sqrt{2}\sigma_b}\right) = \Pr(Q_{v^\star} > Q_{v_t}), \tag{16}$$

where Φ is the standard normal cumulative distribution function.

5 Experiments

Benchmarks and design paradigms. Experiments are conducted on representative NP-hard combinatorial optimization benchmarks, including the Traveling Salesman Problem (TSP), Knapsack Problem (KP), Bin Packing Problem (BPP), Capacitated Vehicle Routing Problem (CVRP), and Multiple Knapsack Problem (MKP). Following prior LLM-based AHD studies Liu et al. (2024); Zheng et al. (2025), heuristic design is evaluated under step-by-step construction Asani et al. (2023); Vinyals et al. (2015) and Ant Colony Optimization (ACO) Dorigo et al. (2006); Kim et al. (2025). LLM-generated heuristics are instantiated either as next-node selection functions for constructive solvers or as heuristic-matrix generation functions for ACO-based solvers Ye et al. (2024). DPTS is further evaluated on the Decap Placement Problem (DPP) in Electronic Design Automation for cross-domain validation. The resulting solvers are evaluated on task-specific instance sets, with detailed problem definitions provided in Appendix E.

Baselines. DPTS is compared with three categories of representative baselines: classical hand-crafted heuristics, including nearest-greedy Rosenkrantz et al. (1977) and ACO Dorigo et al. (2006); neural optimization methods, including POMO Kwon et al. (2020) and DeepACO Ye et al. (2023); and LLM-based AHD methods, including FunSearch Romera-Paredes et al. (2024), EoH Liu et al. (2024), ReEvo Ye et al. (2024), HSEvo Dat et al. (2025), CALM Huang et al. (2026), MoH Shi et al. (2026), as well as MCTS-AHD Zheng et al. (2025), the most closely related tree-search baseline.

Settings. GPT-4o-mini OpenAI (2024) is used as the backbone LLM with a sampling temperature of 1.0. Following prior work Ye et al. (2024); Zheng et al. (2025), all LLM-based AHD methods use the same search budget of $T = 1{,}000$ heuristic evaluations. Consistent with MCTS-AHD Zheng et al. (2025), the elite-set size and maximum search-tree depth are both set to 10, with no additional heuristic-evaluation cost introduced by DPTS. Each generated heuristic is evaluated on the same instances with a timeout of 60 seconds. The search tree is initialized with $N_I = 4$ root-child nodes, and each expansion generates $m = 4$ exploitation offspring and one exploration offspring using $k = 5$ representative nodes. The topology-aggregation decay coefficient is set to $\alpha = 0.2$. Experiment results are averaged over three independent runs.

5.1 Performance on combinatorial optimization problems

Results on TSP under the step-by-step construction framework are reported in Table 1. DPTS achieves the strongest performance among existing LLM-based AHD baselines across all evaluated scales, reducing the optimality gaps of MCTS-AHD Zheng et al. (2025) from 9.69%, 11.79%, and 13.71% to 8.52%, 10.19%, and 11.67% at $N = 50$, $N = 100$, and $N = 200$, respectively. On KP, DPTS also obtains consistently strong LLM-based results across all scales, with detailed results reported in Appendix F, demonstrating the effectiveness of disentangling heuristic merit and evolutionary credit in constructive heuristic search.

Table 1: Constructive TSP performance comparison over three independent runs. Shaded columns denote the training problem size. Optimal solutions are obtained by LKH Lin & Kernighan (1973).

Method	$N = 50$		$N = 100$		$N = 200$	
	Obj.↓	Gap↓	Obj.↓	Gap↓	Obj.↓	Gap↓
Optimal	5.675	–	7.768	–	10.659	–
Greedy	6.959	22.62%	9.706	24.94%	13.461	26.29%
POMO	5.697	0.39%	8.001	3.01%	12.897	20.45%
FunSearch	6.357	12.00%	8.850	13.93%	12.372	15.54%
EoH	6.394	12.67%	8.894	14.49%	12.437	16.68%
ReEvo	6.291	10.85%	8.724	12.31%	12.123	13.73%
CALM	6.273	10.54%	8.691	11.88%	12.104	13.56%
MoH	6.280	10.66%	8.705	12.06%	12.208	14.53%
MCTS-AHD	6.225	9.69%	8.684	11.79%	12.120	13.71%
DPTS	<u>6.158</u>	<u>8.52%</u>	<u>8.559</u>	<u>10.19%</u>	<u>11.903</u>	<u>11.67%</u>
DPTS-K	**6.099**	**7.47%**	**8.487**	**9.25%**	**11.821**	**10.90%**

The ACO-based results tested on 64 instances Zheng et al. (2025) are summarized in Table 2. DPTS improves upon MCTS-AHD across all reported TSP and offline BPP settings. For TSP, DPTS reduces the optimality gap from 0.69% to 0.18% at $N = 50$, and from 1.70% to 0.34% at $N = 100$. For offline BPP, DPTS reduces the objective value from 204.094 to 202.250 at $N = 500$, and from 407.323 to 403.052 at $N = 1{,}000$. Additional results on CVRP and MKP are summarized in Appendix G, demonstrating the effectiveness of DPTS for ACO-based heuristic matrix design. Results on DPP are summarized in Appendix H, further supporting the effectiveness of DPTS in practical optimization.

5.2 Lineage diversity and DPTS-K

The root-level branches of DPTS represent distinct algorithmic lineages, preserving diversity throughout tree-structured evolution. Accordingly, we construct DPTS-K, which selects K repre-

Table 2: ACO-based TSP and offline BPP performance comparison over three independent runs.

Methods	TSP				Offline BPP			
	$N = 50$		$N = 100$		$N = 500, C = 450$		$N = 1,000, C = 150$	
	Obj.↓	Gap↓	Obj.↓	Gap↓	Obj.↓	Gap↓	Obj.↓	Gap↓
ACO	5.992	4.00%	8.948	11.26%	208.828	3.26%	417.938	3.69%
DeepACO	5.842	1.40%	8.282	2.98%	203.125	0.44%	405.172	0.53%
EoH	5.828	1.15%	8.263	2.74%	204.646	1.19%	408.599	1.38%
ReEvo	5.856	1.64%	8.340	3.70%	206.693	2.20%	413.510	2.59%
HSEvo	5.810	0.84%	8.219	2.19%	205.563	1.64%	410.745	1.91%
MCTS-AHD	5.801	0.69%	8.179	1.70%	204.094	0.91%	407.323	1.06%
DPTS	5.772	0.18%	8.070	0.34%	202.250	0.003%	**403.052**	**0.00%**
DPTS-K	**5.762**	**0.00%**	**8.043**	**0.00%**	**202.245**	**0.00%**	403.057	0.001%

sentative experts from different root-level branches. Each branch r is ranked using a Hoeffding-style confidence score Hoeffding (1963) that combines estimated merit with a confidence bonus:

$$\mathrm{Score}(r) = Q_r + (Q_{\max} - Q_{\min}) \sqrt{\log(1/\delta)/2N_r}, \tag{17}$$

where $\delta \in (0, 1)$ controls the confidence level. The highest-fitness heuristic from each selected branch is retained as its representative expert. Further details are provided in Appendix I. As reported in Table 1 and 2, DPTS-K with $K = 5$ and $\delta = 0.1$ improves upon single-heuristic DPTS in most settings, indicating that lineage diversity preserved by tree-structured evolution can be effectively exploited through a multi-heuristic portfolio.

5.3 Ablation studies

Ablation results on TSP under the construction framework are provided in Appendix J. DPTS w.o. evolutionary credit replaces credit-aware exemplar selection with the sampling strategy of MCTS-AHD Zheng et al. (2025), whereas DPTS w.o. topology-aware aggregation uses Q in place of $\widetilde{Q}$. Removing either component degrades performance, with topology-aware aggregation exerting a more pronounced effect on train–test consistency. The objective increases from 6.265 on the training instances to 6.323 on the test instances after removing topology-aware aggregation, whereas full DPTS remains stable at 6.148 and 6.142, respectively. The reduced discrepancy supports the contribution of topology-aware aggregation to robust evaluation under finite samples. Additional hyperparameter ablations are reported in Appendix K, demonstrating that DPTS remains stable across a range of parameter settings.

6 Conclusion

In this paper, DPTS, a dual-process tree search framework for LLM-based AHD, is proposed to disentangle evolutionary credit from heuristic merit within a shared heuristic evolution tree. By attributing exemplar-induced gains to both parent lineages and auxiliary exemplars and aggregating evidence across structurally related heuristic variants, DPTS enables evolution-aware exemplar selection while improving robustness under finite-set evaluation. Theoretical analysis establishes that topology-aware aggregation reduces estimation variance and improves the reliability of frontier ranking. Experiments across representative combinatorial optimization benchmarks demonstrate the effectiveness of DPTS in heuristic discovery under both step-by-step construction and ACO-based design paradigms, while ablation studies validate the contributions of evolutionary-credit-based exemplar selection and topology-aware aggregation. The DPTS-K analysis further indicates that tree-structured evolution maintains complementary algorithmic lineages, facilitating the construction of effective heuristic portfolios. DPTS focuses on multi-node offspring generation to address the limitations of prior tree-based AHD, leaving single-node refinement actions such as mutation-based structural and parametric modification as an additional child-generation action or a fast rollout mechanism for future work. Overall, DPTS provides a principled framework for reliable and generalizable heuristic evolution in LLM-based AHD.

AI USE STATEMENT

In this work, we used OpenAI's GPT for language refinement and limited assistance with code drafting. GPT was not used to develop the research ideas, design the methodology or experiments, generate or analyze experimental data, or draw scientific conclusions. All AI-assisted content was reviewed and verified by the authors, who take full responsibility for the final manuscript and results.

REPRODUCIBILITY STATEMENT

Detailed pseudocode and experimental configurations are provided in the appendix to facilitate reproducibility. The source code and test data will also be publicly released upon acceptance.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Emmanuel O Asani, Aderemi E Okeyinka, and Ayodele Ariyo Adebiyi. A computation investigation of the impact of convex hull subtour on the nearest neighbour heuristic. In *2023 International Conference on Science, Engineering and Business for Sustainable Development Goals (SEB-SDG)*, volume 1, pp. 1–7. IEEE, 2023.

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.

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

Sachin Desale, Akhtar Rasool, Sushil Andhale, and Priti Rane. Heuristic and meta-heuristic algorithms and their relevance to the real world: a survey. *Int. J. Comput. Eng. Res. Trends*, 351(5): 2349–7084, 2015.

Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

Gabriel Duflo, Emmanuel Kieffer, Matthias R Brust, Grégoire Danoy, and Pascal Bouvry. A gp hyper-heuristic approach for generating tsp heuristics. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops*, pp. 521–529. IEEE, 2019.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. *Advances in Neural Information Processing Systems*, 28, 2015.

Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

Ziyao Huang, Weiwei Wu, Kui Wu, Wei-Bin Lee, and Jianping Wang. Calm: Co-evolution of algorithms and language model for automatic heuristic design. In *International Conference on Learning Representations*, volume 2026, pp. 72468–72510, 2026.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer, 2019.

Haeyeon Kim, Minsu Kim, Federico Berto, Joungho Kim, and Jinkyoo Park. Devformer: A symmetric transformer for context-aware device placement. In *International Conference on Machine Learning*, pp. 16541–16566. PMLR, 2023.

Minsu Kim, Sanghyeok Choi, Hyeonah Kim, Jiwoo Son, Jinkyoo Park, and Yoshua Bengio. Ant colony sampling with gflownets for combinatorial optimization. In *International Conference on Artificial Intelligence and Statistics*, pp. 469–477. PMLR, 2025.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pp. 282–293. Springer, 2006.

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

Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.

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

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*, pp. 32201–32223. PMLR, 2024.

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

Laszlo Mero. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, 23(1): 13–27, 1984.

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.

OpenAI. Gpt-4o mini: advancing cost-efficient intelligence. `https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/`, 2024. Accessed: 2026-05-01.

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.

Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, 1977.

Yiding Shi, Jianan Zhou, Wen Song, Jieyi Bi, Yaoxin Wu, Zhiguang Cao, and Jie Zhang. Generalizable heuristic generation through llms with meta-optimization. In *International Conference on Learning Representations*, volume 2026, pp. 119314–119349, 2026.

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

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in Neural Information Processing systems*, 28, 2015.

Chenjun Xiao, Jincheng Mei, and Martin Müller. Memory-augmented monte carlo tree search. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Lei Xu, Pingfan Yan, and Tong Chang. Algorithm cnneim-a and its mean complexity. In *Proc. of 2nd International Conference on Computers and Applications. IEEE Press, Beijing*, pp. 494–499, 1987.

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

Haoran Ye, Jiarui Wang, Zhiguang Cao, Helan Liang, and Yong Li. Deepaco: Neural-enhanced ant systems for combinatorial optimization. *Advances in Neural Information Processing Systems*, 36: 43706–43728, 2023.

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

Dengwei Zhao, Shikui Tu, and Lei Xu. Seea*: Efficient exploration-enhanced a* search by selective sampling. *Advances in Neural Information Processing Systems*, 37:104138–104179, 2024.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. In *International Conference on Machine Learning*, pp. 78338–78373. PMLR, 2025.

A LLM-MEDIATED EXPANSION ACTIONS AND PROMPT TEMPLATES

DPTS expands the heuristic evolution tree through two complementary LLM-mediated actions, namely exploitation and exploration. The exploitation action refines an existing algorithmic lineage, whereas the exploration action introduces a new algorithm lineage at the root level.

Exploitation action. Given a selected parent node p and an auxiliary exemplar e, exploitation performs reference-guided refinement by prompting the LLM to generate an offspring heuristic conditioned on both h_p and h_e:

$$h_{\text{new}} = \mathcal{G}_{\text{exploit}}(h_p, h_e). \tag{18}$$

Here, h_p provides the inherited algorithmic backbone, while h_e supplies transferable design motifs for improving the parent heuristic. The generated offspring is attached as a child of p, thereby extending the corresponding algorithmic lineage.

Exploration action. Given a representative node set $\mathcal{R}$ sampled from existing branches, exploration prompts the LLM to synthesize a heuristic that is structurally distinct from the sampled representatives:

$$h_{\text{new}} = \mathcal{G}_{\text{explore}}\big(\{h_v \mid v \in \mathcal{R}\}\big). \tag{19}$$

The generated heuristic is inserted as a new child of the root node v_0, thereby initiating a new algorithm lineage. This action expands the search toward unexplored regions of the heuristic space and preserves diversity among root-level lineages.

DPTS does not rely on a handcrafted seed heuristic. For the initial expansion, the first heuristic program is generated directly by the LLM from a task-specific prompt. Taking TSP under the step-by-step construction framework as an example, the prompt is formulated as follows.

Initial Prompt for TSP Constructive Heuristic

```
Solving Traveling Salesman Problem (TSP) with constructive
    heuristics. TSP requires finding the shortest path that visits
    all given nodes and returns to the starting node. Please output
    your response in exactly the following format:

<ALGORITHM>
one-sentence description of the design idea and main steps
</ALGORITHM>

<CODE>
# your code here
</CODE>

Requirements:
The algorithm description must be exactly one sentence inside <
    ALGORITHM>. The code inside <CODE> must be valid, directly
    executable Python. The code must strictly implement the
    algorithm described in <ALGORITHM>. Do not output any text
    outside <ALGORITHM> and <CODE>.
The function signature must exactly be: def select_next_node(
    current_node, destination_node, unvisited_nodes, distance_matrix
    ):
Input variable types and meanings are: current_node: int, the
    current node in the partial solution. destination_node: int, the
     starting node that is also the final return destination.
    unvisited_nodes: list[int], the list of currently unvisited
    candidate nodes. distance_matrix: numpy.ndarray, a 2D distance
    matrix of shape (problem_size, problem_size), where
    distance_matrix[i][j] is the distance from node i to node j.
The function must return 1 output(s): next_node.
```

Each existing heuristic is represented by three components: an *algorithm* field summarizing the design rationale, a *code* field containing the executable Python implementation, and an *objective value*

obtained from finite-set evaluation. For the exploitation action, the exemplar heuristic is provided as Algorithm No.1, while the selected parent heuristic is provided as Algorithm No.2. The LLM is then prompted to improve the parent while selectively incorporating useful motifs from the exemplar.

Exploitation Prompt for TSP Constructive Heuristic

```
Solving Traveling Salesman Problem (TSP) with constructive
    heuristics. TSP requires finding the shortest path that visits
    all given nodes and returns to the starting node.

I have 2 existing algorithms with their codes and objective values
    as follows: No.1 algorithm's description, its corresponding code
     and its objective value are: exampler_algorithm, exampler_code,
     exampler_fitness. No.2 algorithm's description, its
    corresponding code and its objective value are: parent_algorithm
    , parent_code, parent_fitness.

Please create a new algorithm by improving Algorithm No.2 while
    selectively incorporating useful ideas from the other algorithms
    . The new algorithm must retain the overall framework of
    Algorithm No.2 but include clear non-trivial changes, rather
    than being identical or nearly identical to any existing
    algorithm. It should aim to achieve a lower objective value than
     all existing algorithms.Please output your response in exactly
    the following format:

<ALGORITHM>
one-sentence description of the design idea and main steps
</ALGORITHM>

<CODE>
# your code here
</CODE>

Requirements:
The algorithm description must be exactly one sentence inside <
    ALGORITHM>. The code inside <CODE> must be valid, directly
    executable Python. The code must strictly implement the
    algorithm described in <ALGORITHM>. Do not output any text
    outside <ALGORITHM> and <CODE>.
The new algorithm should improve Algorithm No.2 with clear non-
    trivial changes, while selectively incorporating useful ideas
    from the other algorithms.
The function signature must exactly be: def select_next_node(
    current_node, destination_node, unvisited_nodes, distance_matrix
    ):

Input variable types and meanings are: current_node: int, the
    current node in the partial solution. destination_node: int, the
     starting node that is also the final return destination.
    unvisited_nodes: list[int], the list of currently unvisited
    candidate nodes. distance_matrix: numpy.ndarray, a 2D distance
    matrix of shape (problem_size, problem_size), where
    distance_matrix[i][j] is the distance from node i to node j.
The function must return 1 output(s): next_node.
```

For the exploration action, k representative heuristics are provided to the LLM as contrastive examples. The LLM is prompted to synthesize a structurally novel heuristic rather than locally modifying the provided examples.

Following MCTS-AHD Zheng et al. (2025), an additional re-description step is applied after code generation to improve the consistency between the natural-language algorithm description and

the executable implementation. Given the generated code and its initial design idea, the LLM is prompted to re-describe the implemented heuristic by analyzing the actual code structure, variable priorities, and decision rules.

Exploration Prompt for TSP Constructive Heuristic

```
Solving Traveling Salesman Problem (TSP) with constructive
    heuristics. TSP requires finding the shortest path that visits
    all given nodes and returns to the starting node. I have k
    existing algorithms with their codes and objective values as
    follows: No.i algorithm's description, its corresponding code
    and its objective value are: ......

Please create a new algorithm that is substantially different from
    the existing algorithms. The new algorithm should aim to achieve
     a lower objective value.Please output your response in exactly
    the following format:

<ALGORITHM>
one-sentence description of the design idea and main steps
</ALGORITHM>

<CODE>
# your code here
</CODE>

Requirements:

The new algorithm must be fundamentally different from the provided
     algorithms and should introduce a distinct core idea, decision
    rule, or overall structure, rather than a minor modification of
    their existing framework. The algorithm description must be
    exactly one sentence inside <ALGORITHM>. The code inside <CODE>
    must be valid, directly executable Python. The code must
    strictly implement the algorithm described in <ALGORITHM>. Do
    not output any text outside <ALGORITHM> and <CODE>.
The function signature must exactly be: def select_next_node(
    current_node, destination_node, unvisited_nodes, distance_matrix
    ):
Input variable types and meanings are: current_node: int, the
    current node in the partial solution. destination_node: int, the
     starting node that is also the final return destination.
    unvisited_nodes: list[int], the list of currently unvisited
    candidate nodes. distance_matrix: numpy.ndarray, a 2D distance
    matrix of shape (problem_size, problem_size), where
    distance_matrix[i][j] is the distance from node i to node j.
The function must return 1 output(s): next_node.
```

Algorithm Re-description Prompt for TSP Constructive Heuristic

```
Solving Traveling Salesman Problem (TSP) with constructive
    heuristics. TSP requires finding the shortest path that visits
    all given nodes and returns to the starting node. Following is
    the Design Idea of a heuristic algorithm for the problem and the
     code with function name 'select_next_node' for implementing the
     heuristic algorithm. The select_next_node function takes as
    input the current node, the destination_node, a list of
    unvisited nodes, and a distance matrix, and returns the next
    node to visit. Design Idea: algorithm Code: code
```

```
The content of the Design Idea cannot fully represent what the
    algorithm has done. Therefore, you should now re-describe the
    algorithm using fewer than 3 sentences.

Hint: You should reference the given Design Idea and highlight the
    most critical design ideas implemented in the code. You may
    analyze the code to describe which variables are assigned higher
     or lower priority, as well as the parameters and structure of
    the implementation.
```

Algorithm 1 Dual-Process Tree Search

Require: finite evaluation set $\mathcal{D}_{\text{eval}}$, maximum evaluation budget T, initial tree size N_I, number of exploitation exemplars m, number of exploration representatives k, elite-set size M

