Generalizable Heuristic Generation Through Large Language Models with Meta-Optimization

Yiding Shi[1], **Jianan Zhou**[1*], **Wen Song**[2*], **Jieyi Bi**[1], **Yaoxin Wu**[3], **Jie Zhang**[1]

[1]Nanyang Technological University
[2]Shandong University
[3]Eindhoven University of Technology

Abstract

Heuristic design with large language models (LLMs) has emerged as a promising approach for tackling combinatorial optimization problems (COPs). However, existing approaches often rely on manually predefined evolutionary computation (EC) optimizers and single-task training schemes, which may constrain the exploration of diverse heuristic algorithms and hinder the generalization of the resulting heuristics. To address these issues, we propose Meta-Optimization of Heuristics (MoH), a novel framework that operates at the optimizer level, discovering effective optimizers through the principle of meta-learning. Specifically, MoH leverages LLMs to iteratively refine a meta-optimizer that autonomously constructs diverse optimizers through (self-)invocation, thereby eliminating the reliance on a predefined EC optimizer. These constructed optimizers subsequently evolve heuristics for downstream tasks, enabling broader heuristic exploration. Moreover, MoH employs a multi-task training scheme to promote its generalization capability. Experiments on classic COPs demonstrate that MoH constructs an effective and interpretable meta-optimizer, achieving state-of-the-art performance across various downstream tasks, particularly in cross-size settings.

1 Introduction

Heuristics have long been integral to solving combinatorial optimization problems (COPs), offering practical and efficient approaches when exact methods become computationally intractable due to their exponential time complexity. Over the past few decades, substantial progress has been achieved in human-designed heuristics. Notable examples include the Lin-Kernighan Heuristic (LKH) [45] for the Traveling Salesman Problem (TSP) and the Best Fit heuristic [36] for the Bin Packing Problem (BPP). However, developing effective heuristics for COPs typically requires an in-depth understanding of each problem's unique structure and the specialized expertise to craft suitable heuristic strategies. As a result, traditional approach to heuristic design is both time-intensive and significantly dependent on expert knowledge. This underscores the growing demand for more powerful approaches to accelerate the development of effective heuristics for COPs.

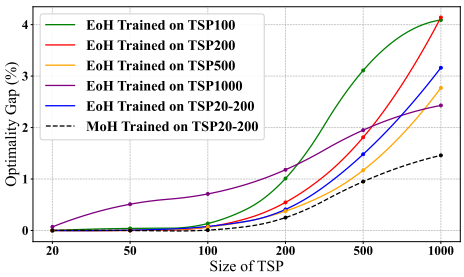

Figure 1: Generalization performance of the evolved improvement heuristics for TSP.

*Corresponding authors.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: DiffCoALG: Differentiable Learning of Combinatorial Algorithms.

With the explosive advancements of large language models (LLMs) in recent years, the landscape of heuristic design has undergone a transformative shift [33, 48]. A prominent trend involves leveraging LLMs to generate effective heuristics aimed at solving NP-hard COPs. Specifically, these methods typically utilize in-context learning to prompt LLMs to produce heuristics, which subsequently become integral components of (meta-)heuristic or learning-based solvers. [58] first demonstrated the feasibility of applying LLMs to heuristic design in this domain. Building on this foundation work, recent approaches have increasingly integrated LLMs with evolutionary computation (EC), giving rise to LLM-EC frameworks [47, 78, 77, 18, 81]. These methods enhance heuristic design by using LLMs to carry out evolutionary operations like crossover and mutation to evolve heuristics.

Despite achieving promising results, existing LLM-EC approaches face two limitations. First, their search space is constrained by manually designed, predefined EC optimizers (e.g., a fixed workflow of crossover followed by mutation), which may restrict the exploration of diverse heuristics and ultimately hinder the discovery of more powerful heuristics [18]. Second, their optimization process is only designed for a single task (i.e., a fixed-size COP), which may limit the generalization of the evolved heuristics. Fig. 1 illustrates the generalization performance of EoH [47], a representative LLM-EC approach, in optimizing improvement heuristics for TSP under various training settings. The results indicate a significant generalization challenge, as performance gaps widen with increasing problem size. Although incorporating cross-size datasets during training can partially mitigate this issue, the overall performance remains suboptimal on large problem sizes (i.e., different tasks).

To overcome these inherent limitations, we introduce Meta-Optimization of Heuristics (MoH), a novel framework leveraging the in-context reasoning and refinement capabilities of LLMs [27, 80] to automate optimizer design. In this paper, *optimizers* are algorithms, such as traditional EC frameworks, that are used to generate or refine *heuristics* for COPs to improve solution quality, whereas *meta-optimizers* are higher-level procedures that adapt and enhance these optimizers. Technically, MoH implements an iterative meta-optimization module within a multi-task framework to encourage both exploration and generalization. At each iteration, the meta-optimizer generates a diverse population of candidate optimizers through (self-)invocation. The most promising optimizer, evaluated by its effectiveness on downstream tasks in optimizing task-specific heuristics, is selected to become the meta-optimizer in the subsequent iteration. By doing so, the optimizers are improved to generate more effective heuristics (see Fig. 2). With its innovative meta-optimization, MoH extends beyond traditional fixed EC optimization frameworks, facilitating broader exploration of the heuristic search space and operating at a higher abstraction level than existing approaches.

Our contributions are summarized as follows: 1) We propose MoH, a novel framework that highlights meta-optimization for producing effective COP heuristics. MoH enables broader heuristic exploration by autonomously discovering novel optimization strategies through an iterative meta-optimization module, thereby addressing inherent limitations of existing LLM-EC approaches. 2) We position MoH within a multi-task training framework to enhance its generalization capability to unseen tasks. 3) Extensive experiments across multiple heuristic algorithms and classical COPs demonstrate that MoH is able to generate effective and interpretable meta-optimizers that consistently outperform baselines. Notably, the resulting heuristics exhibit strong performance on large COP instances.

2 Methodology

An overview of MoH is shown in Fig. 2, which features a two-level optimization process: *an outer loop for optimizer design* and *an inner loop for heuristic design.* Inspired by recent advances in LLMs [83, 80], MoH aims to construct a meta-optimizer capable of generating novel and effective optimization strategies and improve heuristics on downstream tasks concurrently. Concretely, in the outer loop, the meta-optimizer produces a diverse population of candidate optimizers. Then, each generated optimizer is leveraged in the inner loop to evolve task-specific heuristics for downstream tasks. After evaluating the heuristics on the validation dataset, the candidate optimizer with the highest utility score is selected as the new meta-optimizer for the next iteration, enabling MoH to iteratively discover novel and increasingly effective optimization strategies. Moreover, MoH is inherently suited for a multi-task training setting by maintaining diversity among tasks, thereby enhancing its ability to explore a broader range of heuristics, leading to improved performance across diverse tasks. Examples of seed (or initial) and generated optimizers are provided in Appendix E. In the following, we present the technical details of the proposed MoH framework.

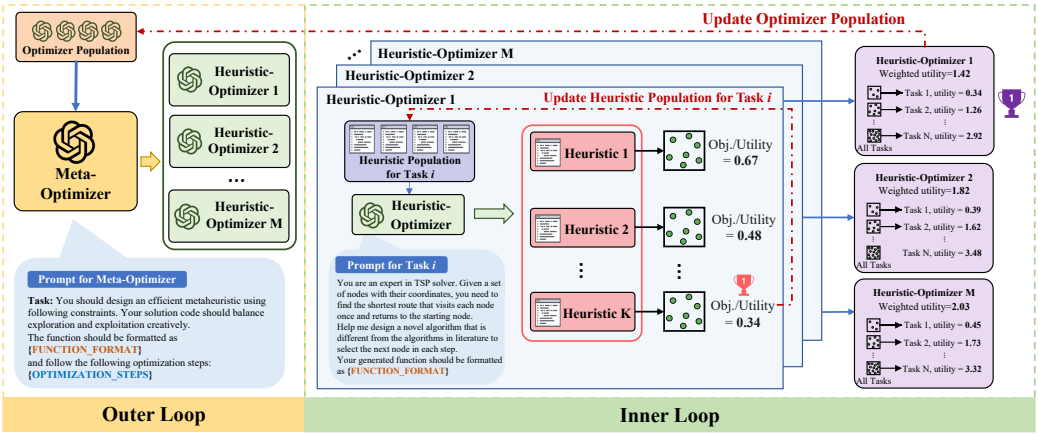

Figure 2: Overview of MoH. In iteration t, the current meta-optimizer $\mathcal{I}^*_{t-1}$ generates M candidate optimizers in the *outer loop*. Each candidate optimizer is then evaluated through the *inner loop*, where it generates K heuristics that are applied to N downstream tasks. For each task, the best heuristic is selected, and its utility contributes to the overall utility of the optimizer. After aggregating utility scores across all tasks, the optimizer with the highest utility is selected as the new meta-optimizer $\mathcal{I}^*_t$.

2.1 Problem Formulation

Suppose there are N downstream tasks, each corresponding to a heuristic design (or optimization) task for a COP. For each task i, let $h_i^{\mathcal{I}}$ denote the heuristic found by the optimizer $\mathcal{I}$, $\mathcal{D}_i$ the validation dataset, and $U_i(h_i^{\mathcal{I}}, \mathcal{D}_i)$ the utility function evaluating the performance of $h_i^{\mathcal{I}}$ on $\mathcal{D}_i$. The objective of *heuristic design* is to discover the best heuristic $\tilde{h}_i^{\mathcal{I}}$ using the optimizer $\mathcal{I}$ as follows,

$$\tilde{h}_i^{\mathcal{I}} = \arg\max_{h_i^{\mathcal{I}} \in \mathbb{H}_i} U_i(h_i^{\mathcal{I}}, \mathcal{D}_i), \tag{1}$$

where $\mathbb{H}_i$ denotes the heuristic search space, comprising all possible heuristics for the task i. The utility function is defined as the negative of the solution optimality gap. Most studies integrate EC as the optimizer $\mathcal{I}$ within LLMs to perform heuristic design. While these LLM-EC approaches offer a certain degree of flexibility, they struggle to effectively explore the huge heuristic search space due to the rigid structure of the fixed optimizer $\mathcal{I}$. Additionally, their heuristic design process necessitates separate training for each task i, making it computationally expensive. An alternative is to incorporate diverse instances from N tasks into the training dataset. However, this simple data mixture results in suboptimal performance, as a single generated heuristic struggles to adapt effectively across different tasks, consistent with the No Free Lunch Theorem [73].

To address the limitations, MoH directly searches for optimizers rather than relying on a fixed one, i.e., the *optimizer design* process. The objective of optimizer design is to discover the best optimizer $\tilde{\mathcal{I}}$, formally defined as:

$$\mathcal{I}^* \leftarrow \tilde{\mathcal{I}} = \arg\max_{\mathcal{I}} \sum_{i=1}^{N} w_i \cdot U_i(\tilde{h}_i^{\mathcal{I}}, \mathcal{D}_i), \text{ with } \tilde{h}_i^{\mathcal{I}} = \arg\max_{h_i^{\mathcal{I}} \in \mathbb{H}_i} U_i(h_i^{\mathcal{I}}, \mathcal{D}_i), \tag{2}$$

where $\mathcal{I}^*$ is the meta-optimizer, and w_i is the task weight. In essence, MoH extends Eq. (1) by introducing an outer loop for meta-optimization. In this outer loop, the meta-optimizer $\mathcal{I}^*$ produces a population of candidate optimizers through (self-)invocation. The best optimizer $\tilde{\mathcal{I}}$, as evaluated by the optimizer utility function $\mathbf{U}(\tilde{\mathcal{I}}) = \sum_{i=1}^{N} w_i \cdot U_i(\tilde{h}_i^{\tilde{\mathcal{I}}}, \mathcal{D}_i)$, is then selected to serve as the new meta-optimizer $\mathcal{I}^*$ in the next iteration.

