Experience-Driven Reflective Co-Evolution of Prompts and Heuristics for Autonomous Algorithm Design

Anonymous ACL submission

Abstract

Combinatorial optimization has long been dominated by manually engineered heuristics, a paradigm requiring substantial expert intuition and implementation overhead. The advent of Large Language Models has disrupted this landscape, enabling the autonomous synthesis and optimization of algorithms. Recent approaches typically iterate on heuristic populations using LLMs as mutators; however, these strategies often suffer from limited exploration, leading to stagnation in local optima. To overcome this, we present the Experience-Driven Reflective Co-**Evo**lution of **P**rompt and **H**euristics (**EvoPH**) for autonomous algorithm design, a novel framework that couples an island migration model with elite selection to maintain population diversity. Uniquely, EvoPH co-evolves both the guiding prompts and the heuristics themselves, using a feedback loop driven by past experience to refine the search process. We demonstrate EvoPH's efficacy on the Traveling Salesman and Bin Packing Problems. Our results show that EvoPH achieves superior accuracy compared to baselines, marking a significant step forward in LLM-aided algorithm design.

1 Introduction

Combinatorial optimization problems (COPs) (Dantzig and Ramser, 1959) form a fundamental branch of mathematical research. They drive progress in areas such as algorithm design and computational complexity theory, while also providing essential methods for addressing real-world challenges in resource allocation and decision-making. Traditionally, solving COPs relied on handcrafted heuristic algorithms, which require researchers to possess substantial domain knowledge (Pillay and Qu, 2018). Moreover, practical applications often demand customized algorithms with distinct processes and parameters, resulting in considerable human effort (Hromkovič, 2013). To alleviate these challenges, researchers have proposed the paradigm of automatic heuristics design (AHD), with Genetic Programming (GP) being one of the most representative examples (Langdon and Poli, 2013). GP iteratively refines heuristics by applying mutation operators (Duflo et al., 2019). However, the effectiveness of GP-based methods is constrained by the human-defined operator, which not only increases implementation difficulty but also limits achievable performance.

In recent years, large language models (LLMs) have demonstrated remarkable effectiveness across diverse domains, notably through prompt engineering that simulates mutation operations, enabling applications in code generation, automated machine learning, scientific discovery, and algorithm design (Zhao et al., 2023; Jiang et al., 2024; Liu et al., 2024b). Nevertheless, current practices often rely on ineffective evolutionary algorithms or fixed prompts, which limit adaptability in complex scenarios. As a consequence, existing approaches tend to converge to local optima, furthermore, syntax or logic errors introduced during code execution frequently propagate across descendant heuristics, leading to repeated failures and substantial computational overhead.

To address these limitations, we propose **EvoPH**, a novel experience-guided reflective co-**Evo**lution framework that can co-evolve **P**rompts and **H**euristics for automatic algorithm design. EvoPH is built upon an iterative cycle of heuristics generation, evaluation, experience storage and reflection. In each iteration, new heuristics are generated by an LLM, followed by assessing their performance through execution, and the outcomes are distilled into experience that informs subsequent heuristic search. During heuristics evolution, the saved experience guides the LLM to evolve heuristics through a diverse set of mutation operators. Specifically, we propose an *island-*

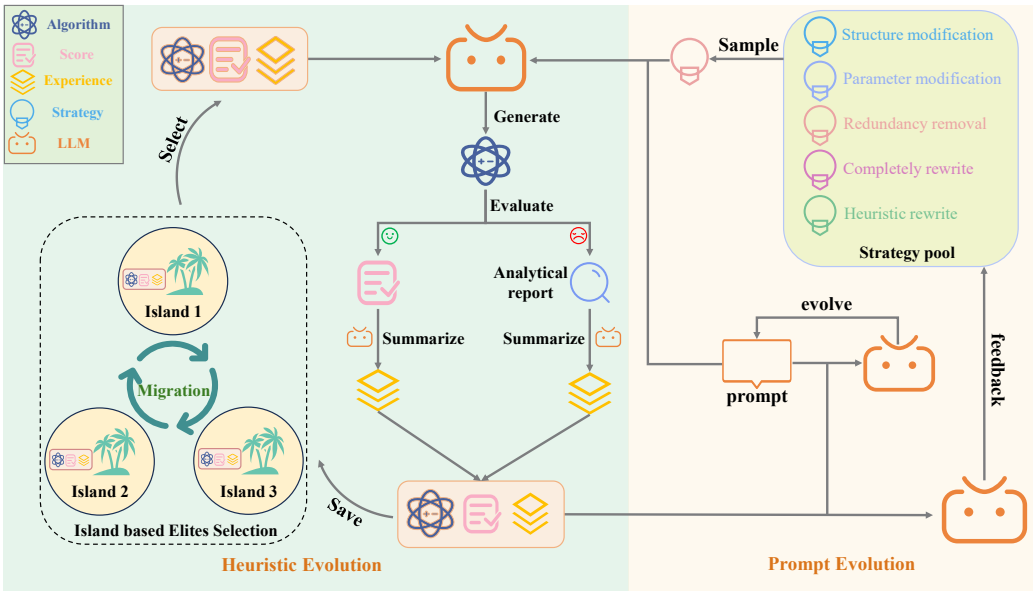

Figure 1: EvoPH comprises two interacting processes. Heuristics Evolution generates, evaluates, and stores candidate algorithms, providing feedback for further search. Prompt Evolution adaptively refines LLM prompts and strategy selection based on this feedback.

based elites selection algorithm, which can preserve diversity while enabling the exchange of high-quality elites across populations. Here, an *island* refers to an independent sub-population that evolves in parallel, occasionally exchanging high-quality elites with others. The core of our EvoPH framework lies in the integration of prompt evolution, where prompts are not only adaptively rewritten but also progressively specialized based on fine-grained execution feedback. This mechanism enables dynamic error correction and knowledge consolidation, allowing prompts to evolve into increasingly task-specific guides that retain effective instructions while continuously steering the evolution of heuristics. Furthermore, we propose an *experience-driven strategy sampling* that selects or combines mutation operators and interacts with prompt evolution to ensure prompts and strategies co-adapt in a self-correcting manner. Through this synergy, prompts function as both adaptive controllers and knowledge carriers, aligning task descriptions with heuristics evolution. To evaluate the performance of EvoPH, we conduct experiments on the Traveling Salesman Problem (TSP) and the Bin Packing Problem (BPP). We used Gurobi or OR-Tools to calculate the optimal solution and relative error to ensure the authority of the evaluation results. Our core contributions can be summarized as follows:

- We propose EvoPH, an automatic algorithm design framework that co-evolves prompts and heuristics. By iteratively evolving prompts based on execution feedback, leveraging stored experience to inform mutation operator choice and dynamically selecting strategies of evolution, EvoPH generates targeted yet diverse prompts. This synergy enables heuristic algorithms to escape local optima during evolution while promptly correcting errors in code execution.