Ensure: Search tree $\mathcal{T}$ and best heuristic $h^\star$

Initialize a rooted heuristic evolution tree $\mathcal{T} = (\mathcal{V}, \mathcal{A})$ with root v_0
Generate N_I initial heuristics $\mathcal{V}_{new}$ with INITFUNCTION(N_I, $\mathcal{D}_{\text{eval}}$)
Attach the initial heuristics as children of v_0
$\mathcal{O} \leftarrow \mathcal{V}_{new}$, $E \leftarrow \text{TopM}_{v\in\mathcal{O}}(g_v, M)$, $t \leftarrow N_I$
while $t < T$ **do**
    $\{(p_i, e_i)\}_{i=1}^m$, $\mathcal{R} \leftarrow$ SELECTION($\mathcal{T}, \mathcal{O}, E, m, k$)
    $\mathcal{V}_{\text{new}}$, $v_{\text{new}}^{(0)} \leftarrow$ EXPANSION($\{(p_i, e_i)\}_{i=1}^m, \mathcal{R}$)
    $\mathcal{G}_{\text{new}}$, $\mathcal{B}_{\text{new}} \leftarrow$ EVALUATION($\mathcal{V}_{\text{new}}, v_{\text{new}}^{(0)}, \mathcal{D}_{\text{eval}}$)
    $\mathcal{T} \leftarrow$ BACKPROPAGATION($\mathcal{T}, \mathcal{V}_{\text{new}}, \{e_i\}_{i=1}^m, \mathcal{G}_{\text{new}}, \mathcal{B}_{\text{new}}$)
    $\mathcal{O} \leftarrow \mathcal{O} \cup \mathcal{V}_{new} \cup \{v_{new}^{(0)}\} \setminus \{p_i\}_{i=1}^m$
    $E \leftarrow \text{TopM}_{v\in E\cup\mathcal{V}_{new}\cup\{v_{new}^{(0)}\}}(g_v, M)$
    $t \leftarrow t + |\mathcal{V}_{\text{new}}| + 1$
end while
$v^\star \leftarrow \arg\max_{v\in\mathcal{V}} g_v$, and $h^\star \leftarrow h_{v^\star}$
return $\mathcal{T}, h^\star$

B OVERALL ALGORITHM OF DPTS

DPTS proceeds through four stages, namely *selection*, *expansion*, *evaluation*, and *back-propagation*. At each iteration, nodes required by the selected action are first chosen from the current heuristic evolution tree, new heuristic programs are then generated by the LLM, evaluated on the finite evaluation set, and finally used to update heuristic merit and evolutionary credit.

The initialization procedure is designed to construct the initial root-level algorithm families without relying on handcrafted seed heuristics. DPTS first invokes the LLM with the initial prompt to generate one heuristic program from the task specification alone. Subsequent initial heuristics are generated through the exploration action, where previously generated heuristics are provided as references and the LLM is prompted to synthesize structurally distinct variants. The resulting N_I heuristics are evaluated on $\mathcal{D}_{\text{eval}}$, initialized with their node statistics, and attached as children of the root node v_0.

The selection procedure consists of exploitation selection and exploration selection. For exploitation, DPTS first samples $N_C = 20$ legal leaf nodes from the open set $\mathcal{O}$ to form a candidate set $\mathcal{V}^C$, and then selects the parent node with the largest heuristic merit Q within this candidate set. This candidate sampling step avoids deterministically exploiting only the current best leaf and improves search diversity Zhao et al. (2024). The auxiliary exemplar is selected from the elite set E by maximizing an exploration-augmented evolutionary credit score $B + U$, where

$$U_v = c\sqrt{\frac{\log(N_{ex}+1)}{N_v^B+1}}.$$

Algorithm 2 InitFunction

Require: Initial tree size N_I, finite evaluation set $\mathcal{D}_{\text{eval}}$
Ensure: Initial heuristic node set $\mathcal{V}_I$

$\mathcal{V}_I \leftarrow \emptyset$
Generate the first heuristic with the initial prompt: $h^{(1)} \leftarrow \mathcal{G}_{\text{init}}()$
Create node $v^{(1)}$ with heuristic $h^{(1)}$
Evaluate $h^{(1)}$ on $\mathcal{D}_{\text{eval}}$ to obtain $g_{v^{(1)}}$, and initialize statistics of $v^{(1)}$
$\mathcal{V}_I \leftarrow \mathcal{V}_I \cup \{v^{(1)}\}$
for $i = 2, \ldots, N_I$ **do**
    Generate a new structurally distinct heuristic with exploration prompt:

$$h^{(i)} \leftarrow \mathcal{G}_{\text{explore}}\big(\{h_v \mid v \in \mathcal{V}_I\}\big)$$

    Create node $v^{(i)}$ with heuristic $h^{(i)}$
    Evaluate $h^{(i)}$ on $\mathcal{D}_{\text{eval}}$ to obtain $g_{v^{(i)}}$ and initialize statistics of $v^{(i)}$
    $\mathcal{V}_I \leftarrow \mathcal{V}_I \cup \{v^{(i)}\}$
end for
return $\mathcal{V}_I$

Algorithm 3 Selection

Require: Search tree $\mathcal{T} = (\mathcal{V}, \mathcal{A})$, open set $\mathcal{O}$, elite set E, number of exploitation pairs m, number of exploration representatives k
Ensure: Parent–exemplar pairs $\{(p_i, e_i)\}_{i=1}^{m}$ and exploration reference set $\mathcal{R}$

$\mathcal{P} \leftarrow \emptyset$ ▷ *Exploitation selection*
for $i = 1, \ldots, m$ **do**
    Sample a candidate set $\mathcal{V}_i^C \subseteq \mathcal{O}$ with $|\mathcal{V}_i^C| = \min(N_C, |\mathcal{O}|)$
    Select parent by heuristic merit: $p_i = \arg\max_{v \in \mathcal{V}_i^C} \tilde{Q}_v$
    Select exemplar by exploration-augmented evolutionary credit:

$$e_i = \arg\max_{v \in E} \left(B_v + c\sqrt{\frac{\log(N_{\text{ex}} + 1)}{N_v^B + 1}} \right)$$

    $\mathcal{P} \leftarrow \mathcal{P} \cup \{(p_i, e_i)\}$
end for
Select $k' = \min(k, |\text{Ch}(v_0)|)$ root-level algorithm-lineage roots: ▷ *Exploration selection*

$$\mathcal{C} = \text{TopK}_{v \in \text{Ch}(v_0)}(Q_v, k')$$

$\mathcal{R} \leftarrow \emptyset$
for $r \in \mathcal{C}$ **do**
    $u \leftarrow r$
    while $u \notin \mathcal{O}$ **do**
        if $\text{Rand}() < \epsilon$ **then**
            $u \leftarrow$ a uniformly sampled child from $\text{Ch}(u)$
        else
            $u \leftarrow \arg\max_{v \in \text{Ch}(u)} Q_v$
        end if
    end while
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{u\}$
end for
return $\mathcal{P}, \mathcal{R}$

Algorithm 4 Expansion

Require: parent–exemplar pairs $\mathcal{P} = \{(p_i, e_i)\}_{i=1}^{m}$, exploration reference set $\mathcal{R}$

Ensure: Newly generated node set $\mathcal{V}_{\text{new}}$ by exploitation, and a generated node $v_{\text{new}}^{(0)}$ by exploration

$\mathcal{V}_{\text{new}} \leftarrow \emptyset$ ▷ *Exploitation expansion*

for $i = 1, \ldots, m$ **do**

Generate offspring code with the exploitation prompt:

$$c_i^{\text{exp}} \leftarrow \mathcal{G}_{\text{exploit}}\big(h_{p_i}, h_{e_i}\big)$$

Generate a faithful algorithm description with the refinement prompt:

$$a_i^{\text{exp}} \leftarrow \mathcal{G}_{\text{refine}}\big(c_i^{\text{exp}}\big)$$

Create node v_i^{exp} with code c_i^{exp} and description a_i^{exp}

Attach v_i^{exp} as a child of p_i

$\mathcal{V}_{\text{new}} \leftarrow \mathcal{V}_{\text{new}} \cup \{v_i^{\text{exp}}\}$

end for

Generate offspring code with the exploration prompt: ▷ *Exploration expansion*

$$c^{\text{explore}} \leftarrow \mathcal{G}_{\text{explore}}\big(\{h_v \mid v \in \mathcal{R}\}\big)$$

Generate a faithful algorithm description with the refinement prompt:

$$a^{\text{explore}} \leftarrow \mathcal{G}_{\text{refine}}\big(c^{\text{explore}}\big)$$

Create node $v_{\text{new}}^{(0)}$ with code c^{explore} and description a^{explore}

Attach v^{explore} as a child of the root v_0

return $\mathcal{V}_{\text{new}}, v_{\text{new}}^{(0)}$

Algorithm 5 Evaluation

Require: Exploitation offspring set $\mathcal{V}_{\text{new}}$, exploration offspring $v_{\text{new}}^{(0)}$, finite evaluation set $\mathcal{D}_{\text{eval}}$

Ensure: Fitness values $\mathcal{G}_{\text{new}}$ and gain scores $\mathcal{B}_{\text{new}}$

$\mathcal{G}_{\text{new}} \leftarrow \emptyset, \mathcal{B}_{\text{new}} \leftarrow \emptyset$

for $v \in \mathcal{V}_{\text{new}}$ **do** ▷ *Evaluate exploitation offspring*

Evaluate h_v on $\mathcal{D}_{\text{eval}}$ to obtain empirical fitness g_v

$\mathcal{G}_{\text{new}} \leftarrow \mathcal{G}_{\text{new}} \cup \{(v, g_v)\}$

Let $p(v)$ and $e(v)$ denote the parent and exemplar used to generate v

Compute evolutionary gain:

$$b_v = \mathbb{I}\big[g_v > g_{p(v)}\big]$$

$\mathcal{B}_{\text{new}} \leftarrow \mathcal{B}_{\text{new}} \cup \{(e(v), b_v)\}$

end for

Evaluate $h_{v_{\text{new}}^{(0)}}$ on $\mathcal{D}_{\text{eval}}$ to obtain empirical fitness $g_{v_{\text{new}}^{(0)}}$ ▷ *Evaluate exploration offspring*

$\mathcal{G}_{\text{new}} \leftarrow \mathcal{G}_{\text{new}} \cup \{(v_{\text{new}}^{(0)}, g_{v_{\text{new}}^{(0)}})\}$

return $\mathcal{G}_{\text{new}}, \mathcal{B}_{\text{new}}$

Here, $c = 1.5$ controls the exploration strength, N denotes the total number of exemplar selections, and N_v^B denotes the number of times node v has been selected as an exemplar. The exploitation selection is repeated m times to obtain m parent–exemplar pairs. For exploration, DPTS selects k root children with the largest heuristic merit Q and performs an ϵ-greedy traversal from each selected root child to a leaf node, yielding k representative heuristics as references for generating a new algorithm lineage.

