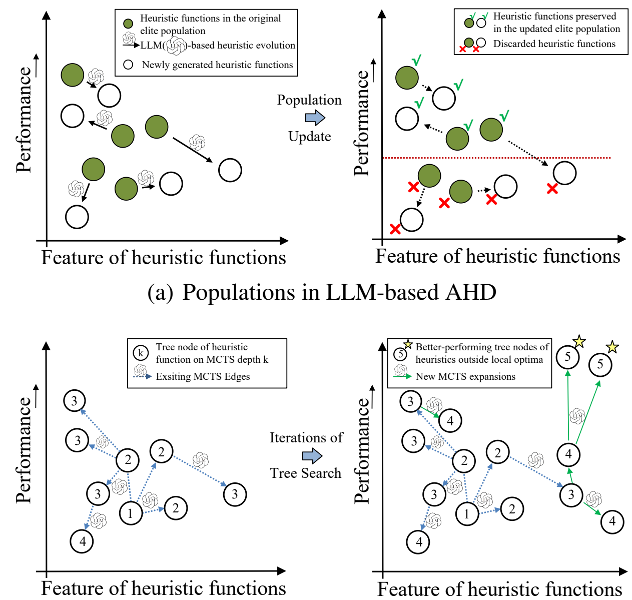
# Monte Carlo Tree Search for Comprehensive Exploration in LLM-Based Automatic Heuristic Design

**Zhi Zheng** [1] **Zhuoliang Xie** [2] **Zhenkun Wang** [2] **Bryan Hooi** [1]

## Abstract

Handcrafting heuristics for solving complex planning tasks (e.g., NP-hard combinatorial optimization (CO) problems) is a common practice but requires extensive domain knowledge. Recently, Large Language Model (LLM)-based automatic heuristics design (AHD) methods have shown promise in generating high-quality heuristics without manual intervention. Existing LLM-based AHD methods employ a population to maintain a fixed number of top-performing LLM-generated heuristics and introduce evolutionary computation (EC) to enhance the population iteratively. However, the population-based procedure brings greedy properties, often resulting in convergence to local optima. Instead, to more comprehensively explore the space of heuristics, we propose using Monte Carlo Tree Search (MCTS) for LLM-based heuristic evolution while preserving all LLM-generated heuristics in a tree structure. With a novel thought-alignment process and an exploration-decay technique, the proposed MCTS-AHD method delivers significantly higher-quality heuristics on various complex tasks. Our code is available at https://github.com/zz1358m/MCTS-AHD-master.




(a) Populations in LLM-based AHD

(b) MCTS for higher-quality LLM-based AHD (Ours)

*Figure 1.* The generally adopted population (a) in existing LLM-based AHD methods (Liu et al., 2024b; Ye et al., 2024a) will discard low-quality heuristics (under the red dashed line in (a)) directly, thus falling into local optima. MCTS allows optimizing low-quality heuristics, which is advantageous in maintaining diversity and exploring heuristic functions with different features.

## 1. Introduction

Manually designed heuristics are promising in addressing complex planning and combinatorial optimization (CO) tasks (Desale et al., 2015). They are widely applied for various real-world applications, including traffic control (He et al., 2011), job scheduling (Rajendran, 1993), and robotics (Tan et al., 2021). However, manually designed heuristics often contain complex workflows and sophisticated parameter settings, whose design is labor-intensive and necessitates extensive task-specific expert knowledge. To achieve easier heuristic design across various tasks, the concept of Automatic Heuristic Design (AHD) (Burke et al., 2013) (also known as Hyper-Heuristics (Ye et al., 2024a)) has attracted extensive attention. AHD aims to find the best-performing heuristic algorithm among valid ones. Genetic Programming (GP) (Langdon & Poli, 2013) is generally adopted for AHD, with GP-based AHD methods introducing a series of mutation operators to gradually update heuristic algorithms (Duflo et al., 2019; Zhao et al., 2023). Nevertheless, the effectiveness of GP-based methods still relies on human definitions of permissible operators (Liu et al., 2024b), which poses additional implementation difficulties.

In recent years, large language models (LLMs) have demonstrated remarkable effectiveness in various fields (Hadi et al., 2023; 2024; Naveed et al., 2023). Leveraging their superb

[1]School of Computing, National University of Singapore, Singapore [2]School of System Design and Intelligent Manufacturing, Southern University of Science and Technology, Shenzhen, China. Correspondence to: Bryan Hooi <bhooi@comp.nus.edu.sg>.

reasoning abilities and abundant inherent knowledge, LLM-based AHD methods (Liu et al., 2024c; Yu & Liu, 2024) are proposed to automatically design high-quality heuristics for various complex tasks. Funsearch (Romera-Paredes et al., 2024) first applies LLMs to AHD by designing a population-based evolutionary computation (EC) procedure. In task solving, there are several established general frameworks for implementing heuristics, e.g., greedy solving frameworks or frameworks with search-based ideas for CO problems. Funsearch concentrates on designing key heuristic functions within a predefined general framework, rather than developing algorithms from scratch. It maintains a population of outstanding heuristic functions based on their performances on an evaluation dataset and iteratively prompts LLMs to generate new heuristic functions taking the existing ones as starting points. Subsequent LLM-based AHD methods propose novel components based on this population-based EC framework. EoH (Liu et al., 2024b) develop several effective prompt strategies, guiding LLMs to generate effective heuristic functions. ReEvo (Ye et al., 2024a) introduces the reflection mechanism (Shinn et al., 2024) to enhance the reasoning of LLMs among heuristic function samples. HSEvo (Dat et al., 2024) introduces diversity metrics and harmony search (Shi et al., 2013) to improve the diversity of heuristic functions without losing effectiveness.

Although these population-based methods can achieve fast convergence by focusing on top-performing heuristic functions, the population structure will overlook the exploration of lower-quality but potentially promising heuristics (as shown in Figure 1), probably leading the evolution of heuristic functions to local optima. Funsearch (Romera-Paredes et al., 2024) and HSEvo (Dat et al., 2024) employ multiple-population structures (Cantú-Paz et al., 1998) and diversity metrics, respectively, to improve the diversity of populations. These components can reduce the probability of quick convergence into local optima but still fall short in exploring the complex space of heuristics.

To comprehensively explore the space of heuristic functions and maintain the focus on better-performing ones, this paper proposes the first tree search method for LLM-based AHD, MCTS-AHD, which employs a tree structure to organize all heuristic functions and incorporates the Monte Carlo Tree Search (MCTS) algorithm with a progressive widening technique (Coulom, 2007) for heuristic evolution. As a structured data structure, the MCTS tree records the evolution history of heuristics, thus providing organized samples for heuristic evolution and LLMs' reasoning. Moreover, we propose an exploration-decay technique, which linearly declines the exploration of MCTS branches as the evolution of heuristic functions progresses. We also introduce a novel thought-alignment procedure for more precise heuristic generation. In experiments, we implement MCTS-AHD to design heuristic functions within several general frameworks for a wide range of NP-hard (Ausiello et al., 2012) CO problems and a Bayesian Optimization (BO)-related planning task. MCTS-AHD can achieve significantly higher-quality heuristics than developed solving algorithms and existing LLM-based AHD methods.

## 2. Preliminary

### 2.1. Definition: AHD & LLM-based AHD

**AHD:** For a given task $P$ (e.g., CO problems), AHD methods search for the best-performing heuristic $h^*$ within a heuristic space $H$ as follows:

$$h^* = \arg\max_{h \in H} g(h). \tag{1}$$

The heuristic space $H$ contains all feasible heuristics for tasks $P$. In solving a task $P$, executing a heuristic $h \in H$ projects the distribution of task inputs (i.e., instances) $I_P$ into a set of solutions $S_P$, i.e., $h : I_P \to S_P$. For example, heuristics for an NP-hard CO task, Traveling Salesman Problem (TSP), project city coordinates (TSP instances) to travel tours (solutions). The function $g(\cdot)$ is a performance function for heuristics $g : H \to \mathbb{R}$. For a CO task $P$ minimizing an objective function $f : S_P \to \mathbb{R}$, the performance function $g$ of AHD is calculated from the expectation of the objective function values for solutions in $S_P$ as follows:

$$g(h) = \mathbb{E}_{\boldsymbol{ins} \in I_P} \left[ \mathbb{E}_{\boldsymbol{s} \in h(\boldsymbol{ins})} [-f(\boldsymbol{s})] \right], \tag{2}$$

where $\boldsymbol{s} \in h(\boldsymbol{ins})$ represents a feasible CO solution by running a heuristic $h$ for an instance $\boldsymbol{ins} \in I_P$. In practice, we can estimate $g(h)$ by running $h$ on a task-specific evaluation dataset $D$ (Zhao et al., 2023). Heuristics in $H$ belong to a wide range of general frameworks. So to concentrate the evaluation budgets on key parts, AHD methods tend to predefine a general framework and only design a **key heuristic function** with specified inputs and outputs within the predefined given general framework. For example, heuristics for solving TSP under the step-by-step construction framework will build the TSP tour one city after another. AHD methods within this framework will design a key heuristic function to choose the next city based on a partial tour of the city visited before. For simplicity of representation, we still denote the function to be designed as $h$.

**LLM-based AHD:** LLM-based AHD introduces LLMs into the search process for the optimal heuristic function $h^*$ within pre-defined general frameworks. Existing LLM-based AHD methods (Liu et al., 2024b; Ye et al., 2024a) maintain a population of $M$ heuristic functions $\{h_1, \ldots, h_M\}$ and employ EC to iteratively update the population. The mutation or crossover operators in EC for heuristic generation are prompted LLMs. Newly generated heuristic functions will be evaluated on an evaluation dataset $D$ and only algorithms (with performance function $g(h)$) that

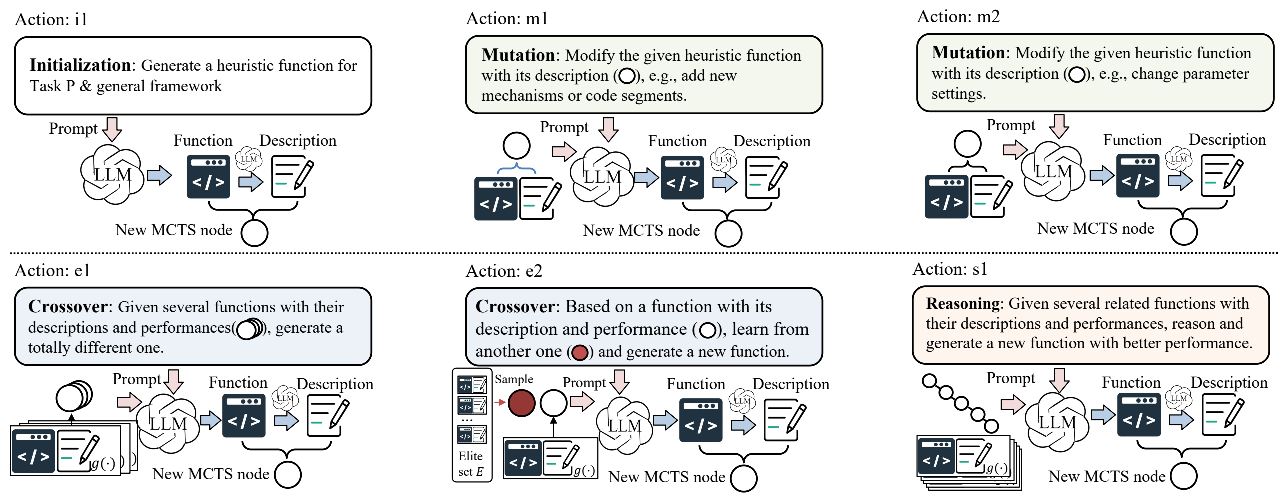



*Figure 2.* Actions for heuristic function evolutions in MCTS-AHD, including an initialization action i1, two mutation actions m1 & m2 for detail designs, two crossover actions e1 & e2, and a novel tree-structure-specific reasoning action s1 for optimization.

exceed the worst-performing algorithm in the original population can be retained, i.e., $g(h) > \min_{i\in\{1,\dots,M\}} g(h_i)$. This property makes it difficult for population-based methods to take a worse-before-better strategy for the optimal heuristic function.

### 2.2. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) (Świechowski et al., 2023) is a decision-making algorithm widely used in games (Silver et al., 2016) and complex decision-making tasks (Fu et al., 2021). Recent studies also verify the power of MCTS to enhance the multi-hop reasoning ability of LLMs (Feng et al., 2023). Each node $n_c$ in the MCTS tree represents a decision state, and MCTS will recurrently develop the most potential state as judged by a UCT algorithm (Kocsis & Szepesvári, 2006). Each MCTS node $n_c$ records a quality value $Q(n_c)$ and a visit count $N(n_c)$. Starting from an initial root node $n_r$, MCTS gradually builds the MCTS tree to explore the entire decision state space. Each round of MCTS consists of four stages:

**Selection**: The selection stage identifies the most potential MCTS tree node for subsequent node expansions. From the root node $n_r$, the selection stage iteratively selects the child node with the largest UCT value until reaching a leaf node. For the current node $n_c$, the UCT values for its child nodes $c \in \text{Children}(n_c)$ are calculated as follows:

$$\text{UCT}(c) = \left(Q(c) + \lambda \cdot \sqrt{\frac{\ln(N(n_c)+1)}{N(c)}}\right). \tag{3}$$

**Expansion**: The expansion stage obtains multiple child nodes from the current node $n_c$ by sampling several actions from the decision state of $n_c$ among all possible options.

**Simulation**: The simulation stage evaluates all newly expanded leaf nodes for their quality value $Q(\cdot)$. In particular, chess games generally adopt a ”rollout” simulation process to evaluate leaf nodes (Silver et al., 2016).

**Backpropagation**: The backpropagation process uses the results of simulations to update the values of $Q(\cdot), N(\cdot)$ for nodes on the tree path from $n_c$ to the root $n_r$.

After several iterations of MCTS, the MCTS tree will contain nodes with high-value decision states for tasks.

**Progressive Widening:** Conventional MCTS is not suitable for tasks with extensive action options or dynamically changing environments (Lee et al., 2020b). Progressive widening (Coulom, 2007) is a technique designed for MCTS to fit such tasks. It gradually adds new child nodes to non-leaf nodes as their visit count value $N(\cdot)$ increases. Formally, for any node $n_c$ with child nodes $\text{children}(n_c)$, a new child node will be added every time the following is satisfied:

$$\lfloor N(n)^{\alpha} \rfloor \geq |\text{children}(n_c)|, \tag{4}$$

where $|\text{children}(n_c)|$ is the number of child nodes of $n_c$.

## 3. MCTS-AHD

Population-based LLM-based AHD faces challenges in escaping local optima and exploring complex heuristic spaces. To comprehensively explore the heuristic space, this paper proposes a novel method MCTS-AHD. It retains all LLM-generated heuristic functions in an MCTS tree and employs MCTS with the progressive widening technique instead of population-based EC for heuristic evolution. The MCTS root node $n_r$ is a meaningless virtual node, and each of the other nodes represents an executable Python code implementation of a heuristic function $h \in H$ and its lin-

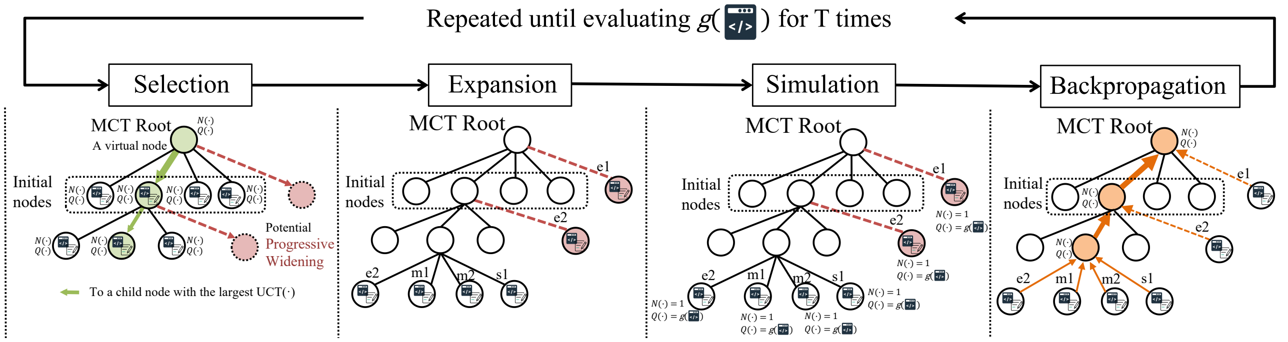



*Figure 3.* The MCTS process in MCTS-AHD contains four stages, i.e., selection, expansion, simulation, and backpropagation. MCTS-AHD simulates the quality value of each node as the performance function values of their heuristics and the MCTS will terminate after total $T$ performance evaluations. MCTS-AHD introduces the progressive widening technique to better crossover original heuristic functions with continuously generated new ones. It conducts the crossover actions e1 for the root node and action e2 for other nodes.

guistic description. For MCTS expansion, MCTS-AHD prompts LLMs in several ways (e.g., mutation, crossover, and reasoning) to simulate operators on heuristic functions. Moreover, to enhance the exploration of the heuristic space $H$ in the early stages of MCTS and ensure convergence in the later stages, MCTS-AHD presents an exploration-decay technique to linearly decay the exploration factor in UCT.