- We construct benchmark datasets for TSP and BPP by adapting TSPlib (Reinelt, 1991) and BPPlib (Delorme et al., 2018), converting them into distance-matrix formats for efficient evaluation. We also adopt Gurobi or OR-Tools as references for optimal solutions and release all data to facilitate future research in automatic heuristic design.

- Experimental results demonstrate that on TSP, EvoPH significantly improves performance compared to prior frameworks. On BPP, EvoPH effectively enhances baseline heuristics, whereas existing frameworks only yield marginal improvements.

2 Related work

Neural combinatorial optimization. Neural combinatorial optimization (NCO) has emerged as

a promising paradigm for solving COPs in an end-to-end manner (Chen et al., 2023; Ma et al., 2023). As a variant of hyper-heuristics (HH), it explores heuristic spaces through neural architectures and training algorithms (Romera-Paredes et al., 2024; Liu et al., 2023a). Existing methods are typically grouped into learning constructive heuristics (LCH), which incrementally build solutions (Liu et al., 2023b; Son et al., 2025); learning improvement heuristics (LIH) (André and Kevin, 2020), which refine existing solutions through neural-guided search and hybrid solvers, which combine neural models with classical algorithms (Gasse et al., 2019; Luo et al., 2023). Applications now span routing, SAT, scheduling and other NP-hard problems (Li et al., 2023; Sun and Yang, 2023). Though challenges remain in scalability, generalization, and closing the gap with state-of-the-art classical solvers (Selsam, 2019).

LLM for Evolutionary computation. Evolutionary computation (EC) is a population-based black-box optimization paradigm well suited for non-convex or discrete problems without gradient information (Eiben and Smith, 2015; Bäck et al., 1997). With the rise of LLMs, recent work explores their integration with EC frameworks (Chauhan et al., 2025). For instance, the LMEA framework uses natural language instructions to guide LLM-based crossover and mutation on textual solution representations (Liu et al., 2024c), while EvoLLM leverages LLMs in a zero-shot manner to execute full evolutionary cycles via ranking-based prompting, achieving strong results on synthetic benchmarks (Lange et al., 2024). These approaches highlight LLMs as intelligent operators or high-level controllers, showing potential in heuristics design and code generation.

3 Method

The EvoPH framework is a closed-loop system for the automatic design and optimization of heuristic algorithms. As illustrated in Figure 1, EvoPH operates through an iterative cycle that integrates two complementary components: heuristics evolution and prompt evolution. The heuristics evolution module employs an island-based elites selection algorithm to refine candidate heuristics, while maintaining diversity through migration across subpopulations. The outcomes are distilled into experience, which provide structured feedback. This feedback, in turn, drives the prompt evolution module and strategy sampling module, guiding the next generation of heuristic algorithms. Together, these processes form a continuous loop of generation, evaluation and adaptation. The overall execution flow of the EvoPH framework is formally outlined in Algorithm 1. In the following sections, we provide a detailed discussion of each sub-process.

Algorithm 1: EvoPH: Experience-Guided Co-Evolution

Input : Initial Heuristics $\mathcal{H}_0$, Prompts P_0, Strategy Pool S, Data I
Settings : Islands K, Max Iterations T, Interval τ

$P \leftarrow P_0$; $\mathcal{E} \leftarrow \emptyset$;
for $k \in \{1, \ldots, K\}$ **do**
  $\mathcal{M}^{(k)} \leftarrow \text{Init}(\mathcal{H}_0)$;
for $t \leftarrow 1$ **to** T **do**
  for $k \in \{1, \ldots, K\}$ **do**
    $h_p \leftarrow \text{Select}(\mathcal{M}^{(k)}, \mathcal{E})$;
    $s \leftarrow \text{Sample}(S, \mathcal{E})$;
    $P_t \leftarrow \text{Mix}_c(P, s)$;
    $h_c \leftarrow \text{LLM}(h_p, P_t)$;
    $res \leftarrow \text{Run}(h_c, I)$;
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\text{Reflect}(h_c, res)\}$;
    Update($\mathcal{M}^{(k)}, h_c, res.\text{score}$);
  $P \leftarrow \text{Evolve}(P, \mathcal{E})$;
  if $t \pmod{\tau} = 0$ **then**
    Migration($\{\mathcal{M}^{(k)}\}$);
return Best found heuristic h;

3.1 Heuristics Evolution

Heuristic Algorithm Generation. In the generation phase, The LLM is used as a high-level semantic mutation operator to generate new candidate algorithms. Starting from the parent algorithms selected from the elite library, the LLM generates a new generation of algorithms under the guidance of carefully designed prompts.

Experience Summarization. After generating candidate heuristic algorithms, EvoPH evaluates them on the given problem instances. When execution produces valid solutions, corresponding performance metrics are extracted; in cases of invalid outputs, systematic analysis and reporting are conducted. Regardless of correctness, the execution results or the analytical report is distilled into

structured experiential knowledge that captures effective strategies and performance characteristics. This accumulated experience is subsequently synthesized into reflective feedback, which in turn guides the next iteration of heuristic search.

Heuristics and Experience Storage. The EvoPH proposes an *island-based elites selection algorithm* to organize and preserve heuristic algorithms together with their experience. The core idea is to partition the global population into N relatively independent subpopulations, referred to as *islands*. Each *island* independently executes a full elites selection process, maintaining its own elite archive. While the islands evolve autonomously, they are not entirely isolated; a periodic migration mechanism enables the exchange of elite individuals, thereby promoting global information sharing and cooperative co-evolution. The details of the elites selection algorithm are as follows:

- **Feature Space Definition**. To guide elites selection, we first define a multidimensional behavioral feature space for program solutions. Intuitively, this space can be viewed as a grid, where each cell in this grid corresponds to a unique combination of behavioral features (e.g., high development potential with low relative error). Each cell stores the best-performing solution found for that feature combination. Formally, for a heuristic $h \in H$, we define a mapping function $F : H \to B$ that projects h into a behavioral descriptor $b \in B$. Each island i maintains an elite archive M_i, in which cells are indexed by descriptors b and record the best heuristic h currently associated with b.