2.2 Overall Workflow

We summarize the MoH training workflow in Alg. 1 and detail each step as follows. We initialize each downstream heuristic design task i with a heuristic population $\mathcal{H}_i$. This is achieved by prompting LLMs to generate diverse heuristic ideas in natural language, accompanied by their corresponding

code implementations. We also initialize an optimizer population $\mathcal{P}$ using a given seed optimizer (see Fig. 8), which serves as the starting point and reference baseline for subsequent iterations. Concretely, at iteration t, the current meta-optimizer $\mathcal{I}_{t-1}^*$ is used to generate a set of candidate optimizers $\{\mathcal{I}_t^1, \ldots, \mathcal{I}_t^M\}$. This generation is accomplished via (self-)invocation of $\mathcal{I}_{t-1}^*$ using LLMs, with prompts constructed from the information in the optimizer population $\mathcal{P}$. Then, each candidate optimizer $\mathcal{I}_t^j$ employs its own LLM-generated optimization strategy to evolve heuristics across all downstream tasks, resulting in N populations of improved heuristics $\{\{h_{i,1}^{\mathcal{I}_t^j}, \ldots, h_{i,K}^{\mathcal{I}_t^j}\}\}_{i=1}^N$. After evaluation, the best heuristic for each task is collected, yielding $\{\tilde{h}_1^{\mathcal{I}_t^j}, \ldots, \tilde{h}_N^{\mathcal{I}_t^j}\}$. The utility score of each candidate optimizer is thus calculated as $\mathbf{U}(\mathcal{I}_t^j) = \sum_{i=1}^N w_i \cdot U_i(\tilde{h}_i^{\mathcal{I}_t^j}, D_i)$. The candidate optimizer with the highest utility score is selected as the meta-optimizer $\mathcal{I}_t^*$ for the next iteration. More details on population management can be found in Section 2.3. During inference, the meta-optimizer $\mathcal{I}_T^*$ can be deployed on novel tasks that differ from those encountered during training, such as tasks with larger problem sizes. By performing several rounds of heuristic design using $\mathcal{I}_T^*$, MoH yields an improved, near-optimal heuristic tailored to the new task.

2.3 Detailed Implementation

As key subroutines of MoH, we further elaborate on the optimizer and heuristic generation (i.e., step a in the heuristic and optimizer design processes in Alg. 1). The main entities are as follows.

Individual and Population Structure. An individual is defined as a structured entity comprising three components: 1) a code implementation (String), 2) a high-level natural language description of the core strategy (String), and 3) a utility score reflecting its performance (Float). This unified format is adopted for both heuristics and optimizers. To ensure diversity and stability, we preserve a population with 10 individuals for both heuristic populations $\mathcal{H}$ and optimizer population $\mathcal{P}$.

Population Management. Individuals in the population are ranked by their utility scores. When a new candidate arrives, it is compared to the current worst-performing individual. If the candidate's utility is higher, it replaces that individual. After insertion, the population is re-sorted to maintain the utility-based ranking. This structure is efficiently handled using a heap.

Optimizer Signature

```
def optimize_algorithm(
    population: dict,
    utility: callable[[dict],
        float],
    language_model: class '
        LanguageModel'
    subtask_prompt: str,
    subtask: str
) -> Tuple[str, str, float]:
```

Figure 3: Optimizer signature implemented by LLMs to iteratively generate or refine optimization strategies.

Optimizer Signature. As shown in Fig. 3, the optimizer $\mathcal{I}$ is formatted as a callable function that takes the following inputs: 1) *population:* the population structure defined above, 2) *utility:* a utility function that evaluates the performance of an individual and returns its utility score, 3) *language_model:* an LLM for generating heuristics or optimizers, 4) *subtask_prompt:* a task-specific prompt to guide the optimization, and 5) *subtask:* a string specifying the name of the task. The optimizer returns the best individual discovered during the optimization process. Notably, this optimizer function is designed to support recursive invocation, allowing it to take its own implementation as input, such as through the population parameter in the first outer loop iteration.

Optimizer Generation Procedure. The optimizer generation procedure in outer loop iteration t follows the standardized steps below: 1) *Individual Selection:* The current meta-optimizer $\mathcal{I}_{t-1}^*$ uses its LLM-generated strategy to select promising candidate optimizers from the optimizer population $\mathcal{P}$. This step aims to balance the exploitation of high-utility individuals with the exploration of diverse candidates. 2) *Idea Generation:* The iterative improvement of algorithms generated by LLMs critically depends on algorithmic reasoning articulated in natural language, as evidenced by [69]. Consequently, the meta-optimizer is encouraged to prompt LLMs to propose exploratory or refinement ideas based on the optimizers selected in the first step. 3) *Implementation Generation:* Guided by the generated ideas and task-specific prompts, the LLM refines or generates new code implementations of selected optimizers through (self-)invocation of $\mathcal{I}_{t-1}^*$, producing a set of candidate optimizers $\{\mathcal{I}_t^1, \ldots, \mathcal{I}_t^M\}$. Each candidate is evaluated using the optimizer utility function $\mathbf{U}(\cdot)$, and the best-performing optimizer is selected as the new meta-optimizer $\mathcal{I}_t^*$ for the next iteration.

Note that our optimizer structure enables flexible exploration of novel optimization strategies. While the specific behavior of a generated optimizer may vary depending on prior optimizers, prompts, and LLM versions, their procedures generally adhere to the above three steps. Appendix E presents examples of LLM-generated optimizers, some of which resemble traditional metaheuristics, while others exhibit hybrid or unconventional strategies.

Heuristic Generation Procedure. Given an optimizer $\mathcal{I}_t^j$ generated by the meta-optimizer $\mathcal{I}_{t-1}^*$ in outer loop iteration t, the inner loop heuristic generation procedure for downstream task i follows the standardized steps below: 1) *Individual Selection:* The optimizer $\mathcal{I}_t^j$ employs its LLM-generated strategy to select promising candidate heuristics from $\mathcal{H}_i$ for evolution. 2) *Idea Generation:* The optimizer $\mathcal{I}_t^j$ prompts LLMs to propose exploratory or refinement ideas based on the heuristics selected in the first step. 3) *Implementation Generation:* Guided by the generated ideas and task-specific prompts, the LLM generates or refines heuristic implementations, resulting in a set of candidate heuristics $\{h_{i,1}^{\mathcal{I}_t^j}, \ldots, h_{i,K}^{\mathcal{I}_t^j}\}$. Each candidate is evaluated using its corresponding utility function $U_i(\cdot)$, and the heuristic population $\mathcal{H}_i$ is updated thereafter.

Despite their procedural similarities, the key differences between optimizer generation in the outer loop and heuristic generation in the inner loop are as follows: 1) *Optimization Target:* The outer loop focuses on generating optimizers using the meta-optimizer, whereas the inner loop applies each generated optimizer to improve heuristics across all downstream tasks. 2) *Invocation Frequency:* In the outer loop, the meta-optimizer is invoked once per iteration to generate M candidate optimizers. In contrast, during the inner loop, each optimizer is individually applied to every downstream task to generate K heuristics. Consequently, the invocation frequency in the inner loop is higher than in the outer loop. Detailed prompts for optimizer and heuristic generation can be found in Appendix D.

3 Experiments

3.1 Setup

We conduct extensive experiments to optimize various heuristic algorithms on classical COP benchmarks, including TSP and online BPP. Additional results on other problem benchmarks (e.g., CVRP, BPP) are provided in Appendix C. All experiments are conducted on servers with NVIDIA GeForce RTX 4090 GPUs and AMD Ryzen Threadripper PRO 7975WX CPU at 4GHz.

Heuristic Settings. 1) *Constructive Heuristic for TSP:* In constructive heuristics, a solution is built incrementally, starting from a random node and iteratively selecting the next promising node based on a predefined rule. The selected node is then appended to the current route to form a valid tour step by step. Since constructive heuristics focus on local optimization at each step rather than the global optimum, their performance is often suboptimal compared to other heuristic algorithms. 2) *Improvement Heuristic for TSP:* Guided Local Search (GLS) [68] is a metaheuristic that penalizes frequently used edges in local optima, steering the search away from less promising regions. In specific, it modifies the cost landscape by adjusting the distance matrix, adding penalties to certain edges to prevent their repeated selection in subsequent iterations. In our experiment, we compare two GLS implementations from [47] and [78]. The implementation in [47] follows a standard GLS approach, combining a basic local search method with dynamic edge penalties to guide the search. In contrast, the approach in [78] aligns more closely with Knowledge-Guided Local Search (KGLS) [7], incorporating domain-specific knowledge from the distance matrix to enhance the standard GLS framework. In our experiments, we use GLS and KGLS to represent two different settings.

Baselines. 1) *Traditional methods:* We employ Concorde [5] and OR-Tools [24] to solve TSP, and compare with classic heuristics, including Nearest Neighbor for TSP. For OR-Tools, we use guided local search as the local search strategy. The time limit for solving each TSP instance is set to 20s for problem size ≤ 100 and 40s for problem size ≥ 200. 2) *LLM-based methods:* We compare MoH with five representative approaches: FunSearch [58], EoH [47], ReEvo [78], HSEvo [18], and MCTS-AHD [81]. We rerun their publicly available implementations in our training settings, as detailed below. 3) *Neural methods:* We also benchmark against neural solvers, such as POMO [41] and LEHD [49], with results reported in Appendix C.

Training and Inference. To ensure a fair comparison, all LLM-based methods are trained under identical experimental conditions for each problem setting and evolved without relying on any

Table 1: Results for constructive and improvement heuristics on TSP.

Methods	Train								Generalization				Average Gap
	20		50		100		200		500		1000		
	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap	
Concorde	3.840	-	5.715	-	7.766	-	10.679	-	16.519	-	23.104	-	-
OR-Tools	3.840	0.000%	5.715	0.001%	7.772	0.089%	10.944	2.478%	17.259	4.479%	24.262	5.011%	2.010%
Nearest Neighbor	4.602	19.806%	7.055	23.406%	9.636	24.072%	13.374	25.228%	20.691	25.252%	28.990	25.474%	23.873%
Constructive Heuristic													
Funsearch	4.261	11.000%	6.523	14.162%	9.018	16.109%	12.615	18.143%	19.531	18.242%	27.571	19.332%	16.165%
EoH	4.204	9.408%	6.402	12.007%	8.774	12.974%	12.233	14.548%	19.029	15.196%	26.890	16.390%	13.420%
ReEvo	4.197	9.250%	6.399	11.966%	8.786	13.133%	12.217	14.403%	19.035	15.232%	26.818	16.076%	13.343%
HSEvo	4.108	6.897%	**6.280**	**9.881%**	8.705	12.102%	12.208	14.320%	19.550	18.349%	27.431	18.727%	13.379%
MCTS-AHD	4.107	6.882%	6.332	10.807%	8.735	12.499%	12.165	13.921%	19.036	15.240%	26.814	16.060%	12.568%
MoH (Ours)	**4.104**	**6.837%**	6.280	9.893%	**8.654**	**11.444%**	**12.100**	**13.307%**	**18.869**	**14.224%**	**26.581**	**15.049%**	**11.792%**
Improvement Heuristic													
EoH-GLS	3.840	0.000%	5.715	0.000%	7.768	0.024%	10.716	0.342%	16.714	1.176%	23.747	2.781%	0.721%
HSEvo-GLS	3.840	0.000%	5.715	0.000%	7.768	0.028%	10.715	0.328%	16.729	1.266%	23.719	2.660%	0.714%
MoH-GLS (Ours)	**3.840**	**0.000%**	**5.715**	**0.000%**	**7.767**	**0.012%**	**10.711**	**0.291%**	**16.674**	**0.936%**	**23.445**	**1.476%**	**0.453%**
ReEvo-KGLS	3.840	0.000%	5.715	0.000%	7.766	0.003%	10.704	0.221%	16.681	0.976%	23.473	1.595%	0.466%
HSEvo-KGLS	3.840	0.000%	5.715	0.000%	7.767	0.004%	10.704	0.221%	16.678	0.958%	23.478	1.615%	0.466%
MCTS-AHD-KGLS	3.840	0.000%	5.715	0.000%	7.767	0.006%	10.702	0.204%	16.662	0.867%	23.425	1.389%	0.411%
MoH-KGLS (Ours)	**3.840**	**0.000%**	**5.715**	**0.000%**	**7.766**	**0.002%**	**10.699**	**0.177%**	**16.652**	**0.805%**	**23.419**	**1.363%**	**0.391%**

predefined seed heuristic. In the TSP scenario, all methods are trained on cross-size datasets comprising four tasks: TSP20, 50, 100 and 200, and generalized to larger instances of sizes 500 and 1000. During training, we fix the number of outer loop iterations to $T = 10$ and maintain a population size of 10 for both optimizer and heuristic populations. We control the computational budget of each method by limiting the number of heuristic evaluations to 1,000. A detailed analysis of computational costs is provided in Appendix C. At inference stage, the trained meta-optimizer is executed for 10 iterations. We use 128 instances for TSP heuristic evaluation. All results reported in Table 1 reflect the average performance over the test dataset of the best-performing heuristic identified across three independent runs. All experiments use GPT 4o-mini (2024-07-18) as the primary LLM.

3.2 Empirical Result

Table 1 presents a comprehensive comparison of our proposed MoH against several baselines on TSP. The table includes results for both constructive and improvement heuristics across TSP20-1000 instances. The optimality gap is calculated as the difference in cost between each heuristic's solution and the optimal solution, obtained using the Concorde solver [5]. In the constructive heuristic setting, MoH achieves the lowest average optimality gap of 11.792%, significantly outperforming existing LLM-based approaches. In the improvement heuristic setting, we evaluate MoH using both GLS and KGLS variants. Our approach consistently achieves the lowest optimality gap of 0.391%, demonstrating superior solution quality across various TSP instances. Moreover, MoH demonstrates strong generalization performance on large-scale instances acorss both settings. These results confirm the effectiveness and adaptability of our approach in both heuristic categories and across different data regimes. Additional results on TSPLib [57] are presented in Appendix C.