### 3.1. LLM-based actions in MCTS-AHD

In employing LLMs to generate new heuristics from existing ones (Liu et al., 2024b), LLM-based AHD methods present several efficient prompt strategies (e.g., heuristic mutation and crossover). Unlike population data structures, the MCTS tree in MCTS-AHD can record the relationships among all the generated heuristics. Leveraging this advantage, as shown in Figure 2, with different prompt strategies, MCTS-AHD presents a novel tree-structure-specific set of actions for LLM-based heuristic evolution, including i1, e1, e2, m1, m2, and s1. Detailed prompts for these actions are provided in Appendix E.1. As a commonality, all prompts contain descriptions of the task $P$, the pre-defined general framework, the inputs and outputs of the key heuristic function, and their meanings. Prompts for actions except action i1 also contain samples of existing heuristic functions. For each action, LLMs are supposed to output the Python code of a heuristic function and its linguistic description.

*Initial Action i1*: As an action for initialization, the action i1 aims to directly generate a heuristic function and its corresponding description from scratch by LLMs.

*Mutation Action m1 & m2*: MCTS-AHD contains two mutation actions, m1 & m2, to attempt more detailed designs within the original function workflow. Based on the inputted heuristic function, the action m1 prompts LLMs to introduce new mechanisms and formulas, and the action m2 prompts LLMs to change the parameter settings.

*Crossover Action e1*: To explore heuristic functions with new workflows, we employ a crossover action e1 to generate a new heuristic function that varies from the multiple existing ones. These heuristic functions are inputted to LLMs with their performances $g(\cdot)$ and descriptions.

*Crossover Action e2*: The crossover action e2 prompts LLMs with a parent heuristic function, a reference heuristic function, and their performances. LLMs are required to identify the beneficial designs in the reference one and design a better heuristic function based on the parent one. The reference is sampled from a **elite heuristic function set** $E$ consisting of heuristic functions with the top 10 performances $g(\cdot)$.

*Tree-path Reasoning Action s1*: The MCTS tree paths from the root $n_r$ to leaf nodes well record the history of heuristic function evolutions. So, utilizing this characteristic, MCTS-AHD presents an action s1 to analyze the unique heuristic function samples from an MCTS tree path up to the root $n_r$, identify advantageous designs in these samples, and generate a better heuristic function.

**Generation of Function Descriptions: The Thought-Alignment Process**. Generating descriptions of heuristic functions can enhance the reasoning of LLMs (Liu et al., 2024b), where EoH prompts LLMs to generate a function description before outputting its code. However, due to LLM hallucination (Huang et al., 2023), such a procedure could lead to an uncorrelation between codes and descriptions. For correlated and detailed descriptions, MCTS-AHD proposes a thought-alignment process that summarizes descriptions after the code generations. Therefore, performing all the MCTS-AHD actions calls LLMs twice. In the first call, LLMs generate a Python implementation of a heuristic function with the action prompts. Then, LLMs are prompted with a thought-alignment prompt (refer to Appendix E.2) for the description of the implementation in up to three sentences. The second LLM call is significantly shorter than the first one, so it will not cause a severe increase in execution time and token cost.

### 3.2. MCTS settings

Figure 3 displays the MCTS process in MCTS-AHD. It first generates $N_I$ initial nodes representing different heuristic functions and links them to a virtual root node $n_r$ that does not represent any heuristic functions. Subsequently, similar to the regular MCTS introduced in Section2.2, MCTS-AHD repeatedly performs the selection, expansion, simulation, and backpropagation stages as follows until the number of evaluations of heuristic functions approaches a limit $T$:

*Selection:* In the selection process of MCTS-AHD, MCTS-AHD normalizes the quality value $Q(\cdot)$ to enhance the homogeneity of different tasks in calculating UCT value for child nodes $c \in \text{Children}(n_c)$ of a node $n_c$ as follows:

$$\text{UCT}(c) = \left(\frac{Q(c) - q_{min}}{q_{max} - q_{min}} + \lambda \cdot \sqrt{\frac{\ln(N(n_c)+1)}{N(c)}}\right), \tag{5}$$

where $q_{max}$ and $q_{min}$ are the upper and lower limits of quality values $Q(\cdot)$ that ever encountered in MCTS, respectively. From the root $n_r$, MCTS iteratively selects a child node with the largest UCT value until reaching a leaf node.

*Expansion*: For the leaf node selected in the selection stage, the expansion stage prompts LLMs with actions e2, m1, m2, and s1 to build its child nodes. To attempt various detail designs, in a single expansion stage, MCTS-AHD generates $k$ child nodes with the actions m1 and m2, one with e2 and s1, respectively ($2k+2$ child nodes in total).

*Simulation*: Then, MCTS-AHD evaluates these newly generated heuristic functions on the evaluation dataset $D$ for their performances $g(\cdot)$. After evaluations, each leaf node $n_l$ with heuristic function $h$ is assigned with the quality value $Q(n_l) = g(h)$ and the visit count $N(n_l) = 1$. Also, the elite set $E$ for the action e2, $q_{max}$, and $q_{min}$ are updated.

*Backpropagation*: The backpropagation process updates the quality values and the visit counts in MCTS as follows:

$$\begin{aligned} Q(n_c) &\leftarrow \max_{c \in \text{Children}(n_c)} Q(c), \\ N(n_c) &\leftarrow \sum_{c \in \text{Children}(n_c)} N(c). \end{aligned} \tag{6}$$

**Progressive Widening Settings**. Since the elite heuristic function set $E$ for the action e2 updates gradually as the search progresses. MCTS-AHD introduces progressive widening to enable the re-exploration of non-leaf nodes, especially nodes with higher visit counts. The progressive widening process will occur when the condition in Eq.(4) is satisfied and $\alpha = 0.5$ in this paper. We call an **action e1** for a new child node of the root $n_r$ when the root node $n_r$ qualifies the condition in Eq.(4), and we call **action e2** when a deeper node qualifies. The heuristic functions inputted to the action e1 are uniformly selected from 2 to 5 different subtrees of MCTS. We implement the progressive widening process for new nodes during the selection stage, and then these nodes will also be processed in the simulation and backpropagation stages.

Eventually, MCTS-AHD will output a heuristic function with the highest performance value throughout MCTS.

### 3.3. Exploration-Decay

The setting of the exploration factor $\lambda$ in Eq.(5) determines the preferences of MCTS on exploration or exploitation (Browne et al., 2012). A larger $\lambda$ promotes the exploration of temporarily inferior nodes, while a smaller $\lambda$ stimulates concentration on nodes with higher quality values $Q(\cdot)$. To facilitate a more comprehensive exploration in the early stage of MCTS and ensure convergence in the later stages, MCTS-AHD presents an exploration-decay technique, linearly decaying the exploration factor $\lambda$ in Eq.(5) as follows:

$$\lambda = \lambda_0 * \frac{T-t}{T}. \tag{7}$$

Although having a task-specific setting $\lambda_0$ is helpful for high-quality heuristics, MCTS-AHD strives to keep the generalization ability, so we set $\lambda_0 = 0.1$ for all tasks.

## 4. Experiments

We evaluate the proposed MCTS on various complex tasks, including NP-hard CO problems and a Cost-aware Acquisition Function (CAF) design task for BO (Yao et al., 2024c). The definitions of these tasks are given in Appendix B. We implement MCTS-AHD to design key functions of a wide range of general frameworks (detailed in Appendix C) for these tasks, e.g., step-by-step construction, Ant Colony Optimization (ACO), Guided Local Search (GLS), and BO.

**Settings.** For all experiments in this section, we set the number of initial tree nodes $N_I = 4$, $\lambda_0 = 0.1$, and $\alpha = 0.5$. The maximum running time of each heuristic function on the evaluation dataset $D$ is confined to 60 seconds. The composition of evaluation datasets $D$ for each task is detailed in Appendix D as well as the parameter settings of general frameworks. Valuable LLM-based AHD methods should be flexible for different pre-trained LLMs, so this paper includes both *GPT-3.5-turbo* and *GPT-4o-mini*.

**Baselines.** To verify the ability of heuristics designed by MCTS-AHD, we introduce four types of heuristics as baselines: **(a)** Manually designed heuristics, e.g., Nearest-greedy (Rosenkrantz et al., 1977), ACO (Dorigo et al., 2006), EI (Mockus, 1974). **(b)** Traditional AHD method: GHPP

*Table 1.* Designing heuristics with the step-by-step construction framework for TSP and KP. We evaluate methods on 6 test sets with 1,000 instances and the scales of in-domain (i.i.d. to the evaluation dataset $D$) test sets are underlined. Since LLM-based AHD has no guarantees for generalization ability, the effect on in-domain datasets is more important. Optimal for TSP is obtained by LKH (Lin & Kernighan, 1973) and Optimal for KP is the result of OR-Tools. Each LLM-based AHD method is run three times and we report the average performances. The best-performing method for each LLM model is shaded, and each test set's overall best result is in bold.

| Task | TSP | | | | | | KP | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N= | $N$=50 | | $N$=100 | | $N$=200 | | $N$=50, $W$=12.5 | | $N$=100, $W$=25 | | $N$=200, $W$=25 | |
| Methods | Obj.↓ | Gap | Obj.↓ | Gap | Obj.↓ | Gap | Obj.↑ | Gap | Obj.↑ | Gap | Obj.↑ | Gap |
| Optimal | 5.675 | - | 7.768 | - | 10.659 | - | 20.037 | - | 40.271 | - | 57.448 | - |
| Greedy Construct | 6.959 | 22.62% | 9.706 | 24.94% | 13.461 | 26.29% | 19.985 | 0.26% | 40.225 | 0.12% | 57.395 | 0.09% |
| POMO | **5.697** | **0.39%** | **8.001** | **3.01%** | 12.897 | 20.45% | 19.612 | 2.12% | 39.676 | 1.48% | 57.271 | 0.09% |
| LLM-based AHD: *GPT-3.5-turbo* | | | | | | | | | | | | |
| Funsearch | 6.683 | 17.75% | 9.240 | 18.95% | 12.808 | 19.61% | 19.985 | 0.26% | 40.225 | 0.12% | 57.395 | 0.09% |
| EoH | 6.390 | 12.59% | 8.930 | 14.96% | 12.538 | 17.63% | 19.994 | 0.21% | 40.231 | 0.10% | 57.400 | 0.08% |
| MCTS-AHD(Ours) | 6.346 | 11.82% | 8.861 | 14.08% | 12.418 | 16.51% | 19.997 | 0.20% | 40.233 | 0.09% | 57.393 | 0.10% |
| LLM-based AHD: *GPT-4o-mini* | | | | | | | | | | | | |
| Funsearch | 6.357 | 12.00% | 8.850 | 13.93% | 12.372 | 15.54% | 19.988 | 0.24% | 40.227 | 0.11% | 57.398 | 0.09% |
| EoH | 6.394 | 12.67% | 8.894 | 14.49% | 12.437 | 16.68% | 19.993 | 0.22% | 40.231 | 0.10% | 57.399 | 0.09% |
| MCTS-AHD(Ours) | 6.225 | 9.69% | 8.684 | 11.79% | **12.120** | **13.71%** | **20.015** | **0.11%** | **40.252** | **0.05%** | **57.423** | **0.04%** |

(Duflo et al., 2019) **(c)** Neural Combinatorial Optimization (NCO) methods under the same general frameworks, e.g. POMO (Kwon et al., 2020) and DeepACO (Ye et al., 2024b). **(d)** Existing LMM-based AHD methods: Funsearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024b), ReEvo (Ye et al., 2024a), and the most recent work HSEvo (Dat et al., 2024). Funsearch, ReEvo, and HSEvo design heuristics from a manually designed seed function, and we provide the same low-quality seed function for each design scenario to avoid providing too much external information. Instead, running EoH and MCTS-AHD does not require manually setting a seed function to initiate the heuristic evolution, so both methods demonstrate better applicability.

We design heuristics with the proposed MCTS-AHD and all other LLM-based AHD baselines on a single Intel(R) i7-12700 CPU. Following similar settings of EoH, for almost all tasks, we set the evaluation budget $T$ for LLM-based AHD methods on the evaluation dataset $D$ to $T = 1,000$. In designing heuristics for each task with each general framework, we conduct three independent runs for each LLM-based AHD method to reduce statistical biases. To verify the significant advantages of MCTS-AHD, we perform more runs on some tasks to obtain the p-value in Appendix F.3. In designing heuristics with the step-by-step construction framework for the 0-1 Knapsack Problem (KP), executing MCTS-AHD with $T$=1,000 takes approximately three hours, 1M input tokens, 0.2M output tokens, about 0.3$ with *GPT-4o-mini*. Compared to LLM-based AHD baselines, there is no significant efficiency degradation or cost improvement.

### 4.1. MCTS-AHD for NP-hard CO Problems

As commonly recognized complex tasks (Korte et al., 2011), we first evaluate the proposed MCTS-AHD on NP-hard CO problems, including TSP, KP, Capacitated Vehicle Routing Problem (CVRP), Multiple Knapsack Problem (MKP), Bin-Packing Problem (BPP) with both online and offline settings (online BPP & offline BPP), and Admissible Set Problem (ASP). We apply MCTS-AHD to automatically design heuristics with several general frameworks to obtain high-quality heuristics of these NP-hard CO problems, including step-by-step construction, ACO, and GLS (shown in Appendix F.2).

**Step-by-Step Construction Framework.** The step-by-step construction framework (also known as the constructive heuristic framework) is simple but flexible for task solving, which constructs nodes in feasible solutions one by one (Asani et al., 2023). It is the most common framework adopted in NCO methods (Vinyals et al., 2015; Bello et al., 2016) and is also used as a popular test scenario for LLM-based AHD methods. We use MCTS-AHD to design heuristics with the step-by-step construction framework for TSP, KP, online BPP, and ASP (shown in Appendix F.1).

*TSP & KP*. We first evaluate MCTS-AHD by designing high-quality TSP and KP heuristic functions for step-by-step construction frameworks. The function should select the next TSP node or the KP item to join based on the solving state (e.g., currently selected and remaining TSP nodes or KP items) as input. This function will be executed recursively until a complete feasible solution is constructed. To design TSP heuristics, the evaluation dataset $D$ for LLM-based AHD methods contains 64 50-node TSP ($N$=50) instances. For KP, it contains 64 100-item KP instances with capacity $W = 25$. Table 1 shows the performance of baseline heuristics and LLM-based AHD methods. The Greedy Construct baseline is the Nearest-greedy heuristic algorithm for TSP and it constructs KP solutions based on the ratio of item values and weights. MCTS-AHD exhibits significant advantages on almost all test scales, surpassing manually designed heuristics and LLM-based AHD methods EoH and Fun-

*Table 2.* Designing heuristics with the ACO general framework for solving TSP, CVRP, MKP, and offline BPP. Each test set contains 64 instances and LLM-based AHD methods' performances are averaged over three runs.

| | TSP | | | | CVRP | | | | MKP | | | | Offline BPP | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Test sets | $N$=50 | | $N$=100 | | $N$=50, $C$=50 | | $N$=100, $C$=50 | | $N$=100,$m$=5 | | $N$=200,$m$=5 | | $N$=500,$C$=150 | | $N$=1,000,$C$=150 | |
| Methods | Obj.↓ | Gap | Obj.↓ | Gap | Obj.↓ | Gap | Obj.↓ | Gap | Obj.↑ | Gap | Obj.↑ | Gap | Obj.↓ | Gap | Obj.↓ | Gap |
| ACO | 5.992 | 3.28% | 8.948 | 9.40% | 11.355 | 27.77% | 18.778 | 25.76% | 22.738 | 2.28% | 40.672 | 4.30% | 208.828 | 2.81% | 417.938 | 3.15% |
| DeepACO | 5.842 | 0.71% | 8.282 | 1.26% | **8.888** | **0.00%** | **14.932** | **0.00%** | 23.093 | 0.75% | 41.988 | 1.20% | **203.125** | **0.00%** | **405.172** | **0.00%** |
| LLM-based AHD: *GPT-4o-mini* | | | | | | | | | | | | | | | | |
| EoH | 5.828 | 0.45% | 8.263 | 1.03% | 9.359 | 5.31% | 15.681 | 5.02% | 23.139 | 0.56% | 41.994 | 1.19% | 204.646 | 0.75% | 408.599 | 0.85% |
| ReEvo | 5.856 | 0.42% | 8.340 | 0.75% | 9.327 | 4.94% | 16.092 | 7.77% | 23.245 | 0.10% | 42.416 | 0.19% | 206.693 | 1.76% | 413.510 | 2.06% |
| MCTS-AHD(Ours) | **5.801** | **0.00%** | **8.179** | **0.00%** | 9.286 | 4.48% | 15.782 | 5.70% | **23.269** | **0.00%** | **42.498** | **0.00%** | 204.094 | 0.48% | 407.323 | 0.53% |

search. Moreover, compared to an advanced NCO method POMO which requires task-specific training, MCTS-AHD can design better heuristics in 200-node TSP and KP test sets, exhibiting its power to solve NP-hard CO problems.