- **Archive Update**. When a new heuristic h_{child} is generated, we first get its descriptor $b_{\text{child}} = F(h_{\text{child}})$. The archive is updated in the following way:

$$M_i(b_{\text{child}}) \leftarrow \max_g\{M_i(b_{\text{child}}),\, h_{\text{child}}\}, \quad (1)$$

  where $g(\cdot)$ denotes the evaluate function of heuristic h and $\max_g\{\cdot,\, \cdot\}$ returns the heuristic with the larger $g(\cdot)$. This update rule ensures that only heuristics with equal or superior performance replace the existing elite. Through this process, each island incrementally explores its search region, while the collective archives promote both potential and solution quality in the global search.

- **Heuristic Selection for Evolution**. After updating the elite archive, EvoPH selects parent heuristics for the next generation through an experience-guided process. First, a candidate island is chosen, within the selected island, EvoPH then adaptively balances exploration and exploitation based on heuristics experience: in the exploration mode, a parent heuristic is randomly sampled to promote behavioral diversity, whereas in the exploitation mode, heuristics demonstrating consistently high quality across multiple descriptors are prioritized. This mechanism enables EvoPH to simultaneously foster innovation and leverage proven solutions, thereby preventing premature convergence to local optima.

- **Island Migration**. At predefined generational intervals, migration events occur. Selected elites from a source island are introduced into the evolutionary cycle of a target island, where they compete with local elites under the same archiving mechanism. These migrated solutions enrich the diversity of the population and strengthen cooperative co-evolution across islands.

3.2 Prompt Evolution

The key idea of prompt evolution is to elevate the evolutionary search from the program level to the prompt level. In this meta-evolutionary framework, as heuristics undergo optimization, the instructional prompts guiding their mutation are co-evolved concurrently. This ensures that mutation operations remain targeted, continuously adapting to the state of the search. Our proposed prompt evolution consists of two primary steps:

Prompt Update. The prompt update step employs a closed-loop mechanism in which adaptation is guided by experiential feedback from heuristics evolution. In each iteration, the performance of generated heuristic algorithm is recorded as experience, which, together with the initial prompts, are fed back into the LLM to guide subsequent prompt refinement. Prompts associated with effective heuristics are reinforced, while those consis-

tently leading to poor outcomes are refined or discarded. Through this iterative process, the system autonomously learns and improves prompts.

Strategy Sampling. To simulate the diverse mutation patterns observed in biological evolution and to introduce greater exploration potential into the evolutionary process, this study pre-designed a variety of differentiated "strategies of evolution". These strategies are modularly embedded within the prompts to guide the LLM in performing different mutation operations during the heuristics evolution process. The selection of strategies is informed by accumulated experience, in which the historical performance of previously generated heuristics is recorded. The experience serves as a reference for matching problem characteristics with suitable strategies, thereby enabling the framework to adaptively sample a strategy from the pool that is most appropriate to the current search state rather than relying on fixed or random selection. A detailed description of these strategies in the pool is provided in Appendix B. The final prompts submitted to the LLM are dynamically combined from the iteratively updated prompts and the evolutionary strategy sampled based on experience. The specific content of each prompt is detailed in Appendix D.

The prompts adaptively update based on experiential feedback, inheriting knowledge from historical successes while avoiding repeated failures. In doing so, accumulated experience guides both the refinement of prompts and the sampling of strategies of evolution, enabling the system to select the most appropriate mutation pathway for the current search context. Such an experience-driven mutation mechanism effectively aids heuristic algorithm populations in escaping local optima, significantly improving algorithm discovery efficiency and solution quality. At the same time, it enhances the effectiveness of individual mutations at the micro level and provides a solid foundation for sustained heuristics evolution at the macro level.

3.3 Comparison to Previous Work

EvoPH advances beyond prior methods in several key aspects. **First**, EvoPH preserves heuristic population diversity through an island-based elite selection mechanism. Unlike EoH (Liu et al., 2024a), which relies on fixed prompting schemes, EvoPH introduces an experience-driven evolutionary loop that continuously refines meta-prompts over time. While ReEvo (Ye et al., 2024) leverages LLM-based reflection, it does not support the co-evolution of prompts; in contrast, EvoPH explicitly enables coordinated prompt evolution, ensuring both population stability and strategic adaptability. **Second**, EvoPH adopts a lightweight feedback mechanism that eliminates the need for auxiliary predictors required by NeRM (Guo et al., 2025). Instead of training additional models, EvoPH dynamically selects mutation strategies, such as parameter tuning or prompt rewriting, from a strategy pool based on accumulated experience. This design allows for more efficient and targeted exploration while significantly reducing computational overhead. **Third**, EvoPH achieves strong performance without relying on computationally intensive LLM fine-tuning, which is central to CALM (Huang et al., 2025). By adapting search strategies within an evolutionary framework rather than modifying model parameters, EvoPH remains model-agnostic and more scalable for a wide range of application scenarios.

4 Experiment

For TSP and BPP, we initialize our population using a series of classic heuristic algorithms. The specific details of each algorithm can be found in Appendix C. In the following sections, we provide a detailed description of the dataset composition, experimental settings and the specific experimental components of our study.

4.1 Benchmarks

Existing automatic heuristic design methods often rely on randomly generated instances, which may lack structural diversity and lead to overfitting on idealized datasets. To address this problem, we construct two robust benchmarks with optimal solutions as ground truth:

- **TSP-Gurobi-Bench (TGB):** Derived from TSPLIB (Reinelt, 1991), TGB converts city coordinates into distance matrices. We employ the Gurobi solver (Gurobi Optimization, LLC, 2022) to compute optimal solutions, retaining 58 instances solved within a 600s limit.