4 Conclusion

In this paper, we propose a novel MoH framework, which leverages LLMs to generate effective meta-optimizers for improving COP heuristics. MoH extends the heuristic design paradigm by incorporating an outer loop for optimizer design and employs a multi-task scheme to improve generalization and enable broader heuristic exploration. Experimental results demonstrate that heuristics discovered by MoH outperform both classical heuristics and existing LLM-based approaches. We believe MoH offers a new perspective on generating promising heuristics, with the potential to surpass human-designed ones in solving NP-hard COPs. We acknowledge certain limitations of MoH, such as the search efficiency. The outer-loop and multi-task optimization inherently increase computational complexity, highlighting the need for more efficient search strategies. Additionally, while our current scope focuses on classical COPs, MoH has the potential to address a broader range of COPs and even other classes of optimization problems, which we leave for future work.

References

[1] Henrik Abgaryan, Ararat Harutyunyan, and Tristan Cazenave. Llms can schedule. *arXiv preprint arXiv:2408.06993*, 2024.

[2] Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (mi) lp solvers and large language models. *arXiv preprint arXiv:2402.10172*, 2024.

[3] Anton Andreychuk, Konstantin Yakovlev, Aleksandr Panov, and Alexey Skrynnik. MAPF-GPT: Imitation learning for multi-agent pathfinding at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 23126–23134, 2025.

[4] Spyros Angelopoulos, Shahin Kamali, and Kimia Shadkami. Online bin packing with predictions. *Journal of Artificial Intelligence Research*, 78:1111–1141, 2023.

[5] David Applegate, Robert Bixby, Vasek Chvatal, and William Cook. Concorde-03.12.19, 2003. Last accessed: 2020/2/26.

[6] David L Applegate. *The traveling salesman problem: a computational study*, volume 17. Princeton university press, 2006.

[7] Florian Arnold and Kenneth Sörensen. Knowledge-guided local search for the vehicle routing problem. *Computers & Operations Research*, 105:32–46, 2019.

[8] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[9] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[10] Federico Berto, Chuanbo Hua, Junyoung Park, Laurin Luttmann, Yining Ma, Fanchen Bu, Jiarui Wang, Haoran Ye, Minsu Kim, Sanghyeok Choi, Nayeli Gast Zepeda, André Hottung, Jianan Zhou, Jieyi Bi, Yu Hu, Fei Liu, Hyeonah Kim, Jiwoo Son, Haeyeon Kim, Davide Angioni, Wouter Kool, Zhiguang Cao, Qingfu Zhang, Joungho Kim, Jie Zhang, Kijung Shin, Cathy Wu, Sungsoo Ahn, Guojie Song, Changhyun Kwon, Kevin Tierney, Lin Xie, and Jinkyoo Park. RL4CO: an extensive reinforcement learning for combinatorial optimization benchmark. In *SIGKDD Conference on Knowledge Discovery and Data Mining*, 2025.

[11] Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, André Hottung, Niels Wouda, Leon Lan, Junyoung Park, Kevin Tierney, and Jinkyoo Park. RouteFinder: Towards foundation models for vehicle routing problems. *arXiv preprint arXiv:2406.15007*, 2024.

[12] Aymeric Blot, Holger H Hoos, Laetitia Jourdan, Marie-Éléonore Kessaci-Marmion, and Heike Trautmann. 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*, pages 32–47. Springer, 2016.

[13] Jürgen Branke, Su Nguyen, Christoph W Pickardt, and Mengjie Zhang. Automated design of production scheduling heuristics: A review. *IEEE Transactions on Evolutionary Computation*, 20(1):110–124, 2015.

[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[15] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

[16] Edmund K Burke, Matthew R Hyde, Graham Kendall, and John Woodward. Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1559–1565, 2007.

[17] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. *Advances in neural information processing systems*, 32, 2019.

[18] Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. HSEvo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. *arXiv preprint arXiv:2412.14995*, 2024.

[19] Tim Dernedde, Daniela Thyssens, Sören Dittrich, Maximilian Stubbemann, and Lars Schmidt-Thieme. Moco: A learnable meta optimizer for combinatorial optimization. *arXiv preprint arXiv:2402.04915*, 2024.

[20] Darko Drakulic, Sofia Michel, and Jean-Marc Andreoli. GOAL: A generalist combinatorial optimization agent learner. In *International Conference on Learning Representations*, 2025.

[21] Darko Drakulic, Sofia Michel, Florian Mai, Arnaud Sors, and Jean-Marc Andreoli. BQ-NCO: Bisimulation quotienting for generalizable neural combinatorial optimization. In *Advances in Neural Information Processing Systems*, 2023.

[22] Johann Dréo. *Metaheuristics for hard optimization: methods and case studies*. Springer Science & Business Media, 2006.

[23] Lizhou Fan, Wenyue Hua, Lingyao Li, Haoyang Ling, and Yongfeng Zhang. NPHardEval: Dynamic benchmark on reasoning ability of large language models via complexity classes. *arXiv preprint arXiv:2312.14890*, 2023.

[24] Vincent Furnon and Laurent Perron. Or-tools routing library, 2023.

[25] Chengrui Gao, Haopu Shang, Ke Xue, Dong Li, and Chao Qian. Towards generalizable neural solvers for vehicle routing problems via ensemble with transferrable local policy. *arXiv preprint arXiv:2308.14104*, 2023.

[26] André Hottung and Kevin Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. In *ECAI 2020*, pages 443–450. IOS Press, 2020.

[27] Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*, 2022.

[28] Benjamin Hudson, Qingbiao Li, Matthew Malencia, and Amanda Prorok. Graph neural network guided local search for the traveling salesperson problem. *arXiv preprint arXiv:2110.05291*, 2021.

[29] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of artificial intelligence research*, 36:267–306, 2009.

[30] Zangir Iklassov, Yali Du, Farkhad Akimov, and Martin Takac. Self-guiding exploration for combinatorial problems. *arXiv preprint arXiv:2405.17950*, 2024.

[31] Jiaming Ji, Tianyi Qiu, Boyuan Chen, Borong Zhang, Hantao Lou, Kaile Wang, Yawen Duan, Zhonghao He, Jiayi Zhou, Zhaowei Zhang, et al. AI Alignment: A comprehensive survey. *arXiv preprint arXiv:2310.19852*, 2023.

[32] Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. LLMOPT: Learning to define and solve general optimization problems from scratch. *arXiv preprint arXiv:2410.13213*, 2024.

[33] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[34] Xia Jiang, Yaoxin Wu, Yuan Wang, and Yingqian Zhang. UNCO: Towards unifying neural combinatorial optimization through large language model. *arXiv preprint arXiv:2408.12214*, 2024.

[35] Xia Jiang, Yaoxin Wu, Chenhao Zhang, and Yingqian Zhang. DRoC: Elevating large language models for complex vehicle routing via decomposed retrieval of constraints. In *13th international Conference on Learning Representations, ICLR 2025*. OpenReview. net, 2025.

[36] David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on computing*, 3(4):299–325, 1974.

[37] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169*, 2023.

[38] Minsu Kim, Sanghyeok Choi, Hyeonah Kim, Jiwoo Son, Jinkyoo Park, and Yoshua Bengio. Ant colony sampling with gflownets for combinatorial optimization. *arXiv preprint arXiv:2403.07041*, 2024.

[39] Minsu Kim, Jinkyoo Park, et al. Learning collaborative policies to solve np-hard routing problems. *Advances in Neural Information Processing Systems*, 34:10418–10430, 2021.

[40] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

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

[42] Yeong-Dae Kwon, Jinho Choo, Iljoo Yoon, Minah Park, Duwon Park, and Youngjune Gwon. Matrix encoding networks for neural combinatorial optimization. *Advances in Neural Information Processing Systems*, 34:5138–5149, 2021.

[43] Sirui Li, Janardhan Kulkarni, Ishai Menache, Cathy Wu, and Beibin Li. Towards foundation models for mixed integer linear programming. *arXiv preprint arXiv:2410.08288*, 2024.

[44] Yang Li, Jinpei Guo, Runzhong Wang, and Junchi Yan. T2T: From distribution learning in training to gradient search in testing for combinatorial optimization. *Advances in Neural Information Processing Systems*, 36:50020–50040, 2023.

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

[46] Fei Liu, Xialiang Tong, Mingxuan Yuan, and Qingfu Zhang. Algorithm evolution using large language model. *arXiv preprint arXiv:2311.15249*, 2023.

[47] Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of Heuristics: Towards efficient automatic algorithm design using large language model. In *Forty-first International Conference on Machine Learning*, 2024.

[48] Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Xialiang Tong, Mingxuan Yuan, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. A systematic survey on large language models for algorithm design. *arXiv preprint arXiv:2410.14716*, 2024.

[49] Fu Luo, Xi Lin, Fei Liu, Qingfu Zhang, and Zhenkun Wang. Neural combinatorial optimization with heavy decoder: Toward large scale generalization. In *Advances in Neural Information Processing Systems*, 2023.

[50] Yining Ma, Zhiguang Cao, and Yeow Meng Chee. Learning to search feasible and infeasible regions of routing problems with flexible neural k-opt. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[51] Silvano Martello and Paolo Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete applied mathematics*, 28(1):59–70, 1990.

[52] Sadegh Mirshekarian and Dusan Sormaz. Machine learning approaches to learning heuristics for combinatorial optimization problems. *Procedia Manufacturing*, 17:102–109, 2018.

[53] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.

[54] Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose A Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew J Parkes, Sanja Petrovic, et al. HyFlex: A benchmark framework for cross-domain heuristic search. In *Evolutionary Computation in Combinatorial Optimization: 12th European Conference, EvoCOP 2012, Málaga, Spain, April 11-13, 2012. Proceedings 12*, pages 136–147. Springer, 2012.

[55] Bo Peng, Jiahai Wang, and Zizhen Zhang. A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. In *Artificial Intelligence Algorithms and Applications: 11th International Symposium, ISICA 2019, Guangzhou, China, November 16–17, 2019, Revised Selected Papers 11*, pages 636–650. Springer, 2020.

[56] Nelishia Pillay and Rong Qu. *Automated Design of Machine Learning and Search Algorithms*. Springer, 2021.

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

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

[59] Steven S Seiden. On the online bin packing problem. *Journal of the ACM (JACM)*, 49(5):640–671, 2002.

[60] Jialong Shi, Qingfu Zhang, and Edward Tsang. EB-GLS: an improved guided local search based on the big valley structure. *Memetic computing*, 10:333–350, 2018.

[61] Jingyan Sui, Shizhe Ding, Boyang Xia, Ruizhi Liu, and Dongbo Bu. NeuralGLS: learning to guide local search with graph convolutional network for the traveling salesman problem. *Neural Computing and Applications*, 36(17):9687–9706, 2024.

[62] Weiwei Sun, Shengyu Feng, Shanda Li, and Yiming Yang. CO-Bench: Benchmarking language model agents in algorithm search for combinatorial optimization. *arXiv preprint arXiv:2504.04310*, 2025.

[63] Yiwen Sun, Furong Ye, Xianyin Zhang, Shiyu Huang, Bingzhen Zhang, Ke Wei, and Shaowei Cai. AutoSAT: Automatically optimize sat solvers via large language models. *arXiv preprint arXiv:2402.10705*, 2024.

[64] Zhiqing Sun and Yiming Yang. DIFUSCO: Graph-based diffusion solvers for combinatorial optimization. *Advances in neural information processing systems*, 36:3706–3731, 2023.

[65] Paolo Toth and Daniele Vigo. Models, relaxations and exact approaches for the capacitated vehicle routing problem. *Discrete Applied Mathematics*, 123(1-3):487–512, 2002.

[66] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[67] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[68] Christos Voudouris, Edward PK Tsang, and Abdullah Alsheddy. Guided local search. In *Handbook of metaheuristics*, pages 321–361. Springer, 2010.

[69] Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*, 2024.

[70] Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems*, 36:30840–30861, 2023.

[71] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

[72] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[73] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[74] Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving routing problems. *IEEE transactions on neural networks and learning systems*, 33(9):5057–5069, 2021.

[75] Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, et al. Chain-of-Experts: When llms meet complex operations research problems. In *The Twelfth International Conference on Learning Representations*, 2023.

[76] Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. Optibench meets resocratic: Measure and improve llms for optimization modeling. *arXiv preprint arXiv:2407.09887*, 2024.

[77] Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. Multi-objective evolution of heuristic using large language model. *arXiv preprint arXiv:2409.16867*, 2024.

[78] Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. *arXiv preprint arXiv:2402.01145*, 2024.

[79] Haoran Ye, Jiarui Wang, Zhiguang Cao, Helan Liang, and Yong Li. DeepACO: Neural-enhanced ant systems for combinatorial optimization. *Advances in neural information processing systems*, 36:43706–43728, 2023.