*Online BPP*. As another widely considered NP-hard CO problem, online BPP is the online variant of BPP. It allows only immediate bin packing decisions once a new item is received. It is generally adopted as a common evaluation task for LLM-based AHD methods. We follow (Liu et al., 2024b) to generate WeiBull BPP instances (Castiñeiras et al., 2012) and use four WeiBull instances with different scales as the evaluation dataset $D$. As shown in Table 3, online BPP heuristics designed by MCTS-AHD demonstrate superior average performances across six scales.

*Table 3.* Design heuristics with the step-by-step construction framework for solving online BPP. The table shows the performance gaps of heuristics to the lower bound and each LLM-based AHD method is run three times for the average gaps. Each test set contains five WeiBull BPP instances and we underline the four (in-domain) scales contained in $D$. The test scales are abbreviated, e.g., 1k_100 represents 1,000 items and capacity $W = 100$.

| Online BPP | | | | | | | |
|---|---|---|---|---|---|---|---|
| Test sets | 1k_100 | 1k_500 | 5k_100 | 5k_500 | 10k_100 | 10k_500 | Avg. |
| Best Fit | 4.77% | 0.25% | 4.31% | 0.55% | 4.05% | 0.47% | 2.40% |
| First Fit | 5.02% | 0.25% | 4.65% | 0.55% | 4.36% | 0.50% | 2.56% |
| LLM-based AHD: *GPT-4o-mini* | | | | | | | |
| Funsearch | 2.45% | 0.66% | 1.30% | 0.25% | 1.05% | 0.21% | 0.99% |
| EoH | 2.69% | 0.25% | 1.63% | 0.53% | 1.47% | 0.45% | 1.17% |
| ReEvo | 3.94% | 0.50% | 2.72% | 0.40% | 2.39% | 0.31% | 1.71% |
| HSEvo | 2.64% | 1.07% | 1.43% | 0.32% | 1.13% | 0.21% | 1.13% |
| MCTS-AHD | 2.45% | 0.50% | 1.06% | 0.32% | 0.74% | 0.26% | **0.89%** |

**Ant Colony Optimization Framework**. The ACO is an optimization algorithm inspired by the foraging behavior of ants, which contains a heuristic matrix and implements a path selection mechanism to solve combinatorial optimization problems by simulating the transfer of pheromones between ants (Dorigo et al., 2006; Kim et al., 2024). LLM-based AHD can design a generation function of the heuristic matrix, thereby transforming ACO into a framework and applying it to a variety of tasks. MCTS-AHD follows Ye et al. (2024a) in designing heuristics based on the ACO framework for four CO problems: TSP, CVRP, MKP, and offline BPP. The design results with *GPT-4o-mini* as LLMs are shown in Table 2, where MCTS-AHD exhibits significant leads to EoH and ReEvo in all in-domain test sets across four CO problems and three out-of-domain test sets. Moreover, the proposed MCTS-AHD can consistently outperform the manually designed ACO heuristics in all eight test sets and surpass a well-performing NCO method DeepACO (Ye et al., 2024b) on TSP and MKP test sets.

### 4.2. MCTS-AHD for Other Complex Tasks

To verify whether MCTS-AHD can still perform well in planning tasks other than NP-hard CO problems, we follow Yao et al. (2024c) to evaluate MCTS-AHD by designing heuristic CAFs for BO. The CAF is an important element for Cost-aware BO to approach the global optimum within a limited budget in a cost-efficient manner, and a series of advanced manually designed heuristics including EI (Mockus, 1974), EIpu (Snoek et al., 2012), and EI-cools (Lee et al., 2020a). We employ two synthetic instances with different landscapes and input dimensions (i.e., Ackley and Rastrigin in Table 4) as the evaluation dataset $D$ for LLM-based AHD and also test the manually and automatically designed heuristics on ten other synthetic instances. During heuristic evolutions, we set the sampling budget to 12 and run 5 independent trials for average performances. As shown in Table 4, heuristic CAFs designed by the proposed MCTS-AHD demonstrate higher qualities that outperform both manually designed heuristics and EoH in six out of twelve synthetic instances. It verifies that MCTS-AHD can not only show significant advantages in NP-hard CO problems but also has great potential in other complex planning tasks.

## 5. Discussion

Experiments have demonstrated the effectiveness of the proposed MCTS-AHD in designing high-quality heuristic functions for a wide range of application scenarios. This section first conducts essential ablation studies. Then, we will also analyze the advantages of utilizing MCTS in LLM-based AHD compared to the original population-based EC.

*Table 4.* Designing CAFs for BO. The table shows the gaps to optimal when running BO on instances with manually designed CAFs and CAFs designed by LLM-based AHD methods. LLM-based AHD methods are run three times for the average gaps. In testing, the evaluation budgets for BO are 30 and we run 10 trials for average gaps. The results of EI, EIpu, and EI-cool are from Yao et al. (2024c).

| Instances | Ackley | Rastrigin | Griewank | Rosenbrock | Levy | ThreeHumpCamel | StyblinskiTang | Hartmann | Powell | Shekel | Hartmann | Cosine8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EI | 2.66% | 4.74% | 0.49% | **1.26%** | **0.01%** | 0.05% | 0.03% | 0.00% | 18.89% | 7.91% | 0.03% | 0.47% |
| EIpu | 2.33% | 5.62% | 0.34% | 2.36% | 0.01% | 0.12% | **0.02%** | 0.00% | 19.83% | 7.92% | 0.03% | 0.47% |
| EI-cool | 2.74% | 5.78% | **0.34%** | 2.29% | 0.01% | 0.07% | 0.03% | **0.00%** | 14.95% | 8.21% | **0.03%** | 0.54% |
| LLM-based AHD: *GPT-4o-mini* | | | | | | | | | | | | |
| EoH | 2.45% | 0.90% | 0.54% | 56.78% | 0.20% | 0.26% | 0.79% | 0.04% | 70.89% | 4.56% | 0.33% | 0.36% |
| MCTS-AHD | **2.40%** | **0.77%** | 0.36% | 1.68% | 0.01% | **0.02%** | 0.20% | 0.01% | **1.27%** | **3.94%** | 0.38% | **0.34%** |

### 5.1. Ablation on Parameters and Components

We conduct ablation studies to validate the necessity of components in MCTS-AHD and to measure the sensitivity of the parameter settings. As shown in Table 5, we first remove three proposed components of MCTS-AHD (Progressive Widening, Thought-alignment, and Exploration-decay) and use these three variants to design heuristics. Table 5 reports their average optimality gaps over 3 runs in the in-domain test sets of Table 1. Ablation variants exhibit a clear performance degradation in at least one task. The actions for expanding the MCTS-AHD nodes are also significant. Actions e1 and e2 are closely associated with progressive widening, so they cannot be ablated individually. Results also show that the MCTS-AHD variant without actions s1, m1, and m2 (*w/o* Action s1, *w/o* Action m1, and *w/o* Action m2) could only design significantly inferior heuristics, demonstrating the importance of each LLM-based action in MCTS-AHD.

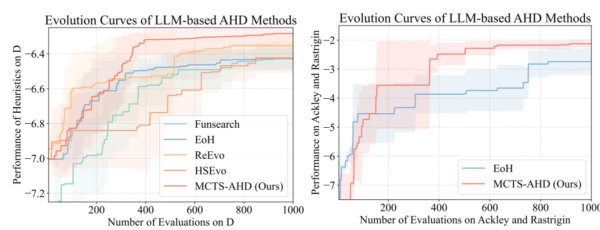
Meanwhile, we analyze the main parameter $\lambda_0$ of MCTS-AHD. The results show that although the TSP and KP tasks may have different preferences in the $\lambda_0$ setting, the default setting (i.e., $\lambda_0 = 0.1$) exhibits generally good quality.

*Table 5.* Ablation studies on the components, actions, and parameter settings of MCTS-AHD. We use MCTS-AHD variants to design heuristics with the step-by-step construction framework and report their optimality gaps averaged over three runs.

| Methods | *TSP50* | *KP100* |
|---|---|---|
| MCTS-AHD (Ours, 10 runs) | 10.661% | 0.059% |
| *w/o* Progressive Widening | 12.224% | 0.068% |
| *w/o* Thought-alignment | 12.023% | 0.046% |
| *w/o* Exploration-decay | 11.409% | 0.088% |
| *w/o* Action s1 | 12.386% | 0.066% |
| *w/o* Action m1 | 10.654% | 0.080% |
| *w/o* Action m2 | 12.172% | 0.070% |
| MCTS-AHD variant $\lambda_0$ = 0.05 | 10.914% | 0.065% |
| MCTS-AHD variant $\lambda_0$ = 0.2 | 12.091% | 0.042% |

### 5.2. MCTS versus Population-based EC

**Ability of MCTS-AHD in Escaping from Local Optima**. As the main contribution of this paper, instead of population-based EC, MCTS-AHD can manage underperforming but potential heuristic functions, achieving a more comprehensive exploration of the heuristic space $H$ thus escaping local optima. To verify this, we plot the performance curves of MCTS-AHD in designing heuristic functions in the step-by-step construction framework for TSP and designing CAF functions in BOs. Each curve is averaged from at least **5** runs. As illustrated in Figure 4, all baseline methods with populations exhibit early convergence to local optima, but MCTS-AHD can demonstrate a steady performance update and eventually convergence to better results.




(a) Step-by-step Construction for TSP (b) Designing CAFs in BO

*Figure 4.* The evolution curves on two diverse tasks.

**Advantage scopes of MCTS-AHD**. Compared to population-based baselines, we claim that MCTS-AHD demonstrates more advantages in application scenarios with more complex heuristic spaces $H$ and application scenarios with more descriptions as knowledge. We analyze these two claims with experiments in Appendix F.7.

## 6. Conclusion

This paper presents the first application of MCTS to LLM-based AHD. The proposed MCTS-AHD achieves a comprehensive exploration of heuristic space, resulting in higher-quality heuristics. The effectiveness of MCTS-AHD shows that MCTS can be a more promising basic structure compared to population-based EC.

**Limitation and Future Work**: As a limitation of MCTS-AHD, the convergence speed of MCTS-AHD can still be improved. In the future, we will consider designing MCTS-population hybrid structures for better evolution efficiency.

## Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

## References


Abgaryan, H., Harutyunyan, A., and Cazenave, T. Llms can schedule. *arXiv preprint arXiv:2408.06993*, 2024.

Alhindi, A., Alhindi, A., Alhejali, A., Alsheddy, A., Tairan, N., and Alhakami, H. Moea/d-gls: a multiobjective memetic algorithm using decomposition and guided local search. *Soft Computing*, 23:9605–9615, 2019.

Ansari, Z. N. and Daxini, S. D. A state-of-the-art review on meta-heuristics application in remanufacturing. *Archives of Computational Methods in Engineering*, 29(1):427–470, 2022.

Arnold, F. and Sörensen, K. Knowledge-guided local search for the vehicle routing problem. *Computers & Operations Research*, 105:32–46, 2019.

Arnold, F., Gendreau, M., and Sörensen, K. Efficiently solving very large-scale routing problems. *Computers & operations research*, 107:32–42, 2019.

Asani, E. O., Okeyinka, A. E., and Adebiyi, A. A. 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.

Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., and Protasi, M. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012.

Bäck, T., Fogel, D. B., and Michalewicz, Z. Handbook of evolutionary computation. *Release*, 97(1):B1, 1997.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Berto, F., Hua, C., Zepeda, N. G., Hottung, A., Wouda, N., Lan, L., Park, J., Tierney, K., and Park, J. Routefinder: Towards foundation models for vehicle routing problems. *arXiv preprint arXiv:2406.15007*, 2024.

Biggs, N. The traveling salesman problem a guided tour of combinatorial optimization, 1986.

Blot, A., Hoos, H. H., Jourdan, L., Kessaci-Marmion, M.-É., and Trautmann, H. Mo-paramils: A multi-objective automatic algorithm configuration framework. In *Learning and Intelligent Optimization: 10th International Conference, LION 10, Ischia, Italy, May 29–June 1, 2016, Revised Selected Papers 10*, pp. 32–47. Springer, 2016.

Brandfonbrener, D., Henniger, S., Raja, S., Prasad, T., Loughridge, C. R., Cassano, F., Hu, S. R., Yang, J., Byrd, W. E., Zinkov, R., et al. Vermcts: Synthesizing multi-step programs using a verifier, a large language model, and tree search. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024.

Brecklinghaus, J. and Hougardy, S. The approximation ratio of the greedy algorithm for the metric traveling salesman problem. *Operations Research Letters*, 43(3):259–261, 2015.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Qu, R. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

Burke, E. K., Hyde, M. R., Kendall, G., Ochoa, G., Özcan, E., and Woodward, J. R. A classification of hyper-heuristic approaches: revisited. *Handbook of metaheuristics*, pp. 453–477, 2019.

Cantú-Paz, E. et al. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10 (2):141–171, 1998.

Castiñeiras, I., De Cauwer, M., and O'Sullivan, B. Weibull-based benchmarks for bin packing. In *International Conference on Principles and Practice of Constraint Programming*, pp. 207–222. Springer, 2012.

Christofides, N. Worst-case analysis of a new heuristic for the travelling salesman problem. In *Operations Research Forum*, volume 3, pp. 20. Springer, 2022.

Coulom, R. Computing "elo ratings" of move patterns in the game of go. *ICGA journal*, 30(4):198–208, 2007.

Dainese, N., Merler, M., Alakuijala, M., and Marttinen, P. Generating code world models with large language models guided by monte carlo tree search. *arXiv preprint arXiv:2405.15383*, 2024.

Dat, P. V. T., Doan, L., and Binh, H. T. T. Hsevo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. *arXiv preprint arXiv:2412.14995*, 2024.

DeLorenzo, M., Chowdhury, A. B., Gohil, V., Thakur, S., Karri, R., Garg, S., and Rajendran, J. Make every move count: Llm-based high-quality rtl code generation using mcts. *arXiv preprint arXiv:2402.03289*, 2024.

Desale, S., Rasool, A., Andhale, S., and Rane, P. 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.

Dorigo, M., Birattari, M., and Stutzle, T. Ant colony optimization. *IEEE computational intelligence magazine*, 1 (4):28–39, 2006.

Drakulic, D., Michel, S., and Andreoli, J.-M. Goal: A generalist combinatorial optimization agent learning. *arXiv preprint arXiv:2406.15079*, 2024.

Duflo, G., Kieffer, E., Brust, M. R., Danoy, G., and Bouvry, P. A gp hyper-heuristic approach for generating tsp heuristics. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 521–529. IEEE, 2019.

Eiben, A. E. and Smith, J. E. *Introduction to evolutionary computing*. Springer, 2015.

Feng, X., Wan, Z., Wen, M., McAleer, S. M., Wen, Y., Zhang, W., and Wang, J. Alphazero-like tree-search can guide large language model decoding and training. *arXiv preprint arXiv:2309.17179*, 2023.

Fischetti, M., Lodi, A., and Toth, P. Exact methods for the asymmetric traveling salesman problem. *The traveling salesman problem and its variations*, pp. 169–205, 2007.

Fu, Z.-H., Qiu, K.-B., and Zha, H. Generalize a small pre-trained model to arbitrarily large tsp instances. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pp. 7474–7482, 2021.

Gao, C., Shang, H., Xue, K., Li, D., and Qian, C. Towards generalizable neural solvers for vehicle routing problems via ensemble with transferrable local policy, 2024.

Guo, P.-F., Chen, Y.-H., Tsai, Y.-D., and Lin, S.-D. Towards optimizing with large language models. *arXiv preprint arXiv:2310.05204*, 2023.

Hadi, M. U., Qureshi, R., Shah, A., Irfan, M., Zafar, A., Shaikh, M. B., Akhtar, N., Wu, J., Mirjalili, S., et al. A survey on large language models: Applications, challenges, limitations, and practical usage. *Authorea Preprints*, 2023.

Hadi, M. U., Al Tashi, Q., Shah, A., Qureshi, R., Muneer, A., Irfan, M., Zafar, A., Shaikh, M. B., Akhtar, N., Wu, J., et al. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. *Authorea Preprints*, 2024.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

He, Q., Head, K. L., and Ding, J. Heuristic algorithm for priority traffic signal control. *Transportation research record*, 2259(1):1–7, 2011.