- **BPP-Ortools-Bench (BOB):** Based on BPPlib (Corvello et al., 2010), we select 92 instances and obtain optimal solutions via Google OR-Tools (Google, 2024) to provide a high-fidelity performance ceiling.

Table 1: Experimental results of different methods on TGB and BOB datasets. The reported relative errors are averaged over five independent executions with different random seeds. "BASE" represents the performance of the initial heuristic. Bold fonts denote the best performance.

Dataset	Heuristics	BASE	FunSearch	EoH	mEoH	ReEvo	EvoPH
TGB	Christofides	20.64%	19.71%	9.64%	16.90%	20.60%	**5.17%**
	2-opt	6.62%	6.62%	7.00%	6.67%	6.58%	**4.20%**
	Nearest Insertion	19.54%	19.54%	8.78%	11.60%	19.50%	**4.41%**
	Farthest Insertion	8.20%	7.20%	8.00%	8.00%	8.20%	**4.05%**
	Nearest Neighbor	24.67%	16.50%	7.80%	24.67%	24.67%	**4.41%**
	Random Insertion	9.43%	8.11%	9.11%	8.90%	9.34%	**4.41%**
BOB	First Fit	4.90%	4.90%	4.90%	4.90%	4.90%	**0.43%**
	Best Fit	28.13%	25.49%	17.20%	23.45%	26.77%	**1.65%**
	Next Fit	5.61%	5.61%	5.61%	5.61%	5.61%	**1.59%**
	Worst Fit	14.66%	7.66%	4.90%	14.66%	14.66%	**1.65%**

Note: All values represent the mean relative error across 5 random seeds.

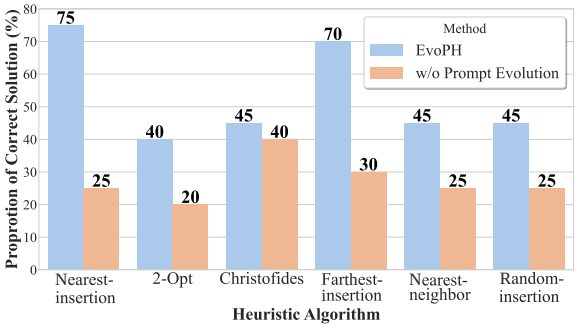

Figure 2: Proportion of generating executable code over 20 iterations of the EvoPH with and without the prompt evolution module.

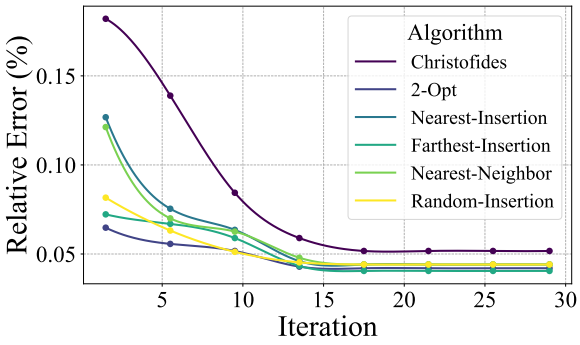

Figure 3: Variation in the lowest relative error of different initial algorithms with evolution iterations.

Evaluation metrics. To ensure objective and standardized performance comparisons across all algorithms, we use the *relative error* as the primary quantitative evaluation metric. The lower the relative error, the better the performance. The *relative error* is defined as follows:

$$\text{Relative Error} = \frac{A_{\text{sol}} - O_{\text{sol}}}{O_{\text{sol}}} \times 100\%, \quad (2)$$

where O_{sol} represents the optimal solution from Gurobi or OR-Tools, and A_{sol} represents the solution obtained by the heuristic algorithm.

4.2 Experiment setup

Baseline. We rigorously evaluate the proposed EvoPH framework in the TGB and BOB. The comparison group includes **FunSearch** (Romera-Paredes et al., 2024), **EoH** (Liu et al., 2024a), **mEoH** (Yao et al., 2025) and **ReEvo** (Ye et al., 2024). By contrasting the performance of our framework with these methods, we aim to assess its superiority and effectiveness in solving COPs.

Implementation Details. We employ Gemini-2.5-pro as the LLM for heuristic generation and evolution, with temperature and top-p set to 0.8 and 0.95, respectively. The evolutionary process is configured with $T = 20$ iterations and $K = 5$ islands to balance global exploration and population diversity. During evaluation, each candidate program is executed with a 600-second timeout threshold to ensure computational efficiency.

4.3 Main Results

As summarized in Table 1, EvoPH consistently achieves substantial performance gains across all initialization heuristics on both TGB and BOB benchmarks. For **TGB**, EvoPH drastically reduces relative errors even from weak starts; e.g., *Christofides* and *nearest-insertion* improve from 20.64% and 19.54% to 5.17% and 4.41%, respectively. It also further refines strong baselines like *2-opt* (from 6.62% to 4.20%).

For **BOB**, EvoPH similarly yields substantial gains across all heuristics. For the *next-fit* and

🧬 (Before Evolve) Evolutionary Goal and Evolutionary Directives

Your primary objective is to enhance the provided function in each evolutionary step, pushing it towards the optimal balance of solution quality and speed. Assume you are receiving a function that already exists and needs improvement.

In each round, you must analyze the function provided to you and then rewrite it to be better.

🧬 (After Evolve) Evolutionary Goal and Evolutionary Directives

Your primary objective is to enhance the provided function in each evolutionary step, pushing it towards the optimal balance of solution quality

The returned code must be valid Python code that executes without errors. Prioritize generating correct and functional code above all else. If a change results in worse performance, revert to the previous version and explore alternative strategies. Critically analyze the traceback and error messages to understand the failure's context within the code.

When the code is not valid, focus exclusively on fixing the errors. Do not attempt optimizations until the code is error-free. Carefully examine the provided `error_reason` and traceback information to pinpoint the root cause. Use print statements or a debugger to understand the program's state at the point of failure. Consider edge cases and boundary conditions, especially when array slicing or manipulating indices. Test your fix thoroughly with small example inputs before submitting.

When the code is valid concentrate on improving performance, Experiment with different neighborhood search strategies, candidate lists, and data structures tailored to these algorithms. Consider the trade-offs between exploration and exploitation.