[80] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-Taught Optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling*, 2024.

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

[82] Jianan Zhou, Zhiguang Cao, Yaoxin Wu, Wen Song, Yining Ma, Jie Zhang, and Xu Chi. MVMoE: Multi-task vehicle routing solver with mixture-of-experts. In *International Conference on Machine Learning*, 2024.

[83] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022.

Appendix

A Additional Related Work

A.1 Heuristic Design for COPs

Traditional heuristic design for NP-hard COPs relies heavily on expert knowledge and is time-consuming to develop [22]. This has motivated the emergence of automatic heuristic design [15] as a more efficient alternative [56], leveraging metaheuristic or ML techniques to automate heuristic generation and optimization [16, 29, 12, 52]. However, these approaches are often constrained by inflexible search and strong domain-specific dependencies [54, 13]. Recently, neural solvers have gained attention as a promising alternative [67, 40], employing deep learning to learn heuristics in a data-driven manner. Despite showing promise, they still face several challenges, including limited scalability and generalization, as well as high training overhead. More recently, the advent of LLMs has transformed the landscape of heuristic design. Their advanced language understanding and reasoning capabilities [14, 72, 71] have been increasingly exploited to enhance heuristic generation for solving COPs [48, 58, 63]. Among recent efforts, most approaches combine the efficiency of evolutionary search with the adaptability of LLM reasoning via few-shot prompting, leading to a surge of interest in LLM-EC frameworks for heuristic design in COPs [47, 78, 77, 18, 81]. However, the use of a fixed optimization strategy (e.g., EC) in these frameworks often restricts exploration of the broader search space.

A.2 Neural Heuristics for COPs

Different from traditional hand-crafted heuristics, neural heuristics for solving COPs have rapidly advanced in recent years [9, 10, 25, 38, 79]. These methods generally fall into two paradigms. 1) For constructive heuristics, Pointer Network (Ptr-Net) [67], a sequence-to-sequence model with differentiable attention mechanisms, was first introduced to directly learn permutation-invariant solutions for TSP through supervised learning. This was extended by using reinforcement learning to improve performance [8], and further applied to CVRP [53]. With the rise of Transformer architectures [66], the attention-based model [40] was proposed to solve various COPs, inspiring a series of subsequent works [21, 39, 41, 49, 55]. More recently, there has been a surge of interest in foundation models that aim to solve multiple COPs using a single, general-purpose model [11, 20, 82]. 2) Improvement heuristics [17, 19, 26, 42, 44, 50, 64, 74] leverage neural networks to guide local search for solution refinement [28, 61]. While these approaches can often produce (near-)optimal solutions with extended inference times, they typically face challenges in scaling to large problem instances and generalizing across diverse problem settings.

A.3 LLMs for CO

LLMs have recently gained widespread recognition and found broad applications across various domains [31, 37], significantly influencing research directions in combinatorial optimization. In particular, recent studies have explored the application of LLMs in multiple facets of CO, including enhancing algorithm design [18, 46, 47, 58, 78], automating the formulation of CO problems [2, 32, 35, 43, 75], developing CO-specific benchmark datasets [23, 62, 76], directly solving COPs[1, 30, 70], and integrating LLMs into domain-specific foundation models to construct unified frameworks capable of addressing a wide spectrum of CO tasks[3, 34]. As LLMs continue to evolve rapidly, they exhibit great potential to support the development of more automated, generalizable, and efficient problem-solving frameworks in the field of CO.

B Preliminaries

In this section, we first introduce two canonical COPs, TSP and online BPP, followed by an introduction of existing LLM-EC approaches and a high-level comparison with our proposed MoH.

Algorithm 1: MoH Training Workflow

Input: Number of downstream tasks N, Number of iterations T, Seed optimizer $\mathcal{I}_0$;
Output: Meta-optimizer $\mathcal{I}_T^*$, Heuristic populations $\mathcal{H}$ across all tasks;
Function $\mathbf{U}(\mathcal{I})$**:**
  $u \leftarrow 0$
  for $i = 1, \ldots, N$ **do**
    $\tilde{h}_i^{\mathcal{I}} \leftarrow \mathcal{I}(\mathcal{H}_i, U_i(\cdot), \text{LLM}, \text{Prompt}, \text{"Task i"})$:
      Steps for heuristic design with optimizer $\mathcal{I}$:
      a. Heuristic Generation: a group of heuristics using $\mathcal{I} \rightarrow \{h_{i,1}^{\mathcal{I}}, \ldots, h_{i,K}^{\mathcal{I}}\}$
      b. Evaluate the utility score of each heuristic through $U_i(h_{i,k}^{\mathcal{I}}, \mathcal{D}_i),\ \forall k \in [1, K]$
      c. Update $\mathcal{H}_i = \text{Top}\mathcal{K}(\mathcal{H}_i \cup \{h_{i,1}^{\mathcal{I}}, \ldots, h_{i,K}^{\mathcal{I}}\})$ by utility and return the best heuristic $\tilde{h}_i^{\mathcal{I}}$
    $u \leftarrow u + \omega_i \cdot U_i(\tilde{h}_i^{\mathcal{I}}, \mathcal{D}_i)$
  return u
▶▶ Initialize heuristic and optimizer populations $\mathcal{H} = \{\mathcal{H}_1, \ldots, \mathcal{H}_N\}, \mathcal{P} = \{\mathcal{I}_0\}$
for $t = 1, \ldots, T$ **do**
  $\tilde{\mathcal{I}}_t \leftarrow \mathcal{I}_{t-1}^*(\mathcal{P}, \mathbf{U}(\cdot), \text{LLM}, \text{Prompt}, \text{"Optimizer"})$:
    Steps for optimizer design with meta-optimizer $\mathcal{I}_{t-1}^*$:
    a. Optimizer Generation: a group of optimizers via (self-)invocation of $\mathcal{I}_{t-1}^* \rightarrow \{\mathcal{I}_t^1, \ldots, \mathcal{I}_t^M\}$
    b. Evaluate the utility score of each optimizer through $\mathbf{U}(\mathcal{I}_t^j),\ \forall j \in [1, M]$
    c. Update $\mathcal{P} = \text{Top}\mathcal{K}(\mathcal{P} \cup \{\mathcal{I}_t^1, \ldots, \mathcal{I}_t^M\})$ by utility and return the best optimizer $\tilde{\mathcal{I}}_t$
  $\mathcal{I}_t^* \leftarrow \tilde{\mathcal{I}}_t$
return $\mathcal{I}_T^*, \mathcal{H}$

Traveling Salesman Problem. TSP is a well-known NP-hard COP [6]. A TSP instance is defined over a complete graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, where $\mathcal{V} = \{v_1, \ldots, v_n\}$ is the set of cities and $\mathcal{E} = \{e(v_i, v_j) | v_i, v_j \in \mathcal{V}, i \neq j\}$ is the set of edges, representing possible travel routes between cities. Each edge $e(v_i, v_j)$ is associated with a distance d_{ij}, where $d : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}^+$ defines the travel cost between any pair of cities. The objective of TSP is to find a Hamiltonian cycle (i.e., a permutation of $\mathcal{V}$ that starts and ends at the same city) with the minimum total travel cost, subject to the constraint that each city is visited exactly once before returning to the starting city.

Online Bin Packing Problem. BPP aims to pack a set of items $\{i_1, i_2, \ldots, i_n\}$, each with an associated weight w_i, into bins of capacity C. In its online version [59], items arrive sequentially in an unknown order, and an immediate, irrevocable placement decision must be made for each item. The objective is to minimize the number of bins used, subject to the constraint that the total weight of items in each bin does not exceed its capacity C.

Existing LLM-EC Approaches. Early approaches [47] leverage a fixed EC optimizer to discover effective heuristics through LLMs for solving COPs. Specifically, they maintain a fixed-size population of heuristics tailored to a specific COP task. The EC optimizer refines this population by iteratively selecting promising candidates and applying crossover and mutation operations to generate new and potentially improved heuristic variants. Although this method can quickly converge to reasonably good heuristics, its performance is inherently constrained by the insufficient exploration of the vast search space. Although subsequent works propose various EC variants, such as incorporating a reflection mechanism [78] or integrating with Monte Carlo Tree Search (MCTS) [81], these methods still suffer from limited exploration or high computational cost. In summary, existing approaches primarily focus on heuristic design using a fixed EC optimizer, whereas MoH targets optimizer design, operating at a higher level of abstraction beyond prior methods, as illustrated in Fig. 2.

C Additional Results

C.1 Results on Online BPP

3) *Online BPP:* We follow the settings of [58] to develop a heuristic that assigns incoming items to bins in real time. The heuristic utilizes a scoring function to determine the most suitable bin for each item dynamically [4]. We evaluate the generated heuristics on 100 Weibull instances for each

Table 2: Results on Online BPP.

Bin Capacity	Item Size	Best Fit	First Fit	FunSearch	EoH	ReEvo	HSEvo	MCTS-AHD	MoH (Ours)
100	1k	4.621%	5.038%	3.165%	3.294%	3.475%	3.748%	**2.543%**	2.553%
	5k	4.149%	4.488%	2.165%	0.827%	2.022%	1.088%	1.769%	**0.600%**
	10k	4.030%	4.308%	2.008%	0.436%	1.821%	0.734%	1.647%	**0.414%**
200	1k	1.825%	2.025%	0.938%	1.645%	1.825%	1.825%	1.238%	**0.848%**
	5k	1.555%	1.665%	0.543%	0.366%	1.549%	1.555%	1.062%	**0.262%**
	10k	1.489%	1.578%	0.459%	0.188%	1.489%	1.489%	1.036%	**0.141%**
300	1k	1.131%	1.265%	0.654%	1.086%	1.131%	1.131%	0.922%	**0.581%**
	5k	0.919%	0.984%	0.352%	0.254%	0.919%	0.919%	0.785%	**0.161%**
	10k	0.882%	0.924%	0.316%	0.115%	0.882%	0.882%	0.765%	**0.079%**
400	1k	0.815%	0.835%	0.519%	0.815%	0.815%	0.815%	0.755%	**0.498%**
	5k	0.624%	0.672%	0.275%	0.191%	0.621%	0.624%	0.608%	**0.104%**
	10k	0.603%	0.639%	0.243%	0.098%	0.595%	0.603%	0.579%	**0.054%**
500	1k	0.546%	0.522%	**0.324%**	0.695%	0.546%	0.546%	0.496%	0.373%
	5k	0.472%	0.507%	0.214%	0.119%	0.472%	0.472%	0.447%	**0.090%**
	10k	0.448%	0.487%	0.196%	0.075%	0.445%	0.448%	0.430%	**0.032%**
Average		1.607%	1.729%	0.825%	0.680%	1.240%	1.125%	1.006%	**0.453%**

problem size, ranging from 1,000 to 10,000, with bin capacities varying from 100 to 500. The lower bound lb for each instance is calculated as the ceiling of the total item weight divided by the capacity of a single bin: $lb = \left\lceil \frac{\sum_{i=1}^{n} w_i}{c} \right\rceil$, where w_i is the weight of the item i and c is the bin capacity [51]. The heuristics used as baseline are Best Fit and First Fit for online BPP. All methods are trained on two tasks: 1,000 items with bin capacity 1,000, and 5,000 items with bin capacity 1,000. Each heuristic are evaluated on 100 instances.

Table 2 summarizes the performance of MoH on the Online BPP, evaluated over a variety of settings with different bin capacities (100 to 500) and item set sizes (1k, 5k, 10k). The reported metric is the proportion of excess bins used relative to the theoretical lower bound. Our method outperforms all competing baselines and traditional heuristics Best Fit, First Fit across nearly all instance settings, achieving lower average bin usage. These results indicate that MoH performs well on online packing tasks, demonstrating strong adaptability and generalization in dynamic environments. Examples of the best-performing heuristics discovered by MoH in large-scale settings are provided in Appendix E.

C.2 Results on TSPLib

We further evaluate our method on the widely used TSPLib dataset across various instance sizes. As shown in Table 3 and 4, we compare the TSP-GLS and TSP-KGLS settings against EoH, ReEvo, HSEvo, MCTS-AHD and some neural solvers across instances of different scales. The results are split into two tables: one for instances smaller than 200, and another for sizes ranging from 200 to 1000. Consistent with the results in Table 1, our generated heuristics outperform both GLS and KGLS baselines on most TSPLib instances. Heuristic 2 and 3 show examples of the best heuristics generated for the TSP-GLS and TSP-KGLS settings of size 1000, respectively.