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

Hemberg, E., Moskal, S., and O'Reilly, U.-M. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21, 2024.

Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B., et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 2023.

Huang, X., Liu, W., Chen, X., Wang, X., Wang, H., Lian, D., Wang, Y., Tang, R., and Chen, E. Understanding the planning of llm agents: A survey. *arXiv preprint arXiv:2402.02716*, 2024.

Hudson, B., Li, Q., Malencia, M., and Prorok, A. Graph neural network guided local search for the traveling salesperson problem. *arXiv preprint arXiv:2110.05291*, 2021.

Jiang, X., Wu, Y., Wang, Y., and Zhang, Y. Unco: Towards unifying neural combinatorial optimization through large language model. *arXiv preprint arXiv:2408.12214*, 2024.

Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L., and Murthy, A. Llms can't plan, but can help planning in llm-modulo frameworks. *arXiv preprint arXiv:2402.01817*, 2024.

Kim, M., Choi, S., Kim, H., Son, J., Park, J., and Bengio, Y. Ant colony sampling with gflownets for combinatorial optimization. *arXiv preprint arXiv:2403.07041*, 2024.

Kocsis, L. and Szepesvári, C. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Kool, W., Van Hoof, H., and Welling, M. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

Korte, B. H., Vygen, J., Korte, B., and Vygen, J. *Combinatorial optimization*, volume 1. Springer, 2011.

Kwon, Y.-D., Choo, J., Kim, B., Yoon, I., Gwon, Y., and Min, S. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.

Lam, R., Willcox, K., and Wolpert, D. H. Bayesian optimization with a finite budget: An approximate dynamic programming approach. *Advances in Neural Information Processing Systems*, 29, 2016.

Langdon, W. B. and Poli, R. *Foundations of genetic programming*. Springer Science & Business Media, 2013.

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

Lee, E. H., Perrone, V., Archambeau, C., and Seeger, M. Cost-aware bayesian optimization. *arXiv preprint arXiv:2003.10870*, 2020a.

Lee, J., Jeon, W., Kim, G.-H., and Kim, K.-E. Monte-carlo tree search in continuous action spaces with value gradients. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 4561–4568, 2020b.

Lehman, J., Gordon, J., Jain, S., Ndousse, K., Yeh, C., and Stanley, K. O. Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pp. 331–366. Springer, 2023.

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

Liu, F., Lin, X., Wang, Z., Zhang, Q., Xialiang, T., and Yuan, M. Multi-task learning for routing problem with cross-problem zero-shot generalization. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 1898–1908, 2024a.

Liu, F., Xialiang, T., Yuan, M., Lin, X., Luo, F., Wang, Z., Lu, Z., and Zhang, Q. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Forty-first International Conference on Machine Learning*, 2024b.

Liu, F., Yao, Y., Guo, P., Yang, Z., Lin, X., Tong, X., Yuan, M., Lu, Z., Wang, Z., and Zhang, Q. A systematic survey on large language models for algorithm design. *arXiv preprint arXiv:2410.14716*, 2024c.

Liu, F., Zhang, R., Xie, Z., Sun, R., Li, K., Lin, X., Wang, Z., Lu, Z., and Zhang, Q. Llm4ad: A platform for algorithm design with large language model. *arXiv preprint arXiv:2412.17287*, 2024d.

Liu, S., Chen, C., Qu, X., Tang, K., and Ong, Y.-S. Large language models as evolutionary optimizers. In *2024 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8. IEEE, 2024e.

López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L. P., Birattari, M., and Stützle, T. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

Luong, P., Nguyen, D., Gupta, S., Rana, S., and Venkatesh, S. Adaptive cost-aware bayesian optimization. *Knowledge-Based Systems*, 232:107481, 2021.

Lusby, R. M., Larsen, J., Ehrgott, M., and Ryan, D. An exact method for the double tsp with multiple stacks. *International Transactions in Operational Research*, 17 (5):637–652, 2010.

Mei, Y., Chen, Q., Lensen, A., Xue, B., and Zhang, M. Explainable artificial intelligence by genetic programming: A survey. *IEEE Transactions on Evolutionary Computation*, 27(3):621–641, 2022.

Merz, P. and Freisleben, B. Genetic local search for the tsp: New results. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*, pp. 159–164. IEEE, 1997.

Meyerson, E., Nelson, M. J., Bradley, H., Gaier, A., Moradi, A., Hoover, A. K., and Lehman, J. Language model crossover: Variation through few-shot prompting. *arXiv preprint arXiv:2302.12170*, 2023.

Mockus, J. On bayesian methods for seeking the extremum. In *Proceedings of the IFIP Technical Conference*, pp. 400–404, 1974.

Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., and Mian, A. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*, 2023.

Qi, Z., Ma, M., Xu, J., Zhang, L. L., Yang, F., and Yang, M. Mutual reasoning makes smaller llms stronger problem-solvers. *arXiv preprint arXiv:2408.06195*, 2024.

Rajendran, C. Heuristic algorithm for scheduling in a flowshop to minimize total flowtime. *International Journal of Production Economics*, 29(1):65–73, 1993.

Reinelt, G. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J., Ellenberg, J. S., Wang, P., Fawzi, O., et al. Mathematical discoveries from program search with large language models. *Nature*, 625 (7995):468–475, 2024.

Rosenkrantz, D. J., Stearns, R. E., and Lewis, II, P. M. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977.

Sánchez-Díaz, X., Ortiz-Bayliss, J. C., Amaya, I., Cruz-Duarte, J. M., Conant-Pablos, S. E., and Terashima-Marín, H. A feature-independent hyper-heuristic approach for solving the knapsack problem. *Applied Sciences*, 11(21): 10209, 2021.

Sengupta, L., Mariescu-Istodor, R., and Fränti, P. Which local search operator works best for the open-loop tsp? *Applied Sciences*, 9(19):3985, 2019.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and De Freitas, N. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104 (1):148–175, 2015.

Shi, W. W., Han, W., and Si, W. C. A hybrid genetic algorithm based on harmony search and its improving. In *Informatics and Management Science I*, pp. 101–109. Springer, 2013.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

Stützle, T. and López-Ibáñez, M. Automated design of metaheuristic algorithms. *Handbook of metaheuristics*, pp. 541–579, 2019.

Sui, J., Ding, S., Xia, B., Liu, R., and Bu, D. Neuralgls: learning to guide local search with graph convolutional network for the traveling salesman problem. *Neural Computing and Applications*, 36(17):9687–9706, 2024.

Świechowski, M., Godlewski, K., Sawicki, B., and Mańdziuk, J. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023.

Tan, C. S., Mohd-Mokhtar, R., and Arshad, M. R. A comprehensive review of coverage path planning in robotics using classical and heuristic algorithms. *IEEE Access*, 9: 119310–119342, 2021.

Tuononen, J. Analysis of rebuild local search operator for tsp. Master's thesis, Itä-Suomen yliopisto, 2022.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

Voudouris, C. and Tsang, E. Guided local search and its application to the traveling salesman problem. *European journal of operational research*, 113(2):469–499, 1999.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Weston, J. and Sukhbaatar, S. System 2 attention (is something you might need too). *arXiv preprint arXiv:2311.11829*, 2023.

Williams, C. K. and Rasmussen, C. E. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

Wu, Y., Song, W., Cao, Z., Zhang, J., and Lim, A. Learning improvement heuristics for solving routing problems.. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

Wu, Z., Qi, Q., Zhuang, Z., Sun, H., and Wang, J. Pre-tokenization of numbers for large language models. In *The Second Tiny Papers Track at ICLR 2024*, 2024.

Xin, L., Song, W., Cao, Z., and Zhang, J. Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. *Advances in Neural Information Processing Systems*, 34: 7472–7483, 2021.

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers, 2024. URL https://arxiv.org/abs/2309.03409.

Yao, S., Liu, F., Lin, X., Lu, Z., Wang, Z., and Zhang, Q. Multi-objective evolution of heuristic using large language model. *arXiv preprint arXiv:2409.16867*, 2024a.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024b.

Yao, Y., Liu, F., Cheng, J., and Zhang, Q. Evolve cost-aware acquisition functions using large language models. In *International Conference on Parallel Problem Solving from Nature*, pp. 374–390. Springer, 2024c.

Ye, H., Wang, J., Cao, Z., Berto, F., Hua, C., Kim, H., Park, J., and Song, G. Reevo: Large language models as hyper-heuristics with reflective evolution. *arXiv preprint arXiv:2402.01145*, 2024a.

Ye, H., Wang, J., Cao, Z., Liang, H., and Li, Y. Deepaco: neural-enhanced ant systems for combinatorial optimization. *Advances in Neural Information Processing Systems*, 36, 2024b.

Ye, H., Wang, J., Liang, H., Cao, Z., Li, Y., and Li, F. Glop: Learning global partition and local construction for solving large-scale routing problems in real-time. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024c.

Yin, H., Kononova, A. V., Bäck, T., and van Stein, N. Controlling the mutation in large language models for the efficient evolution of algorithms. *arXiv preprint arXiv:2412.03250*, 2024.

Yu, H. and Liu, J. Deep insights into automated optimization with large language models and evolutionary algorithms. *arXiv preprint arXiv:2410.20848*, 2024.

Zhang, D., Huang, X., Zhou, D., Li, Y., and Ouyang, W. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*, 2024a.

Zhang, D., Zhoubian, S., Hu, Z., Yue, Y., Dong, Y., and Tang, J. Rest-mcts*: Llm self-training via process reward guided tree search. *arXiv preprint arXiv:2406.03816*, 2024b.

Zhang, Y., Pan, Y., Wang, Y., and Cai, J. Pybench: Evaluating llm agent on various real-world coding tasks. *arXiv preprint arXiv:2407.16732*, 2024c.

Zhao, Q., Duan, Q., Yan, B., Cheng, S., and Shi, Y. A survey on automated design of metaheuristic algorithms. *arXiv preprint arXiv:2303.06532*, 2023.

Zheng, Z., Yao, S., Li, G., Han, L., and Wang, Z. Pareto improver: Learning improvement heuristics for multi-objective route planning. *IEEE Transactions on Intelligent Transportation Systems*, 2023.

Zheng, Z., Yao, S., Wang, Z., Tong, X., Yuan, M., and Tang, K. Dpn: Decoupling partition and navigation for neural solvers of min-max vehicle routing problems. *arXiv preprint arXiv:2405.17272*, 2024a.

Zheng, Z., Zhou, C., Xialiang, T., Yuan, M., and Wang, Z. Udc: A unified neural divide-and-conquer framework for large-scale combinatorial optimization problems. *arXiv preprint arXiv:2407.00312*, 2024b.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

# A. Related Work

This section presents a detailed literature review of various research areas related to LLM, AHD, MCTS, and CO problems.

## A.1. AHD

Automatic Heuristic Design, also known as Hyper-Heuristics, (Burke et al., 2013; Stützle & López-Ibáñez, 2019) aims to find the best-performing heuristic among an extensive set of valid heuristics, i.e., the heuristic space. AHD methods present various effective methodologies and frameworks (Blot et al., 2016; López-Ibáñez et al., 2016; Burke et al., 2019) to update heuristic algorithms automatically. GP (Mei et al., 2022) is a prevailing and effective approach for AHD. However, it requires hand-crafted operators for heuristic mutation and crossover (Duflo et al., 2019; Sánchez-Díaz et al., 2021).

## A.2. Neural Combinatorial Optimization (NCO)

Traditional methods of solving CO problems contain exact methods (Fischetti et al., 2007; Lusby et al., 2010) and approximate methods (Merz & Freisleben, 1997; Helsgaun, 2017). Leveraging the recent development of deep learning techniques (He et al., 2016; Vaswani et al., 2017), Neural Combinatorial Optimization (NCO) methods can obtain near-optimal results on various CO problems with less time consumption compared to traditional methods (Bello et al., 2016; Kool et al., 2018). NCO methods train neural networks for decision-making with diverse solving process (e.g., the LKH algorithm (Xin et al., 2021), the GLS algorithm (Sui et al., 2024; Hudson et al., 2021), improvement-based framework (Wu et al., 2021; Zheng et al., 2023), or step-by-step construction (Vinyals et al., 2015; Zheng et al., 2024a)). So, NCO methods can be regarded as a special kind of Hyper-Heuristics (AHD) (Ye et al., 2024a), where the solving process is the general framework of AHD and the training process of NCO methods searches for the optimal parameter settings within a parameterized heuristic space by gradient decent. In contrast to traditional heuristics, NCO does not require expert knowledge, so some NCO methods can be applied to multiple CO problems (Ye et al., 2024c; Zheng et al., 2024b; Liu et al., 2024a; Drakulic et al., 2024; Berto et al., 2024).

Compared to existing NCO methods, designing heuristics with LLM-based AHD methods for NP-hard CO problems demonstrates better efficiency, better applicability, and lower implementation difficulty. Training an NCO model on a given framework may take several days or even several weeks on an advanced GPU (Kwon et al., 2020), whereas a high-quality heuristic can be generated in a few hours using the LLM-based AHD methods without GPU requirements. Considering applicability, LLM-based AHD can be more quickly applied to various complex NP-hard CO problems without task-specific training, demonstrating remarkable flexibility. NCO methods bring more implementation difficulties when creating complicated environments for model training (Zheng et al., 2024b). LLM-based AHD methods, instead, only need some linguistic descriptions for the environment. Moreover, LLM-based AHD methods can even demonstrate better results than NCO methods in some application scenarios, as shown in Table 1, Table 2, and Table 8, heuristics designed by the proposed MCTS-AHD can outperform advanced NCO methods under the same solving framework (e.g. POMO (Kwon et al., 2020), DeepACO (Ye et al., 2024b)) in 10 out of 16 test sets.

## A.3. LLM for EC

EC is a generic optimization principle inspired by natural evolution (Bäck et al., 1997; Eiben & Smith, 2015). The core idea of EC is to generate a set of candidate solutions (called populations) by simulating genetic variation and natural selection in the process of biological evolution and to gradually optimize these solutions through an iterative process over multiple generations. Some recent work focuses on stimulating the crossover or mutation operators in the original evolutionary computation framework by prompting LLMs (Lehman et al., 2023; Meyerson et al., 2023; Lange et al., 2024) or introducing LLMs to generate auxiliary information (Ye et al., 2024a). LLM-based EC approaches can be specified to plenty of application scenarios, including planning (Kambhampati et al., 2024), code generation (Hemberg et al., 2024), and LLM-based AHD discussed in this paper (Romera-Paredes et al., 2024).

## A.4. LLM for AHD

LLM-based AHD methods with populations can be summarized as a four-part process (Romera-Paredes et al., 2024; Liu et al., 2024b; Ye et al., 2024a; Dat et al., 2024; Liu et al., 2024d), **1):** Heuristics Initialization: LLM-based AHD methods employ LLMs to generate multiple initial heuristic functions or directly handcraft seed functions as initial heuristic functions. **2):** Operator selection: Existing methods design a variety of prompts corresponding to mutation or crossover operators on parent heuristic functions. In this step, population-based methods will randomly select an operator to be executed. Afterward, these methods also select the parent heuristic(s) to be operated from the current population (Yin et al., 2024). **3):** New heuristic generation: Next, LLM-based AHD methods will prompt LLMs for the Python code implementation of a new heuristic function based on the description of the task $P$, general framework, the prompt of selected operators, and the selected parent heuristics. **4):** Population update: As the final step in each iteration, each newly generated heuristic function $h$ will be evaluated by the training set $D$ to obtain $g(h)$, the invalid heuristic functions will also be discarded. The population will keep only the heuristic functions with top performances $g(h)$. The last three processes will run in a loop until the number of heuristic evaluations reaches a given upper limit of $T$.

Yao et al. (2024a) considers multiple objectives (for example, heuristic performance and execution efficiency) in LLM-based AHD. This paper focuses on heuristic performance, so we do not involve it as a baseline.

### A.5. LLM for CO problem

Besides LLM-based AHD, there are also ways to utilize LLMs for CO problems. Existing methods provide two more approaches: LLM-as-optimizer methods and (fine-tuned) LLM-as-solver methods. Yang et al. (2024) first developed LLMs as an optimizer on TSP tours. LLM-as-optimizer methods (Guo et al., 2023; Liu et al., 2024e) initially generate several possible solutions for a single CO instance. In each iteration, the LLM is prompted to generate better solutions by taking the original top-performing solutions and their objective values as in-context information. Then, the newly generated solution is evaluated for its objective value and inserted back into the in-context. LLM-as-solver methods directly treat LLMs as end-to-end CO instance solvers (Abgaryan et al., 2024; Jiang et al., 2024). These methods consider LLMs as pre-trained NCO models and establishing environments to fine-tune LLMs for better performances on CO instances.

