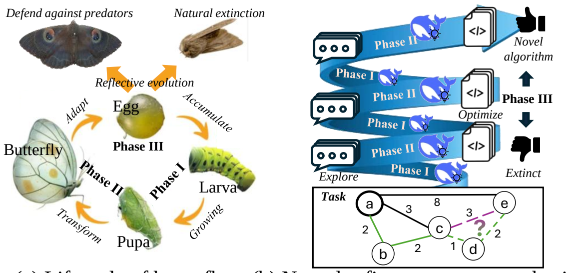
Nested-Refinement Metamorphosis: Reflective Evolution for Efficient Optimization of Networking Problems

Shuhan Guo[1], Nan Yin[2,*], James T. Kwok[2], Quanming Yao[1,3]
[1]Department of Electronic Engineering, Tsinghua University
[2]Department of Computer Science and Engineering,
Hong Kong University of Science and Technology
[3]State Key Laboratory of Space Network and Communications, Tsinghua University
{guoshuhan, qyaoaa}@tsinghua.edu.cn, yinnan8911@gmail.com, jamesk@cse.ust.hk

Abstract

Large Language Models (LLMs) excel in network algorithm design but suffer from inefficient iterative coding and high computational costs. Drawing inspiration from butterfly metamorphosis—where structured developmental phases (Phase I: larval nutrient accumulation → Phase II: pupal transformation) enable adaptive evolution—we propose Nested-Refinement Metamorphosis (NeRM). Building on this principle, we introduce Metamorphosis on Prompts (MoP) to iteratively refine task descriptions (e.g. latency / bandwidth constraints) and Metamorphosis on Algorithms (MoA) to generate more effective solutions (e.g. appropriate network processing architecture). Their nested refinement ensures task-algorithm alignment, systematically improving both task descriptions and algorithmic solutions for more efficient algorithm design. To further enhance efficiency, we incorporate predictor-assisted code evaluation, mimicking natural selection by filtering out weak candidates early and reducing computational costs. Experimental results on TSP (routing), MKP (resource allocation), and CVRP (service-network coordination) demonstrate that NeRM consistently outperforms state-of-the-art approaches in both performance and efficiency.



(a) Lifecycle of butterfly. (b) Nested-refinement metamorphosis.

Figure 1: (a) The lifecycle of a butterfly. Phase I: initial growth, Phase II: a profound transformation, and Phase III: natural filtering. (b) NeRM consists of MoP (Phase I), MoA (Phase II), and their interaction (Phase III).

1 Introduction

Large Language Models (LLMs)(Wei et al., 2022; Chen et al., 2023; Madaan et al., 2024) for algorithm design have gained significant attention across various domains. These models have proven effective in tasks such as automated code generation(Chen et al., 2021; Ni et al., 2023; Liu et al., 2023), program synthesis(Li et al., 2024; Barke et al., 2024; Khan et al., 2024), and algorithm optimization (Huang et al., 2024; Romera-Paredes et al., 2024; Ye et al., 2024; Liu et al., 2024a). These applications demonstrate the ability of LLMs to automate and optimize traditional processes, improving development workflows and enabling innovative solutions across fields.

Currently, LLM-based automatic algorithm design (Romera-Paredes et al., 2024; Ye et al., 2024; Liu et al., 2024a) primarily focuses on code evolution, where models iteratively generate and refine code through search-based strategies. These approaches require generating a large pool of candidate solutions and individually evaluating each one to identify high-quality algorithms. However, without a mechanism to refine task descriptions, evolved code often lacks alignment with problem requirements, leading to inefficient search that produces redundant or suboptimal solutions (Pearce et al., 2022). Moreover, the necessity of evaluating every candidate imposes significant computational costs, making the process impractical for large-scale problems. These challenges highlight the need for a more structured framework that integrates both prompt refinement and algorithm evolution, enabling more targeted exploration while reducing evaluation overhead.

Inspired by biological metamorphosis[1], we take the butterfly lifecycle as an analogy (Figure 1a), which unfolds of three key phases: **Phase I**: meta-

*Corresponding author

[1]https://en.wikipedia.org/wiki/Metamorphosis

Findings of the Association for Computational Linguistics: ACL 2025, pages 17398–17429

morphosis from larva to pupa, where the larva undergoes growth and nutrient accumulation in preparation for transformation into a pupa; **Phase II**: metamorphosis from pupa to butterfly, a structural transition where the butterfly adapts to its environment and emerges as a fully developed adult; and **Phase III**, where the butterfly's adaptation shapes the evolutionary trajectory of the next generation of larvae. We identify the challenges of algorithm design as *Nested-Refinement Metamorphosis* (Figure 1b), which mirrors the butterfly's transformation by iteratively refining both prompts and algorithms to enhance their effectiveness and efficiency.

Specifically, we propose a novel framework named **Ne**sted-**R**efinement **M**etamorphosis: Reflection and Evolution of Prompts and Code for Efficient Algorithm Design (NeRM), a novel framework that enables the generation of effective algorithms with high efficiency. NeRM introduces a nested metamorphosis process that combines Metamorphosis on Prompts (MoP) and Metamorphosis on Algorithms (MoA). MoP iteratively refines task descriptions to improve guidance for solution generation, while MoA leverages these refined prompts to evolve algorithms through reflection-driven adaptation (Wang et al., 2024a) and evolutionary optimization (Liu et al., 2024c), ensuring better task alignment. The interaction between MoP and MoA forms a self-improving feedback loop, continuously enhancing algorithm performance. To further improve efficiency, NeRM incorporates a predictor-assisted evaluation mechanism, which mimics natural selection by filtering out weak candidates early. This prioritization of promising solutions reduces computational overhead while maintaining solution quality.

Our contributions can be summarized as follows:

- We introduce the concept of biologically inspired *Nested-Refinement Metamorphosis*, which captures the interdependent refinement of both task descriptions and algorithmic solutions, ensuring continuous adaptation and improvement.
- We propose NeRM, combining MoP for iterative task refinement and MoA for adaptive algorithm evolution, and accelerate the evolution using a code predictor.
- We evaluate the proposed NeRM across a diverse range of complex optimization problems, demonstrating that NeRM surpasses state-of-the-art methods in efficiency and effectiveness.