C.3 Results on CVRP

We further evaluate our MoH on another VRP variant, i.e., the Capacitated Vehicle Routing Problem (CVRP) [65], which is a widely studied optimization problem in the fields of logistics and operations research. It builds upon the classic TSP by incorporating the crucial real-world constraint of limited vehicle capacity. Specifically, a CVRP instance can be defined over a complete graph $\mathcal{G} = \{\mathcal{V} \cup v_0, \mathcal{E}\}$, where $\mathcal{V} = \{v_1, \ldots, v_n\}$ denotes the set of customer nodes, v_0 represents the depot, and $\mathcal{E} = \{e(v_i, v_j) | v_i, v_j \in \mathcal{V} \cup v_0, i \neq j\}$ is the edge set that includes all the possible travel routes between any two nodes, either customers or depot. Each edge $e(v_i, v_j)$ is associated with a non-negative travel cost or distance d_{ij}, where $d : (\mathcal{V} \cup v_0) \times (\mathcal{V} \cup v_0) \to \mathbb{R}^+$ defines the travel cost between any pair of two nodes. Each customer $v_i \in \mathcal{V}$ has a demand $q_i > 0$, while the depot has $q_0 = 0$. The fleet consists of m vehicles, each with capacity Q. The objective is to determine a set of m' routes (usually

Table 3: Results on TSPLib instances with sizes smaller than 200.

Instance	Neural Solver			GLS Algorithms					TSP-GLS			TSP-KGLS			
	AM [40]	POMO [41]	LEHD [49]	GNNGLS [28]	NeuralGLS [61]	GLS [68]	EBGLS [60]	KGLS [7]	EoH	HSEvo	MoH	ReEvo	HSEvo	MCTS-AHD	MoH
eil51	1.63	0.83	1.64	0.00	0.00	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67
berlin52	4.17	0.04	0.03	0.14	0.00	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
st70	1.74	0.31	0.33	0.76	0.00	0.31	0.31	0.31	0.31	0.31	0.31	0.31	0.31	0.31	0.31
eil76	1.99	1.18	2.54	0.16	0.00	1.37	1.18	1.18	1.18	1.18	1.18	1.18	1.18	1.18	1.18
pr76	0.82	0.00	0.22	0.04	0.82	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
rat99	2.65	2.39	1.10	0.55	0.72	1.55	0.74	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.68
kroA100	4.02	0.41	0.12	0.73	0.03	0.02	0.02	0.06	0.02	0.02	0.02	0.02	0.02	0.02	0.02
kroB100	5.14	0.32	0.26	0.15	0.88	0.23	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00
kroC100	0.97	0.18	0.32	1.57	1.77	0.50	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
kroD100	2.72	0.84	0.38	0.57	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
kroE100	1.47	0.45	0.43	1.22	1.05	0.49	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00
rd100	3.41	0.01	0.01	0.46	0.00	0.01	0.01	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01
eil101	2.99	1.84	2.31	0.20	0.36	3.28	1.91	2.07	1.78	1.82	1.78	1.78	1.78	1.78	1.78
lin105	1.74	0.52	0.34	0.61	0.65	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
pr107	3.93	0.52	11.24	0.44	0.81	0.40	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
pr124	3.68	0.60	1.11	0.76	0.08	0.60	0.60	0.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00
bier127	5.91	13.72	4.76	1.95	2.73	0.59	0.29	0.42	0.01	0.01	0.01	0.01	0.04	0.10	0.01
ch130	3.18	0.16	0.55	3.52	1.19	1.09	0.46	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
pr136	5.06	0.93	0.45	3.39	2.32	2.01	0.28	0.24	0.00	0.00	0.00	0.00	0.01	0.00	0.00
pr144	7.64	0.53	0.19	3.58	0.74	0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ch150	4.58	0.53	0.52	2.11	2.49	0.68	0.37	0.04	0.04	0.33	0.04	0.04	0.04	0.04	0.04
kroA150	3.78	0.70	1.40	2.98	0.77	1.75	0.26	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00
kroB150	2.44	1.17	0.76	3.26	3.11	1.01	0.00	0.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00
pr152	7.49	1.05	12.14	3.12	0.00	0.19	0.19	0.19	0.00	0.00	0.00	0.19	0.19	0.19	0.00
u159	7.55	0.95	1.13	1.02	0.90	0.74	0.78	0.96	0.00	0.00	0.00	0.00	0.00	0.00	0.00
rat195	6.89	8.15	1.42	1.67	0.48	0.61	0.61	0.97	0.90	0.80	0.56	0.65	0.47	0.61	0.47
d198	373.02	17.29	9.23	4.77	1.28	2.08	1.87	0.31	0.32	0.26	0.21	0.20	0.28	0.43	0.20
kroA200	7.11	1.58	0.64	2.03	0.86	0.75	0.18	0.71	0.13	0.04	0.00	0.23	0.09	0.04	0.00
kroB200	8.54	1.44	0.16	2.59	3.74	1.43	1.27	0.89	0.08	0.05	0.04	0.01	0.01	0.01	0.01
Average	16.77	2.02	1.92	1.53	0.96	0.78	0.42	0.36	0.21	0.22	**0.19**	0.21	0.20	0.21	**0.19**

Table 4: Results on TSPLib instances with sizes ranging from 200 to 1000.

Instance	Neural Solver	TSP-GLS			TSP-KGLS			
	POMO [41]	EoH	HSEvo	MoH	ReEvo	HSEvo	MCTS-AHD	MoH
ts225	3.60	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tsp225	3.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00
pr226	1.42	0.00	0.04	0.00	0.00	0.00	0.00	0.00
pr264	2.80	0.00	0.00	0.00	0.00	0.00	0.00	0.00
a280	5.23	0.55	0.80	0.30	0.30	0.30	0.30	0.30
pr299	4.94	0.07	0.53	0.11	0.13	0.07	0.08	0.01
lin318	4.72	0.55	1.13	0.38	0.32	0.27	0.43	0.29
rd400	6.37	0.78	0.82	0.29	0.17	0.38	0.45	0.20
fl417	8.51	0.64	0.68	0.62	0.60	0.49	0.70	0.49
pr439	7.87	1.09	2.46	0.28	1.01	1.21	1.23	0.56
pcb442	5.36	0.59	1.12	0.79	0.28	0.35	0.15	0.08
d493	9.67	1.12	1.11	0.51	0.61	0.67	1.42	0.45
u574	11.86	1.12	0.87	0.80	0.86	1.49	1.39	0.83
rat575	12.46	2.55	1.25	1.35	1.67	1.60	1.47	1.04
p654	11.30	0.21	0.31	0.12	0.15	1.57	0.11	0.10
d657	12.72	1.42	1.26	0.87	1.02	1.08	0.96	0.54
u724	16.57	1.06	2.07	0.99	0.91	0.92	0.88	0.72
rat783	18.11	2.46	1.98	2.18	1.68	1.65	1.86	1.11
pr1002	20.00	1.83	1.11	1.14	1.21	1.27	1.17	0.90
Average	8.77	0.84	0.92	**0.57**	0.57	0.70	0.66	**0.40**

$m' < m$) with minimized total travel cost across all routes, while satisfying the following constraints: 1) each route starts and ends at the depot, 2) each customer is visited exactly once by a single vehicle, and 3) the total demand on any route does not exceed Q.

We follow the experimental setup from [78], which designs heuristics for CVRP under the Ant Colony Optimization (ACO) framework, a setting also adopted by [18] and [81]. Fig. 5 compares the best objective of these methods with ours.

C.4 Results on Offline BPP

The Offline Bin Packing Problem (Offline BPP) differs from the Online version in Section B by assuming complete prior knowledge of item sizes and quantities, thereby allowing for global optimization. Following the approach in [78], we adopt an ACO framework to iteratively construct bin packing solutions, where the ants assign items to bins based on pheromone trails and heuristic cues, with pheromone updates progressively enhancing solution quality. Figure 5 compares the best objective values achieved by these methods and ours.

Table 5: Results for CVRP and Offline BPP.

Problems	CVRP				Offline BPP			
Methods	Train		Generalization		Train		Generalization	
	20	50	100	200	N=100, C=150	N=500, C=150	N=500, C=300	N=1000, C=300
ReEvo	4.826	9.339	15.901	28.224	41.984	207.406	102.438	204.438
HSEvo	4.858	9.250	15.940	28.598	41.766	205.500	102.438	204.656
MCTS-AHD	4.843	9.165	15.630	28.041	41.656	**204.609**	102.625	204.734
MoH	**4.704**	**9.059**	**15.563**	**27.512**	**41.625**	205.453	**102.125**	**203.750**

C.5 Cost and Evaluation Comparison

We present a comprehensive cost comparison across methods by reporting their average computational metrics, i.e., the LLM request counts, token usage, evaluation numbers, and performance, for the two studied problem settings: TSP-GLS (Table 6) and CVRP (Table 7). For TSP-GLS, we use instances of size 200 for both training and inference to ensure fair comparison between MoH and baseline methods. For CVRP, training involves instance sizes of 20 and 50. All results are averaged over three runs, with token usage evaluated using the GPT-4o-mini API.

Table 6: Cost of different methods on TSP-GLS setting.

Methods	Time/mins	Input Tokens	Output Tokens	Total Tokens	LLM requests	Evaluation times	Results-Gap
MoH-Train	238.4	743837.0	458358.7	1202195.7	1248.3	971.7	0.373%
MoH-Inference	62.7	398064.3	138350.0	536414.3	276.7	240.3	0.398%
EoH	326.4	1437973.0	446583.7	1884556.7	1256.0	992.0	0.535%
HSEvo	291.7	1188412.3	390665.0	1579077.3	679.7	1005.0	0.448%

Table 7: Cost of different methods on CVRP+ACO setting.

Methods	Time/mins	Input Tokens	Output Tokens	Total Tokens	LLM requests	Evaluation times	Results-Obj. 20	Results-Obj. 50
MoH	321.1	986216.0	740292.3	1726508.3	1456.7	938.3	4.831	9.262
ReEvo	231.2	2122802.0	677652.0	2800454.0	1431.0	1000.0	4.877	9.521
HSEvo	350.7	2140030.7	670590.0	2810620.7	1076.7	1001.7	4.977	9.536
MCTS	1330.3	2601902.0	825037.3	3426939.3	1490.0	1002.7	4.881	9.233

C.6 Ablation Study and Further Analysis

In this section, we provide a more in-depth analysis of MoH. The experiments shown in Fig. 4 and Tables 8 and 9 are conducted under the improvement heuristic (i.e., GLS) setting on TSP. Results are averaged over three runs, using TSP100 and TSP200 as the downstream tasks during training.

Idea Generation. A key strength of LLMs lies in their powerful natural language processing capabilities. Integrating code generation and optimization tasks with natural language algorithm descriptions is therefore a natural approach. We incorporate these descriptions into the code generation process as *ideas*, enabling LLMs to fully utilize their language understanding abilities. This strategy goes beyond simple repeated sampling within the code space, allowing LLMs to explore a broader and more diverse solution space. As shown in Fig. 4 and Table 9, we compare the performance of methods with and without natural language ideas. The utility score reflects the average performance across two downstream tasks during training. The results clearly demonstrate that incorporating such natural language descriptions improves training performance.

Population size. For each downstream task, MoH maintains a heuristic population that allows the optimizer to iteratively select, reference and refine promising candidates. Given the well-established effectiveness of few-shot prompting, it is crucial to retain elite heuristics from previous iterations to guide subsequent optimization steps, supporting both exploration and exploitation. As shown in Fig. 4 and Table 9, we evaluate the impact of different population sizes during MoH training.

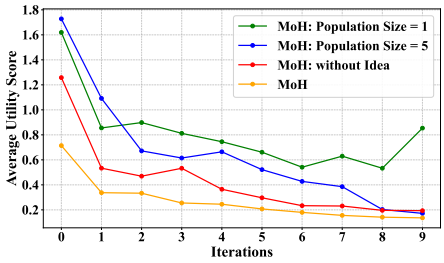

Figure 4: Training convergence curves under different settings.

Table 8: Ablation results of MoH-GLS on different LLMs for TSP.

LLM	100	200	500	1000
4o mini	0.035%	**0.332%**	**1.045%**	**1.710%**
o1-mini	**0.024%**	0.353%	1.280%	2.119%
Deepseek-V3	0.108%	0.413%	1.314%	2.527%
Qwen-Plus	0.057%	0.375%	1.753%	3.277%

Table 9: Ablation results of MoH-GLS on idea generation and population size for TSP.

Setting	Population Size	w. idea	100	200	500	1000
	1	✓	0.120%	0.563%	2.355%	3.300%
	5	✓	0.058%	0.337%	1.475%	2.338%
	10	✗	0.043%	0.390%	1.644%	2.420%
Default	10	✓	**0.035%**	**0.332%**	**1.045%**	**1.710%**

The results indicate that a small population size limits the LLM's ability to effectively leverage top-performing candidates when generating improved ones. To balance computational cost, training time, and exploration breadth in downstream heuristic tasks, we set the population size to 10. This choice ensures sufficient solution diversity while keeping overhead manageable.