LLM-based AHD methods, especially MCTS-AHD, are still the most promising directions for solving complex NP-hard CO problems. LLM-as-optimizer methods pose high demand for LLM's reasoning ability and knowledge storage, and current LLMs (Huang et al., 2024) can only offer extremely limited results on infamous and large-scale CO instances (refer to comparisons in Table 11). It should be noted that Kambhampati et al. (2024) also recommends using LLMs for module design instead of instance solving for complex planning tasks. The module design process corresponds to the idea of LLM-based AHD. (Fine-tuned) LLM-as-solver methods (Abgaryan et al., 2024; Jiang et al., 2024) require additional training and face difficulties in configuring training environments. Moreover, unlike NCO models that represent each coordinate value as a piece of embeddings, LLMs face challenges in tokenizing high-precision coordinate numbers in a linguistic way (Wu et al., 2024).

### A.6. LLM inference with MCTS

With the great success of CoT (Wei et al., 2022) and ToT (Yao et al., 2024b) in enhancing the ability of LLM reasoning, a series of System 2 techniques (Weston & Sukhbaatar, 2023) have been proposed. Among them, MCTS has recently emerged as a powerful technique to enhance the reasoning capabilities of LLMs (Zhang et al., 2024b). These methods mainly construct MCTS in two ways (Feng et al., 2023), representing each node with a complete answer refined from the parents (Zhang et al., 2024a; Zhou et al., 2023) or a reasoning step following its parents (Qi et al., 2024). The MCTS-AHD in this paper belongs to the former, where each node represents a complete piece of executable function and its description. When dealing with commonsense QA or mathematical problems, System 2 techniques often use self-evaluation for MCTS simulation (Zhang et al., 2024a), but MCTS-AHD does not involve the self-evaluation or rollout (Silver et al., 2016) since AHD methods can easily obtain performances $g(\cdot)$ for tasks.

### A.7. LLM for code generation

Recent LLMs have strong coding capabilities (Zhang et al., 2024c), and some recent papers have designed a series of structures based on reflection and MCTS (Dainese et al., 2024; DeLorenzo et al., 2024) to promote this advantage. The LLM for code generation is a similar domain to LLM-based AHD. Compared to code generation which aims at passing algorithm tests, LLM-based AHD faces more challenges in finding the optimal function (Liu et al., 2024c). LLM-based AHD needs to consider more on exploring the entire heuristic space and optimizing code performances by modifying parameter settings and detail designs.

Most similar to this paper, Brandfonbrener et al. (2024) uses the process of MCTS with progressive widening for code generation, but its specific process in MCTS including progressive widening is different from the proposed MCTS-AHD.

## B. Defination of Tasks

### B.1. NP-hard CO Problems

This paper conducts experiments on six NP-hard representative CO problems, including TSP, CVRP, KP, MKP, ASP, and BPP. For the BPP, we consider both an online setting and an offline setting. Next, we will introduce the mathematical definitions of these problems in detail.

**Traveling Salesman Problem** TSP is one of the most representative COPs (Biggs, 1986), which aims at finding the shortest path to visit each city once and returns to the starting point. An $N$-node TSP instance $\boldsymbol{ins}$ contains distance matrix $\{D = d_{j,k}, j = 1, \ldots, N, k = 1 \ldots, N\} \in \mathbb{R}^{N\times N}$, where $d_{j,k}$ denotes the cost between nodes $k$ and $j$, the goal is to minimize the following objective function (Zheng et al., 2024b):

$$\text{minimize} \quad f(\boldsymbol{s}) = \sum_{t=1}^{N-1} d_{x_t,x_{t+1}} + d_{x_N,x_1}, \tag{8}$$

where the solution $\boldsymbol{s} = (s_1, s_2, \ldots, s_N)$ is a permutation of all nodes. All the feasible solutions satisfy the constraint of node degree being two and containing no loop with lengths less than N.

**Capacitated Vehicle Routing Problem** CVRP aims to plan several capacity-constrained vehicles departing from a depot, meeting the demands of multiple customers, and minimizing the total travel distance. Each CVRP instance contains a depot (the 0-th node) and several customers. With a distance matrix $\{D = d_{j,k}, j = 0, \ldots, N, k = 0 \ldots, N\}$, the CVRP can be expressed as follows:

$$\begin{aligned}
&\text{minimize} \quad f(\boldsymbol{s}) = \sum_{j=1}^{q} C(\boldsymbol{\rho}^j), \quad C(\boldsymbol{\rho}^j) = \sum_{t=0}^{|\boldsymbol{\rho}^j|-1} d_{\rho^j_t,\rho^j_{t+1}} + d_{\rho^j_{n_j},\rho^j_0},\\
&\text{s.t.} \qquad 0 \le \delta_i \le C, \quad \sum_{i\in\boldsymbol{\rho}^j} \delta_i \le C, \quad i \in \{1, \ldots, n\}, j \in \{1, \ldots, q\},
\end{aligned} \tag{9}$$

where $\boldsymbol{s}$ is a solution representing the complete route of vehicles and consists of $q$ sub-routes $\boldsymbol{s} = \{\boldsymbol{\rho}^1, \boldsymbol{\rho}^2, \ldots, \boldsymbol{\rho}^q\}$. Each sub-route $\boldsymbol{\rho}^j = (\rho^j_1, \ldots, \rho^j_{n_j}),\ j \in \{1, \ldots, q\}$ starts from the depot $s_0$ and backs to $s_0$, $n_j$ represents the number of customer nodes in it. $n = \sum_{j=1}^{q} n_j$ is the total number of customer nodes; $\delta_i$ denotes the demand of node $i$; $C$ denotes the capacity of the vehicle. This paper follows the settings of ReEvo (Ye et al., 2024a) when generating CVRP data sets, fixing the coordinates of the depot to (0.5, 0.5).

**0-1 Knapsack Problem** KP is a typical CO problem, consider filling a $W$-capacity knapsack with items of a maximum total value, and each item can only be picked once. KP solution $\boldsymbol{s} \subseteq \{1, 2, ..., N\}$ records the selection item indexes. A KP instance $\boldsymbol{ins}$ records the value $v_i \sim \text{Uniform}(0, 1)$ and weight $w_i \sim \text{Uniform}(0, 1)$ of each candidate item. We follow the settings in Kool et al. (2018) in generating instances and have $W = 25$ for 100-item and 200-item KP instances and $W = 12.5$ for 50-item ones.

**Multiple Knapsack Problem** We follow the settings for MKP instances in ReEvo, setting the value $v_i \sim \text{Uniform}(0, 1)$ and weight for the $m$ knapsack $w_{ij} \sim \text{Uniform}(0, 1), i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\}$. We also uniformly sample the capacity of the $m$ knapsacks $C_i, i \in \{1, \ldots, m\}$ from $(\max_j w_{ij}, \sum_j w_{ij})$.

**Admissible Set Problem** ASP constructs a set of $n$-dimensional vectors $\mathcal{A}$ (called an admissible set) where vectors $\boldsymbol{a} \in \mathcal{A} \subset \{0, 1, 2\}^n$ and the number of non-zero items is constrained to be $w$. ASP aims to maximize the admissible set $\mathcal{A}$ size under a certain constraint that for any three distinct vectors there is a coordinate in which their three respective values are $\{0, 0, 1\}$, $\{0, 0, 2\}$, or $\{0, 1, 2\}$. We formulate the objective function and constraints as follows:

$$\begin{aligned}
&\text{maximize} \quad |\mathcal{A}|\\
&\text{s.t.} \qquad \sum_{i=1}^{n} \mathbb{I}[a_i \ne 0] = w, \exists i \in \{1, \ldots, n\}, \{a_i, b_i, c_i\} \in \{\{0, 0, 1\}, \{0, 0, 2\}, \{0, 1, 2\}\}, \forall \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathcal{A},
\end{aligned} \tag{10}$$

where $\mathbb{I}[a_i \ne 0]$ represents the number of non-zero items in vector $\boldsymbol{a} = (a_1, \ldots, a_n)$.

**Bin Packing Problem** BPP is an important and classic NP-hard CO problem that aims to place a set of items with different sizes into as few $W$-capacity bins as possible. Online BPP requires making an immediate decision on which bin to place as soon as a new item is inputted and offline BPP does not require this. We follow Romera-Paredes et al. (2024) and Liu et al. (2024b) to generate WeiBull instances for online BPP and follow Ye et al. (2024a) to generate offline BPP instances with $W = 150$ and the size of items uniformly sampled from 20 to 100.

### B.2. Cost-Aware Acquisition Function Design in Bayesian Optimization

Please refer to Appendix C.4 for relevant introduction.

## C. Defination of General Frameworks

For NP-hard combinatorial optimization (CO) problems, we employ MCTS-AHD to design key functions within given general frameworks. To verify the framework-agnosticism of MCTS-AHD, we include three general frameworks in experiments, e.g., step-by-step construction, GLS, and ACO. Next, in this section, we will introduce these involved general frameworks in detail.

### C.1. Step-by-Step Construction

Step-by-step construction is an intuitive framework capable of handling almost all CO problems. It considers gradually expanding a solution ($\boldsymbol{s}$) of an NP-hard CO problem from scratch until the solution is finally constructed. In each step of construction (i.e., solution expansion), step-by-step construction gives priority to each candidate and selects the one with the highest priority.

In the step-by-step construction framework, MCTS-AHD and LLM-based AHD baselines design the same key function that is repeatedly executed to calculate the priority of candidates and decide how to expand the solution. In this paper, the step-by-step construction framework is considered for solving four CO problems: TSP, KP, ASP, and online BPP. Based on this framework, Nearest-greedy is a common manually designed heuristic that scores candidate nodes entirely based on their distance from the current node and is prevailing for TSP. Similarly, a greedy process that evaluates KP items based on their value-weight-ratio is also commonly used. We consider these two handcrafted heuristics as Greedy Construct in Table 1. Additionally, in both TSP and KP, there is a series of NCO methods training deep neural networks based on the step-by-step construction solving framework. Their neural networks can be considered as more efficient and sophisticated key functions for evaluating all candidate nodes or items.

- For TSP, MCTS-AHD is responsible for designing a function to select the next node to visit based on node coordinates, starting point, distance matrix, and all unvisited nodes.
- For KP, MCTS-AHD is responsible for selecting the next item to add to the knapsack based on the value and weight of all items to be chosen and the remaining capacity of the knapsack.
- For ASP, the key function provides a score for the current vector to determine to what extent it is suitable to be added to the admissible set ($\mathcal{A}$).
- In MCTS-AHD, the function required for online BPP gives a preference score for adding the current newly input item to each bin based on the item's size and the remaining capacity of all bins.

### C.2. Ant Colony Optimization

ACO is a metaheuristic and evolutionary algorithm (EA) inspired by the behavior of ants to find the shortest route between their colony and food sources (Dorigo et al., 2006; Ansari & Daxini, 2022).

ACO records a pheromone matrix $\boldsymbol{\tau}$ and a heuristic matrix $\boldsymbol{\eta}$. Each item in the matrix $\tau_{ij}$ indicates the desirability of including that edge $(i, j)$ in a solution. The pheromone trails are iteratively updated based on the quality of the solutions found, encouraging future ants to follow better paths. The heuristic information on each edge (i.e., $\eta_{ij}$) is a problem-specific measure that indicates the immediate benefit of choosing a particular path. For solving TSP with ACO and a manually designed heuristic matrix, $\eta_{ij}$ is often set to the inverse of the distance between cities $i$ and $j$, that is, $\eta_{ij} = \frac{1}{d_{ij}}$. LLM-based AHD methods are employed to produce the heuristic matrix $\boldsymbol{\eta}$ based on the necessary problem-specific inputs.

Ants construct solutions by moving from node to node, probabilistically choosing the next node based on a combination of pheromone and heuristic information. After all ants have constructed their solutions, the pheromone levels are updated. An ACO iteration typically involves solution construction, optional local search, and pheromone update. By iteratively applying these steps, ACO algorithms can effectively explore the solution space and converge toward optimal or near-optimal solutions for NP-hard CO problems. We follow the settings in Ye et al. (2024a), evaluating MCTS-AHD by designing the generation functions of heuristic metrics for TSP, CVRP, MKP, and offline BPP.

- For TSP, the function inputs the distance matrix. We adopt the parameter settings in ReEvo (Ye et al., 2024a), the number of ants is set to 30 during heuristic evaluation (Evaluating on the training dataset $D$), and the number of iterations is 100. In testing, we allow more optimization for all ACO baselines and increase the number of iterations to 500.

- Functions for CVRP input the distance matrix, coordinates and demands of nodes, and the capacity $C$. Numbers of ants and iterations are set the same as TSP.
- For MKP, the function inputs the values and weights of items. The number of ants is set to 10. The number of iterations is 50 for evaluating the dataset $D$ and 100 for test sets.
- For offline BPP, the function inputs the size of items and the capacity of bins. The number of ants is set to 20. The number of iterations is 15 for evaluating the dataset $D$ and 100 for test sets.

**ACO for Black-box Settings** Black-box settings are proposed in ReEvo (Ye et al., 2024a) as novel application scenarios for LLM-based AHD. Under black-box settings, tasks are provided with no descriptions related to the task $P$ and the general framework. The names of inputs in the to-be-designed key function will also be erased. These settings can simulate the application scenarios that cannot find any linguistic descriptions. This paper will evaluate AHD methods on both regular settings and black-box settings in Appendix F.7. To generate stable initial nodes, we set $N_I = 10$ for MCTS-AHD in designing heuristics with black-box settings.

### C.3. Guided Local Search

The GLS framework uses local search algorithms (e.g., using 2-opt operators) for solution iteration and introduces a penalty mechanism to guide the search process escaping local optima. It shows capability on a wide range of CO problems with manually designed key functions (Voudouris & Tsang, 1999; Alhindi et al., 2019). We use LLM-based AHD methods to design functions for generating knowledge-based matrices in the knowledge-guided local search (KGLS) framework (Arnold & Sörensen, 2019). Taking the TSP as an example, KGLS maintains a TSP solution while also preserving the ever-encountered best-performing solution. In each iteration of KGLS, KGLS first perturbs the TSP solution based on the generated knowledge matrix and then performs a local search using both 2-opt and relocate operators (Sengupta et al., 2019; Tuononen, 2022). We conduct 1200 iterations in both the heuristic function evolution process and the testing procedure for each CO instance. The number of perturbations to each solution is set to 30.

### C.4. Bayesian Optimization

BO (Shahriari et al., 2015) is a method for solving the black-box optimization problem where the objective is to find the global minimum of an unknown function $f(\mathbf{x})$ over a search space $\mathcal{X}$, represented as:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x}). \tag{11}$$

Two core components of BO are the probabilistic surrogate model and the acquisition function (Mockus, 1974; Lam et al., 2016). In each iteration of BO, the probabilistic surrogate model is first trained using the available samples (BO typically employs a Gaussian process (GP) model (Williams & Rasmussen, 2006)). Then, the acquisition function utilizes the posterior information of the surrogate model to guide the subsequent search. Specifically, the next solution to evaluate at iteration $t$, denoted as $\mathbf{x}_t$, is chosen by maximizing the acquisition function:

$$\mathbf{x}_t = \arg\max_{\mathbf{x}\in\mathcal{X}} \alpha(\mathbf{x}, M_t), \tag{12}$$

where $M_t$ represents the information from the surrogate model at iteration $t$, and $\alpha(\mathbf{x}, M_t)$ represents the acquisition function value at $\mathbf{x}$. After performing an expensive evaluation at $\mathbf{x}_t$, this point is added to the available dataset. BO iteratively executes surrogate benchmark training and sampling based on the acquisition function until a termination criterion is met. Finally, the best solution among those evaluated is returned as the solution to the problem.

This article focuses on the cost-aware BO (Yao et al., 2024c; Luong et al., 2021; Snoek et al., 2012), which takes the number of expensive evaluations as the termination criterion. It not only focuses on optimizing the objective function but also considers the cost of evaluating the objective function. This method is beneficial in practical applications, especially when the evaluation cost of the objective function is high, like experimental design and hyperparameter optimization of machine learning models. BO employs a CAF as the acquisition function in cost-aware settings to manage the sample efficiency, which is the design focus of the cost-aware BO methods. We adopt LLM-based AHD methods for designing the CAF. During CAF evolutions, MCTS-AHD and EoH run 5 independent cost-aware BO trails with at most 12 function samples. In testing, we set the evaluation budget to 30 and report the average result of 10 trials.

## D. Details of Evaluations & Experiments

In this section, we provide a more detailed introduction on the setup of evaluation budgets $T$ and evaluation datasets $D$ used in the heuristic evaluation phase. We want to take as fair a setup as possible so most evaluation settings are adopted from Funsearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024b), and ReEvo (Ye et al., 2024a).