2 Related Works

LLMs for Algorithm Design. LLMs have been widely applied in different aspects of algorithm design. Specifically, LLMs are used for automated code generation (Chen et al., 2021; Ni et al., 2023; Liu et al., 2023), program synthesis (Li et al., 2024; Barke et al., 2024; Khan et al., 2024), and algorithm optimization (Huang et al., 2024; Romera-Paredes et al., 2024; Ye et al., 2024; Liu et al., 2024a).

The most related work to NeRM includes FunSearch (Romera-Paredes et al., 2024) and EoH (Liu et al., 2024a), both of which leverage LLMs to evolve code generation for function search. While these methods have demonstrated promising results, often surpassing handcrafted algorithms in specific tasks, they focus exclusively on code evolution without refining task descriptions, which is crucial for guiding the search toward more effective solutions. Moreover, their reliance on exhaustive candidate evaluation leads to high computational costs, as every generated algorithm must be fully executed and assessed, making the process inefficient for large-scale or complex problems.

Prompt Evolution with LLMs. Prompting has emerged as a powerful technique for leveraging LLMs in specialized tasks, with performance heavily influenced by prompt selection. Recent advancements in automatic prompt optimization have drawn significant attention, particularly in continuous prompt tuning, where input token embeddings are fine-tuned to improve model responses (Li and Liang, 2021; Liu et al., 2021; Lester et al., 2021). Despite its effectiveness, this approach requires access to LLM parameters, often unavailable for black-box APIs, and soft prompts lack interpretability, making them harder to analyze and refine (Lester et al., 2021). In contrast, discrete prompting, which involves predefined tokens or task instructions (Schick and Schütze, 2020), offers a more interpretable and human-friendly interface. However, relying solely on prompt evolution for code generation can introduce inconsistencies, as prompts cannot directly modify or optimize the logic of generated code (Marcus, 2020).

3 Methodology

This work introduces NeRM, comprising two components: **Nested Refinement Metamorphosis** and **Predictor-Assisted Code Evaluation**. Nested Refinement Metamorphosis integrates Metamorphosis on Prompts (MoP), which simulates Phase I of

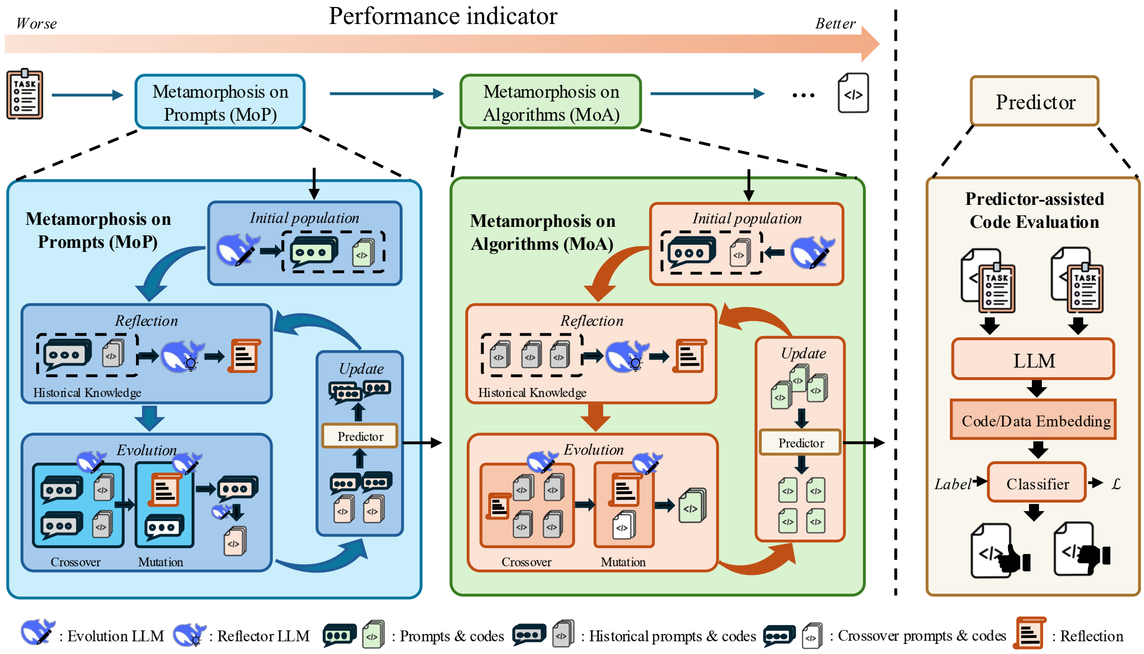


Figure 2: Overview of NeRM, which integrates nested-refinement metamorphosis and predictor-assisted code evaluation. The nested-refinement metamorphosis consists of the nests between Metamorphosis on Prompts (MoP) and Metamorphosis on Code (MoA) to generate algorithms effectively. Additionally, NeRM incorporates predictor-assisted code evaluation to efficiently filter high-quality solutions, reducing computational overhead.

biological metamorphosis by refining task descriptions, and Metamorphosis on Algorithms (MoA), which mimics Phase II by optimizing algorithms. Their nested interaction simulates Phase III, driving continuous co-evolution for more effective solutions. Predictor-Assisted Code Evaluation filters out low-quality algorithms early, reducing computational costs while maintaining solution quality. An overview of NeRM is shown in Figure 2.

3.1 Nested-Refinement Metamorphosis

Nested-refinement metamorphosis overcomes the limitations of existing methods that iteratively generate code without refining task descriptions to better guide the search process, leading to inefficient exploration and high computational costs. Inspired by biological evolution, which mimics the phases of biological development, we propose (i) *Metamorphosis on Prompts (MoP)*, which corresponds to the growth and accumulation phase, where the task description is iteratively refined to guide exploration more effectively; and (ii) *Metamorphosis on Algorithms (MoA)*, which mirrors the transformation phase, where the generated code undergoes iterative refinement, continuously adapting to task requirements and enhancing its quality.