Given the selected parent-exemplar pairs and exploration references, the expansion procedure invokes the LLM with action-specific prompts to generate new heuristic programs. For exploitation, each selected pair (p_i, e_i) is passed to the exploitation prompt, where h_{p_i} provides the algorithmic backbone and h_{e_i} provides transferable design motifs for refinement. For exploration, the representative set $\mathcal{R}$ is passed to the exploration prompt to synthesize a structurally novel heuristic. After

each offspring program is generated, a refinement prompt is further applied to produce a faithful natural-language description of the implemented code. Exploitation offspring are attached as children of their corresponding parents, whereas the exploration offspring is attached as a new child of the root v_0, initiating a new algorithm lineage.

The evaluation procedure assigns empirical fitness to all newly generated heuristics on the finite evaluation set $\mathcal{D}_{\text{eval}}$. Let $\mathcal{V}_{\text{new}}$ denote the exploitation offspring set and $v_{\text{new}}^{(0)}$ denote the exploration offspring. For each exploitation offspring, an evolutionary gain score is additionally computed by comparing its fitness with that of its parent. A positive improvement is assigned $b = 1$, while non-improvement is assigned $b = 0$. The exploration offspring is evaluated only for heuristic merit and does not contribute exemplar credit.

The back-propagation procedure updates the two statistics maintained by DPTS through distinct propagation paths. For heuristic merit, the empirical fitness of each newly generated node is propagated along its parent path toward the root, with a discount factor $\gamma = 0.9$ applied to attenuate credit over longer ancestor distances. For evolutionary credit, each gain score b_v is propagated from the corresponding exemplar node $e(v)$ toward the root, updating the exemplar usage statistics and evolutionary credit. Thus, parent lineages are credited for producing high-fitness offspring, while exemplar lineages are credited for inducing beneficial reference-guided improvements.

Algorithm 6 Back-Propagation

Require: Fitness values $\mathcal{G}_{\text{new}}$, gain scores $\mathcal{B}_{\text{new}}$
Ensure: Updated search tree $\mathcal{T}$

for $v \in \mathcal{V}_{\text{new}} \cup \{v_{\text{new}}^{(0)}\}$ **do** ▷ *Merit back-propagation*
  Retrieve empirical fitness g_v from $\mathcal{G}_{\text{new}}$
  $u \leftarrow v$, $d \leftarrow 0$
  while u exists **do**
    $N_u \leftarrow N_u + 1$
    $W_u \leftarrow W_u + \gamma^d g_v$
    $Q_u \leftarrow W_u / N_u$
    $u \leftarrow parent(u)$
    $d \leftarrow d + 1$
  end while
end for
for $(e, b) \in \mathcal{B}_{\text{new}}$ **do** ▷ *Exemplar-credit back-propagation*
  $u \leftarrow e$
  while u exists **do**
    $N_u^B \leftarrow N_u^B + 1$
    $W_u^B \leftarrow W_u^B + b$
    $B_u \leftarrow W_u^B / N_u^B$
    $u \leftarrow parent(u)$
  end while
end for
return $\mathcal{T}$

C ALGORITHM OF TOPOLOGY-AWARE MERIT AGGREGATION

After back-propagation, each ancestor node u carries a subtree-level merit estimate Q_u, since fitness evaluations from its descendants have been accumulated along the parent path. Topology-aware merit aggregation further performs a forward propagation step to transfer such subtree-level evidence to leaf nodes in the open set $\mathcal{O}$. For a leaf node $v \in \mathcal{O}$, merit estimates along its root-to-leaf path provide evidence from algorithmic variants that share ancestors with v, including variants from sibling branches. Since a more distant shared ancestor indicates weaker structural similarity, DPTS assigns smaller weights to farther ancestors through a truncated linear decay. The resulting topology-aggregated merit $\widetilde{Q}_v$ evaluates a leaf node by combining its own lineage quality with the broader performance of its algorithm lineage.

Algorithm 7 Topology-Aware Merit Aggregation

Require: Search tree $\mathcal{T} = (\mathcal{V}, \mathcal{A})$, open set $\mathcal{O}$, decay coefficient α
Ensure: Topology-aggregated merits $\{\widetilde{Q}_v\}_{v\in\mathcal{O}}$
for $v \in \mathcal{O}$ **do**
  Let $\mathcal{L}(v) = \{v_0, v_1, \ldots, v_l = v\}$ denote the root-to-leaf path of v
  for $i = 0, \ldots, l$ **do**
    Compute unnormalized structural weight:$w_i = \max\{0,\, 1 - \alpha|l - i|\}$
  end for
  Normalize weights:$\bar{w}_i = \frac{w_i}{\sum_{j=0}^{l} w_j}, \qquad i = 0, \ldots, l$
  Compute topology-aggregated merit: $\widetilde{Q}_v = \sum_{i=0}^{l} \bar{w}_i Q_{v_i}$
end for
return $\{\widetilde{Q}_v\}_{v\in\mathcal{O}}$

D Proof of Theorem 1

Proof. We consider a minimization problem and analyze the effect of the propagation and aggregation mechanism on the estimation of frontier nodes. For analytical convenience, we set the merit-propagation discount factor to $\gamma = 1.0$ in the theoretical analysis.

For any node v_i, let N_{v_i} denote the number of evaluated descendants whose information is propagated to v_i. Under the Gaussian noise assumption, the propagated merit estimate satisfies

$$Q_{v_i} \sim \mathcal{N}\left(\mu_{v_i}, \frac{\sigma_b^2}{N_{v_i}}\right). \tag{20}$$

For an OPEN node v_k with ancestral path

$$\mathcal{L}(v_k) = \{v_0, v_1, \ldots, v_k\},$$

DPTS computes the topology-aware aggregated merit

$$\widetilde{Q}_{v_k} = \sum_{i=0}^{k} \bar{w}_i Q_{v_i}, \tag{21}$$

where

$$\bar{w}_i \geq 0, \qquad \sum_{i=0}^{k} \bar{w}_i = 1.$$

Assuming independence among the propagated merit estimates,

$$\widetilde{Q}_{v_k} \sim \mathcal{N}\left(\widetilde{\mu}_{v_k}, \widetilde{\sigma}_{v_k}^2\right), \tag{22}$$

with

$$\widetilde{\mu}_{v_k} = \sum_{i=0}^{k} \bar{w}_i \mu_{v_i}, \tag{23}$$

and

$$\widetilde{\sigma}_{v_k}^2 = \sigma_b^2 \sum_{i=0}^{k} \frac{\bar{w}_i^2}{N_{v_i}}. \tag{24}$$

Since $N_{v_i} \geq 1$,

$$\widetilde{\sigma}_{v_k}^2 \leq \sigma_b^2 \sum_{i=0}^{k} \bar{w}_i^2. \tag{25}$$

Furthermore, because the weights are nonnegative and normalized,

$$\sum_{i=0}^{k} \bar{w}_i^2 \leq \left(\sum_{i=0}^{k} \bar{w}_i\right)^2 = 1. \tag{26}$$

Therefore,

$$\widetilde{\sigma}^2_{v_k} \leq \sigma^2_b. \tag{27}$$

The inequality is strict whenever multiple estimates with positive weights contribute to the aggregation. Hence, propagation and topology-aware aggregation reduce the estimation variance relative to an individual node estimate.

We next consider the expected ordering between optimal and non-optimal nodes. Let $\mu^f_0 < \mu^f_1$ denote the expected ground-truth values of optimal and non-optimal nodes, respectively. Let $\rho(v_i)$ denote the proportion of optimal descendants contributing to the propagated estimate at v_i. Then

$$\mu_{v_i} = \rho(v_i)\mu^f_0 + \big(1 - \rho(v_i)\big)\mu^f_1. \tag{28}$$

Since $\mu^f_0 < \mu^f_1$, μ_{v_i} decreases monotonically with $\rho(v_i)$.

Consider an optimal frontier node v_k and a non-optimal frontier node v_t after their paths diverge. Along the non-optimal branch, no optimal descendant is contained in the corresponding subtree, giving $\rho(v_i) = 0$. In contrast, nodes along the optimal branch retain $\rho(v_i) > 0$. Consequently,

$$\widetilde{\mu}_{v_k} < \widetilde{\mu}_{v_t}, \tag{29}$$

and the expected ordering between the optimal and non-optimal frontier nodes is preserved by the aggregation mechanism.

Define the positive expected gap

$$\Delta = \widetilde{\mu}_{v_t} - \widetilde{\mu}_{v_k} > 0. \tag{30}$$

Because $\widetilde{Q}_{v_k}$ and $\widetilde{Q}_{v_t}$ are Gaussian, their difference is also Gaussian:

$$\widetilde{Q}_{v_t} - \widetilde{Q}_{v_k} \sim \mathcal{N}\left(\Delta, \widetilde{\sigma}^2_{v_k} + \widetilde{\sigma}^2_{v_t}\right). \tag{31}$$

The probability of correctly ranking the optimal node is therefore

$$\Pr\left(\widetilde{Q}_{v_k} < \widetilde{Q}_{v_t}\right) = \Phi\left(\frac{\Delta}{\sqrt{\widetilde{\sigma}^2_{v_k} + \widetilde{\sigma}^2_{v_t}}}\right), \tag{32}$$

where $\Phi(\cdot)$ denotes the cumulative distribution function of the standard normal distribution.

For a fixed positive expected gap Δ, the above probability increases monotonically as the estimation variances decrease. Since topology-aware propagation and aggregation reduce the variance while preserving the expected ordering, they reduce the overlap between the score distributions of optimal and non-optimal frontier nodes and thereby improve the probability of correctly selecting the optimal node. This completes the proof. □

E PROBLEM DEFINITIONS

In this paper, DPTS is evaluated on representative NP-hard combinatorial optimization problems, including the Traveling Salesman Problem (TSP), Knapsack Problem (KP), and Bin Packing Problem (BPP). These tasks cover routing, resource allocation, constrained selection, and packing, providing diverse testbeds for assessing heuristic design across different objective structures and feasibility constraints.

Traveling Salesman Problem. Given a complete graph $G = (V, E)$ with node set $V = \{1, \ldots, n\}$ and distance matrix $D \in \mathbb{R}^{n \times n}$, TSP aims to find a Hamiltonian cycle with minimum total length. Let $\pi = (\pi_1, \ldots, \pi_n)$ denote a permutation of all nodes. The objective is

$$\min_{\pi} \left[\sum_{i=1}^{n-1} D_{\pi_i, \pi_{i+1}} + D_{\pi_n, \pi_1}\right]. \tag{33}$$

In the constructive setting, a heuristic sequentially selects the next unvisited node until a complete tour is formed. In the ACO setting, a heuristic is instantiated as a heuristic matrix that guides the probabilistic transition rule, biasing ants toward promising next nodes during tour construction.

Knapsack Problem. Given n items with values p_i, weights w_i, and capacity C, KP seeks a subset of items with maximum total value under a single capacity constraint. Let $x_i \in \{0,1\}$ indicate whether item i is selected. The problem is formulated as

$$\max_{x\in\{0,1\}^n} \sum_{i=1}^{n} p_i x_i, \qquad \text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \le C. \tag{34}$$

A constructive heuristic determines the order or priority by which feasible items are selected.