**The Setting of** $T$. The number of generations in EoH is set to 20. And its population size is 20 for online bin packing and 10 for TSP and FSSP. So, this paper designs a similar design for the maximum number of evaluations $T$, setting the $T$ to 2,000 for online BPP (under the step-by-step construction framework) and setting $T = 1,000$ for other tasks.

**The Setting of** $D$. For most problems, MCTS-AHD adopts the settings in LLM-based baseline methods (e.g., EoH, ReEvo, Funsearch). For (Online) BPP under the step-by-step construction framework, baselines adopt 5 5,000-item WeiBull BPP instances with $W$=100, However, such a dataset design often leads to algorithms designed to fail completely at other scales, so we used a varying-scale setup (Gao et al., 2024) that included data with different characteristics in the evaluation dataset $D$. The detailed settings of the evaluation datasets are exhibited in Table 6

**Pre-Trained Large Language Models**. The pre-trained LLM is *gpt-4o-mini-2024-07-18* for *GPT-4o-mini* and *gpt-3.5-turbo-0125* for *GPT-3.5-turbo*

*Table 6.* Compositions of the evaluation dataset $D$ on involved general frameworks and tasks.

| Framework | Step-by-step Construction | |
|---|---|---|
| Task | TSP | KP |
| Evaluation dataset $D$ | 64 50-node TSP instances | 64 100-item KP instances (W=25) |
| Framework | Step-by-step Construction | |
| Task | ASP | Online BPP |
| Evaluation dataset $D$ | 1 instance (n=15, w=10) | 4 WeiBull BPP instances<br>1,000-item instance with W=100<br>1,000-item instance with W=500<br>5,000-item instance with W=100<br>5,000-item instance with W=500 |
| Framework | ACO | |
| Task | TSP | CVRP |
| Evaluation dataset $D$ | 5 50-node TSP instances | 10 50-node CVRP instances |
| Framework | ACO | |
| Task | MKP | Offline BPP |
| Evaluation dataset $D$ | 5 100-item MKP instances (m=5) | 5 500-item BPP instances (W=150) |
| Framework | GLS | CAF Design |
| Task | TSP | - |
| Evaluation dataset $D$ | 10 TSP200 instances | 2 synthetic instances<br>The Ackley instance<br>The Rastrigin instance |

**Other Parameters.** MCTS-AHD adopts the same set of parameters for all tasks involved, and here we will summarize the other parameters of MCTS-AHD in the evaluation phase. The number of initial nodes $N_I = 4$, the maximum depth of the tree $H = 10$, the number of mutations in each expansion $k = 2$, the maximum number of references in action e1 $p \in \{2, 3, 4, 5\}$, the initial exploration parameter $\lambda_0 = 0.1$, and the progressive widening parameter $\alpha = 0.5$. We consider $\lambda_0 = 0.1$ to be the most important setting of these and therefore ablate it in the main text, and we discuss the rest of the settings in Appendix F.6 to illustrate that none of these parameters take sensitive values.

# E. Detailed Methodology

## E.1. Prompts of MCTS Actions

MCTS-AHD contains 6 distinct actions i1, e1, e2, m1, m2, and s1 for MCTS initialization and expansion. Next, we describe the meaning and prompt engineering of each action. these prompts need to be adapted to the problem description, context, function name, input name, and output name according to the task $P$ and the current MCTS. The rest of the section is generic, in which we highlight the highlights of each prompt in the red color (For all actions we use the TSP under the step-by-step construction framework as an example):

- **Initial Action i1**: Action i1 represents directly getting an idea of designing a heuristic function from scratch and a Python implementation directly through LLM.

**Prompt for Action i1**

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.

First, describe the design idea and main steps of your algorithm in one sentence. The description must be inside a brace outside the code implementation. Next, implement it in Python as a function named 'select_next_node'.

This function should accept 4 input(s): 'current_node', 'destination_node', 'unvisited_nodes', 'distance_matrix'. The function should return 1 output(s): 'next_node'. The select_next_node function takes as input the current node, the destination_node, a set of unvisited nodes, and a distance matrix, and returns the next node to visit.

Do not give additional explanations.

- **Crossover Action e1**: Action e1 inputs several distinct heuristic functions with their codes, their descriptions, and their performances. The prompt asks the LLM to get an idea for a new heuristic function different from all these heuristic functions and its corresponding Python implementation.

**Prompt for Action e1**

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 as follows:
No.1 algorithm's description, its corresponding code and its objective value are:
# Its Description
# Its Python Code Implementation of A Function
Objective value: # The Objective Value of the heuristic algorithm with the python code on a evaluation dataset.
...
No.k algorithm's description, its corresponding code and its objective value are:
# Its Description
# Its Python Code Implementation of A Function
Objective value: # The Objective Value of the heuristic algorithm with the python code on a evaluation dataset.

Please create a new algorithm that has a totally different form from the given algorithms. Try generating codes with different structures, flows or algorithms. The new algorithm should have a relatively low objective value.

First, describe the design idea and main steps of your algorithm in one sentence. The description must be inside a brace outside the code implementation. Next, implement it in Python as a function named 'select_next_node'.

This function should accept 4 input(s): 'current_node', 'destination_node', 'unvisited_nodes', 'distance_matrix'. The function should return 1 output(s): 'next_node'. The select_next_node function takes as input the current node, the destination_node, a set of unvisited nodes, and a distance matrix, and returns the next node to visit.

Do not give additional explanations.

- **Crossover Action e2**: Action e2 inputs one parent heuristic function (with its code, description, and performance) and one reference heuristic function from a 10-size top-performing elite heuristic function set $E$. The following prompt asks the LLM to design a new heuristic function based on the parent one and learn from the advantageous designs of the reference one.

**Prompt for Action e2**

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 as follows:
No.1 algorithm's description, its corresponding code and its objective value are:
# Its Description
# Its Python Code Implementation of A Function
Objective value: # The Objective Value of the heuristic algorithm with the python code on a evaluation dataset.

No.2 algorithm's description, its corresponding code and its objective value are:
# Its Description
# Its Python Code Implementation of A Function
Objective value: # The Objective Value of the heuristic algorithm with the python code on a evaluation dataset.

Please create a new algorithm that has a similar form to the No.2 algorithm and is inspired by the No.1 algorithm. The new algorithm should have an objective value lower than both algorithms.

Firstly, list the common ideas in the No.1 algorithm that may give good performances. Secondly, based on the common idea, describe the design idea based on the No.len(indivs) algorithm and main steps of your algorithm in one sentence. The description must be inside a brace. Next, implement it in Python as a function named 'select_next_node'.

This function should accept 4 input(s): 'current_node', 'destination_node', 'unvisited_nodes', 'distance_matrix'. The function should return 1 output(s): 'next_node'. The select_next_node function takes as input the current node, the destination_node, a set of unvisited nodes, and a distance matrix, and returns the next node to visit.

Do not give additional explanations.

- **Mutation Action m1**: Action m1 inputs a heuristic function with its description and code, it attempts to introduce more new mechanisms and new formulas or program segments to the input code through LLM.

**Prompt for Action m1**

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 one algorithm with its code as follows.:
# Its Description

# Its Python Code Implementation of A Function

Please create a new algorithm that has a different form but can be a modified version of the provided algorithm. Attempt to introduce more novel mechanisms and new equations or programme segments.

First, describe the design idea and main steps of your algorithm in one sentence. The description must be inside a brace outside the code implementation. Next, implement it in Python as a function named 'select_next_node'.

This function should accept 4 input(s): 'current_node', 'destination_node', 'unvisited_nodes', 'distance_matrix'. The function should return 1 output(s): 'next_node'. The select_next_node function takes as input the current node, the destination_node, a set of unvisited nodes, and a distance matrix, and returns the next node to visit.

Do not give additional explanations.

- **Mutation Action m2**: Action m2 also inputs the description and implementation of a heuristic function, it attempts to generate a heuristic function with different parameter settings through LLM.

**Prompt for Action m2**

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 one algorithm with its code as follows.:
# Its Description
# Its Python Code Implementation of A Function

Please identify the main algorithm parameters and help me in creating a new algorithm that has different parameter settings to equations compared to the provided algorithm.

First, describe the design idea and main steps of your algorithm in one sentence. The description must be inside a brace outside the code implementation. Next, implement it in Python as a function named 'select_next_node'.

This function should accept 4 input(s): 'current_node', 'destination_node', 'unvisited_nodes', 'distance_matrix'. The function should return 1 output(s): 'next_node'. The select_next_node function takes as input the current node, the destination_node, a set of unvisited nodes, and a distance matrix, and returns the next node to visit.

Do not give additional explanations.

- **Tree Path Reasoning Action s1**: Action s1 is a tree-special action that takes all diverse heuristic functions (with their codes, descriptions, and performances) on a leaf-to-root tree path as input. The following prompt asks the LLM to get a better heuristic function based on these inputted in-contexts.

**Prompt for Action s1**

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 as follows:
No.1 algorithm's description, its corresponding code and its objective value are:
# Its Description
# Its Python Code Implementation of A Function

Objective value: # The Objective Value of the heuristic algorithm with the python code on a evaluation dataset.
...
No.k algorithm's description, its corresponding code and its objective value are:
# Its Description
# Its Python Code Implementation of A Function
Objective value: # The Objective Value of the heuristic algorithm with the python code on a evaluation dataset.

Please help me create a new algorithm that is inspired by all the above algorithms with its objective value lower than any of them.

Firstly, list some ideas in the provided algorithms that are clearly helpful to a better algorithm. Secondly, based on the listed ideas, describe the design idea and main steps of your new algorithm in one sentence. The description must be inside a brace. Thirdly, implement it in Python as a function named 'select_next_node'.

This function should accept 4 input(s): 'current_node', 'destination_node', 'unvisited_nodes', 'distance_matrix'. The function should return 1 output(s): 'next_node'. The select_next_node function takes as input the current node, the destination_node, a set of unvisited nodes, and a distance matrix, and returns the next node to visit.

Do not give additional explanations.

### E.2. The Prompt of the Thought-Alignment Process

**The thought-alignment Prompt**

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 set of unvisited nodes, and a distance matrix, and returns the next node to visit.

Design Idea:
# The Design Idea

Code:
# The Python Code Implementation of A Function

The content of the Design Idea idea cannot fully represent what the algorithm has done informative. So, now you should re-describe the algorithm using less than 3 sentences. Hint: You should reference the given Design Idea and highlight the most critical design ideas of the code. You can analyse the code to describe which variables are given higher priorities and which variables are given lower priorities, the parameters and the structure of the code.

### E.3. Examples of LLM Outputs

This part provides two examples of LLM output in designing heuristics with MCTS-AHD. Every single LLM-based heuristic generation includes two LLM calls. The first call generates **the Python code of a heuristic function** with action prompts and the second call generates **linguistic descriptions** for heuristic functions with thought-alignment prompts.

**Example of LLM output #1: Designing TSP heuristics within the step-by-step solving framework.**

**First Call Input:**
The prompt for an MCTS-AHD action.
**First Call Output:**
{The new algorithm will utilize a hybrid approach that weighs the immediate distance (1.2) more heavily while still considering the connection strength (0.6), ensuring that we select paths that are both cost-effective in the short term and reasonable in the longer term by optimizing the total accumulated cost.}

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

    next_node = None
    min_combined_cost = float('inf')

    for candidate in unvisited_nodes:
        immediate_distance = distance_matrix[current_node][candidate]
        connection_strength = sum(distance_matrix[candidate][n] for n in
            unvisited_nodes if n != candidate) / (max(len(unvisited_nodes) - 1, 1))

        total_cost = immediate_distance
        current_candidate = candidate
        remaining_nodes = unvisited_nodes - {candidate}

        while remaining_nodes:
            best_next_node = min(remaining_nodes, key=lambda node: distance_matrix[
                current_candidate][node])
            total_cost += distance_matrix[current_candidate][best_next_node]
            current_candidate = best_next_node
            remaining_nodes.remove(best_next_node)

        total_cost += distance_matrix[current_candidate][destination_node]

        combined_cost = (1.2 * immediate_distance) + (0.6 / (connection_strength +
            1)) + total_cost

        if combined_cost < min_combined_cost:
            min_combined_cost = combined_cost
            next_node = candidate

    return next_node
```

**First Call Input:**
The prompt of the thought-alignment process.
**First Call Output:**
The algorithm utilizes a hybrid approach to solve the Traveling Salesman Problem by prioritizing immediate distance (with a weight of 1.2) over connection strength (with a weight of 0.6) to select the next node to visit from a set of unvisited nodes. It calculates a combined cost that considers the immediate distance to a candidate node, the average connection strength to remaining unvisited nodes, and the total cost of visiting all remaining nodes before returning to the destination. The algorithm systematically evaluates candidate nodes based on this combined cost, ensuring a balance between short-term efficiency and overall path optimization.

**Example of LLM output #2: Designing CAF for Cost-aware BO.**

**First Call Input:**
The prompt for an MCTS-AHD action.
**First Call Output:**
{The new algorithm evaluates the potential improvement over the best observed value while incorporating a compound risk factor based on both uncertainty and evaluation costs, adjusts the utility through a combination of historical performance scaling, and normalizes the results across the test inputs to favor those with high expected returns and low associated costs.}

```
import torch

def utility(train_x, train_y, best_x, best_y, test_x, mean_test_y, std_test_y,
    cost_test_y, budget_used, budget_total):
    # Calculate remaining budget
    remaining_budget = budget_total - budget_used

    # Adjust the mean based on uncertainty
    risk_factor = std_test_y / (std_test_y + 1e-10)
    adjusted_mean = mean_test_y * (1 - risk_factor)

    # Improvement over the best observed value
    improvement_score = adjusted_mean - best_y

    # Calculate cost-effectiveness with exponential decay based on evaluation costs
    cost_effectiveness = (improvement_score / (cost_test_y + 1e-10)) * torch.exp(-
        cost_test_y / (remaining_budget + 1e-10))

    # Historical risk/variance for budget scaling
    historical_variance = torch.var(train_y, unbiased=False)
    budget_scaling = remaining_budget / (remaining_budget + historical_variance + 1e
        -10)

    # Combine cost-effectiveness with budget scaling
    final_utility = cost_effectiveness * budget_scaling

    # Normalize the utility values
    utility_value = (final_utility - torch.min(final_utility)) / (torch.max(
        final_utility) - torch.min(final_utility) + 1e-10)