MoP and MoA. Both MoP and MoA begin with a diverse initial population, where MoP combines manually crafted prompts to integrate human expertise with LLM-generated prompts for diversity, while MoA generates an initial heuristic population guided by MoP prompts and an LLM-based generator. In the reflection phase, a reflector LLM extracts insights from past refinements, synthesizing key takeaways, and suggesting improvements to ensure continuous evolution based on previous performance. During the evolution phase, MoP and MoA apply crossover and mutation to generate new candidates: MoP combines and modifies prompts, while MoA generates new heuristics through LLM-driven refinements. Finally, in the update phase, both processes evaluate the candidates generated in a development set, using a code predictor to retain high-quality prompts and algorithms while filtering out weaker ones. The details of the MoP and MoA design are described in Table 1.

Nested-Refinement. NeRM achieves effective algorithm generation through iterative refinement of both MoP and MoA. The process begins with initialization, where a diverse set of prompts and algorithms are generated. In the MoP phase, task specifications are continuously refined based on past performance to improve alignment and robustness. The MoA phase then evolves heuristic solutions using genetic-inspired operations, guided by the refined prompts. Finally, through iterative refinement, MoP and MoA update each other over multiple cycles, ensuring both prompts and algo-

Table 1: Details of the Metamorphosis on Prompts and Code

Module	Metamorphosis on Prompts (MoP)	Metamorphosis on Algorithms (MoA)
Initial population	In contrast to most automated prompt-based methods that disregard human expertise, we initialize the population with manually crafted prompts to incorporate prior knowledge while enhancing diversity with LLM-generated ones.	MoA creates a heuristic population using the generator LLM, guided by MoP prompts and an initial seed heuristic. This approach provides examples that encourage valid heuristics and search toward promising solutions.
Reflection	MoP refines heuristics through iterative reflections, where the reflector LLM generates summaries and provides suggestions to improve design. These reflections can start from scratch or incorporate predefined manual hints.	Similar to MoP, MoA systematically gathers knowledge through historical iterative reflections. The reflector LLM synthesizes key insights and offers strategic recommendations to improve heuristic algorithm design.
Evolution	The prompt evolution process involves crossover and mutation operations. Crossover combines parent prompts to generate offspring, while mutation introduces random changes to specific components.	With the provided prompts and a pair of parent heuristics, MoA guides LLMs to generate a new offspring heuristic. It then applies a mutation strategy, prompting the generator LLM to refine heuristics for further improvement.
Update	We assess the generated candidate prompts/algorithms using a development set and keep those that demonstrate strong performance. To further enhance efficiency, we leverage the code predictor to filter high-quality candidates.	

rithms co-evolve, progressively converging toward a more effective algorithm. The details of the process are illustrated in Algorithm 1.

3.2 Predictor-assisted Code Evaluation

Existing algorithm design methods suffer from inefficiency due to their reliance on exhaustive validation, where every generated candidate must be fully evaluated to determine its quality. This process leads to high computational costs, especially as iterative refinement of MoP and MoA produces large volumes of algorithms. Inspired by natural selection in biological evolution, where weaker individuals are eliminated early to conserve resources for stronger candidates, we introduce *Predictor-assisted Code Evaluation* to prioritize high-quality solutions without exhaustive testing.

Design of the Predictor. Traditional token-based prediction methods (Papineni et al., 2002; Lin, 2004; Denkowski and Lavie, 2014) rely on textual similarity, failing to capture functional correctness and leading to inaccurate performance evaluation. Additionally, LLM-based methods (Zhuo, 2024; Tong and Zhang, 2024) introduce inconsistency due to model stochasticity and incur high computational costs, making them inefficient for large-scale heuristic ranking. To address these limitations, we leverage LLM embeddings to map code and data into a shared space, enabling execution-based performance estimation. Unlike traditional embeddings that focus on syntax or token similarity, LLM embeddings capture semantic and structural properties, leading to more accurate heuristic assessment. Additionally, our pairwise ranking model efficiently selects high-quality candidates, significantly reducing computational overhead while ensuring scalability and effectiveness.

Specifically, we first construct a training set for predictor training (detailed in Appendix A.1) using pairwise algorithm comparisons (e.g., $\mathcal{D}_{ij} = (\text{code}_i, \text{data}_i, \text{code}_j, \text{data}_j)$), where the labels are:

$$\text{label}(\mathcal{D}_{ij}) = \begin{cases} 1 & \text{if } \ \text{gap}_i > \text{gap}_j, \\ 0 & \text{otherwise.} \end{cases}$$

During training, we first utilize the text embedding LLM to encode both the heuristic code and associated problem data, which is formulated as:

$$\mathbf{E}_i = \mathsf{Enc}(\text{code}_i), \mathbf{D}_j = \mathsf{Enc}(\text{data}_j),$$

where $\mathbf{E}_c$, $\mathbf{E}_d$ are the embeddings of code and data, Enc is the text embedding encoder. Then, for each pair of heuristic code, we concate the embeddings of codes and data and fed them into a Multi-Layer Perceptron (MLP) for performance comparison:

$$p_{ij} = \mathsf{MLP}\big(\mathbf{E}_i||\mathbf{D}_i||\mathbf{E}_j||\mathbf{D}_j\big),$$

where p_{ij} is the probability that code_i outperforms code_j. We optimize the predictor with the objective function: $\mathcal{L} = \sum l\big(p_{ij}, \text{label}(\mathcal{D}_{ij})\big)$, where l is Binary Cross Entropy (BCE) loss (Bishop and Nasrabadi, 2006).

Algorithm 1 Learning Process of NeRM.

Input: Problem description; initial algorithms; LLM; interactive times t of MoP and MoA; interactive times T of nested-refinement.
Output: Generated optimal algorithm.

// Nested-Refinement Metamorphosis
Initial population with a diverse set of prompts and heuristics.
for $k = 1, 2 \cdots, T$ **do**
  // Metamorphosis on Prompts (MoP)
  for $i = 1, 2, \cdots, t$ **do**
    Initial population;
    Reflection & Evolution;
    Update prompts with code predictor;
    Update predictor parameters;
  end for
  // Metamorphosis on Algorithms (MoA)
  for $j = 1, 2, \cdots, t$ **do**
    Initial population;
    Reflection & Evolution;
    Update algorithms with code predictor;
    Update predictor parameters;
  end for
end for

Evolution of Predictor. During prediction, we begin by randomly sampling K code instances as initial candidates for validation, forming an algorithm pool of size K. The remaining generated codes are then evaluated against these K samples using the predictor, which estimates their performance. Higher-performing candidates are retained, while weaker ones are discarded, ensuring that only the most promising algorithms remain.