Bin Packing Problem. Given n items with sizes s_i and bins of capacity C, BPP aims to pack all items into the minimum number of bins without exceeding bin capacities. Let $y_b \in \{0,1\}$ indicate whether bin b is used and $x_{ib} \in \{0,1\}$ indicate whether item i is assigned to bin b. The standard formulation is

$$\min_{x,y} \sum_{b=1}^{n} y_b, \qquad \text{s.t.} \quad \sum_{b=1}^{n} x_{ib} = 1,\ \forall i, \qquad \sum_{i=1}^{n} s_i x_{ib} \le C y_b,\ \forall b. \tag{35}$$

In the offline setting, all item sizes are known before packing, allowing heuristics to exploit global item information.

In this paper, two optimization frameworks are considered, inclduing the step-by-step construction framework and the Ant Colony Optimization (ACO) framework.

Step-by-step Construction Framework. The step-by-step construction framework builds a feasible solution through a sequence of local decisions. At each construction step t, a partial solution s_t is maintained, and a heuristic function selects the next feasible component from a candidate set $\mathcal{C}(s_t)$:

$$a_t = h(s_t, \mathcal{C}(s_t), \xi), \qquad s_{t+1} = s_t \oplus a_t, \tag{36}$$

where ξ denotes task-specific information such as distances, demands, capacities, or item weights, and $\oplus$ denotes the operation of adding the selected component to the partial solution. The process terminates when a complete feasible solution is obtained. In this framework, LLM-based AHD designs the decision rule h, such as a next-node selection function for TSP or an item-selection rule for KP.

Ant Colony Optimization Framework. Ant Colony Optimization (ACO) constructs solutions through stochastic transitions guided by pheromone information and heuristic desirability. Given a partial solution s_t, the probability of selecting a feasible component $a \in \mathcal{C}(s_t)$ is typically defined as

$$P(a \mid s_t) = \frac{\tau(a)^\alpha \eta(a)^\beta}{\sum_{a'\in\mathcal{C}(s_t)} \tau(a')^\alpha \eta(a')^\beta}, \tag{37}$$

where $\tau(a)$ denotes the pheromone value, $\eta(a)$ denotes the heuristic desirability, and α, β control their relative influence.

In the ACO framework, LLM-based AHD designs the heuristic component η, typically instantiated as a heuristic matrix or task-specific scoring function that biases probabilistic construction toward promising decisions. Unlike deterministic construction, ACO evaluates a heuristic through its effect on a population of sampled solutions, making η responsible for shaping the global search distribution rather than a single greedy trajectory. For TSP, η_{ij} estimates the desirability of visiting city j after city i. For BPP, the heuristic scores item–bin assignments, favoring placements that improve capacity utilization while preserving feasibility for subsequent items.

F Additional Constructive Results on KP

Table 3 reports the results on the Knapsack Problem (KP) under the step-by-step construction framework. DPTS consistently improves upon existing LLM-based AHD baselines across all evaluated problem sizes, achieving optimality gaps of 0.02%, 0.01%, and 0.01% at $N = 50$, $N = 100$, and $N = 200$, respectively. DPTS-K further improves the objective values across all three scales, with corresponding gaps of 0.01%, 0.00%, and 0.01%. The improvements are maintained at both the training-time scale and larger test scales.

Table 3: Performance comparison on KP under the step-by-step construction framework. The shaded columns indicate the training-time problem size. Optimal solutions are obtained by OR-Tools. Among LLM-based AHD methods, the best result is shown in bold and the second-best result is underlined.

Method	$N = 50, W = 12.5$		$N = 100, W = 25$		$N = 200, W = 25$	
	Obj.↑	Gap↓	Obj.↑	Gap↓	Obj.↑	Gap↓
Optimal	20.037	–	40.271	–	57.448	–
Greedy	19.985	0.26%	40.225	0.12%	57.395	0.09%
POMO	19.612	2.12%	39.676	1.48%	57.271	0.31%
FunSearch	19.988	0.24%	40.227	0.11%	57.398	0.09%
EoH	19.993	0.22%	40.231	0.10%	57.399	0.09%
MCTS-AHD	20.015	0.11%	40.252	0.05%	57.423	0.04%
DPTS	<u>20.034</u>	<u>0.02%</u>	<u>40.269</u>	<u>0.01%</u>	<u>57.444</u>	<u>0.01%</u>
DPTS-K	**20.035**	**0.01%**	**40.270**	**0.00%**	**57.445**	**0.01%**

G Additional Results on CVRP and MKP

CVRP introduces vehicle-capacity constraints and multi-route coordination, while MKP involves multiple coupled resource constraints. We further evaluate DPTS on both problems to assess its generalization to more constrained optimization settings. The results are reported in Table 4.

Table 4: ACO-based CVRP and MKP performance comparison over three independent runs. Shaded columns denote the training problem size. Among LLM-based AHD methods, the best result is shown in bold and the second-best result is underlined.

Methods	CVRP				MKP			
	$N = 50, C = 50$		$N = 100, C = 50$		$N = 100, m = 5$		$N = 200, m = 5$	
	Obj.↓	Gap↓	Obj.↓	Gap↓	Obj.↑	Gap↓	Obj.↑	Gap↓
ACO	11.355	27.77%	18.778	25.76%	22.738	2.47%	40.672	4.64%
DeepACO	8.888	0.00%	14.932	0.00%	23.093	0.95%	41.988	1.55%
EoH	9.359	5.31%	15.681	5.02%	23.139	0.75%	41.994	1.54%
ReEvo	9.327	4.94%	16.092	7.77%	23.245	0.30%	42.416	0.55%
HSEvo	9.349	5.19%	16.282	9.04%	23.276	0.17%	42.494	0.37%
MCTS-AHD	9.286	4.48%	15.782	5.70%	23.269	0.20%	42.498	0.36%
DPTS	<u>9.136</u>	<u>2.79%</u>	<u>15.340</u>	<u>2.73%</u>	<u>23.279</u>	<u>0.15%</u>	<u>42.614</u>	<u>0.08%</u>
DPTS-K	**9.119**	**2.60%**	**15.302**	**2.48%**	**23.315**	**0.00%**	**42.650**	**0.00%**

DPTS consistently improves upon existing LLM-based AHD methods across all CVRP and MKP settings. Compared with MCTS-AHD, DPTS reduces the optimality gaps from 4.48% to 2.79% and from 5.70% to 2.73% on CVRP at $N = 50$ and $N = 100$, respectively. On MKP, the corresponding gaps decrease from 0.20% to 0.15% at $N = 100$ and from 0.36% to 0.08% at $N = 200$. DPTS-K further achieves the best performance among LLM-based AHD methods across all four settings. The results demonstrate the generalization of DPTS to more constrained optimization problems.

H Cross-Domain Evaluation on DPP

To further evaluate the applicability of DPTS beyond conventional combinatorial optimization benchmarks, we consider the Decap Placement Problem (DPP), an application-oriented task in Electronic Design Automation. The results are reported in Table 5. DPTS and DPTS-K achieve objective values of 12.697 and 12.911, respectively, both exceeding the genetic algorithm baseline. DPTS-K further surpasses DevFormer-CSE Kim et al. (2023), indicating that the proposed search framework remains effective in an application-oriented optimization domain distinct from the primary benchmarks.

Table 5: Performance comparison on the Decap Placement Problem. Higher values are better.

Method	Objective ↑
Genetic Algorithm	12.410
DevFormer-CSE	12.880
DPTS	12.697
DPTS-K	**12.911**

I DPTS-K EXPERT EXTRACTION

DPTS-K is a multi-heuristic extension of DPTS for extracting a diverse expert portfolio from the learned search tree. As each root child in DPTS corresponds to a distinct algorithm lineage, selecting experts at the branch level avoids concentrating the portfolio on redundant variants from a single lineage. For each branch rooted at $r \in \mathrm{Ch}(v_0)$, a Hoeffding-style upper confidence score Hoeffding (1963) is used to balance estimated branch merit and convergence reliability:

$$Score(r) = Q_r + (Q_{\max} - Q_{\min}) \sqrt{\frac{\log(1/\delta)}{2N_r}}, \tag{38}$$

where $\delta \in (0, 1)$ controls the confidence level, N_r denotes the visit count of branch root r, and $Q_{\max}$ and $Q_{\min}$ denote the maximum and minimum merit estimates among root branches. The top-K branches are selected according to $Score(r)$, and the highest-fitness heuristic within each selected branch is retained as its representative expert. The resulting portfolio combines complementary algorithmic families, enabling different heuristics to address different problem instances. For each test instance, all K extracted heuristics are evaluated, and the best-performing result is reported.

Algorithm 8 DPTS-K Expert Extraction

Require: Search tree $\mathcal{T} = (\mathcal{V}, \mathcal{A})$, root node v_0, portfolio size K, confidence parameter δ
Ensure: Heuristic expert portfolio $\mathcal{P}_K$

$\mathcal{P}_K \leftarrow \emptyset$
$\mathcal{C} \leftarrow \mathrm{Ch}(v_0)$ ▷ root-level algorithm families
$Q_{\max} \leftarrow \max_{r \in \mathcal{C}} Q_r, \quad Q_{\min} \leftarrow \min_{r \in \mathcal{C}} Q_r$
for $r \in \mathcal{C}$ **do**
    Compute branch confidence score:

$$Score(r) = Q_r + (Q_{\max} - Q_{\min}) \sqrt{\frac{\log(1/\delta)}{2N_r}}$$

end for
Select top-K branches:

$$\mathcal{C}_K = \mathrm{TopK}_{r \in \mathcal{C}} \left(Score(r), K \right)$$

for $r \in \mathcal{C}_K$ **do**
    $v_r^\star \leftarrow \arg\max_{v \in \mathrm{Subtree}(r)} g_v$
    $\mathcal{P}_K \leftarrow \mathcal{P}_K \cup \{h_{v_r^\star}\}$
end for
return $\mathcal{P}_K$

The following heuristic experts are extracted by DPTS-K under the TSP constructive framework. Although all five heuristics share the same next-node selection interface, they instantiate distinct decision principles across different algorithmic families. Specifically, h_1 extends greedy distance minimization with tabu-style candidate filtering, neighborhood-distance calibration, and cumulative penalty terms; h_2 implements a parsimonious return-aware nearest-neighbor rule that jointly accounts for immediate travel cost and depot-return cost; h_3 introduces adaptive exploratory scoring by combining cumulative-distance estimates, distance-distribution penalties, and local-minimum escape; h_4 performs rollout-based lookahead by approximating the completion cost induced by each

candidate through greedy simulation; and h_5 further strengthens lookahead with MST-based residual lower bounds, density-aware proximity penalties, and graph-structured neighborhood information. The spectrum from local greedy selection to rollout and lower-bound-based lookahead demonstrates that DPTS-K retrieves heterogeneous heuristic experts from distinct search-tree branches rather than redundant variants of a single construction pattern.