Figure 4: Comparison of prompts before and after evolution.

worst-fit baselines, it lowers errors from 5.61% and 14.66% to 1.59% and 1.65%, respectively. These results underscore EvoPH's robust capability to rectify poorly performing heuristics and its superior search efficiency.

4.4 Further Analysis

4.4.1 Ablation Study

To investigate the individual contributions of each core component within our proposed framework, we ablation experiments on the TGB dataset, comparing the outcomes with those obtained from the complete framework. We design the following three ablation variants: (1) *w/o Strategy Sampling*: This variant removes the strategy sampling component from the framework; (2) *w/o Prompt Evolution*: This variant replaces the dynamic prompt evolution module with a fixed prompt strategy; (3)*w/o Island-Based Elites Selection*: This variant removes the island model and the elites selection algorithm from the framework.

As shown in Table 2, the ablation results clearly demonstrate the necessity and effectiveness of each component within our framework. Specifically, without Strategy Sampling, performance drops across all tasks, showing that maintaining policy diversity is essential to ensure broader exploration. Without Prompt Evolution, the performance drops noticeably (e.g., Christofides from 5.17% to 9.24%), indicating that adaptive prompt updates are crucial for guiding effective mutations. Without Island-based Elites Selection, performance deteriorates significantly (e.g., nearest-insertion from 4.41% to 9.70%), confirming that the island model based elites selection act as the foundational mechanism for sustaining both robustness and high-quality solutions.

Table 2: Performance comparison of the EvoPH framework and its different ablation versions on various heuristic algorithms. (SS: *Strategy Sampling*, PE: *Prompt Evolution*, IES: *Island-based Elites Selection*)

	Nearest Insert	**2-opt**	**Christofides**	**Farthest Insert**	**Nearest Neighbor**	**Random Insert**
EvoPH	4.41%	4.20%	5.17%	4.05%	4.41%	4.41%
w/o SS	5.17%	4.38%	5.17%	5.17%	5.17%	4.41%
w/o PE	5.17%	5.17%	9.24%	5.50%	5.17%	5.17%
w/o IES	9.70%	5.17%	7.03%	5.99%	6.48%	6.90%

4.4.2 Robustness and Effectiveness of EvoPH

We conducted further experiments to assess the robustness of EvoPH in terms of heuristic executability and convergence behavior.

Executability. Under the same experimental settings, we examined the proportion of executable heuristics generated over 20 iterations, specifically comparing EvoPH with and without the prompt evolution module. As shown in Figure 2, incorporating prompt evolution leads to higher success rates. This demonstrates that evolving meta-prompts improves the likelihood of producing reliable code within a limited number of iterations.

Convergence. We also investigated the convergence behavior of multiple heuristics starting from different initializations. As illustrated in Figure 3, despite initial performance gaps, all algorithms follow a similar trajectory: rapid quality improvements in early iterations, followed by gradual gains and eventual stabilization. This consistent trend highlights the general effectiveness of EvoPH across diverse initialization strategies.

Stochastic Stability. Notably, heuristics evolved by EvoPH often involve stochastic components. While we enforced fixed random seeds in our main experiments to strictly compare algorithmic logic,

Before Evolution

```
def solve_tsp_approximate(dist_matrixnp.ndarray):
...
 improved=True
  while improved:
      improved=False
      for I in range(n-1):
        for j in range(i+2,n):
          a,b,c,d=tour[i-1],tour[i],tour[j-1],tour[j]
          delta=dist_matrix[a,c]+dist_matrix[b,d]
                -dist_matrix[a,b]-dist_matrix[c,d]
          if delta<0:
            tour[i:j]=tour[i:j][::1],
            improved=True,
            break
          if improved:
            break

...
```

After Evolution

```
def solve_tsp_approximate(dist_matrix: np.ndarray):
...
    improved = True
      while improved:
        improved = False,best_change = 0,best_swap = None
        for i in range(n - 1):
          for j in range(i + 2, n):
            a, b = tour[i], tour[(i + 1) % n]
            c, d = tour[j], tour[(j + 1) % n]
            change=dist_matrix[a, b]+dist_matrix[c,d]-(dist_matrix[a,c] + dist_matrix[b,d])
            if change > best_change:
              best_change = change
              best_swap = (i, j)
            if  best_swap:
               i, j = best_swap
               tour[i+1:j+1] = reversed(tour[i+1:j+1])
               improved = True
...
```

Figure 5: Comparison of heuristic algorithm before and after evolution.

we further evaluated the algorithms in an unconstrained setting to ensure the reported superiority was not an artifact of specific seeds. Each algorithm was executed five times with distinct random seeds. As shown in the box plots in Figure 6, the low variance indicates that our generated heuristics maintain consistent and high performance even under stochastic variations.

Table 3: Performance Comparison of Different Models on Heuristics. Note that **Gemini** denotes gemini-2.5-pro, **GPT** denotes gpt-4o, **DeepSeek** denotes deepseek-r1, and **Llama** denotes Llama3.1-405B-Instruct.

Heuristic	Gemini	GPT	Deepseek	Llama3.1
2-opt	4.20%	5.17%	5.17%	5.70%
Christofides	5.17%	6.62%	5.74%	6.62%
Nearest Insertion	4.41%	6.83%	5.46%	7.82%
Farthest Insertion	4.05%	7.43%	6.62%	7.54%
Nearest Neighbor	4.41%	8.71%	7.54%	13.67%
Random Insertion	4.41%	7.92%	5.25%	8.34%

4.4.3 Case Study

Case of prompt evolution. As shown in Figure 4, the evolved prompts demonstrate clearer structure and stronger task orientation than their pre-evolution counterparts. They adopt a hierarchical instruction framework that prioritizes code correctness and functionality. When the code is invalid, the prompts restrict the task to error repair, specifying detailed steps . Once validity is ensured, the focus shifts to performance optimization with explicit strategies.

Case of heuristics evolution. According to Figure 5, compared with the pre-evolutionary algorithm that adopts the first-improvement strategy, the key advantage of the evolved algorithm lies in its exhaustive evaluation of all possible exchanges, where instead of stopping at the first improving move, the algorithm scans through every candidate swap and identifies the globally optimal modification. By applying this best-improvement mechanism in each iteration, the evolved algorithm achieves a more consistent and thorough enhancement of solution quality.