Once MoP or MoA prediction is complete, we obtain the top K algorithms, which are then fully validated to determine their actual performance. The validation results are subsequently used to fine-tune the predictor's parameters, following the approach outlined in Section 3.2, enhancing its alignment with real-world performance. This adaptive selection process significantly reduces computational overhead while ensuring the generation of high-quality algorithms.

3.3 Comparison with Existing Works

FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024a), and ReEvo (Ye et al., 2024) rely solely on code evolution, lacking prompt reflection and evolution, making them highly dependent on initial prompts and limiting adaptability. FunSearch mutates code without refining task descriptions or incorporating solution feedback, leading to inefficient exploration. EoH evolves parent-generated heuristics without optimizing prompt guidance, making it prone to local optima and computationally expensive due to exhaustive evaluation. ReEvo introduces code reflection but lacks prompt refinement, requiring full execution for candidate assessment, increasing computational costs. In contrast, NeRM integrates MoP and MoA, enabling the co-evolution of prompts and algorithms, improving task alignment, and leveraging predictor-assisted evaluation to enhance efficiency while maintaining solution quality. The comparison is shown in Table 2.

Table 2: Comparisons with existing works.

Method	MoP	MoA		Predictor
		Reflection	Evolution	
FunSearch	✗	✗	✓	✗
EoH	✗	✗	✓	✗
ReEvo	✗	✓	✓	✗
NeRM	✓	✓	✓	✓

4 Experiments

4.1 Experimental Settings

Benchmarks. We conduct the proposed NeRM on three different combinatorial optimization problems to evaluate its effectiveness and efficiency. (1) *Traveling Salesman Problem (TSP):* Its target is to find the shortest possible route for a salesman to visit a set of cities exactly once and return to the origin city. (2) Multiple Knapsack Problem (MKP): This challenge involves selecting items, each with a given weight and value, to maximize the total value across multiple knapsacks, each with a specified capacity constraint. We also report the performance of the Capacitated Vehicle Routing Problem (CVRP) in Appendix A.3.

Baselines. For TSP, we compare NeRM against hand-crafted heuristics, including GLS (Voudouris et al., 2010), KGLS (Arnold and Sörensen, 2019), and EBGLS (Shi et al., 2018); prompt-based methods such as EEPE (Yu et al., 2023), CoAPT (Lee et al., 2024), and APR (Liu et al., 2024b); and evolution-based methods, including EoH (Liu et al., 2024a) and ReEvo (Ye et al., 2024). For MKP, we similarly compare NeRM with hand-crafted heuristics, specifically VNS (Mladenović and Hansen,

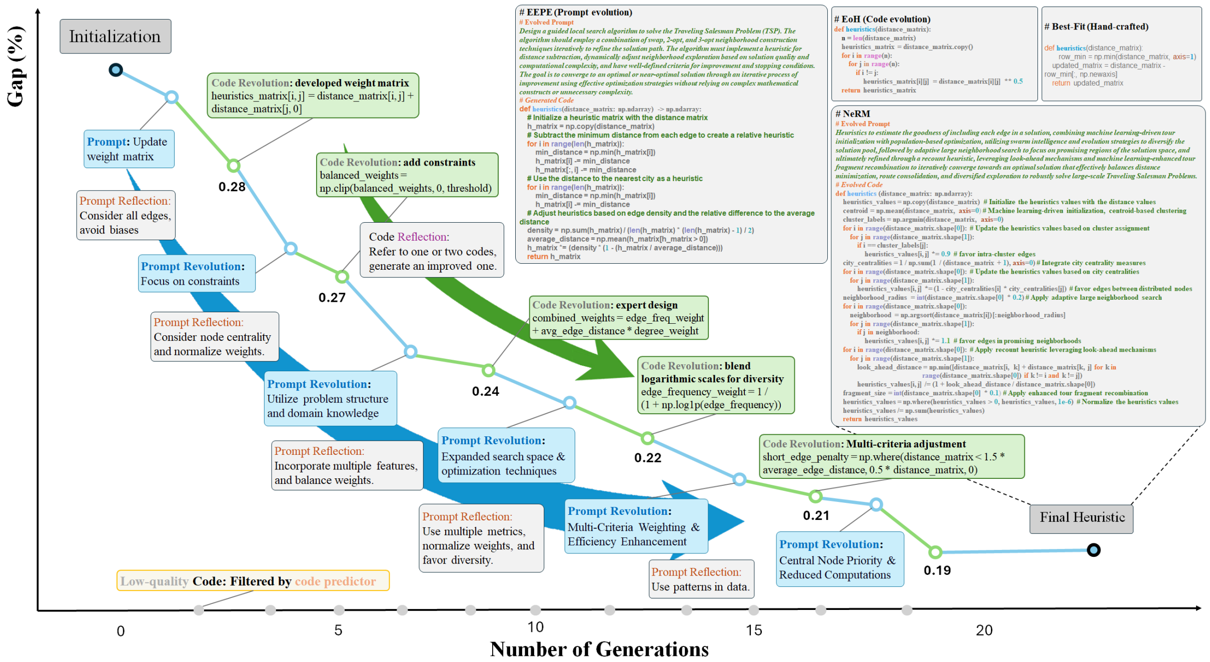


Figure 3: Evolution of NeRM for TSP. We outline the key prompts and best codes generated from NeRM. Moreover, we present the best algorithm in the final iteration and compare it with the best-fit heuristic, EoH and EEPE.

1997), TS (Fred Glover, 1997), and ACO (Dorigo and Di Caro, 1999), while keeping the comparisons with prompt and evolution-based methods consistent with those used for TSP. Details of experimental settings are illustrated in Appendix A.2.

Table 3: Comparisons of different methods on TSP. **Bold** results indicate the best performance.