h_1: Penalty-Augmented Greedy Search with Tabu Control

```
import numpy as np
def select_next_node(current_node, destination_node,
    unvisited_nodes, distance_matrix):
   if not unvisited_nodes:
      return None
   tabu_size = max(3, len(unvisited_nodes) // 3)
   tabu_list = []
   tabu_list.append(current_node)
   if len(tabu_list) > tabu_size:
      tabu_list.pop(0)
   candidates = [node for node in unvisited_nodes if node not in
       tabu_list]
   if not candidates:
      candidates = unvisited_nodes
   best_next_node = None
   min_score = float('inf')
   total_distance_travelled = sum(distance_matrix[current_node][
       node] for node in unvisited_nodes)
   for candidate in candidates:
      distance_to_candidate = distance_matrix[current_node][
          candidate]
      distances_to_unvisited = [
         distance_matrix[candidate][neighbor]
         for neighbor in unvisited_nodes
         if neighbor != candidate
      ]
      avg_distance_to_unvisited = np.mean(distances_to_unvisited)
          if distances_to_unvisited else 0
      weight_factor = 0.4 + (1 - len(unvisited_nodes) / len(
          distance_matrix)) * 0.3
      weighted_avg_distance = (
         weight_factor * avg_distance_to_unvisited
         + (1 - weight_factor) * distance_to_candidate
      )
      recent_visit_penalty = (
         (1.0 if current_node in tabu_list else 0)
         * (distance_to_candidate - avg_distance_to_unvisited)
         * 0.5
      )
      cumulative_penalty = (
         total_distance_travelled / (len(unvisited_nodes) + 1)
      ) * 0.1 + (distance_to_candidate * 0.05)

      score = distance_to_candidate + recent_visit_penalty +
          cumulative_penalty
      if score < min_score:
         min_score = score
         best_next_node = candidate
   return best_next_node
```

h_2: Return-Aware Nearest-Neighbor Selection

```
import numpy as np
def select_next_node(current_node, destination_node,
    unvisited_nodes, distance_matrix):
   if not unvisited_nodes:
      return None
   best_next_node = None
   min_total_cost = float('inf')
   return_distance = distance_matrix[current_node][destination_node
       ]
   for next_node in unvisited_nodes:
      current_distance = distance_matrix[current_node][next_node]
      total_cost = current_distance + return_distance
      if total_cost < min_total_cost:
         min_total_cost = total_cost
         best_next_node = next_node
   return best_next_node
```

h_3: Adaptive Exploratory Greedy Search

```
import numpy as np
def select_next_node(current_node, destination_node,
    unvisited_nodes, distance_matrix):
   if not unvisited_nodes:
      return None
   cumulative_distances = {node: float('inf') for node in
       unvisited_nodes}
   cumulative_distances[current_node] = 0
   for node in unvisited_nodes:
      distance_to_node = distance_matrix[current_node][node]
      cumulative_distances[node] = cumulative_distances[
          current_node] + distance_to_node
   avg_distance_to_unvisited = np.mean([
      distance_matrix[current_node][node]
      for node in unvisited_nodes
   ])
   exploration_factor = min(
      0.9,
      max(
         0.6,
         avg_distance_to_unvisited / (np.sum(list(
             cumulative_distances.values())) + 1)
      )
   )
   next_node = None
   min_cost = float('inf')
   for node, total_cost in cumulative_distances.items():
      if node != current_node:
         distance_to_node = distance_matrix[current_node][node]
         median_distance = np.median([
            distance_matrix[node][n]
            for n in unvisited_nodes
            if n != node
         ])
         penalty = (
            (distance_to_node - median_distance) * 0.3
            if distance_to_node > median_distance
            else 0
         )
```

```
            weighted_score = (
                exploration_factor * distance_to_node
                + (1 - exploration_factor) * total_cost
                + penalty
            )
            if weighted_score < min_cost:
                min_cost = weighted_score
                next_node = node
    local_threshold = 1.5 * np.mean(list(cumulative_distances.values
        ()))
    if min_cost > local_threshold:
        next_node = max(unvisited_nodes, key=lambda n:
            cumulative_distances[n])
    return next_node
```

h_4: Rollout-Based Completion-Cost Estimation

```
import numpy as np
import random
def select_next_node(current_node, destination_node,
    unvisited_nodes, distance_matrix):
    if not unvisited_nodes:
        return None
    proximity_penalty_weight = 0.3
    exploration_factor = 0.7
    recent_visit_penalty_weight = 0.2
    distances = [distance_matrix[current_node][node] for node in
        unvisited_nodes]
    avg_distance_to_unvisited = np.mean(distances)
    recent_visits = {node: 0 for node in unvisited_nodes}
    cost_to_completion = []
    for node in unvisited_nodes:
        remaining_nodes = unvisited_nodes.copy()
        remaining_nodes.remove(node)
        if remaining_nodes:
            total_cost = distance_matrix[current_node][node]
            current_node_for_next = node
            while remaining_nodes:
                next_node = min(
                    remaining_nodes,
                    key=lambda n: distance_matrix[current_node_for_next
                        ][n]
                )
                total_cost += distance_matrix[current_node_for_next][
                    next_node]
                current_node_for_next = next_node
                remaining_nodes.remove(next_node)

            total_cost += distance_matrix[current_node_for_next][
                destination_node]
            cost_to_completion.append(total_cost)
        else:
            cost_to_completion.append(
                distance_matrix[current_node][node]
                + distance_matrix[node][destination_node]
            )
    combined_scores = []
    for i, node in enumerate(unvisited_nodes):
        distance_to_candidate = distances[i]
```

```
        proximity_penalty = distance_to_candidate *
            proximity_penalty_weight
        adaptive_penalty = (
            (avg_distance_to_unvisited - distance_to_candidate) * 0.2
            if distance_to_candidate < avg_distance_to_unvisited
            else 0
        )
        recent_visit_penalty = recent_visits[node] *
            recent_visit_penalty_weight
        total_score = (
            distance_to_candidate
            + proximity_penalty
            + adaptive_penalty
            + recent_visit_penalty
            + cost_to_completion[i]
        )
        combined_scores.append(total_score)
    min_score = min(combined_scores)
    candidates = [
        unvisited_nodes[i]
        for i in range(len(combined_scores))
        if combined_scores[i] == min_score
    ]
    if len(candidates) > 1:
        weights = [
            1 / exploration_factor * (min_score - combined_scores[i])
            for i in range(len(combined_scores))
            if combined_scores[i] == min_score
        ]
        next_node = random.choices(candidates, weights=weights, k=1)
            [0]
    else:
        next_node = candidates[0]
    recent_visits[next_node] += 1
    return next_node
```

h_5: MST-Enhanced Graph-Structured Lookahead

```
import numpy as np
from scipy.sparse.csgraph import minimum_spanning_tree
def calculate_mst_cost(unvisited_nodes, distance_matrix):
    if not unvisited_nodes:
        return 0
    sub_matrix = distance_matrix[np.ix_(unvisited_nodes,
        unvisited_nodes)]
    return minimum_spanning_tree(sub_matrix).toarray().sum()

def cluster_unvisited_nodes(unvisited_nodes, distance_matrix):
    clusters = []
    visited = set()
    for start in unvisited_nodes:
        if start in visited:
            continue
        current_cluster = [start]
        visited.add(start)
        stack = [start]
        while stack:
            node = stack.pop()
            for neighbor in unvisited_nodes:
                if (
```

```
                neighbor not in visited
                and distance_matrix[node][neighbor]
                < np.mean(distance_matrix[node][unvisited_nodes]) *
                    0.50
            ):
                visited.add(neighbor)
                current_cluster.append(neighbor)
                stack.append(neighbor)
        clusters.append(current_cluster)
    return clusters

def select_next_node(current_node, destination_node,
    unvisited_nodes, distance_matrix):
    if not unvisited_nodes:
        return None
    min_total_cost = float('inf')
    best_candidate = None
    unvisited_count = len(unvisited_nodes)
    clusters = cluster_unvisited_nodes(unvisited_nodes,
        distance_matrix)
    for candidate in unvisited_nodes:
        remaining_nodes = unvisited_nodes.copy()
        remaining_nodes.remove(candidate)
        current_path_length = distance_matrix[current_node][candidate
            ]
        current_node_for_next = candidate
        while remaining_nodes:
            next_node = min(
                remaining_nodes,
                key=lambda n: distance_matrix[current_node_for_next][n]
            )
            current_path_length += distance_matrix[
                current_node_for_next][next_node]
            current_node_for_next = next_node
            remaining_nodes.remove(next_node)
        current_path_length += distance_matrix[current_node_for_next
            ][destination_node]
        mst_cost = calculate_mst_cost(remaining_nodes,
            distance_matrix)
        closest_distances = sorted([
            distance_matrix[candidate][node]
            for node in unvisited_nodes
        ])
        proximity_penalty = 0
        if closest_distances:
            average_closest_distance = np.mean(
                closest_distances[:min(3, len(closest_distances))]
            )
            density_weight = 1.0 if unvisited_count > 5 else 0.5
            proximity_penalty = max(
                0,
                (distance_matrix[current_node][candidate] -
                    average_closest_distance)
                * density_weight
            )
        total_cost = (
            current_path_length
            + mst_cost * 0.45
            + proximity_penalty * (1 + 0.5 * (unvisited_count / len(
                distance_matrix)))
        )
        if total_cost < min_total_cost:
            min_total_cost = total_cost
```

```
            best_candidate = candidate
    return best_candidate
```

J Component Ablation Studies

We conduct component ablations on constructive TSP to assess the contributions of evolutionary credit and topology-aware aggregation. DPTS w.o. evolutionary credit replaces credit-aware exemplar selection with the sampling strategy of MCTS-AHD Zheng et al. (2025), whereas DPTS w.o. topology-aware aggregation uses Q in place of $\tilde{Q}$. Results are reported in Table 6.

Table 6: Component ablation results on constructive TSP.

Method	$N = 50$		$N = 100$		$N = 200$	
	Obj.↓	Gap↓	Obj.↓	Gap↓	Obj.↓	Gap↓
MCTS-AHD Zheng et al. (2025)	6.225	9.69%	8.684	11.79%	12.120	13.71%
DPTS w.o. evolutionary credit	6.165	8.63%	8.583	10.49%	11.928	11.91%
DPTS w.o. topology-aware aggregation	6.323	11.42%	8.842	13.83%	12.412	16.45%
DPTS	**6.158**	**8.52%**	**8.559**	**10.19%**	**11.903**	**11.67%**

Removing evolutionary credit consistently degrades performance across all three problem sizes, supporting the contribution of evolutionary utility to exemplar selection. A more pronounced degradation occurs without topology-aware aggregation, where the optimality gaps increase from 8.52%, 10.19%, and 11.67% to 11.42%, 13.83%, and 16.45%, respectively. The results support the role of topology-aware aggregation in reducing sensitivity to finite-set evaluation and improving out-of-sample robustness.