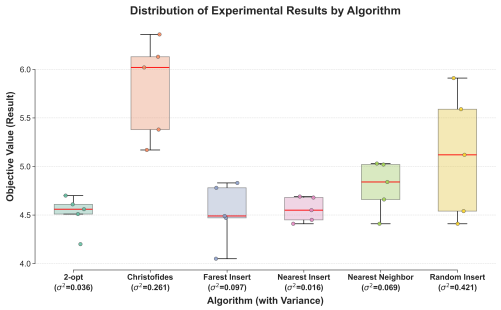

Figure 6: Distribution of experimental results and variances across different heuristic algorithms.

5 Conclusion

This paper proposes a structured co-evolutionary framework, EvoPH, designed to efficiently solve combinatorial optimization problems through the iterative evolution of prompts and heuristic algorithms. The synergy between macro-level population management and micro-level co-evolution facilitates nested optimization, avoiding local optima and enhancing the performance of algorithm design. Extensive experimental results on the TGB and BOB demonstrate that EvoPH outperforms existing evolutionary computation methods in terms of solution quality. Future research will focus on expanding this framework to a broader range of combinatorial optimization problems.

Limitations

While EvoPH demonstrates significant improvements in automatic algorithm design, several limitations remain. First, our evaluation is currently restricted to the TSP and BPP; extending the framework to more complex, multi-objective, or highly constrained real-world scenarios requires further validation. Second, despite avoiding the high computational costs of model fine-tuning, the reliance on iterative LLM inference for population evolution and island migration incurs cumulative latency and token overhead, which may hinder deployment in strict real-time environments. Finally, the inherent stochasticity of LLM generation introduces variability in convergence trajectories, necessitating rigorous statistical aggregation to ensure the reliability of the evolved heuristics.

References

Hottung André and Tierney Kevin. 2020. *Neural Large Neighborhood Search for the Capacitated Vehicle Routing Problem*. IOS Press.

Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. 1997. Handbook of evolutionary computation. *Release*, 97(1):B1.

Dikshit Chauhan, Bapi Dutta, Indu Bala, Niki van Stein, Thomas Bäck, and Anupam Yadav. 2025. Evolutionary computation and large language models: A survey of methods, synergies, and applications. *arXiv preprint arXiv:2505.15741*.

Jinbiao Chen, Jiahai Wang, Zizhen Zhang, Zhiguang Cao, Te Ye, and Siyuan Chen. 2023. Efficient meta neural heuristic for multi-objective combinatorial optimization. *Advances in Neural Information Processing Systems*, 36:56825–56837.

V. Corvello, J. V. de Carvalho, J. P. de Sousa, C. Oliveira, M. Carravilla, P. S. Martins, and J. F. Oliveira. 2010. BPPLIB: a bin packing problem library. In *Proceedings of the 3rd International Symposium on Engineering, MONACO'10*, Monaco.

George B Dantzig and John H Ramser. 1959. The truck dispatching problem. *Management science*, 6(1):80–91.

Maxence Delorme, Manuel Iori, and Silvano Martello. 2018. Bpplib: a library for bin packing and cutting stock problems. *Optimization Letters*, 12(2):235–250.

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

Agoston E Eiben and James E Smith. 2015. *Introduction to evolutionary computing*. Springer.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. 2019. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32.

Google. 2024. OR-Tools.

Shuhan Guo, Nan Yin, James Kwok, and Quanming Yao. 2025. Nested-refinement metamorphosis: Reflective evolution for efficient optimization of networking problems. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 17398–17429.

Gurobi Optimization, LLC. 2022. *Gurobi Optimizer Reference Manual*. Version 9.5.2.

Juraj Hromkovič. 2013. *Algorithmics for hard problems: introduction to combinatorial optimization, randomization, approximation, and heuristics*. Springer Science & Business Media.

Ziyao Huang, Weiwei Wu, Kui Wu, Jianping Wang, and Wei-Bin Lee. 2025. Calm: Co-evolution of algorithms and language model for automatic heuristic design. *arXiv preprint arXiv:2505.12285*.

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

William B Langdon and Riccardo Poli. 2013. *Foundations of genetic programming*. Springer Science & Business Media.

Robert Lange, Yingtao Tian, and Yujin Tang. 2024. Large language models as evolution strategies. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 579–582.

Zhaoyu Li, Jinpei Guo, and Xujie Si. 2023. G4satbench: Benchmarking and advancing sat solving with graph neural networks. *arXiv preprint arXiv:2309.16941*.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024a. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*.

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

Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Zhe Zhao, Xi Lin, Xialiang Tong, Mingxuan Yuan, Zhichao Lu, Zhenkun Wang, and 1 others. 2024b. A systematic survey on large language models for algorithm design. *arXiv preprint arXiv:2410.14716*.

Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. 2024c. Large language models as evolutionary optimizers. In *2024 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.

Shengcai Liu, Yu Zhang, Ke Tang, and Xin Yao. 2023b. How good is neural combinatorial optimization? a systematic evaluation on the traveling salesman problem. *IEEE Computational Intelligence Magazine*, 18(3):14–28.

Fu Luo, Xi Lin, Fei Liu, Qingfu Zhang, and Zhenkun Wang. 2023. Neural combinatorial optimization with heavy decoder: Toward large scale generalization. *Advances in Neural Information Processing Systems*, 36:8845–8864.

Zeyuan Ma, Hongshu Guo, Jiacheng Chen, Zhenrui Li, Guojun Peng, Yue-Jiao Gong, Yining Ma, and Zhiguang Cao. 2023. Metabox: A benchmark platform for meta-black-box optimization with reinforcement learning. *Advances in Neural Information Processing Systems*, 36:10775–10795.

Nelishia Pillay and Rong Qu. 2018. *Hyper-heuristics: theory and applications*. Springer.

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

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, and 1 others. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475.

Daniel Selsam. 2019. *Neural Networks and the Satisfiability Problem*. Stanford University.

Jiwoo Son, Zhikai Zhao, Federico Berto, Chuanbo Hua, Changhyun Kwon, and Jinkyoo Park. 2025. Neural combinatorial optimization for real-world routing. *arXiv preprint arXiv:2503.16159*.

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

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

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

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, and 1 others. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2).