Method	TSP100	TSP200	TSP500	TSP1000
GLS	$0.66_{\pm 0.00}$	$2.70_{\pm 0.00}$	$3.86_{\pm 0.00}$	$4.22_{\pm 0.00}$
KGLS	$0.01_{\pm 0.00}$	$0.28_{\pm 0.00}$	$0.92_{\pm 0.00}$	$1.55_{\pm 0.00}$
EBGLS	$0.44_{\pm 0.00}$	$1.36_{\pm 0.00}$	$2.65_{\pm 0.00}$	$4.09_{\pm 0.00}$
EEPE	$0.00_{\pm 0.00}$	$0.24_{\pm 0.04}$	$0.81_{\pm 0.07}$	$1.40_{\pm 0.17}$
CoAPT	$0.01_{\pm 0.00}$	$0.24_{\pm 0.23}$	$0.93_{\pm 0.99}$	$1.58_{\pm 1.64}$
APR	$0.01_{\pm 0.00}$	$0.24_{\pm 0.24}$	$0.80_{\pm 0.78}$	$1.46_{\pm 1.39}$
EoH	$0.04_{\pm 0.01}$	$0.36_{\pm 0.05}$	$1.48_{\pm 0.13}$	$2.26_{\pm 0.09}$
ReEvo	$0.00_{\pm 0.00}$	$0.23_{\pm 0.24}$	$0.85_{\pm 0.87}$	$1.42_{\pm 1.44}$
NeRM	$\mathbf{0.00}_{\pm 0.00}$	$\mathbf{0.19}_{\pm 0.04}$	$\mathbf{0.77}_{\pm 0.04}$	$\mathbf{1.31}_{\pm 0.08}$

4.2 Implementation Details

All experiments are conducted using GPT-3.5 Turbo in a computational environment equipped with an NVIDIA RTX 3070Ti GPU, an Intel i7-11700K CPU, and 32GB of RAM. Each NeRM is evaluated on 10 instances during training, generating 30 code candidates per iteration of MoP and MoA. The code predictor is then used to retain the top 3 candidates for the next generation. To assess the performance and stability of different evolution methods, each experiment is run three times, with the mean and variance reported.

Table 4: Comparisons of different methods on MKP. **Bold** results indicate the best performance.

Method	MKP100	MKP300	MKP500
VNS	$21.97_{\pm 0.00}$	$58.95_{\pm 0.00}$	$96.85_{\pm 0.00}$
TS	$21.91_{\pm 0.00}$	$50.84_{\pm 0.00}$	$76.28_{\pm 0.00}$
ACO	$22.86_{\pm 0.00}$	$58.99_{\pm 0.00}$	$97.03_{\pm 0.00}$
EEPE	$22.85_{\pm 0.30}$	$59.56_{\pm 1.46}$	$98.46_{\pm 2.55}$
CoAPT	$22.66_{\pm 0.14}$	$58.99_{\pm 0.52}$	$97.49_{\pm 0.83}$
APR	$22.95_{\pm 0.26}$	$60.27_{\pm 1.35}$	$99.78_{\pm 2.46}$
EoH	$22.93_{\pm 0.14}$	$59.39_{\pm 0.88}$	$98.10_{\pm 1.57}$
ReEvo	$22.91_{\pm 0.26}$	$59.64_{\pm 1.07}$	$98.57_{\pm 2.12}$
NeRM	$\mathbf{23.03}_{\pm 0.08}$	$\mathbf{61.13}_{\pm 0.37}$	$\mathbf{101.56}_{\pm 0.79}$

4.3 Performance Comparison

Effective of NeRM. Table 3 and 4 present performance comparisons for TSP and MKP, respectively. In both TSP and MKP, the prompt-based (e.g., EEPE, CoAPT, APR) and evolution-based methods (e.g., EoH, ReEvo) outperform traditional hand-crafted heuristics, demonstrating the potential of LLMs to generate effective heuristic solutions. Additionally, prompt-based methods achieve better performance than evolution-based methods, highlighting the importance of prompt refinement in guiding the generation of more effective algorithms. Lastly, NeRM consistently outperforms both prompt-based and evolution-based methods, demonstrating that the nested interaction between MoP and MoA effectively balances exploration and optimization, leading to more efficient and high-

Table 5: Performance of different prediction methods on the TSP200. **Bold** results indicate the best performance.

Type	Method	Accuracy
Token-based	BLEU	0.40
	chrF	0.41
	ROUGE-L	0.43
	CodeBLEU	0.39
	METEOR	0.31
	RUBY	0.42
LLM-based	VANILLA	0.45
	ICE-Score	0.53
	CodeJudge	0.60
Ours	NeRM	**0.78**

quality algorithm generation.

Additionally, we visualize the evolution of NeRM for TSP in Figure 3, illustrating key steps along with the generated prompts and algorithms. The fitness value (gap) gradually decreases from 0.28 to 0.19, demonstrating progressive improvement. Compared to the best-fit heuristic, as well as the algorithms generated by the code evolution method (EoH) and the prompt evolution method (EEPE), NeRM generates prompts that more precisely capture problem requirements and algorithms with a hybrid function composed of multiple components, enabling more comprehensive and adaptable solutions. Additional generated prompts and algorithms are provided in Appendix A.5.

Efficiency of NeRM. To demonstrate the efficiency of NeRM, we compare its time consumption and convergence speed against EoH and ReEvo in Figure 4. EoH represents a heuristic-based evolutionary approach, while ReEvo leverages LLMs as hyper-heuristics for adaptive algorithm search. These methods serve as strong baselines, as they both explore algorithm evolution but differ in refinement strategies. As shown in Figure 4a, the evaluation time for EoH and ReEvo increases exponentially with problem size, while NeRM maintains a linear growth trend. This efficiency gain can be attributed to the code predictor, which estimates algorithm performance without exhaustive evaluation of every candidate. Additionally, as shown in Figures 4b and 4c, NeRM achieves faster convergence and higher-quality code in TSP200 and MKP100. This improvement is largely due to the code predictor, which filters out low-quality candidates early in the evolution process, reducing unnecessary evaluations and accelerating convergence while ensuring that only promising solutions are refined. More results are shown in Appendix A.3.

Effect of Code Predictor. To assess the impact of the code predictor, we compare our approach with two alternative predictor types: token-based predictors, including BLEU (Papineni et al., 2002), chrF (Popović, 2015), ROUGE-L (Lin, 2004), CodeBLEU (Ren et al., 2020), METEOR (Denkowski and Lavie, 2014), RUBY (Tran et al., 2019), which estimate code quality based on textual similarity metrics; and LLM-based predictors, such as VANILLA (which directly queries LLMs to compare codes), ICE-Score (Zhuo, 2024), and CodeJudge (Tong and Zhang, 2024), which leverage pre-trained models to assess the effectiveness of generated algorithms. These predictor types represent fundamentally different evaluation strategies, enabling a comprehensive comparison of how different prediction mechanisms influence both efficiency and solution quality. Details of the implementations are provided in Appendix A.2.