K Hyperparameter Sensitivity

We evaluate the sensitivity of DPTS to the hyperparameters α, c, and γ on constructive KP. A one-at-a-time analysis is conducted by varying one parameter while fixing the others at their default values, with $\alpha = 0.2$, $c = 1.5$, and $\gamma = 0.9$. The results are reported in Table 7. Performance remains stable across the evaluated settings of all three hyperparameters, with only minor variations across KP50, KP100, and KP200. DPTS also remains competitive with or superior to MCTS-AHD throughout the reported configurations. The results indicate limited sensitivity to moderate variations in α, c, and γ, supporting the robustness of DPTS to hyperparameter selection.

L Examples of generated heuristics

For each benchmark task, heuristic functions are generated according to the specified input-output interface and integrated into the corresponding solver to guide decision-making. In this section, representative examples of generated heuristic functions are provided for different tasks.

For constructive TSP, the generated heuristic is implemented as a `select_next_node` function that determines the next city to visit during sequential tour construction. For each unvisited candidate, the function first evaluates the downstream effect of committing to that candidate as the next city. A nearest-neighbor rollout is used to approximate the residual tour cost, capturing lookahead information beyond the immediate transition. An MST-based residual term is then incorporated as a graph-structured lower-bound estimate over the remaining nodes, while a density-aware proximity penalty accounts for local neighborhood structure. The candidate with the lowest aggregated cost is selected.

Table 7: Hyperparameter sensitivity on constructive KP. Higher values are better, and default settings are shown in bold.

Parameter	Value	KP50	KP100	KP200
MCTS-AHD Zheng et al. (2025)		20.015	40.252	57.423
α	0.1	20.034	40.269	57.443
	0.2	20.027	40.262	57.427
	0.4	20.029	40.264	57.437
c	1.0	20.024	40.262	57.436
	1.5	20.027	40.262	57.427
	3.0	20.029	40.269	57.438
γ	0.7	20.032	40.265	57.437
	0.9	20.027	40.262	57.427
	0.95	20.024	40.262	57.436

Generated Heuristic for TSP Constructive Solver

```
import numpy as np
from scipy.sparse.csgraph import minimum_spanning_tree

def calculate_mst_cost(unvisited_nodes, distance_matrix):
    if not unvisited_nodes:
        return 0
    sub_matrix = distance_matrix[np.ix_(unvisited_nodes,
        unvisited_nodes)]
    return minimum_spanning_tree(sub_matrix).toarray().sum()

def cluster_unvisited_nodes(unvisited_nodes, distance_matrix):
    clusters = []
    visited = set()

    for start in unvisited_nodes:
        if start in visited:
            continue

        current_cluster = [start]
        visited.add(start)
        stack = [start]

        while stack:
            node = stack.pop()
            for neighbor in unvisited_nodes:
                if neighbor not in visited and distance_matrix[node][
                    neighbor] < np.mean(distance_matrix[node][
                    unvisited_nodes]) * 0.50:
                    visited.add(neighbor)
                    current_cluster.append(neighbor)
                    stack.append(neighbor)

        clusters.append(current_cluster)

    return clusters

def select_next_node(current_node, destination_node,
    unvisited_nodes, distance_matrix):
    if not unvisited_nodes:
        return None
```

```
    min_total_cost = float('inf')
    best_candidate = None
    unvisited_count = len(unvisited_nodes)

    clusters = cluster_unvisited_nodes(unvisited_nodes,
        distance_matrix)

    for candidate in unvisited_nodes:
        remaining_nodes = unvisited_nodes.copy()
        remaining_nodes.remove(candidate)

        # Calculate path length using a nearest neighbor approach
        current_path_length = distance_matrix[current_node][candidate
            ]
        current_node_for_next = candidate

        while remaining_nodes:
            next_node = min(remaining_nodes, key=lambda n:
                distance_matrix[current_node_for_next][n])
            current_path_length += distance_matrix[
                current_node_for_next][next_node]
            current_node_for_next = next_node
            remaining_nodes.remove(next_node)

        current_path_length += distance_matrix[current_node_for_next
            ][destination_node]

        # Calculate minimum spanning tree cost for remaining nodes
        mst_cost = calculate_mst_cost(remaining_nodes,
            distance_matrix)

        # Enhanced adaptive proximity penalty based on node density,
            average distance, and current path length
        closest_distances = sorted([distance_matrix[candidate][node]
            for node in unvisited_nodes])
        proximity_penalty = 0

        if closest_distances:
            average_closest_distance = np.mean(closest_distances[:min
                (3, len(closest_distances))])
            density_weight = 1.0 if unvisited_count > 5 else 0.5
            proximity_penalty = max(0, (distance_matrix[current_node][
                candidate] - average_closest_distance) * density_weight
                )

        # Dynamic adjustment of the path length, MST cost, and
            proximity penalty weights
        total_cost = current_path_length + mst_cost * 0.45 +
            proximity_penalty * (1 + 0.5 * (unvisited_count / len(
            distance_matrix)))

        if total_cost < min_total_cost:
            min_total_cost = total_cost
            best_candidate = candidate

    return best_candidate
```

For constructive KP, the generated heuristic is implemented as a `select_next_item` function that determines which item should be added under the remaining capacity constraint. The function first filters feasible items and computes their value-to-weight ratios as a primary efficiency measure. It then performs a limited combinatorial lookahead over the top ratio-ranked candidates to identify item subsets with higher attainable value within the residual capacity, rather than relying solely on a

single greedy choice. When no high-value feasible combination is identified, a fallback mechanism samples from lower-ratio feasible items to preserve selection flexibility. The first item from the best feasible combination is returned as the next selected item.

Generated Heuristic for KP Constructive Solver

```
import numpy as np
from itertools import combinations
import random

def select_next_item(remaining_capacity, weights, values):
    if remaining_capacity <= 0 or len(weights) == 0:
        return -1 # No valid items can be selected

    # Calculate value-to-weight ratios
    value_per_weight = values / (weights + 1e-5)
    valid_indices = np.where(weights <= remaining_capacity)[0]

    if len(valid_indices) == 0:
        return -1 # No valid items can be selected

    # Step 1: Attempt to select the best single item based on value-
        to-weight ratio
    best_single_index = valid_indices[np.argmax(value_per_weight[
        valid_indices])]

    if weights[best_single_index] <= remaining_capacity:
        best_value = values[best_single_index]
    else:
        best_value = 0 # No item can be selected

    # Step 2: Explore combinations of items to maximize total value
    best_combo_value = best_value
    best_combo_indices = [best_single_index] if best_value > 0 else
        []
    # Try combinations of selected top N items based on their value-
        to-weight ratio
    sorted_valid_indices = valid_indices[
        np.argsort(value_per_weight[valid_indices])[::-1]
    ]

    for r in range(1, min(5, len(sorted_valid_indices)) + 1):
        for combo in combinations(sorted_valid_indices[:5], r):
            combo_weight = sum(weights[i] for i in combo)
            combo_value = sum(values[i] for i in combo)
            if combo_weight <= remaining_capacity and combo_value >
                best_combo_value:
                best_combo_value = combo_value
                best_combo_indices = combo

    # Step 3: Consider fallback options using lower ratio candidates
    if not best_combo_indices and remaining_capacity > 0:
        low_ratio_candidates = [
            i for i in valid_indices
            if value_per_weight[i] < value_per_weight[
                best_single_index]
        ]
        if low_ratio_candidates:
            best_combo_indices = [np.random.choice(
                low_ratio_candidates)]

    return best_combo_indices[0] if best_combo_indices else -1
```

For TSP under the ACO framework, the generated heuristic function constructs a heuristic matrix by repeatedly simulating tours, reinforcing edges that appear in high-quality solutions, and accumulating edge-wise desirability scores over multiple iterations. In each iteration, candidate tours are generated through probabilistic node transitions based on the current heuristic weights and pairwise distances, after which the best tour is refined by a 2-opt local improvement step. The refined best tour, together with a subset of elite tours from the same iteration, contributes additional weight to its traversed edges, with shorter edges in better tours receiving stronger reinforcement. After all iterations, the accumulated edge weights are row-normalized to produce the final heuristic matrix, which assigns each node a distribution of relative preferences over its outgoing edges.

Generated Heuristic for TSP Constructive Solver

```
import numpy as np

def heuristics(distance_matrix):
    problem_size = distance_matrix.shape[0]
    heuristics_matrix = np.zeros((problem_size, problem_size))

    # Parameters for the enhanced algorithm
    num_ants = 15
    max_iterations = 100
    initial_pheromone = 1.0
    evaporation_rate = 0.2
    alpha = 1.0
    beta = 3.0
    elite_fraction = 0.25
    mutation_rate = 0.2

    # Initialize pheromone levels
    pheromones = np.full((problem_size, problem_size),
        initial_pheromone)
    heuristics_weights = np.zeros((problem_size, problem_size))

    for iteration in range(max_iterations):
        all_paths = []
        for ant in range(num_ants):
            start = np.random.randint(problem_size)
            visited = [start]
            current_node = start

            for _ in range(problem_size - 1):
                probabilities = []
                for next_node in range(problem_size):
                    if next_node not in visited:
                        prob = pheromones[current_node][next_node]**alpha
                            * (1 / (distance_matrix[current_node][
                                next_node] + 1e-10))**beta
                        probabilities.append(prob)
                    else:
                        probabilities.append(0)

                probabilities = probabilities / np.sum(probabilities)
                    if np.sum(probabilities) > 0 else probabilities
                next_node = np.random.choice(range(problem_size), p=
                    probabilities)
                visited.append(next_node)
                current_node = next_node

            all_paths.append(visited)

        # Select the best tour from the current iteration
```

```
        best_local_tour = min(all_paths, key=lambda path: sum(
            distance_matrix[path[i], path[(i + 1) % problem_size]] for
             i in range(problem_size)))
        best_local_length = sum(distance_matrix[best_local_tour[i],
            best_local_tour[(i + 1) % problem_size]] for i in range(
            problem_size))

        # Apply 2-opt local search to improve the best tour
        improve = True
        while improve:
            improve = False
            for i in range(problem_size):
                for j in range(i + 1, problem_size):
                    if j != (i + 1) % problem_size and i != 0:
                        new_tour = best_local_tour[:i] + best_local_tour[
                            i:j+1][::-1] + best_local_tour[j+1:]
                        new_length = sum(distance_matrix[new_tour[k],
                            new_tour[(k + 1) % problem_size]] for k in
                            range(problem_size))
                        if new_length < best_local_length:
                            best_local_tour = new_tour
                            best_local_length = new_length
                            improve = True

        # Update pheromones based on the best tour
        contribution = 1 / (best_local_length + 1e-10)
        for i in range(problem_size):
            pheromones[best_local_tour[i]][best_local_tour[(i + 1) %
                problem_size]] += contribution * (1 / distance_matrix[
                best_local_tour[i]][best_local_tour[(i + 1) %
                problem_size]])

        # Evaporate pheromones
        pheromones *= (1 - evaporation_rate)

        # Update using elite tours
        elites = sorted(all_paths, key=lambda x: sum(distance_matrix[
            x[i], x[(i + 1) % problem_size]] for i in range(
            problem_size)))[:int(num_ants * elite_fraction)]
        for elite_tour in elites:
            elite_length = sum(distance_matrix[elite_tour[i],
                elite_tour[(i + 1) % problem_size]] for i in range(
                problem_size))
            elite_contribution = 1 / (elite_length + 1e-10)
            for i in range(problem_size):
                pheromones[elite_tour[i]][elite_tour[(i + 1) %
                    problem_size]] += elite_contribution * (1 /
                    distance_matrix[elite_tour[i]][elite_tour[(i + 1) %
                    problem_size]])

        # Update heuristics weights based on pheromones
        for i in range(problem_size):
            for j in range(problem_size):
                if i != j:
                    heuristics_weights[i][j] += pheromones[i][j] / (
                        distance_matrix[i][j] + 1e-10)

    # Normalize heuristics weights to get the heuristics matrix
    heuristics_matrix = heuristics_weights / heuristics_weights.sum(
        axis=1, keepdims=True)
    return heuristics_matrix
```