    return utility_value
```

**First Call Input:**
The prompt of the thought-alignment process.
**First Call Output:**
The algorithm evaluates potential improvements over the best observed value by leveraging a risk-adjusted mean of the predicted function, where uncertainty is factored in through the standard deviation. It incorporates cost-effectiveness by weighing the expected improvement against evaluation costs, applying an exponential decay influenced by the remaining budget. Finally, the utility values are scaled by historical variance to prioritize inputs that promise higher expected returns while accounting for their associated costs, ultimately normalizing the output to facilitate comparison across test inputs.

### E.4. Total Algorithm

Algorithm 1 provides a pseudo-code for the proposed MCTS-AHD method, which provides details of the flow and parameter settings. The procedure of MCTS-AHD mainly includes an initialization process (up to line 8) and a four-stage MCTS process (line 8 to line 35).

**Algorithm 1** MCTS-AHD: Monte Carlo Tree Search for Automatic Heuristic Design

1: **Input:** Evaluation dataset $D$, The number of initial nodes $N_I$, Maximal evaluation times $T$, Maximal tree depth $H = 10$, Action set $\{i1, e1, e2, m1, m2, s1\}$, The number of mutation in each expansion $k = 2$, The maximal number of references in action e1 $p$, UCT initial exploration parameter $\lambda_0 = 0.1$, Progressive widening parameter $\alpha = 0.5$.
2: **Output:** Code $C^*$ for the best found heuristic functions $h^*$.
3: Initialize a virtual root node $n_r$. *// The virtual root node does not contain any codes.*
4: Set $t \leftarrow 0$. *// $t$ represents the current evaluation times.*
5: Initialize a code and its description with Action i1, and get all other $N_I - 1$ initial nodes with Action e1.
6: $q_{max} \leftarrow -1e5$, $q_{min} \leftarrow 0$
7: Link all the $N_I$ nodes to the MCTS root.
8: Evaluating the newly generated codes with evaluation dataset $D$, setting the Q, N values for these nodes.
9: **while** $t \leq T$ **do**
10: $\quad \lambda \leftarrow \lambda_0 * \frac{T-t}{T}$ ***// Exploration-decay.***
11: $\quad$ **if** $\lfloor N(n_r)^\alpha \rfloor \geq |\text{Children}(n_r)|$ **then**
12: $\quad\quad$ Expand $n_r$ with action e1 and doing backpropagation. ***// Progressive Widening for the root node.***
13: $\quad\quad t \leftarrow t + 1$. *// Updating evaluation time $t$.*
14: $\quad$ **end if**
$\quad$ *// MCTS Selection: Selecting a node to expand.*
15: $\quad n_c \leftarrow n_r$ *// Selecting from root node.*
16: $\quad$ **while** $n_c$ is not a leaf and its depth is less than $H$ **do**
17: $\quad\quad n_c \leftarrow \arg\max_{c \in \text{Children}(n_c)} \left( \frac{Q(c) - q_{min}}{q_{max} - q_{min}} + \lambda \cdot \sqrt{\frac{\ln N(n_c + 1)}{N(c)}} \right)$
18: $\quad\quad$ **if** $\lfloor N(n_c)^\alpha \rfloor \geq |\text{Children}(n_c)|$ **then**
19: $\quad\quad\quad$ Expand $n_c$ with action e2 and doing backpropagation. ***// Progressive Widening for non-root nodes.***
20: $\quad\quad\quad t \leftarrow t + 1$. *// Updating evaluation time $t$.*
21: $\quad\quad$ **end if**
22: $\quad$ **end while**
23: $\quad$ *// MCTS Expansion: Add new nodes to the tree.*
$\quad$ Conducting action e2, s1, $k$ times of m1, and $k$ times of m2 to the $n_c$ by LLM.
24: $\quad$ *// MCTS Simulation:*
$\quad$ Evaluating the newly generated codes with evaluation dataset $D$, setting the Q, N value for these nodes.
$\quad$ *// MCTS Backpropagation:*
25: $\quad$ Updating the code $C^*$ if a better heuristic function is found.
26: $\quad t \leftarrow t + 2k + 2$. *// Updating evaluation time $t$.*
27: $\quad$ **for** $c \in \text{Children}(n_c)$ **do**
28: $\quad\quad q_{max} \leftarrow \max(q_{max}, Q(c))$, $q_{min} \leftarrow \min(q_{min}, Q(c))$ *// Updating $q_{max}$ and $q_{min}$.*
29: $\quad$ **end for**
30: $\quad$ **while** $n_c$ is not a leaf and its depth is less than $H$ **do**
31: $\quad\quad Q(n_c) \leftarrow \max_{c \in \text{Children}(n_c)} Q(c)$ *// Updating Q value.*
32: $\quad\quad N(n_c) \leftarrow N(n_c) + 2k + 2$ *// Updating visit times.*
33: $\quad\quad n_c \leftarrow \text{Father}(n_c)$
34: $\quad$ **end while**
35: **end while**
36: **Return** the best found code $C^*$ for heuristics.

# F. Experiment Details

## F.1. Designing Heuristics with the Step-by-step Construction Framework for ASP

We apply MCTS-AHD to another NP-hard CO problem ASP (Romera-Paredes et al., 2024). ASP constructs a set of n-dimensional vectors $\mathcal{A}$ (named an admissible set) in which each vector $\in \{0,1,2\}^n$ and the number of non-zero items in each vector is constrained to be $w$. ASP aims to maximize the size of the admissible set $\mathcal{A}$ with another certain constraint between the vectors in the admissible set (detailed in Appendix B.1). LLM-based AHD methods are responsible for designing a priority function: $\{0,1,2\}^n \to \mathbb{R}$, which is executed iteratively to construct $\mathcal{A}$ step by step. We use a single instance with $n$=15 and $w$=10 as the evaluation dataset $D$. The experiments in Table 7 test the heuristics on four different scales where MCTS-AHD also exhibits the best results compared to other existing LLM-based AHD methods in most test sets.

*Table 7.* Heuristics with the step-by-step construction framework for ASP. Each LLM-based AHD method is run three times for its average performance. The test set of each scale contains only one instance and we underline the in-domain scale. The best-performing method for each LLM model is shaded and the overall best result is in bold.

| Task | ASP | | | |
|---|---|---|---|---|
| Methods | $n$=12,$w$=7 | <u>$n$=15,$w$=10</u>: | $n$=21,$w$=15: | $n$=24,$w$=17: |
| Optimal | 792 | 3003 | 43596 | 237984 |
| LLM-based AHD: *GPT-3.5-turbo* | | | | |
| Funsearch | 612.0 | 2057.0 | 10664.0 | 37323.0 |
| EoH | 772.0 | 2759.0 | 30869.3 | 147323.0 |
| MCTS-AHD (Ours) | 784.0 | 2784.0 | 30608.0 | 150729.0 |
| LLM-based AHD: *GPT-4o-mini* | | | | |
| Funsearch | 622.0 | 2410.0 | 10758.7 | 43217.0 |
| ReEvo | **784.0** | 2733.0 | 25753.7 | 115709.0 |
| HSEvo | 744.0 | 2307.0 | 18756.3 | 81185.0 |
| EoH | 779.0 | 2776.0 | 32716.7 | 160024.0 |
| MCTS-AHD (Ours) | 775.0 | **2780.0** | **32900.3** | **163832.0** |

## F.2. Guided Local Search Framework

To evaluate the performance of MCTS-AHD in designing the penalty heuristics of GLS frameworks, we follow the framework in Ye et al. (2024a), using MCTS-AHD to design a function for generating the knowledge matrix of KGLS (Arnold et al., 2019) and employing 10 TSP200 instances for performance evaluations. The number of local search iterations in GLS is set to 1200 for both heuristic evolutions and tests. We compare MCTS-AHD to LLM-based AHD baselines, NCO methods with GLS frameworks, and the original KGLS with manually designed heuristic functions. As shown in Table 8, the proposed MCTS-AHD can refine the KGLS on a scale of 200 and 500. MCTS-AHD generally outperforms other LLM-based AHD methods under the black-box pre-trained LLM *GPT-4o-mini*.

*Table 8.* Designing heuristic functions in the GLS general framework for solving TSP. KGLS, NCO methods, and heuristics from LLM-based AHD allow 1200 iterations of local search. For NeuralGLS* and GNNGLS*, we use the results reported in Ye et al. (2024a). The table shows the gaps to optimal and each LLM-based AHD method is run three times for the average optimality gaps.

| TSP-GLS | | | | |
|---|---|---|---|---|
| N= | 100 | <u>200</u> | 500 | 1,000 |
| Optimal | 0.0000% | 0.0000% | 0.0000% | 0.0000% |
| KGLS | **0.0034%** | 0.2270% | 0.9578% | **1.5348%** |
| *NCO methods with the GLS general framework* | | | | |
| VRP-DACT | 1.7943% | 91.9267% | - | - |
| NeuOpt | 0.2950% | 0.9152% | - | - |
| NeuralGLS* | 0.470%* | 3.622%* | - | - |
| GNNGLS* | 0.705%* | 3.522%* | - | - |
| *LLM-based AHD: GPT-4o-mini* | | | | |
| EoH | 0.0065% | **0.2025%** | 0.9534% | 1.6083% |
| ReEvo | 0.0076% | 0.2210% | 0.9993% | 1.6155% |
| MCTS-AHD (Ours) | 0.0060% | 0.2106% | **0.9495%** | 1.5985% |

### F.3. P-values for Significance

Although running LLM-based AHD methods three times can reduce their deviation, in practice, the variance in the quality of the designed heuristics is still non-negligible. In this section, we introduce the p-value to demonstrate the significant advantage of MCT-AHD compared to the main LLM-based AHD baseline EoH. We employ both EoH and MCTS-AHD to design up to ten heuristics for a portion of CO problems considered in this paper, and the results and p-values for each run are shown in Table 9. In any of the four application scenarios, there is at least a 96% confidence in satisfying the hypothesis that MCTS-AHD leads compared to EoH.

*Table 9.* Up to ten runs of EoH and MCTS-AHD on a portion of NP-hard CO problems with step-by-step construction frameworks and ACO frameworks. "avg" in the Table below represents the average and "std" means the standard variant. The p-value is calculated with single-tailed t-tests.

| CO Problem | Methods | run1 | run2 | run3 | run4 | run5 | run6 | run7 | run8 | run9 | run10 | avg | std | p-value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| General Framework: Step-by-step Construction, LLM: *GPT-4o-mini* | | | | | | | | | | | | | | |
| TSP50 | EoH | 6.452 | 6.447 | 6.284 | 6.386 | 6.316 | 6.372 | 6.480 | 6.480 | 6.259 | 6.388 | 6.386 | 0.080 | |
| | MCTS-AHD | 6.174 | 6.156 | 6.347 | 6.356 | 6.285 | 6.274 | 6.257 | 6.365 | 6.289 | 6.302 | 6.280 | 0.071 | 0.002855655 |
| KP100, $W = 25$ | EoH | 40.229 | 40.231 | 40.232 | 40.231 | 40.234 | 40.264 | 40.240 | 40.235 | 40.229 | 40.235 | 40.236 | 0.010 | |
| | MCTS-AHD | 40.259 | 40.265 | 40.231 | 40.236 | 40.233 | 40.262 | 40.264 | 40.233 | 40.233 | 40.262 | 40.248 | 0.015 | 0.027524885 |
| General Framework: ACO, LLM: *GPT-4o-mini* | | | | | | | | | | | | | | |
| TSP50 | EoH | 5.827 | 5.825 | 5.831 | 5.830 | 5.828 | - | - | - | - | - | 5.828 | 0.003 | |
| | MCTS-AHD | 5.798 | 5.779 | 5.827 | 5.742 | 5.830 | - | - | - | - | - | 5.795 | 0.036 | 0.039230447 |
| MKP100, $m = 5$ | EoH | 23.149 | 23.133 | 23.136 | 23.311 | 23.266 | - | - | - | - | - | 23.199 | 0.083 | |
| | MCTS-AHD | 23.235 | 23.284 | 23.287 | 23.294 | 23.294 | - | - | - | - | - | 23.279 | 0.025 | 0.037268388 |

### F.4. Results on TSPLib: Compare to GP-based AHD Methods

We conduct tests on a well-known real-world TSP benchmark TSPLib (Reinelt, 1991) (in this paper we adopt instances with nodes less than 500) to compare the quality of LLM-based heuristic to a GP-based AHD method GHPP (Duflo et al., 2019). Christofides (Christofides, 2022), Greedy (Brecklinghaus & Hougardy, 2015), Nearest insertion, and nearest-greedy (Rosenkrantz et al., 1977) are famous manually designed heuristics for TSP. We use the reported results of these algorithms in the article (Duflo et al., 2019). Meanwhile, the results of GHPP also come from Duflo et al. (2019).

For LLM-based AHD methods EoH and MCTS-AHD, we use the best-performing heuristic function in their three heuristic evolution runs with *GPT-4o-mini* for their performances. We use the best-performing heuristic of ReEvo as their report in Ye et al. (2024a). We run the heuristics of EoH, ReEvo, and MCTS-AHD three times with different starting nodes for an average performance. As shown in Table 10, the heuristic from MCTS-AHD can surpass manually designed baselines, the GP-based AHD method GHPP, and LLM-based AHD baselines in the average optimality gap.

*Table 10.* Results of GP-based AHD method GPHH, LLM-based methods on designing heuristics for TSP with the step-by-step construction framework. Christofides, Greedy, Nearest insertion, and Nearest-greedy are manually designed heuristics for TSP where their results are also drawn from (Duflo et al., 2019). We report the optimality gap of each instance and heuristics designed by LLM-based AHD methods are run 3 times with different starting nodes for average performances. The leading LLM-designed heuristic on each instance is marked in shaded and the overall best heuristic is in bold.

| Instance | Christofides | Greedy | Nearest insertion | Nearest-greedy | GPHH-best | EoH | ReEvo | MCTS-AHD |
|---|---|---|---|---|---|---|---|---|
| ts225.tsp | 5.67% | **5.38%** | 19.93% | 16.82% | 7.71% | 5.57% | 6.56% | 10.84% |
| rat99.tsp | **9.43%** | 22.30% | 21.05% | 21.79% | 14.09% | 18.78% | 12.41% | 10.46% |
| bier127.tsp | 13.03% | 19.50% | 23.05% | 23.25% | 15.64% | 14.05% | 10.79% | **7.56%** |
| lin318.tsp | 13.80% | 18.75% | 24.44% | 25.78% | 14.30% | **14.03%** | 16.63% | 14.07% |
| eil51.tsp | 15.18% | 13.03% | 16.14% | 31.96% | 10.20% | 8.37% | **6.47%** | 15.98% |
| d493.tsp | **9.52%** | 16.68% | 20.39% | 24.00% | 15.58% | 12.41% | 13.43% | 11.73% |
| kroB100.tsp | **9.82%** | 16.59% | 21.53% | 26.26% | 14.06% | 13.46% | 12.20% | 11.43% |
| kroC100.tsp | 9.08% | 12.94% | 24.25% | 25.76% | 16.22% | 16.85% | 15.88% | **8.27%** |
| ch130.tsp | 10.09% | 28.40% | 19.21% | 25.66% | 14.77% | 12.26% | **9.40%** | 10.18% |
| pr299.tsp | **11.23%** | 31.42% | 25.05% | 31.42% | 18.24% | 23.58% | 20.63% | **11.23%** |
| fl417.tsp | 15.57% | 12.64% | 25.52% | 32.42% | 22.72% | 20.47% | 19.15% | **10.20%** |
| kroA150.tsp | 13.44% | 20.24% | 19.09% | 26.08% | 15.59% | 18.36% | 11.62% | **10.08%** |
| pr264.tsp | **11.28%** | 11.89% | 34.28% | 17.87% | 23.96% | 18.03% | 16.78% | 12.27% |
| pr226.tsp | 14.17% | 21.44% | 28.02% | 24.65% | 15.51% | 19.90% | 18.02% | **7.15%** |
| pr439.tsp | **11.16%** | 20.08% | 24.67% | 27.36% | 21.36% | 21.96% | 19.25% | 15.12% |