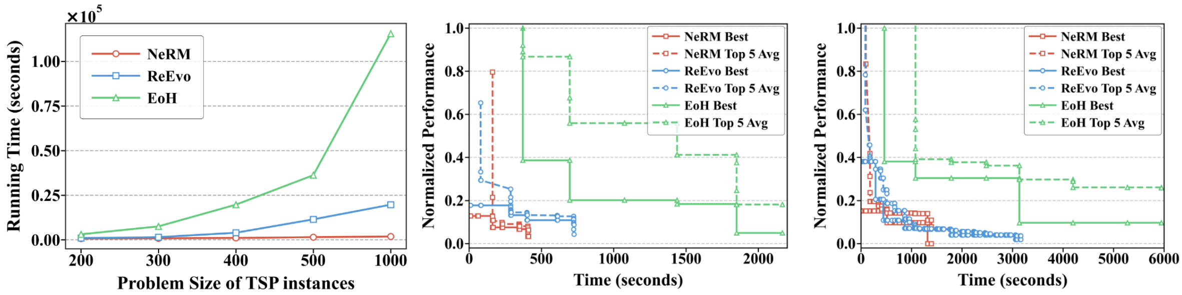
The results are reported in Table 5. We find that LLM-based methods outperform token-based predictors, demonstrating that leveraging contextual understanding and semantic reasoning enables more accurate and reliable performance evaluation of generated algorithms. Additionally, the predictor-assisted approach significantly outperforms token-based and LLM-based methods in performance prediction. We attribute this to its ability to leverage execution-based evaluation and iterative refinement, enabling more accurate, efficient, and scalable identification of high-quality solutions.

4.4 Ablation Study

Effect of Sub-module. We conduct the effect of each sub-module to examine the contribution of each component in NeRM: (1) *w/o predictor*: It removes the module of predictor-assisted code evaluation; (2) *w/o reflection*: It removes the reflection step in both MoP and MoA; (3) *w/o evolution*: It removes the revolution step in MoP and MoA; (4) *w/o MoP*: It removes the module of MoP and simply utilize MoA for code generation; (5) *w/o MoA*: It utilize MoP for code generation.

Table 6: The results of ablation studies on TSP200. **Bold** results indicate the best performance.

Method	Gap (%)	Time (s)
w/o predictor	$\mathbf{0.18}_{\pm 0.02}$	104
w/o reflection	$0.22_{\pm 0.01}$	63
w/o revolution	$0.24_{\pm 0.04}$	62
w/o MoP	$0.34_{\pm 0.16}$	80
w/o MoA	$0.23_{\pm 0.04}$	72
NeRM	$0.19_{\pm 0.04}$	83



(a) Time consumption on TSP200. (b) Performance comparison on MKP100. (c) Performance comparison on TSP200.

Figure 4: Performance comparison of different methods. (a) shows the time consumption for each method across different TSP sizes. (b) (c) illustrate the fitness trends of different methods on MKP100 and TSP200 over time.

Table 7: Performance comparison using different LLMs as the backbone of NeRM. **Bold** values indicate the best performance, while underlined values represent the second-best performance.

Method	TSP200	TSP300	TSP400	TSP500
GPT-3.5-Turbo	0.21	0.63	0.90	1.10
GPT-4-Turbo	**0.18**	0.40	0.55	<u>0.72</u>
Llama-3 70B	0.22	**0.29**	<u>0.54</u>	<u>0.72</u>
Gemini-1.5-Pro	0.22	<u>0.33</u>	0.57	0.77
DeepSeek-V3	0.20	0.42	**0.52**	**0.69**
GLM-4-Flash	<u>0.19</u>	0.35	0.57	0.75

Experimental results are shown in Table 6. From the table, we find that: (1) *w/o predictor* significantly increases computational costs without yielding an increase in solution quality, indicating that the predictor efficiently filters out low-quality heuristics before evaluation, reducing the need for exhaustive testing and accelerating the evolutionary process. (2) *w/o reflection* and *w/o revolution* lead to a noticeable performance decline. Reflection enables iterative refinement by leveraging historical outputs, while revolution, through mutation and crossover, ensures diverse solution exploration. Without these mechanisms, heuristic evolution relies more on random variations, hindering efficient convergence to optimal solutions. (3) *w/o MoP* and *w/o MoA* significantly degrade performance, but in different ways. Without MoP, the framework loses its ability to refine task specifications dynamically, which is crucial for guiding LLM-generated heuristics. This results in suboptimal code generation, as the model lacks effective guidance in exploring the search space. On the other hand, removing MoA prevents continuous heuristic refinement through structured code evolution, leading to stagnation in heuristic improvement after the initial generation.

Effect of Different LLMs. Table 7 presents the performance of different LLMs as the backbone of NeRM across TSP200-TSP500, highlighting variations in effectiveness across different problem sizes. From the table, we find that, GPT-4-Turbo delivers the best performance for TSP200 and remains highly competitive for larger instances, securing the second-best score for TSP500. DeepSeek-V3 excels in handling larger problems, achieving the best performance for TSP400 and TSP500, showcasing its strength in scaling to more complex problem sizes. Llama-3 70B demonstrates strong performance in TSP300 but shows a slight decline in larger instances, indicating that while it is effective for medium-sized problems, it may struggle with the scalability of larger datasets. Meanwhile, GLM-4 Flash maintains stable and consistent performance across all problem sizes, securing the second-best result in TSP200 and remaining competitive in TSP300 and TSP500, highlighting its robustness in various settings. This variation in performance across different LLMs underscores the importance of model selection in optimizing algorithm performance for specific problem sizes.