For offline BPP under the ACO framework, the generated heuristic is implemented as a `heuristics` function that constructs a heuristic matrix for guiding probabilistic item grouping. The function first sorts items in decreasing demand and applies a first-fit packing strategy to form preliminary bins. It then performs local reassignment by attempting feasible item moves across bins, which refines the initial packing structure under the capacity constraint. Finally, the heuristic matrix assigns higher desirability to item pairs that can feasibly coexist in the same bin, thereby biasing ACO toward compact and capacity-compatible packings.

Generated Heuristic for Offline BPP ACO Solver

```
import numpy as np

def heuristics(demand, capacity):
    n = demand.shape[0]
    heuristics_matrix = np.zeros((n, n))
    sorted_indices = np.argsort(-demand)
    packed_bins = []

    # Step 1: Initial greedy packing with the first-fit approach
    for index in sorted_indices:
        item_size = demand[index]
        placed = False

        # First-fit strategy with look-ahead
        for bin in packed_bins:
            current_bin_size = sum(demand[i] for i in bin)
            if current_bin_size + item_size <= capacity:
                bin.append(index)
                placed = True
                break

        # If not placed, create a new bin
        if not placed:
            packed_bins.append([index])

        # Step 2: Optimize packing by checking potential item swaps
        for i in range(len(packed_bins)):
            current_bin = packed_bins[i]
            remaining_capacity = capacity - sum(demand[j] for j in
                current_bin)

            # Only evaluate swapping if there is significant space
            if remaining_capacity > 0:
                for item in current_bin:
                    # Attempt to place item in other bins
                    for j in range(len(packed_bins)):
                        if j != i and item not in packed_bins[j]:
                            potential_fit = sum(demand[k] for k in
                                packed_bins[j]) + demand[item]
                            if potential_fit <= capacity:
                                # Remove item from the current bin and
                                    place in new bin
                                packed_bins[i].remove(item)
                                packed_bins[j].append(item)
                                break # Break after the first successful
                                    swap

    # Step 3: Create heuristics_matrix based on packed_bins
    for bin in packed_bins:
        for i in bin:
            for j in bin:
                if i != j:
```

```
                heuristics_matrix[i][j] = 1 if demand[i] + demand[j]
                    <= capacity else 0

    return heuristics_matrix
```

M EVOLUTIONARY TRAJECTORY OF GENERATED HEURISTICS

In this section, the evolutionary trajectory of DPTS is illustrated using constructive TSP as an example. All generated heuristics share the same `select_next_node` interface, while their decision rules progressively evolve from local selection toward more structured lookahead-based evaluation. This trajectory $\{h_1, h_2, \cdots, h_{10}\}$ provides a concrete example of how DPTS performs intra-lineage refinement within a fixed algorithmic interface.

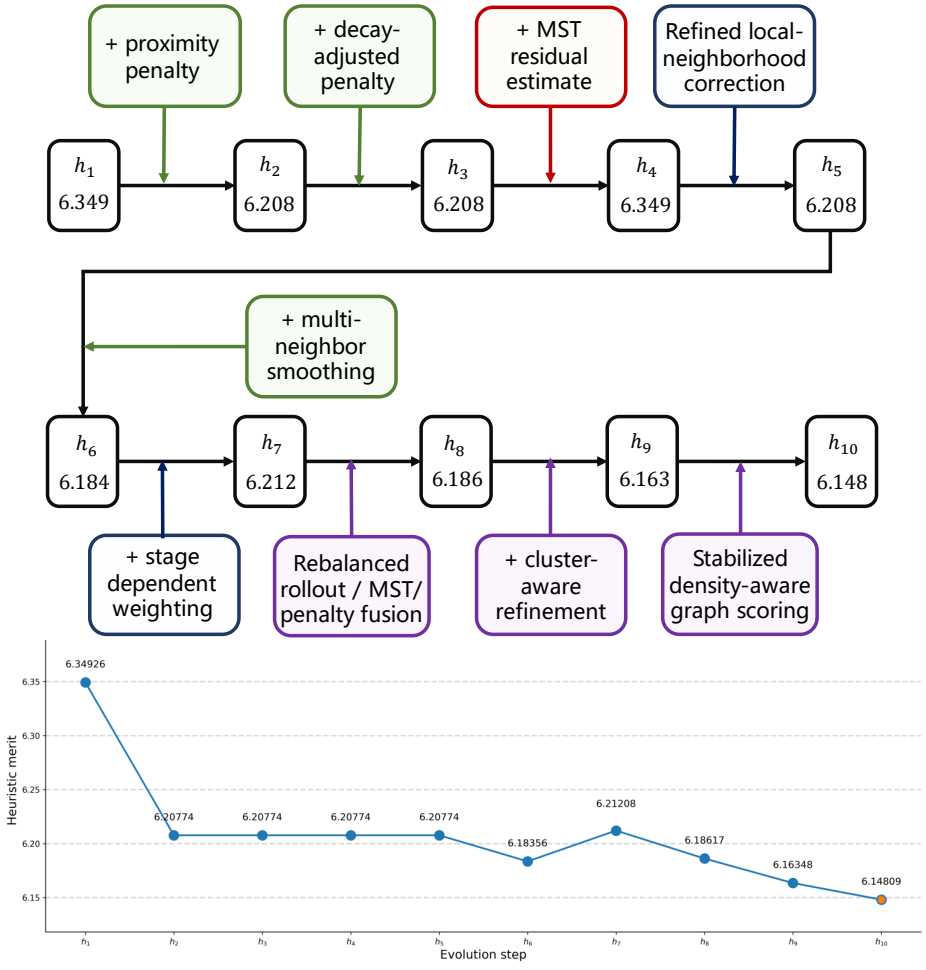


Figure 4: An evolution path of DPTS in TSP construction problem.

Heuristic h_1: rollout-based nearest-neighbor evaluation. The initial heuristic, with heuristic merit 6.34926, starts from a nearest-neighbor candidate and evaluates each unvisited node by simulating the remaining tour with a greedy nearest-neighbor rollout. Compared with a purely myopic nearest-neighbor rule, this design introduces a basic lookahead mechanism by estimating the downstream cost induced by each candidate.

Heuristic h_2: proximity-penalized rollout. The second heuristic, with heuristic merit 6.20774, augments the rollout cost with a proximity penalty based on the candidate's relative position among the remaining nodes. This modification discourages candidates that are close to the current node but poorly connected to the residual node set.

Heuristic h_3: decay-adjusted proximity penalty. The third heuristic, with heuristic merit 6.20774, further introduces a decay factor for the proximity penalty, where the penalty strength is modulated by the distance to the closest unvisited node. The correction therefore becomes adaptive to local neighborhood structure rather than being applied with a fixed strength.

Heuristic h_4: MST-enhanced residual estimation. The fourth heuristic, with heuristic merit 6.20774, incorporates a minimum spanning tree cost over the remaining nodes as a graph-structured residual estimate. The scoring rule combines greedy rollout cost, MST-based lower-bound information, and local proximity correction, yielding a more global candidate evaluation.

Heuristic h_5: nearest-neighborhood proximity refinement. The fifth heuristic, with heuristic merit 6.20774, retains the MST residual term but refines the proximity penalty using the two closest unvisited nodes. This shifts the correction from global average distance to local neighborhood connectivity, emphasizing whether a candidate is well embedded in the nearby residual graph.

Heuristic h_6: multi-neighbor proximity smoothing. The sixth heuristic, with heuristic merit 6.18356, extends the local correction to multiple nearest neighbors, producing a smoother estimate of local density. Compared with the previous two-neighbor penalty, this version reduces sensitivity to individual nearest-node fluctuations.

Heuristic h_7: stage-dependent penalty weighting. The seventh heuristic, with heuristic merit 6.21208, introduces a dynamic proximity weight conditioned on the number of remaining nodes. This allows different correction strengths at different construction stages, reflecting the changing role of local density during early and late tour construction.

Heuristic h_8: weighted fusion of rollout, MST, and proximity terms. The eighth heuristic, with heuristic merit 6.18617, explicitly rebalances the scoring components by down-weighting the MST residual term and amplifying the stage-dependent proximity penalty. This prevents the graph lower bound from dominating candidate evaluation and yields a more balanced integration of rollout cost, structural residual cost, and local density information.

Heuristic h_9: cluster-aware structural refinement. The ninth heuristic, with heuristic merit 6.16348, introduces a clustering procedure over unvisited nodes and strengthens the density-aware proximity correction using both average and maximum local distances. This refinement moves beyond pairwise distance scoring toward modeling the spatial organization of the residual graph.

Heuristic h_{10}: stabilized density-aware MST lookahead. The final heuristic, with heuristic merit 6.14809, consolidates the previous components into a stabilized scoring rule. It retains greedy rollout, MST residual estimation, and density-aware proximity correction, while adopting more conservative MST weighting and a clearer density-dependent penalty. The resulting heuristic represents a mature graph-structured lookahead strategy for constructive TSP.

Overall, the ten heuristics follow a coherent evolutionary path within the same constructive TSP interface. They belong to the same rollout-based graph-structured constructive heuristic lineage and share a common decision backbone that evaluates each candidate by estimating the downstream tour cost after selecting it as the next city. As summarized in Figure 4, their differences mainly arise from progressively refined correction terms, including proximity penalties, MST-based residual estimates, dynamic weighting, and density- or cluster-aware neighborhood information. The progression from local nearest-neighbor selection to rollout-based lookahead, MST-based residual estimation, and finally density-aware graph-structured scoring shows that DPTS performs systematic intra-lineage refinement rather than generating unrelated heuristic variants. The corresponding heuristic merit also improves from 6.34926 to 6.14809, indicating that the evolutionary process progressively enhances the quality of the shared algorithmic backbone.