A Problem Introduction

In this section, We start by presenting two representative NP-hard problems, the Traveling Salesman Problem and the Bin Packing Problem, which serve as running examples throughout this work.

A.1 Traveling Salesman Problem

The TSP is a fundamental NP-hard problem in combinatorial optimization. It is defined on a fully weighted graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ is a set of n vertices, and E is the set of edges containing all unordered pairs of distinct vertices, i.e., $E = \{\{v_i, v_j\} \mid v_i, v_j \in V, i \neq j\}$. A weight function $w : E \to \mathbb{R}^+$ assigns a non-negative cost $w(v_i, v_j)$ to each edge. The goal is to find a Hamiltonian cycle, i.e., a cycle that visits each vertex in V exactly once, with the minimum total weight. If a tour is represented by a permutation π of the vertex indices, the optimization objective is:

$$\pi^* = \arg\min_{\pi} \left(\sum_{i=1}^{n-1} w(v_{\pi_i}, v_{\pi_{i+1}}) + w(v_{\pi_n}, v_{\pi_1}) \right). \tag{3}$$

A.2 Bin Packing Problem

The BPP is another classical NP-hard problem in combinatorial optimization. An instance consists of a set of n items $I = \{i_1, i_2, ..., i_n\}$ with associated sizes $S = \{s_1, s_2, ..., s_n\}$, and an infinite supply of bins, each with a fixed capacity C. The objective is to partition the item set I into the minimum number of disjoint subsets $B_1, B_2, ..., B_k$, where each subset corresponds to the contents of a bin, subject to the capacity constraint:

$$\forall j \in 1, ..., k, \quad \sum_{i_m \in B_j} s_m \leq C. \tag{4}$$

The optimization goal is to minimize k, the total number of bins used. In the offline setting, all items are known in advance, while in the online setting, items arrive sequentially and must be placed before the next item is revealed. In this work, we focus on the offline setting.

B Strategies of evolution

In this section, we introduce the evolutionary strategies that guide mutation within EvoPH, ranging from parameter modification to completely rewrite. These strategies, embedded in the prompt design, balance stability with exploration, ensuring effective heuristics evolution. The complete set and their respective roles are summarized below:

- **Parameter modification**. The most conservative strategy instructs LLM to focus on identifying and fine-tuning hard-coded constants, thresholds, or hyper-parameters in the algorithm to explore the potential of existing algorithms without changing the core logic.
- **Redundancy Removal**. Focuses on algorithm optimization and efficiency, requiring LLM to analyze and remove unnecessary calculations, repeated logical judgments, or simplified code snippets to improve algorithm execution efficiency.
- **Structural modification**. A moderately exploratory strategy that guides LLM to adjust the existing code structure, such as changing the nesting of loops, replacing data structures, or adjusting the order of function calls.
- **Heuristic rewrite**. A more radical strategy requires LLM to identify a core heuristic rule or submodule in the algorithm and try to rewrite it with a completely new, functionally equivalent or better logic, aiming to achieve innovation in key steps.
- **Completely rewrite**. The most exploratory strategy, instructing LLM to completely abandon the existing implementation and write a completely new version from scratch while retaining the original algorithm intent (solving a specific problem). This strategy is used to make a disruptive attempt when the evolution has reached a serious stagnation.

C Initial heuristics for evolution

C.1 Heuristic algorithm for TSP

This subsection introduces several representative heuristic algorithms for the TSP, ranging from simple greedy methods to more sophisticated approaches with theoretical guarantees.

- **Nearest Neighbor**. The Nearest Neighbor algorithm is a simple greedy heuristic that constructs a tour by starting at an arbitrary city and repeatedly traveling to the closest unvisited city. This process continues until every

```
You are a top expert in combinatorial optimization and algorithms. Your task is to **iteratively evolve and refine** an existing Python function, `solve_tsp_approximate(dist_matrix)`, to solve the Traveling Salesperson Problem (TSP).
## 🚀 Evolutionary Goal
Your primary objective is to enhance the provided function in each evolutionary step, pushing it towards the optimal balance of solution quality and speed. Assume you are receiving a function that already exists and needs improvement.

## 📈 Key Optimization Metrics
Your success is measured on a trade-off between two competing goals:
1. **Minimize Relative Error (Solution Quality)**: The solution's path length must be as close to the known optimum as possible. This is the top priority.
2. **Minimize Execution Time (Computational Efficiency)**: The function must execute extremely quickly within the evaluator's strict time limit. A faster solution is often better than a marginally more accurate but slower one.

## 📝 Your Evolutionary Directives
In each round, you must analyze the function provided to you and then rewrite it to be better. Follow this process:
1. **Analyze**: Quickly understand the current function's strategy. What heuristic is it using? What are its potential weaknesses (e.g., slow loops, a simple heuristic that gets stuck)?
2. **Strategize**: Decide on the best evolutionary step.
   * Is the current algorithm good but implemented inefficiently? **Refine it** by optimizing loops or using better data structures.
   * Is the algorithm too basic? **Replace it** with a more powerful one from the toolbox.
   * Does the function already have a strong local search? **Enhance it** .
   * Can two ideas be combined? **Hybridize** different techniques for a better result.
3. **Implement**: Rewrite the function with your proposed improvements, ensuring it remains robust and efficient.

## 💡 Performance & Implementation Tips
* **NumPy is Your Friend**: Always favor `NumPy` for vectorized and matrix operations to maximize speed.
* **Smart Search**: For local search, a "first improvement" strategy (find one good swap and restart the search) is often faster than "best improvement" (testing all possible swaps).
* **Time-Awareness**: The best algorithms are useless if they time out. The function must return a solution within the time limit.
* **Complete provision**:Provide complete and executable code, The input parsing of the main function and data needs to be completely consistent with the source code.
```