5 Conclusion

In this work, we introduced Nested-Refinement Metamorphosis (NeRM), a framework that iteratively refines prompts and evolves heuristic algorithms through reflection and optimization. Inspired by biological metamorphosis, NeRM integrates Metamorphosis on Prompts (MoP) to refine task descriptions and Metamorphosis on Algorithms (MoA) to enhance solutions. The nested refinement between MoP and MoA balances exploration and optimization, enabling adaptive algorithm design. To improve efficiency, we added a predictor-assisted evaluation mechanism, filtering out low-quality heuristics early and reducing computational overhead. Experiments on TSP, MKP, and CVRP show that NeRM outperforms state-of-the-art prompt-based and evolution-based methods in both quality and efficiency. In future work, we plan to expand NeRM to broader domains, enhance predictor models for better assessment, and improve scalability for large-scale algorithm design.

Limitation

While NeRM demonstrates strong performance in algorithm design, several limitations remain. Its effectiveness is influenced by the underlying LLM, leading to performance variability depending on model capabilities and computational costs. Although the predictor-assisted evaluation reduces computational overhead, iterative refinement of both prompts and code still poses scalability challenges for extremely large problem instances or real-time applications. Additionally, while NeRM effectively explores and optimizes heuristic solutions, its search space is constrained by predefined prompt structures and evolutionary operations, potentially limiting further diversity in exploration. The framework has been primarily evaluated on combinatorial optimization tasks, and its generalization to fundamentally different problem domains, such as continuous optimization or symbolic reasoning, remains an open question. Furthermore, the predictor approximates performance instead of executing all candidates, which, while improving efficiency, may lead to occasional misclassification of promising solutions. Addressing these limitations could further enhance NeRM's adaptability, scalability, and effectiveness in broader algorithm design applications.

Acknowledgments

This work is supported by National Key Research and Development Program of China (under Grant No.2023YFB2903904), CCF-Huawei Populus Grove Fund, CCF-Zhipu Large Model Innovation Fund (CCF-Zhipu202402), and the Research Grants Council of the Hong Kong Special Administrative Region (Grants 16202523, FB453 and HKU C7004-22G).

References

D. Applegate, R. Bixby, V. Chvátal, and W. Cook. 2006. Concorde tsp solver.

Florian Arnold and Kenneth Sörensen. 2019. Knowledge-guided local search for the vehicle routing problem. *Computers and Operations Research*.

Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. 2024. Hysynth: Context-free llm approximation for guiding program synthesis. *arXiv*.

Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Springer.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv*.

Michael Denkowski and Alon Lavie. 2014. Meteor universal: Language specific translation evaluation for any target language. In *Workshop on Statistical Machine Translation*.

M. Dorigo and G. Di Caro. 1999. Ant colony optimization: a new meta-heuristic. In *Congress on Evolutionary Computation*.

Manuel Laguna Fred Glover. 1997. *Tabu Search Principles*. Springer.

Yingbing Huang, Lily Jiaxin Wan, Hanchen Ye, Manvi Jha, Jinghua Wang, Yuhong Li, Xiaofan Zhang, and Deming Chen. 2024. New solutions on llm acceleration, optimization, and application. In *IEEE Design Automation Conference*.

Zaid Khan, Vijay Kumar BG, Samuel Schulter, Yun Fu, and Manmohan Chandraker. 2024. Self-training large language models for improved visual program synthesis with visual reinforcement. In *IEEE Conference on Computer Vision and Pattern Recognition*.

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

Gun Lee, Subin An, Sungyong Baik, and Soochahn Lee. 2024. CoAPT: Context attribute words for prompt tuning. *arXiv*.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv*.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Association for Computational Linguistics*.

Yixuan Li, Julian Parsert, and Elizabeth Polgreen. 2024. Guiding enumerative program synthesis with large language models. In *International Conference on Computer Aided Verification*.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*.

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. In *International Conference on Machine Learning*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*.

Qingyi Liu, Jinghui Qin, Wenxuan Ye, Hao Mou, Yuxuan He, and Keze Wang. 2024b. Adaptive prompt routing for arbitrary text style transfer with pretrained language models. *AAAI Conference on Artificial Intelligence*.

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

Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Lam Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2021. P-tuning v2: Prompt tuning can be comparable to finetuning universally across scales and tasks. *arXiv*.

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*.

Gary Marcus. 2020. The next decade in ai: four steps towards robust artificial intelligence. *arXiv*.

N. Mladenović and P. Hansen. 1997. Variable neighborhood search. *Computers and Operations Research*.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Association for Computational Linguistics*.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *IEEE Symposium on Security and Privacy*.

Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Workshop on Statistical Machine Translation*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis.

Bernardino Romera-Paredes, Mohammadamin Barekatain, et al. 2024. Mathematical discoveries from program search with large language models. *Nature*.

Timo Schick and Hinrich Schütze. 2020. Exploiting cloze questions for few shot text classification and natural language inference. *arXiv*.

Jialong Shi, Qingfu Zhang, and Edward Tsang. 2018. Eb-gls: an improved guided local search based on the big valley structure. *Memetic computing*.

Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating code generation with large language models. In *Empirical Methods in Natural Language Processing*.

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien N. Nguyen. 2019. Does bleu score work for code migration? In *International Conference on Program Comprehension*.

Christos Voudouris, Edward P.K. Tsang, and Abdullah Alsheddy. 2010. *Guided Local Search*.

Haoyu Wang, Tao Li, Zhiwei Deng, Dan Roth, and Yang Li. 2024a. Devil's advocate: Anticipatory reflection for llm agents. *arXiv*.

Yang Wang, Ya-Hui Jia, Wei-Neng Chen, and Yi Mei. 2024b. Distance-aware attention reshaping: Enhance generalization of neural solver for large-scale vehicle routing problems.

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

Haoran Ye, Jiarui Wang, Zhiguang Cao, et al. 2024. Reevo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*.

Fangyi Yu, Lee Quartey, and Frank Schilder. 2023. Exploring the effectiveness of prompt engineering for legal reasoning tasks. In *Association for Computational Linguistics*.

Terry Yue Zhuo. 2024. ICE-score: Instructing large language models to evaluate code. In *Association for Computational Linguistics*.