Different LLMs. In Table 8, we evaluate several LLMs beyond GPT 4o-mini (2024-07-18) to assess the adaptability of MoH, including o1-mini (2024-09-12), deepseek-v3 (2024/12/26) and Qwen-plus-0919. The results demonstrate that our framework performs well across different LLMs.

Furthermore, we observe that LLMs with more learnable parameters and larger context windows tend to produce longer and more complex optimization strategies within the (meta-)optimizer. However, increased optimizer complexity does not necessarily lead to better performance of downstream heuristics.

For instance, the more advanced o1-mini does not outperform 4o-mini on large-scale TSP instances.

Analysis of Meta-Optimizer. We take a deeper look into the meta-optimizers generated by MoH. In Appendix E, we present representative examples and analyze their underlying strategies. While some follow the EC framework (Fig. 9), similar to existing approaches, others adopt classical optimization paradigms, such as Ant Colony Optimization (ACO) in Fig. 10, Particle Swarm Optimization (PSO) in Fig. 11, Simulated Annealing in Fig. 12, Tabu Search in Fig. 13, and hybrid strategies in Fig. 15, which achieved the best performance in our evaluations. By leveraging diverse optimization principles and generating tailored prompts, MoH facilitates broader exploration of the extensive search space, enabling the discovery of more effective heuristics.

D Prompt Design

D.1 Prompts for Meta Optimizer Generation

In this section, we present the prompts used to generate the meta-optimizer, along with those embedded within it to guide idea generation and code synthesis for both the optimizer and downstream tasks, as illustrated in Fig. 5. Beyond the predefined prompt constraints, we also integrate additional judgment mechanisms into our framework to ensure the explainability and efficiency of generated hyper-heuristics, mitigating the potential impact of LLM output uncertainty on MoH performance.

D.2 Prompts for formulating heuristic generation

In this section, we present a prompt example used to guide the generation of downstream COP heuristics, as shown in Fig. 6. For different tasks, only the function signature and corresponding problem size are modified accordingly.

D.3 Prompts for Heuristic Initialization

To maintain population diversity, we initialize a population before training and retain elite solutions for idea modification during inference. Accordingly, we present the prompts used to generate diverse ideas that guide code generation at the start of each stage.

Prompt for Meta-Optimizer Generation

Task: You should design an efficient metaheuristic using the following constraints. Your solution code should balance exploration and exploitation creatively.

Firstly, describe your meta-heuristic, including optimization strategy and main optimization steps in one sentence. The description must be inside a brace and marked as a comment.

Next, implement it in Python as a function named 'optimize_algorithm'. This function should accept five inputs: 'population', 'utility', 'language_model', 'subtask_prompt' and 'subtask'. The function should return three output: 'best_idea', 'best_solution', 'best_utility'. 'utility' is a function that evaluates solutions based on a score function, 'subtask_prompt' is the format for model responses, 'task' is the name of the problem to be optimized. The function returns 'best_idea','best_solution','best_utility' which are the idea behind best solution, best code together with its utility.

Note:'language_model' is an instance of the language model class used for code generation, with function "def prompt_batch(expertise,message_batch,temperature), return responses_list" for multiple request to LLM and "def prompt(self, expertise, message, temperature) return result" for single request to LLM, 'population' is a dictionary of several historical best solutions of the task, you only use the following functions to operate: "def get_solution_by_index(self, task_name, index): return item", to get a solution by its utility rank; "def get_random_solution(self, task_name): return item", to get a random solution from the population; the item returned above is a dictionary with keys 'best_sol' and 'utility'. Other functions you can use are: "def get_subtask_size(self, task_name):" to return the size of the population.

Prompts inside Seed Optimizer

1. Idea Generation:
Given the following heuristic for task: ['best_sol'] with its idea: ['idea'] and utility score: ['utility'], " "Summarize the key idea from this heuristic, then provide several totally different or refined ideas from the given one to design improved algorithms with lower utility score. " "Provide a single string as the answer, less than 50 words. Your response should be formatted as a json structure.

2. Heuristic Code Solution Generation:
Improve the following solution: {selected_solution}. You must return an improved solution. Formatted as follows:{subtask_prompt}. To better solve the problem, you are encouraged to develop new solutions based on the direction proposed: {direction} You will be evaluated based on a score function. The lower the score, the better the solution is. Be as creative as you can under the constraints.

Figure 5: Prompts for generating the meta-optimizer and those embedded within the seed optimizer.

You are an expert in TSP solver. Given a set of nodes with their coordinates, you need to find the shortest route that visits each node once and returns to the starting node. The task can be solved step-by-step by starting from the current node and iteratively choosing the next node. Help me design a novel algorithm that is different from the algorithms in literature to select the next node in each step.
First, describe your new algorithm and main steps in one sentence. The description must be inside a brace and marked as a comment. Next, implement it in Python as a function named "select_next_node". This function should accept 4 input(s): "current_node","destination_node","univisited_nodes","distance_matrix". The function should return 1 output(s): "next_node". 'current_node', 'destination_node', 'next_node', and 'unvisited_nodes' are node IDs. 'distance_matrix' is the distance matrix of nodes. All are Numpy arrays. Do not give additional explanations. Your solution should be designed and fit for the task {prob} with problem size {size}.

Figure 6: Prompts for generating constructive heuristics for TSP.

1. Generate seed direction in training stage.
You are an expert in the domain of optimization heuristics and combinatorial optimization problems. Your task is to design heuristics that can effectively solve optimization problems.The problem is {problem} with corresponding size {size}. According to the task description: {task_description} Provide several high-level directions for generating the seed prompt, each aimed at minimizing the utility as a result. Format your response as a JSON codeblock below: {{ "direction": ["content": "Your first direction suggestion here.", "content": "Your second direction suggestion here.", "content": "Your third direction suggestion here.", ... "content": "..."] }}

2. Generate/Modify seed direction in inference stage.
You are an expert in optimization heuristics, tasked with summarizing key insights to design improved algorithms. Given the following heuristics for the problem: {problem}: {solution}, please summarize the key insights in the heuristics to design improved algorithms for larger sized problem problem with corresponding size {size}. Formatted as a json structure: “‘json{{"insights":["content","content","content", ... ,"content"]}}“. Remember each insight inside the list should be one sentence less than 50 words.

3. Generate Code by Idea.
You are an expert in the domain of optimization heuristics and combinatorial optimization problems. Your task is to design heuristics that can effectively solve optimization problems. Write a function that will implement a Python algorithm to solve a problem as well as possible. The optimization problem is {problem} and the size you should focus on is {size}. The output function is formatted as follows:“‘python{formula_str}“‘ You are encouraged to develop the algorithm that follows the direction: {direction}.

Figure 7: Prompts for code and idea generation during the initialization of training and inference.

E Examples of LLM-Generated Heuristics and Meta-Optimizer

In this section, we present the best-performing heuristics for the largest instance size of each problem in Heuristic 1-6, along with several examples of generated meta-optimizers shown in Fig 8-15. These examples demonstrate that MoH can produce diverse, explainable, and effective optimizers that extend beyond traditional LLM-EC optimizer, incorporating a wide range of optimization strategies and generate high-quality heuristics for downstream tasks. For clarity and space efficiency, non-essential code elements are omitted while preserving the core optimization logic.

```python
import numpy as np
def select_next_node(current_node, destination_node, unvisited_nodes, distance_matrix):
    num_unvisited = len(unvisited_nodes)
    if num_unvisited == 0:
        return None
    distances = distance_matrix[current_node, unvisited_nodes]
    avg_distance = np.mean(distances)
    threshold = 0.5 * avg_distance
    close_nodes = unvisited_nodes[distances <= threshold]
    scores = {}
    if len(close_nodes) > 0:
        for node in close_nodes:
            immediate_distance = distance_matrix[current_node, node]
            future_savings = np.sum(distance_matrix[node, close_nodes]) / (len(close_nodes) - 1) if len(close_nodes) > 1
      else 0
            diversity_score = np.mean(distance_matrix[node, unvisited_nodes]) / (immediate_distance + 1)
            scores[node] = immediate_distance + (0.6 * (1 - future_savings)) - (0.4 * diversity_score)
    if not scores:
        far_nodes = unvisited_nodes[distances > threshold]
        for node in far_nodes:
            scores[node] = distance_matrix[current_node, node]
    next_node = min(scores, key=scores.get) if scores else None
    return next_node
```

Heuristic 1: Best constructive heuristic discovered for TSP with size 1000.

```python
import numpy as np
def update_edge_distance(edge_distance, local_opt_tour, edge_n_used):
    updated_edge_distance = np.copy(edge_distance)
    num_nodes = len(local_opt_tour)
    window_size = 5
    for i in range(num_nodes):
        current_city = local_opt_tour[i]
        for j in range(1, window_size + 1):
            next_index = (i + j) % num_nodes
            next_city = local_opt_tour[next_index]
            used_edge_count = edge_n_used[current_city, next_city]
            if used_edge_count >= 2:
```

```
                scaling_factor = np.log(used_edge_count + 1) * 0.5
                updated_edge_distance[current_city, next_city] *= scaling_factor
                updated_edge_distance[next_city, current_city] *= scaling_factor
            else:
                decay_factor = np.exp(-0.1 * used_edge_count)
                updated_edge_distance[current_city, next_city] *= decay_factor
                updated_edge_distance[next_city, current_city] *= decay_factor
            edge_quality = edge_distance[current_city, next_city] / (used_edge_count + 1)
            updated_edge_distance[current_city, next_city] += edge_quality
            updated_edge_distance[next_city, current_city] += edge_quality
    return updated_edge_distance
```

Heuristic 2: Best improvement heuristic discovered for TSP-GLS with size 1000.

```
import numpy as np
def adaptive_indicators(distance_matrix):
    num_nodes = distance_matrix.shape[0]
    indicators = np.zeros((num_nodes, num_nodes))
    min_edge = np.full(num_nodes, np.inf)
    min_edge[0] = 0
    visited = np.zeros(num_nodes, dtype=bool)
    total_mst_cost = 0
    for _ in range(num_nodes):
        u = np.argmin(np.where(visited, np.inf, min_edge))
        visited[u] = True
        total_mst_cost += min_edge[u]
        for v in range(num_nodes):
            if not visited[v] and distance_matrix[u, v] < min_edge[v]:
                min_edge[v] = distance_matrix[u, v]
    inverted_distance_matrix = 1 / (distance_matrix + np.eye(num_nodes))
    total_density = np.sum(inverted_distance_matrix, axis=1)
    for i in range(num_nodes):
        for j in range(num_nodes):
            if i != j:
                base_indicator = (total_density[i] * total_density[j]) / (1 + total_density[i] + total_density[j])
                edge_cost = distance_matrix[i, j] - (total_mst_cost / (num_nodes - 1))
                cycle_penalty = np.sum((inverted_distance_matrix[i, :] + inverted_distance_matrix[j, :] <
      inverted_distance_matrix[i, j]) * distance_matrix[i, j] * 0.2)
                indicators[i, j] = max(0, (base_indicator - cycle_penalty) * edge_cost)
    max_indicator = np.max(indicators)
    if max_indicator > 0:
        indicators /= max_indicator
    return indicators
```

Heuristic 3: Best improvement heuristic for TSP-KGLS with size 1000.

```
import numpy as np
def score(item, bins):
    scores = np.zeros_like(bins, dtype=float)
    feasible_bins = bins[bins > item]
    if feasible_bins.size == 0:
        return scores
    max_capacity = np.max(feasible_bins)
    scores[bins == max_capacity] = -np.inf
    remaining_capacity = (feasible_bins - item) / feasible_bins
    item_ratio = item / feasible_bins
    proximity_penalty = np.where(feasible_bins >= item * 0.90, -5, 0) + np.where(feasible_bins < item * 0.80, -7, 0)
    underutilization_penalty = -3 * np.maximum(0, item - 0.5 * feasible_bins)
    scores[bins > item] = (remaining_capacity + proximity_penalty + underutilization_penalty - (1 - item_ratio) ** 3)
    return scores
```

Heuristic 4: Best heuristic for online BPP with 10000 items and a bin capacity of 500.

```
def compute_edge_scores(distance_matrix, coordinates, demands, capacity):
    import numpy as np
    num_nodes = distance_matrix.shape[0]
    edge_promisingness = np.zeros((num_nodes, num_nodes))
    total_demand = np.sum(demands)
    decay_factor = 0.95
    adaptive_alpha = 1.5
    adaptive_beta = 2.5
    for i in range(num_nodes):
        for j in range(num_nodes):
            if i != j and demands[j] <= capacity:
                distance_score = (1 / (distance_matrix[i, j] + 1e-6)) ** adaptive_beta
                demand_score = demands[j] / total_demand if total_demand > 0 else 0
                pheromone_level = 1.0 / (distance_matrix[i, j] + 1e-6) * decay_factor
                exploration_factor = (1 + demands[j] / capacity)
                edge_promisingness[i, j] = (distance_score ** adaptive_beta) * (demand_score ** adaptive_alpha) *
      pheromone_level * exploration_factor
    return edge_promisingness
```

Heuristic 5: Best heuristic for CVRP_ACO with size 200.

```python
import numpy as np
def compute_pair(demand, capacity):
    n = demand.shape[0]
    heuristic_matrix = np.zeros((n, n))
    valid_indices = np.where(demand <= capacity)[0]
    for i in valid_indices:
        for j in valid_indices:
            if i != j:
                total_demand = demand[i] + demand[j]
                if total_demand <= capacity:
                    heuristic_matrix[i][j] = capacity - total_demand + min(demand[i], demand[j])
    for i in range(n):
        for j in range(n):
            if i != j:
                single_demand = demand[i]
                if single_demand <= capacity:
                    heuristic_matrix[i][j] = max(heuristic_matrix[i][j], capacity - single_demand + demand[j])
    frequency_count = np.sum(heuristic_matrix > 0, axis=1)
    for i in range(n):
        for j in range(n):
            if i != j and heuristic_matrix[i][j] > 0:
                heuristic_matrix[i][j] -= frequency_count[i] * 0.1
    demand_group = np.digitize(demand, bins=np.linspace(0, capacity, num=5))
    for group in range(1, 5):
        group_indices = np.where(demand_group == group)[0]
        if len(group_indices) > 1:
            for i in group_indices:
                for j in group_indices:
                    if i != j and heuristic_matrix[i][j] > 0:
                        heuristic_matrix[i][j] += 0.05
    for i in range(n):
        for j in range(n):
            if i != j and frequency_count[i] > 1 and frequency_count[j] > 1:
                heuristic_matrix[i][j] -= 0.2 * (frequency_count[i] + frequency_count[j]) / 2
    return heuristic_matrix
```

Heuristic 6: Best heuristic for Offline BPP with 1000 items and a bin capacity of 300.