| Average Gap | 11.50% | 18.09% | 23.11% | 25.41% | 16.00% | 15.87% | 13.95% | **11.10%** |

### F.5. Compare to LLM-as-Optimizer Methods

As discussed in Appendix A, another LLM-based approach for CO problems, LLM-as-optimizer methods cannot achieve outstanding performance in large-scale instances. Here we compare this type of method with the proposed MCTS-AHD. As shown in Table 11, LLM-as-optimizer methods LEMA (Liu et al., 2024e) and OPRO (Yang et al., 2024) can provide better solutions in very-small-scale TSP20 instances, but as the scale increases to TSP50, these methods will fail to achieve convergence in LLM-based solution optimizations.

*Table 11.* Comparison of LLM-as-optimizer methods and MCTS-AHD designed heuristics. We display the optimality gap of MCTS-AHD on TSP20 and TSP50 test sets with 1,000 instances, respectively. Results of LEMA and OPRO are drawn from the original literature and the LLM for all methods in this table is *GPT-3.5-turbo*.

| Methods | LEMA* | OPRO* | MCTS-AHD(step-by-step construction) |
|---|---|---|---|
| TSP20 | 3.94% | 4.40% | 7.71% |
| TSP50 | - | 133.00% | 11.82% |

### F.6. Discussion: Ablation of Other Parameters

In 5.1, we have provided partial ablation experiments, focusing mainly on verifying that $\lambda - 0 = 0.1$ is a reasonable setting and validating the effectiveness of the three proposed components (Progressive Widening, Thought Alignment, and Exploration decay) and MCTS-AHD's expansion actions. MCTS-AHD also includes other parameters, such as the number of initial solutions $N_I$, the threshold parameter for progressive widening $\alpha$, and the number of actions m1 and m2 in each expansion $k$. We believe that MCTS-AHD is not sensitive to the settings of these parameters. We will also introduce the reasons for their default settings and demonstrate their flexibility through ablation experiments.

$N_I$ determines the number of initial nodes (i.e., heuristic samples), which affects the perception of heuristic space by LLM in the early stages of the heuristic function evolution. Table 12 shows that adjusting it to $N_I = 10$ has no significant effect on the results, indicating that using $N_I = 4$ is sufficient. Under the setting of evaluating $T$ times in total, progressive widening allows the maximum number of level-1 tree nodes connected to the root to be at most $\lfloor T^{\alpha} \rfloor$. Larger $\alpha$ settings will drive the MCTS tree shallow, affecting the quality of multi-hop reasoning brought by MCTS. Therefore, we choose to set $\alpha = 0.5$ to balance the depth of the MCTS tree and the importance of progressive widening. According to Table 12, changing this setting to $\alpha = 0.6$ does not result in significant effect degradation. The setting of $k = 2$ allows MCTS to utilize the randomness brought about by LLM in trying different search directions in each expansion, and setting it to $k = 1$ does not cause degradation in performance.

*Table 12.* Ablation on the number of initial solution $N_I$, the threshold parameter for progressive widening $\alpha$, the number of actions m1 and m2 in each expansion $k$.

| | TSP50 | KP100 |
|---|---|---|
| $N_I = 4$ (Default Setting in MCTS-AHD, 10 runs) | 10.661% | 0.059% |
| $N_I = 10$ | 10.333% | 0.048% |
| $\alpha = 0.5$ (Default Setting in MCTS-AHD, 10 runs) | 10.661% | 0.059% |
| $\alpha = 0.6$ | 11.487% | 0.045% |
| $k = 2$ (Default Setting in MCTS-AHD, 10 runs) | 10.661% | 0.059% |
| $k = 1$ | 10.927% | 0.044% |

### F.7. Discussion: The Advantage Scope of MCTS

MCTS-AHD shows greater strengths in application scenarios with a more complex heuristic space $H$ and tasks with more descriptions as knowledge.

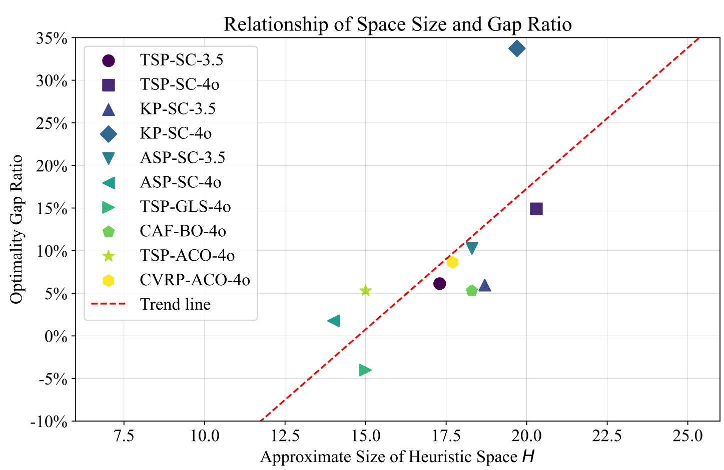



*Figure 5.* The relation of approximate heuristic space size $n$ and the optimality gap ratio between EoH and MCTS-AHD (i.e., $1 - \text{Gap}_{MCTS-AHD}/\text{Gap}_{EoH}$). We consider EoH as a baseline for its applicability in all tasks. We do not include tasks with unavailable optimal objective values (e.g., online BPP). The legends have been simplified, for example, TSP-GLS-4o represents designing GLS heuristics for TSP with *GPT-4o-mini*. SC in the figure is an abbreviation for step-by-step construction.

- **Better in Application Scenarios with a More Complex Heuristic Space** $H$. Each heuristic function can be expressed in the form of $a_1 f_1(x) + a_2 f_2(x) + a_3 f_3(x) + \ldots + a_n f_n(x)$, representing linear combinations of different sub-functions, where $x$ denotes the particular input ($\boldsymbol{ins}$) of the heuristic function $h$. The heuristic space $H$ to be explored for an application scenario can be defined as the set of all meaningful sub-functions, i.e., $H = \text{Span}\{f_1(x), \ldots\}$. To estimate the size of the set consisting of all meaningful sub-functions, we use Openai o1-mini-2024-09-12 to analyze the top10 heuristic functions of EoH and MCTS-AHD in their respective 3 runs (6 runs and 60 heuristic functions in total), ordering LLM to break functions down into linearly independent sub-functions and use the number of sub-functions $n$ to estimate the size of the heuristic space.

We hypothesize that for more complex heuristic spaces, MCTS-AHD can explore the heuristic space more comprehensively compared to population-based methods such as EoH. To test this hypothesis, in Figure 5, we plot the relation of estimated heuristic space size $n$ and the leads in optimality gap between MCTS-AHD and EoH (the y-axis in Figure, $1 - \text{Gap}_{MCTS-AHD}/\text{Gap}_{EoH}$). The results verify our hypothesis. As the trend line demonstrates, MCTS-AHD tends to achieve a more significant lead compared to the population-based baseline EoH in application scenarios with larger $n$. It indicates that MCTS-AHD might be more suitable for application scenarios with more complex heuristic spaces $H$.

- **Better in Application Scenarios with More Descriptions**. As shown in Table 13, MCTS-AHD demonstrates a significant decrease in effectiveness in black-box settings, taking the lead only in the TSP task. This suggests that MCTS-AHD will perform better in application scenarios with more descriptions (e.g.,white-box cases).

*Table 13.* Implementing MCTS-AHD on Black-box CO tasks with ACO general frameworks. We follow the settings of Ye et al. (2024a) in heuristic evolution and run each LLM-based method three times for average performance. The white-box results are the same as Table 2.

| | TSP | CVRP | MKP | Offline BPP |
|---|---|---|---|---|
| N= | $N$=50 | $N$=50, $C$=50 | $N$=100,$m$=5 | $N$=500,$C$=150 |
| Methods | Obj.↓ | Obj.↓ | Obj.↑ | Obj.↓ |
| ACO | 5.992 | 11.355 | 22.738 | 208.828 |
| DeepACO | 5.842 | 8.888 | 23.093 | 203.125 |
| White-box Setting: *GPT-4o-mini* | | | | |
| EoH | 5.828 | 9.359 | 23.139 | 204.646 |
| ReEvo | 5.856 | 9.327 | 23.245 | 206.693 |
| MCTS-AHD(Ours) | 5.801 | 9.286 | 23.269 | 204.094 |
| Black-box Setting: *GPT-4o-mini* | | | | |
| EoH | 5.831 | 9.401 | 23.240 | 204.615 |
| ReEvo | 5.860 | 9.404 | 23.196 | 206.021 |
| MCTS-AHD(Ours) | 5.830 | 9.444 | 23.191 | 205.375 |

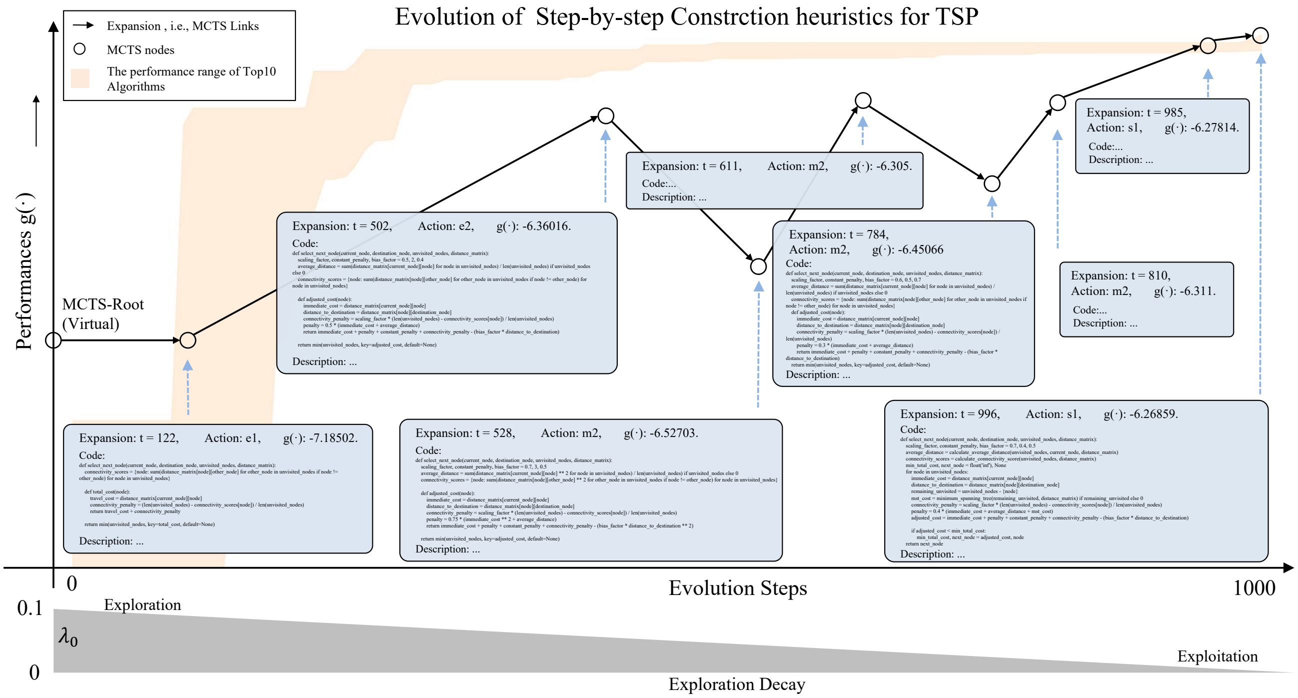
We can explain that MCTS-AHD's better performance in application scenarios with complex heuristic space is highly beneficial from its ability to explore complex function spaces and escape from local optima.

For its mediocre performance in black-box application scenarios, I believe this is mainly because compared to MCTS, which only performs limited expansion on each heuristic, the population structure is computationally intensive and often involves many rounds of LLM-based operations on a heuristic function in the population. So, the effectiveness of MCTS-AHD may require LLMs more on its generation quality in limited number of MCTS expansions. In contrast, black-box application scenarios make it difficult for LLMs to guarantee this condition, so these application scenarios will be tough for MCTS-AHD.

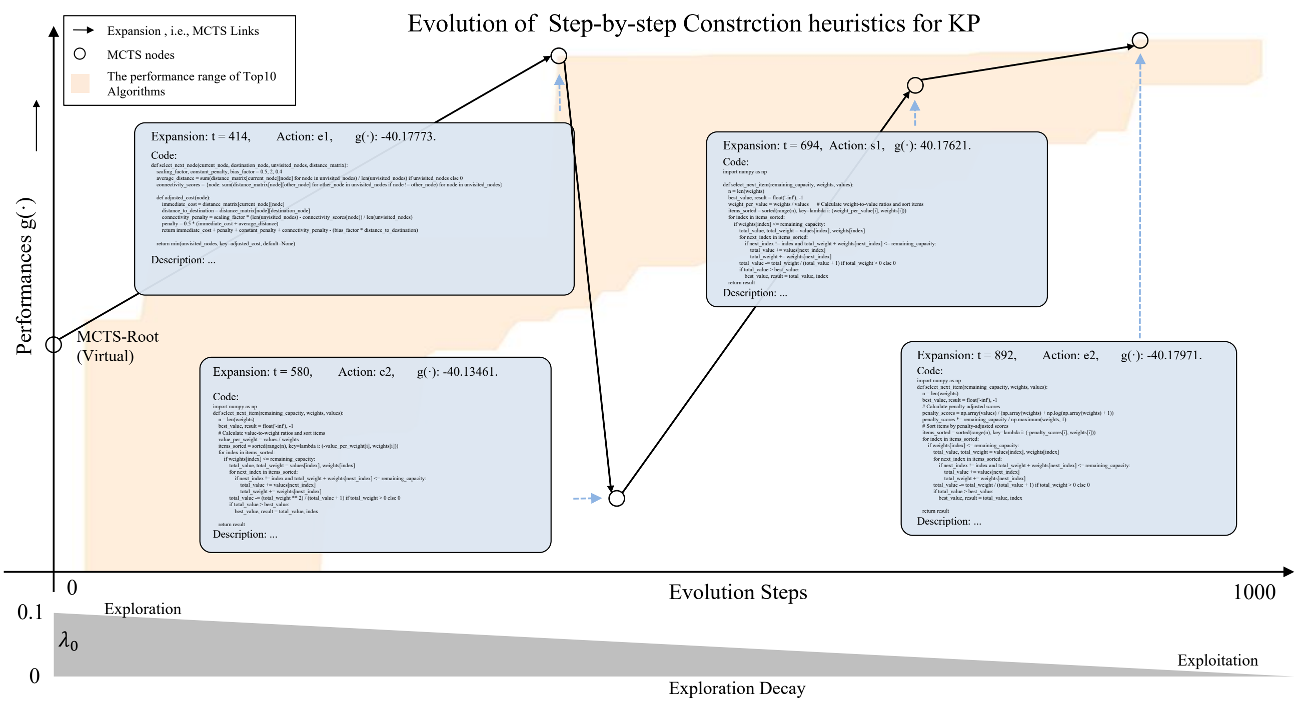
## G. Examples of Evolution

To visually demonstrate the workflow of MCTS-AHD and its ability to escape local optima, we provide two examples of the evolution of heuristic functions in two tasks, as shown in Figure 6.

The two examples of evolution clearly exhibit how MCTS-AHD jumps out of the local optimum. For example, in designing heuristics with a step-by-step construction framework for TSP, MCTS-AHD can expand potential child nodes from nodes (e.g., MCTS node with "Expansion: t=611") that are not among the top 10 optimal ones (the performance range of the top 10 optimal heuristics is the yellow shade), and ultimately reach the best heuristic. It reflects the superiority of employing MCTS as an optimization framework instead of population-based EC. Population-based LLM-based AHD methods such as EoH, ReEvo, and HSEvo lack consideration of worse-performing but potential heuristics. These algorithms will be obsessed with processing top-performance heuristic functions, and when the top 10 algorithms cannot be easily updated, these methods will trap the evolution of heuristics into local optima.




(a) An Example of MCTS-AHD Evolution on Designing Step-by-step Construction Heuristics for TSP




(b) An Example of MCTS-AHD Evolution on Designing Step-by-step Construction Heuristics for KP

*Figure 6.* Two examples of the heuristic evolution process of MCTS-AHD. We provide the performance function value $g(\cdot)$ of the heuristic function within each MCTS tree node. The "Action" in each node represents one of the six actions of expanding this node. Codes are simplified by OpenAI *GPT* to reduce the space occupations.

## H. Baselines & Licenses

**For the optimal value** in Table 1 and Table 8. We run LKH3 for TSP instances with adopting the commonly used setting in NCO methods (Kool et al., 2018). We set the LKH parameters RUNS = 10 and MAX_TRAILS = 1,0000. For KP instances, we use Google OR-Tools for the optimal value.

**For manually designed heuristics with general frameworks**, this paper includes Greedy Construct, the ACO algorithm (Dorigo et al., 2006), the KGLS algorithm (Arnold & Sörensen, 2019), and EI (Mockus, 1974), EIpu (Snoek et al., 2012), EI-cools (Lee et al., 2020a) for the CAF design task in BO. We implement KGSL by setting the heuristic matrix to the distance matrix, using the implementation of ACO in DeepACO (Ye et al., 2024b), and use the results reported in Yao et al. (2024c) for EI, EIpu, and EI-cools.

**For NCO baselines**, this article adopts POMO, DeepACO, VRP-DACT, and NeuOpt. For a fair comparison, in Table 1, the reported POMO solutions are from a single start and generated without augmentations. The maximum operation on a solution is set to $T = 1200$ for VRP-DACT and NeuOpt.

**For AHD baselines**, on TSPLib, we adopt the results of the GHPP reported in (Duflo et al., 2019).

**For LLM-based AHD baselines**, this article considers Funsearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024b), ReEvo (Ye et al., 2024a), and HSEvo (Dat et al., 2024), we follow all their parameter settings in evolutions (e.g., setting population size $M = 20$ for online BPP and $M = 10$ for other tasks). We try not to introduce too much external expert information. However, in implementations, Funsearch needs to maintain 10 relatively independent multiple populations from a certain inferior seed heuristic function. ReEvo relies more on the seed heuristic function, and the ReEvo method will become unavailable when all individuals in the elite population have the same objective function value. So, in some application scenarios (e.g., step-by-step construction for the KP task), it is too hard to find a good seed heuristic function that ensures the availability of the algorithm while trying not to provide too much external knowledge.

In experiments, we use the same seed functions for baselines (Funsearch, ReEvo, and HSEvo), using seed functions proposed in Ye et al. (2024a) for ACO frameworks and GLS frameworks, random selection functions for step-by-step constructing TSP and ASP, and the best-known function proposed in Romera-Paredes et al. (2024) for online BPP. The proposed MCTS-AHD, exhibits better applicability and superior performance in most application scenarios, without the requirement of seed function design. In our seed function settings, ReEvo and its follow-up method HSEvo are not available in some application scenarios (e.g., KP, ACO), where we do not report their effects.

### H.1. License

The license and URL of baselines are listed in Table 14.

*Table 14.* A summary of licenses.

| Resources | Type | License | URL |
|---|---|---|---|
| LKH3 | Code | Available for academic research use | `http://webhotel4.ruc.dk/~keld/research/LKH-3/` |
| OR-Tools | Code | MIT License | `https://developers.google.com/optimization/pack/knapsack?hl=zh-cn` |
| POMO | Code | Available online | `https://github.com/yd-kwon/POMO/tree/master` |
| DeepACO | Code | MIT License | `https://github.com/henry-yeh/DeepACO` |
| VRP-DACT | Code | MIT License | `https://github.com/yining043/VRP-DACT` |
| NeuOpt | Code | MIT License | `https://github.com/yining043/NeuOpt` |
| Funsearch | Code | Apache License | `https://github.com/google-deepmind/funsearch` |
| EoH | Code | MIT License | `https://github.com/FeiLiu36/EoH/tree/main` |
| ReEvo | Code | MIT License | `https://github.com/ai4co/reevo` |
| HSEvo | Code | Available online | `https://github.com/datphamvn/HSEvo` |
| Synthetic problems | Dataset | Available Online | `https://github.com/FeiLiu36/EoH/tree/main/examples/user_bo_caf` |