(a) Prompt template for iteratively evolving heuristics on the TSP.

```
You are a world-class authority on combinatorial optimization and high-performance computing. Your mission is to **iteratively transform and perfect** an existing Python function to solve the Bin Packing Problem (BPP). The function you evolve must strictly adhere to the signature:
`solve(capacity, items)`

## 🚀 Evolutionary Goal
Your primary directive is to continuously enhance the provided function, treating each version as a candidate in an evolutionary process. The ultimate aim is to produce a solution that achieves the lowest possible number of bins for any given problem instance. You are improving upon existing work, not starting from scratch.

## 📈 Key Optimization Metrics
Your performance is judged on this critical, prioritized metrics:
**Minimize Relative Error (The Primary Objective)**: The absolute priority is to minimize the number of bins used. The closer this number is to the theoretical optimum, the higher the quality of the solution.

## 📝 Your Evolutionary Directives
In each evolutionary cycle, dissect the current function and rebuild it to be superior. Adhere to this methodology:
1. **Analyze**: Evaluate the incumbent function's methodology. Is it a simple, non-sorted greedy algorithm? Does it handle large items effectively? Identify its primary limitation.
2. **Strategize**: Formulate a clear plan for the next evolutionary leap, considering a hierarchy of improvements like adding sorting (FFD/BFD), introducing advanced heuristics (RFF, Grouping), or adding post-hoc optimization.
3. **Implement**: Rewrite the `solve` function to incorporate your new strategy.

## 💡 Performance & Implementation Tips for BPP
* **Adhere to the Capacity Constraint**: This is the fundamental rule of BPP. The sum of item sizes in any bin must **never** exceed `capacity`. Every item provided must be packed.
* **Core BPP Concepts**: Leverage the "offline" advantage by pre-sorting items. Aim to build upon "Decreasing" heuristics like FFD and BFD as a baseline.
```

(b) Prompt template for iteratively evolving heuristics on the BPP.

```
You are an expert in optimizing prompt words. Please analyze the following information and generate a better system prompt:

### Original prompt words:
{original_prompt}

### Parent program metrics:
{json.dumps(parent_metrics, indent=2)}

### Child program indicators:
{json.dumps(child_metrics, indent=2)}

### Complete dialogue history:
{json.dumps(history_entry, indent=2)}

###Requirement:
1. If the subroutine performs worse, point out the defect of the original prompt word
2. Generate an improved system prompt (output directly without explanation)
3. Keep prompt words concise and effective, focus on code evolution tasks
4. The format of the generated prompt should be as similar as possible to the initial prompt
5. If the code execution is effective, generate prompt words to prompt improvement. If the code execution fails (i.e. returns a large negative value), generate prompt words to prompt correction of errors in the code
6. Generate system prompt words that are more detailed than the initial system prompt words as much as possible, with more information that can be included in the prompt words
7. Keep the role description section(first part) unchanged
```

(c) Meta-prompt designed to refine and improve existing prompts based on execution feedback and performance metrics.

Figure 7: Initialization and evolution prompts used in EvoPH

city has been visited, at which point the tour is completed by returning to the starting city.

- **Nearest Insertion**. The Nearest Insertion algorithm builds a tour incrementally by starting with a small sub-tour of two cities and progressively adding more. In each step, it identifies the unvisited city that is closest to any city already on the sub-tour and then inserts it into the position along the tour's edge that causes the smallest increase in total length. This process is repeated until all cities have been incorporated into the tour.

- **Farthest Insertion**. The Farthest Insertion algorithm also builds a tour incrementally but uses an opposite selection criterion from Nearest Insertion. It starts with a small sub-tour and, at each step, selects the unvisited city that is the farthest from any city currently in the sub-tour. It then inserts this selected city into the edge of the sub-tour that results in the least additional travel distance.

- **Random Insertion**. The Random Insertion algorithm constructs a tour by starting with a small initial sub-tour and then inserting the remaining cities one by one in a completely random order. For each randomly selected city, the algorithm evaluates all possible insertion points along the edges of the current sub-tour and places the city in the position that minimizes the increase in the tour's total length.

- **2-Opt**. The 2-Opt algorithm is an improvement heuristic designed to refine an existing tour by systematically eliminating edge crossings. It works by iteratively selecting two non-adjacent edges in the tour and checking if swapping their endpoints to reconnect the path in a different order would shorten the total distance. If a beneficial swap is found, the tour is updated, and the process is repeated until no more length-reducing swaps are possible, resulting in a locally optimal solution.

- **Christofides Algorithm**. The Christofides algorithm is an advanced heuristic that provides a theoretical performance guarantee, ensuring the resulting tour is no more than 1.5 times the length of the optimal solution. It operates by first creating a Minimum Spanning Tree (MST) of all cities, then identifying all vertices with an odd degree and finding a minimum-weight perfect matching for them. By combining the MST and the matching, it forms an Eulerian circuit, which is then converted into a valid TSP tour by taking shortcuts to avoid revisiting cities.

D Prompt used in evolution

In this section, we present the initialization and evolution prompts that guide the EvoPH framework. Specifically, the initialization prompts for the TSP and the BPP define the evolutionary objectives, optimization metrics, and implementation directives for the respective tasks. In addition, a meta-level prompt is introduced to refine existing prompts based on execution feedback and performance indicators. These prompts, which serve as the foundation for both algorithmic evolution and reflective prompt refinement, are illustrated in Figure 7.

D.1 Heuristic algorithm for BPP

This subsection presents several classical heuristic algorithms for the BPP, highlighting their strategies for item placement and trade-offs between efficiency and packing quality.

- **First Fit**. The First Fit algorithm is an intuitive online algorithm. It processes each item sequentially and places it in the first bin with enough free space. If no bins are found that can hold the item, the algorithm moves to a new, empty bin and places the item there.

- **Best Fit**. The Best Fit algorithm aims to use space most efficiently. For each item, it searches for the box that can accommodate it and has the least amount of remaining space, also known as the "most compact" box. If all existing boxes cannot accommodate it, it will open a new box. This strategy attempts to avoid leaving large, unusable fragmented spaces in the box, but because it needs to check all boxes, it is slightly slower than the first fit algorithm.

- **Next Fit**. The Next Fit algorithm is the simplest and fastest heuristic, but it's generally the least efficient. It maintains a single, "current" active chest and attempts to place the next item into it. If it fits, it does so. If not, the algorithm simply "closes" the current chest (no longer considering it) and opens a new one for the item.

- **Worst Fit**. The Worst Fit algorithm is the inverse of the best-fit strategy. For each item, it searches for the bin with the largest remaining space that can accommodate it. The goal is to preserve a large, contiguous area to accommodate potentially large items in the future. However, this strategy often performs poorly in practice because it tends to prematurely occupy multiple bins, resulting in inefficient overall packing.