F Broader Impacts

This work explores a general framework for improving COP heuristics through LLMs. By introducing a meta-optimization structure, our method demonstrates how LLMs can autonomously generate and improve heuristics across diverse problem domains such as TSP, CVRP, and BPP. Potential social impacts of MoH may include: 1) Improved optimization capabilities in practical applications such as logistics, manufacturing, and resource allocation; 2) Bridging AI and Operations Research (OR) by designing a unified framework that benefits both communities, especially when solving problems with larger sizes; 3) Lower barrier to high-quality algorithm design, especially in low-resource or less-studied problem domains where handcrafted heuristics are not readily available. Meanwhile, a potential negative impact of our method lies in the reliance on LLMs, where both training and inference involve substantial token usage. This can lead to increased energy consumption and raise environmental concerns due to the computational resources required.

G Declaration of LLM Usage

LLMs are employed as a core component of our methodology to generate and refine optimizers, as well as to support downstream heuristic generation. More precisely, LLMs are used to produce and optimize code implementations aimed at developing high-performing heuristics for solving COPs. A detailed workflow of LLM involvement has been presented in Section 2.

H Licenses

We list all the used assets and their licenses in Table 10.

Table 10: Used assets and their licenses.

Type	Asset	License	Usage
Code	Concorde [5]	Available for academic research use	Evaluation
	OR-Tools [24]	Apache-2.0 license	Evaluation
	FunSearch [58]	MIT License	Evaluation
	EoH [47]	MIT License	Evaluation
	ReEvo [78]	MIT License	Evaluation
	HSEvo [18]	MIT License	Evaluation
	MCTS-AHD [81]	MIT License	Evaluation
	POMO [41]	MIT License	Evaluation
	LEHD [49]	MIT License	Evaluation
Dataset	TSPLib [57]	Available for any non-commercial use	Evaluation

Seed Optimizer Example

```
def optimize_algorithm(population, utility, language_model, subtask_prompt,
    subtask):
    expertise = "You are an expert in the domain of designing meta optimization
        strategy and combinatorial optimization problems. Your task is to design
        heuristics that can effectively solve optimization problems."
    # Step 1: Select a random solution from the population
    selected_solution = population.get_random_solution(subtask)
    # Step 2: Generate directions for improvement
    direction_prompt = (
        f"Given the following heuristic for subtask: {selected_solution['best_sol
            ']} with its idea: {selected_solution['idea']} and utility score: {
            selected_solution['utility']}, "
        "Summarize the key idea from this heuristic, then provide several totally
             different ideas from the given one to design improved algorithms
            with lower utility score. "
        "Provide a single string as the answer, less than 50 words. Your response
             should be formatted as a json structure: "
        "```json\n{{\"insights\":[\"content\",\"content\",\"content\", ... ,\"
            content\"]}}\n```."
    )
    response = language_model.prompt(expertise, direction_prompt, temperature=1)
    directions = json.loads(extract_code(response))["insights"]
    # Step 3: Create messages based on generated directions
    message_batch = []
    for direction in directions:
        message = (
            f"Improve the following solution:\n"
            f"```python\n{selected_solution}\n```\n"
            f"You must return an improved solution. Formatted as follows:\n{
                subtask_prompt}\n"
            f"To better solve the problem, you are encouraged to develop new
                solutions based on the direction proposed: {direction}. "
            "You will be evaluated based on a score function. The lower the score,
                 the better the solution.\n"
            "Your response must firstly provide a summary of the key idea inside a
                 brace and marked as a comment, followed by the code
                implementation. "
            "Be as creative as you can under the constraints."
        )
        message_batch.append(message)
    # Step 4: Generate new solutions using the language model
    responses = language_model.prompt_batch(expertise, message_batch, temperature
        =1)
    new_solutions = extract_code(responses)
    new_ideas = extract_idea(responses)
    # Step 5: Evaluate new solutions
    solutions_with_utilities = [(idea, solution, utility(solution,idea,subtask))
        for idea, solution in zip(new_ideas, new_solutions)]
    best_idea, best_solution, best_utility = min(solutions_with_utilities, key=
        lambda x: x[2])
    return best_idea, best_solution, best_utility
```

Figure 8: The seed optimizer used for training, which randomly selects previous solutions and generates new directions for improvement.

Genetic Algorithm (GA)

```
def optimize_algorithm(population, utility, language_model, subtask_prompt,
    subtask):# Adaptive Genetic Algorithm incorporating selection, crossover,
    and mutation to enhance solution diversity and convergence
   expertise = "You are an expert in the domain of designing meta optimization
       strategy and combinatorial optimization problems. Your task is to design
       heuristics that can effectively solve optimization problems."
   # Step 1: Select top-performing solutions for breeding
   elite_count = max(1, population.get_subtask_size(subtask) // 10) # Top 10% as
        elites
   elites = [population.get_solution_by_index(subtask, i) for i in range(
       elite_count)]
   # Step 2: Generate directions for diversity through crossover and mutation
   direction_prompt = ( "Given the top-performing solutions, suggest innovative
       crossover and mutation strategies to create diverse and high-quality
       offspring. Provide your response as a JSON with keys 'crossover_methods'
       and 'mutation_methods', each containing a list of strategies." )
   response = language_model.prompt(expertise, direction_prompt, temperature
       =0.7)
   directions = json.loads(extract_code(response))
   crossover_methods = directions.get("crossover_methods", [])
   mutation_methods = directions.get("mutation_methods", [])
   # Step 3: Create offspring solutions using the generated strategies
   offspring = []
   for method in crossover_methods:
       for i in range(elite_count):
           parent1 = elites[i]
           parent2 = elites[(i + 1) % elite_count]
           crossover_prompt = ( f"Apply the following crossover strategy to
               combine these two solutions:\nSolution 1: ```python\n{parent1['
               best_sol']}\n```\nSolution 2: ```python\n{parent2['best_sol']}\n
               ```\nStrategy: {method}\nProvide the new offspring solution as a
               JSON with keys 'idea' and 'best_sol'.")
           offspring_response = language_model.prompt(expertise, crossover_prompt
               , temperature=0.7)
           offspring_data = json.loads(extract_code(offspring_response))
           offspring.append(offspring_data)
   for method in mutation_methods:
       for elite in elites:
           mutation_prompt = (f"Apply the following mutation strategy to this
               solution:\nSolution: ```python\n{elite['best_sol']}\n```\nStrategy
               : {method}\nProvide the mutated solution as a JSON with keys '
               idea' and 'best_sol'.")
           mutation_response = language_model.prompt(expertise, mutation_prompt,
               temperature=0.7)
           mutation_data = json.loads(extract_code(mutation_response))
           offspring.append(mutation_data)
   # Step 4: Evaluate offspring and select the best
   solutions_with_utilities = [(child['idea'], child['best_sol'], utility(child
       ['best_sol'], child['idea'], subtask)) for child in offspring]
   # Include elites to maintain the best solutions
   elite_solutions = [(elite['idea'], elite['best_sol'], elite['utility']) for
       elite in elites]
   all_candidates = solutions_with_utilities + elite_solutions
   best_idea, best_solution, best_utility = min(all_candidates, key=lambda x: x
       [2])
   return best_idea, best_solution, best_utility
```

Figure 9: An example of the meta-optimizer generated by LLM, which employs Genetic Algorithm (GA) to balance exploration and expoitation, similar to previous LLM-EC optimizer.
```

Ant Colony Optimization (ACO)

```python
# {This algorithm employs Ant Colony Optimization (ACO) principles to simulate
    the foraging process of ants for efficient bin packing solutions, enabling
    adaptive pheromone adjustment and heuristic guidance to balance exploration
    and exploitation for optimal results.}
def optimize_algorithm(population, utility, language_model, subtask_prompt,
    subtask):
    expertise = "You are an expert in the domain of designing meta optimization
        strategies and combinatorial optimization problems. Your task is to
        design heuristics that can effectively solve optimization problems."
    # Parameters for ACO and solution selection
    ant_count = 10
    elite_count = 3
    population_size = population.get_subtask_size(subtask)
    # Step 1: Select elite solutions based on utility for initial pheromone
        distribution
    elite_solutions = [population.get_solution_by_index(subtask, i)
        for i in range(min(elite_count, population_size))]
    # Initialize pheromone levels for directions based on elite solutions
    pheromone_levels = {sol['best_sol']: 1.0 for sol in elite_solutions}
    # Step 2: Generate solution directions using Ant Colony Optimization
        principles
    direction_prompts = []
    for _ in range(ant_count):
        direction_prompt = f"Using the ACO principles for the task '{subtask}',
            generate a new solution direction that addresses the bin packing
            problem. Consider the current elite solutions: {[sol['best_sol'] for
            sol in elite_solutions]} And ensure that the output follows this
            format: {subtask_prompt}. Provide a summary comment of the key idea
            in braces."
        direction_prompts.append(direction_prompt)
    # Step 3: Get new directions from the language model
    responses = language_model.prompt_batch(expertise, direction_prompts,
        temperature=0.7)
    new_directions = extract_code(responses)
    # Step 4: Evaluate new solutions
    solutions_with_utilities = []
    for direction in new_directions:
        try:
            # Evaluate the new solution's utility
            score = utility(direction, "Derived from ACO strategy", subtask)
            solutions_with_utilities.append((direction, score))
            # Update pheromone based on the quality of the direction
            pheromone_levels[direction] = pheromone_levels.get(direction, 1.0) +
                2.0 / (score + 1e-6)
        except Exception as e:
            continue # Skip if utility evaluation fails
    # Step 5: Select the best new solution based on its utility score
    if not solutions_with_utilities:
        # Fallback to the best existing solution if no new solutions are valid
        best_existing = population.get_solution_by_index(subtask, 0)
        return best_existing.get('idea'), best_existing.get('best_sol'),
            best_existing.get('utility')
    best_direction, best_utility = min(solutions_with_utilities, key=lambda x: x
        [1])
    return "Ant Colony optimized direction", best_direction, best_utility
```

Figure 10: An example of the LLM-generated meta-optimizer that utilizes Ant Colony Optimization (ACO) as its underlying mechanism.

Particle Swarm Optimization (PSO)

```python
def optimize_algorithm(population, utility, language_model, subtask_prompt,
    subtask):
    expertise = "You are an expert in ..."
    # Parameters for PSO
    particle_count = 15
    iterations = 10
    inertia_weight = 0.5 # Controls exploration versus exploitation
    cognitive_param = 0.8 # Personal attraction/learning factor
    social_param = 1.2 # Societal attraction/learning factor
    # Step 1: Initialize particles with random solutions in the population
    particles=[population.get_random_solution(subtask) for _ in range(
        particle_count)]
    best_personal_solutions = particles.copy()
    global_best_solution = min(particles, key=lambda x: utility(x['best_sol'], "
        Initial PSO", subtask)) # Initialize global best solution
    # Step 2: Iterate through the PSO process
    for _ in range(iterations):
        for particle in particles:
            current_score = utility(particle['best_sol'], "PSO iteration", subtask
                )
            # Update personal best if current score is better
            if current_score < utility(best_personal_solutions[particles.index(
                particle)]['best_sol'], "PSO personal best", subtask):
                best_personal_solutions[particles.index(particle)] = particle
            # Update global best if current score is better
            if current_score < utility(global_best_solution['best_sol'], "PSO
                global best", subtask):
                global_best_solution = particle
        # Step 3: Generate new directions using PSO influences
        new_directions = []
        for particle in particles: # Compute new velocities and positions for PSO
            r1, r2 = np.random.rand(), np.random.rand()
            cognitive_velocity = cognitive_param * r1 * (best_personal_solutions[
                particles.index(particle)]['best_sol'] - particle['best_sol'])
            social_velocity = social_param * r2 * (global_best_solution['best_sol'
                ] - particle['best_sol'])
            new_position = particle['best_sol'] + inertia_weight * (
                cognitive_velocity + social_velocity)
            prompt = f"Using PSO principles, generate a new solution for the task
                '{subtask}' based on the solution '{new_position}'. Ensure to
                follow this format: {subtask_prompt}. Provide a summary in braces.
                "
            new_directions.append(prompt)
        # Step 4: Get new solutions from the language model
        responses = language_model.prompt_batch(expertise, new_directions,
            temperature=0.7)
        generated_solutions = extract_code(responses)
        # Step 5: Evaluate new solutions
        for new_solution in generated_solutions:
            try:
                score = utility(new_solution, "Derived from PSO", subtask)
                if score < utility(global_best_solution['best_sol'], "Final global
                    best", subtask):
                    global_best_solution = {'best_sol': new_solution, 'utility':
                        score}
            except Exception:
                continue # Skip errors in utility evaluation
    return "Particle Swarm optimized direction", global_best_solution['best_sol'
        ], global_best_solution['utility']
```

Figure 11: An example of the meta-optimizer generated by LLM, which employs Particle Swarm Optimization (PSO) to balance exploration and expoitation.

Combination of Adaptive Simulated Annealing with Differential Evolution

```
def optimize_algorithm(population, utility, language_model, subtask_prompt,
    subtask):
    swarm_size = 5
    parent_solutions = [population.get_random_solution(subtask) for _ in range(
        swarm_size)]
    current_utilities = [parent['utility'] for parent in parent_solutions]
    initial_temperature = 1.0
    cooling_rate = 0.95
    iterations = 5
    best_solution = None
    best_utility = np.inf
    temperature = initial_temperature
    for iteration in range(iterations):
        # Dynamic temperature adjustment based on utility improvements
        temperature = max(initial_temperature * (cooling_rate ** iteration), 0.01)
             # Ensure temperature doesn't go below a threshold
        initial_prompts = f"Using the following solutions: {[sol['best_sol'] for
            sol in parent_solutions]} with utilities: {current_utilities}, devise
             advanced enhancement strategies by blending Adaptive Simulated
            Annealing and Differential Evolution for the task: {subtask}. Return
            strategies in JSON format: ```json\n{{\"strategies\": [\"strategy1\",
             \"strategy2\", ..., \"strategyN\"]}}```."
        response = language_model.prompt("You are an expert optimization
            strategist.", initial_prompts, temperature=0.7)
        potential_strategies = json.loads(extract_code(response))["strategies"]
        for strategy in potential_strategies:# Apply each strategy to parent
            solutions
            modified_solutions = []
            for parent in parent_solutions:
                message = f"Implement the strategy: {strategy} on the parent
                    solution: ```python\n{parent['best_sol']}\n```. Generate a
                    modified solution adhering to {subtask_prompt}. Focus on
                    creating diverse and high-quality outputs."
                response = language_model.prompt("You are an expert optimization
                    strategist.", message, temperature=0.7)
                new_solution = extract_code(response)
                new_utility = utility(new_solution, "Optimized by Adaptive
                    Simulated Annealing and DE", subtask)
                if new_utility < best_utility:
                    best_utility = new_utility
                    best_solution = new_solution
                # Explore new areas based on probability
                elif np.random.rand() < np.exp((utility(parent['best_sol'], '
                    Original', subtask) - new_utility) / temperature):
                    modified_solutions.append(new_solution)
        # Optional: Implement differential evolution by combining best solutions
        if modified_solutions:
            best_of_modifications = min(modified_solutions, key=lambda s: utility(
                s, "Modified", subtask))
            new_utility = utility(best_of_modifications, "From Differential
                Evolution", subtask)
            if new_utility < best_utility:
                best_utility = new_utility
                best_solution = best_of_modifications
    best_idea = 'Implemented a hybrid approach of Adaptive Simulated Annealing
        and Differential Evolution for enhanced search efficiency.'
    return best_idea, best_solution, best_utility
```

Figure 12: An example of the meta-optimizer generated by LLM, which combines Adaptive Simulated Annealing with Differential Evolution to explore solution space while refining candidates.

Tabu Search

```python
def optimize_algorithm(population, utility, language_model, subtask_prompt,
    subtask):
    # This algorithm utilizes Tabu Search to dynamically explore and adapt
        solutions, avoiding cycling while maximizing job performance and
        minimizing makespan.
    selected_solution = population.get_random_solution(subtask)
    best_solution, best_utility = selected_solution['best_sol'],
        selected_solution['utility']
    tabu_list = set()
    iterations = 0
    max_iterations = 5
    while iterations < max_iterations:
        # Generate candidate solutions
        candidates = []
        for _ in range(3): # Generate 3 candidate solutions
            direction_prompt = (
                f"Given the heuristic: {selected_solution['best_sol']} with its
                    idea: "
                f"{selected_solution['idea']} and utility score: {
                    selected_solution['utility']}, "
                "Generate a modified or new solution. Format as: "
                f"`{subtask_prompt}`. Ensure it is innovative and consider new job
                     scheduling metrics."
            )
            response = language_model.prompt("You are an expert in optimization.",
                 direction_prompt, temperature=0.7)
            candidate_code = extract_code(response)
            candidates.append(candidate_code)
        # Evaluate candidates
        candidate_utilities = []
        for candidate in candidates:
            candidate_idea = extract_idea(candidate)
            candidate_utility = utility(candidate_code, candidate_idea, subtask)
            candidate_utilities.append((candidate_idea, candidate,
                candidate_utility))
        # Apply Tabu Search logic
        feasible_candidates = [c for c in candidate_utilities if c[1] not in
            tabu_list]
        if feasible_candidates:
            best_candidate = min(feasible_candidates, key=lambda x: x[2])
            best_candidate_idea, best_candidate_solution, best_candidate_utility =
                 best_candidate
            # Update best solution if found a better one
            if best_candidate_utility < best_utility:
                best_solution, best_utility = best_candidate_solution,
                    best_candidate_utility
                selected_solution['idea'], selected_solution['best_sol'],
                    selected_solution['utility'] = best_candidate_idea,
                    best_candidate_solution, best_candidate_utility
            # Update Tabu list
            tabu_list.add(best_candidate_solution)
            if len(tabu_list) > 10: # Maintain fixed size
                tabu_list.pop() # Remove the oldest entry
        iterations += 1
    return selected_solution['idea'], best_solution, best_utility
```

Figure 13: An example of the meta-optimizer generated by LLM, which employs Tabu Search to dynamically explore and adapt solutions.

A Neighborhood Search-like Strategy

```
def optimize_algorithm(population, utility, language_model, subtask_prompt,
    subtask):
    # This approach utilizes Variable Neighborhood Search to explore different
        solution neighborhoods dynamically to enhance optimization performance.

    def generate_neighborhood(solution):
        # Generate modified solutions (neighborhoods) by tweaking routes based on
            the original solution
        neighborhoods = []
        # Simple route perturbation: swap two routes or modify delivery sequences
        for i in range(len(solution['best_sol'])):
            if isinstance(solution['best_sol'][i], list) and len(solution['
                best_sol'][i]) > 1:
                # Swap two locations in the route
                new_solution = solution['best_sol'][:]
                new_solution[i] = new_solution[i][:2] + new_solution[i][0:2] #
                    Swap first two for simplicity
                neighborhoods.append({"best_sol": new_solution, "utility": utility
                    (new_solution, subtask)})
        return neighborhoods

    selected_solution = population.get_random_solution(subtask)
    neighborhoods = generate_neighborhood(selected_solution)

    # Collect insights to improve solutions based on the generated neighborhoods
    message_batch = []
    for neighbor in neighborhoods:
        message = (
            f"Improve the following solution:\n"
            f"```python\n{neighbor}\n```\n"
            f"You must return an improved solution. Formatted as follows:\n{
                subtask_prompt}\n"
            "To better solve the problem, consider how the neighborhood structure
                changes the optimization landscape. "
        )
        message_batch.append(message)

    responses = language_model.prompt_batch("You are an expert in optimizations."
        , message_batch, temperature=0.7)
    new_solutions = extract_code(responses)
    new_ideas = extract_idea(responses)

    # Evaluate the new solutions and select the best one
    solutions_with_utilities = [(idea, solution, utility(solution, idea, subtask)
        ) for idea, solution in zip(new_ideas, new_solutions)]
    best_idea, best_solution, best_utility = min(solutions_with_utilities, key=
        lambda x: x[2])

    return best_idea, best_solution, best_utility
#"best_sol": new_solution, "utility": utility(new_solution, task)
```

Figure 14: An example of the meta-optimizer generated by LLM, which utilizes Variable Neighborhood Search to dynamically explore solution neighborhoods to enhance optimization performance.

Adaptive Exploration-Exploitation Strategy with Dynamic and Tabu Mechanisms

```
def optimize_algorithm(population, utility, language_model, subtask_prompt, subtask):
    elite_count = 4 # Parameter Initialization
    diversity_count = 3
    pheromone_levels = {}
    # Step 1: Select elite and diverse solutions
    population_size = population.get_subtask_size(subtask)
    elite_solutions = [population.get_solution_by_index(subtask, i) for i in range(min(elite_count,
          population_size))]
    diverse_solutions = [population.get_random_solution(subtask) for _ in range(diversity_count)]
    selected_solutions = elite_solutions + diverse_solutions
    # Step 2: Generate dynamic exploration insights with adaptive rates
    for solution in selected_solutions:
        temperature = 1 if solution['utility'] > 0 else 0.75 # Dynamic adjustment
        prompt = f"Given the solution '{solution['best_sol']}' with utility score '{solution['utility']}',
              please suggest innovative optimization strategies that could enhance this code. Return your
             recommendations in JSON format: ```json {{\"insights\":[\"content\",\"content\",...]}} ```."
        response = language_model.prompt(expertise, prompt, temperature=temperature)
        try:
            insights = json.loads(extract_code(response))["insights"]
            for insight in insights:
                pheromone_levels[insight] = pheromone_levels.get(insight, 1.0) + 1.0
        except (json.JSONDecodeError, KeyError):
            continue
    # Step 3: Rank directions based on pheromone levels (dynamic evaluation)
    sorted_insights = sorted(pheromone_levels.items(), key=lambda x: x[1], reverse=True)
    top_insights = [insight[0] for insight in sorted_insights if insight[1] > 1.0]
    # Step 4: Create batch messages for generating optimized solutions based on directions
    message_batch = []
    for direction in top_insights:
        message = f"Refine the solution for the task '{subtask}' by focusing on this optimization approach
              : {direction}. Consider elite solutions: {[sol['best_sol'] for sol in elite_solutions]}.
             Ensure your output adheres to the following format: {subtask_prompt}. In addition, provide a
             summary of changes made."
        message_batch.append(message)
    # Step 5: Generate new solutions using the language model through batch prompts
    responses = language_model.prompt_batch(expertise, message_batch, temperature=0.9)
    # Step 6: Evaluate and keep the best-performing solutions
    solutions_with_utilities = []
    for response in responses:
        try:
            new_solution = extract_code(response)
            important_idea = extract_idea(response)
            score = utility(new_solution, important_idea, subtask)
            pheromone_levels[important_idea] = pheromone_levels.get(important_idea, 1.0) + (2.0 / (score +
                  1e-6))
            solutions_with_utilities.append((important_idea, new_solution, score))
        except Exception:
            continue
    if not solutions_with_utilities: # Best Solution Selection
        best_existing = population.get_solution_by_index(subtask, 0)
        return best_existing['idea'], best_existing['best_sol'], best_existing['utility']
    best_idea, best_solution, best_utility = min(solutions_with_utilities, key=lambda x: x[2])#
          Adjustments for ongoing adaptability of pheromones
    for key in pheromone_levels:
        pheromone_levels[key] *= 0.9 # Mild evaporation to allow exploration
    return best_idea, best_solution, best_utility
```

Figure 15: An example of the meta-optimizer generated by LLM, which employs an adaptive exploration-exploitation strategy that combines real-time performance evaluation of solutions with dynamic exploration rates. This approach customizes search focus within a genetic algorithm framework enhanced by adaptive, tabu-like mechanisms for efficient solution refinement, achieving the **best performance** during inference.