A Appendix

A.1 Dataset Construction for Code Predictor

We first construct a training set for predictor training. This dataset is generated by leveraging LLMs to produce heuristic code for various problems. Each generated heuristic is then executed to evaluate its performance (e.g., $(code, data, performance)$). For instance, in the Traveling Salesman Problem (TSP), we generate heuristic solutions for problem sizes of 50, 100, 150, and 200 nodes, with their optimal solution gaps computed using the Concorde (Applegate et al., 2006). Given that smaller-scale TSP instances can be solved efficiently and the predictor requires only a single training phase to generalize across different problem instances, the computational overhead remains negligible. Once the dataset of code-gap pairs is obtained, we randomly sample pairs of heuristic codes (e.g., $\mathcal{D}_{ij} = (\text{code}_i, data_i, \text{code}_j, data_j)$) and compare their performance gaps. If the first sample outperforms the second, it is assigned a label of 1; otherwise, it is labeled as 0:

$$\text{label}(\mathcal{D}_{ij}) = \begin{cases} 1 & \text{if } \text{gap}_i > \text{gap}_j, \\ 0 & \text{if } \text{gap}_i \leq \text{gap}_j. \end{cases}$$

A.2 Details of Experimental Settings

Traveling Salesman Problems (TSP). The heuristic evolution is conducted on a set of 10 TSP200 instances, where TSP200 means that each TSP problem instance contains 200 cities. For the testing phase, each problem size encompassed 64 problem instances, and the optimal paths were obtained using the Concorde (Applegate et al., 2006) to obtain the average gap.

Methods in comparison includes:

- Hand-crafted heuristics: Two commonly used iterative solution methods, Guided Local Search (GLS) (Voudouris et al., 2010) and Knowledge-Guided Local Search (KGLS) (Arnold and Sörensen, 2019), are selected here for comparison. An improved GLS method EBGLS (Shi et al., 2018) is selected as the comparison algorithm. Their core improvement is to leverage domain-specific knowledge to improve the efficiency and accuracy of solving optimal path finding tasks through advanced language model techniques. For each problem instance, during the training phase, the number of iterations is 1200 and the computation is limited to a maximum time of 20 seconds. During the testing phase, the number of iterations for TSP50, TSP100, TSP200, TSP500, TSP1000 is 500, 1800, 800, 800, 800, and 800 respectively.

- Prompt engineering methods: Recent prompt engineering approaches optimize designs to enhance LLMs' adaptability and performance. EEPE (Yu et al., 2023) focuses on tailored prompts for specific domains. CoAPT (Lee et al., 2024) uses context attribute words for accurate code generation. APR (Liu et al., 2024b) introduces adaptive prompt routing for diverse text style transfer in code generation.

- Evolution of heuristics approaches: Recently, these methods leverage LLMs to achieve significant advancements in program search and algorithm design. EoH (Liu et al., 2024a) uses different strategies to search for heuristic codes to achieve efficient automatic algorithm design; and ReEvo (Ye et al., 2024) employs LLMs as hyper-heuristics with reflective evolution to enhance the adaptability of heuristic codes.

Multiple Knapsack Problems (MKP). It refers to the optimization challenge of selecting items, each with a given weight and value, to maximize the total value in multiple knapsacks, each with a specified capacity constraint. The weights and values are both uniformly sampled from the interval [0,1], and the capacity for each instance is uniformly sampled from $(\max_j w_{ij}, \sum_j w_{ij})$ following the settings in ReEvo.

Methods in comparison includes:

- Hand-crafted heuristics: Three commonly utilized solution algorithms, Variable Neighborhood Search (VNS) (Mladenović and Hansen, 1997), Tabu Search (TS) (Fred Glover, 1997), and Ant

Colony Optimization (ACO) (Dorigo and Di Caro, 1999), are employed for comparison. These heuristics are designed to exploit problem-specific knowledge and patterns to efficiently explore the solution space, aiming to identify high-quality solutions that balance the constraints and objectives of the MKP.

- Prompt engineering methods and Evolution of heuristics approaches are the same as TSP.

Performance Predictor. The classification task of this module is to predict the performance of two given pieces of code to identify the quality of the code and whether it needs to be evaluated during actual execution. The datasets consists of around 5000 generated codes for each scenario during the execution of the dual-branch framework and their evaluated fitness values. The label is a simple Boolean value that represents whether the first code is better. Considering that the performance predictor needs to be trained, for each problem, we randomly sampled 6000 pairs of codes for training; Then, 1000 code pairs were randomly sampled from unused code to form a test set, and the prediction accuracy of all methods was validated on this test set.

Methods in comparison includes:

- Token-based methods: Traditional methods used for assessing machine translation or text generation have been adapted for evaluating code. BLEU (Papineni et al., 2002) measures modified n-gram precision and incorporates a penalty for brevity. ROUGE-L (Lin, 2004) evaluates sequence n-grams based on the longest common sub-sequence. METEOR (Denkowski and Lavie, 2014) focuses on the recall and precision of unigrams, taking into account the order of matched words. ChrF (Popović, 2015) calculates character-level n-gram precision and recall. Additionally, CodeBLEU (Ren et al., 2020) and RUBY (Tran et al., 2019) further extend these traditional token-based techniques specifically for code evaluation.

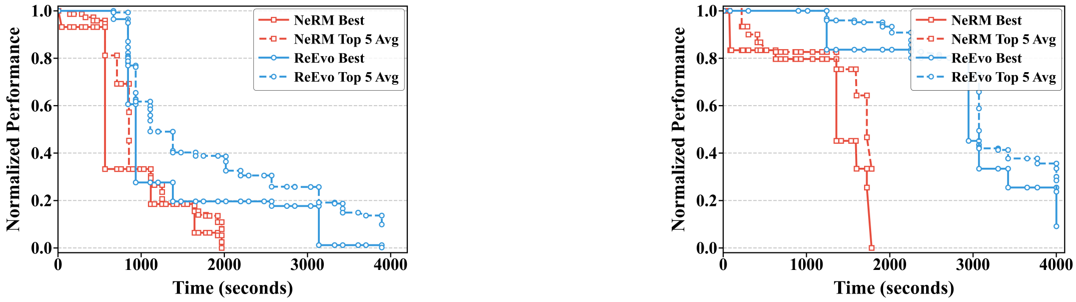
- LLM-based methods: ICE-Score (Zhuo, 2024) and CodeJudge (Tong and Zhang, 2024) are two baseline methods for evaluating code generation, with ICE-Score focusing on semantic matching using LLMs, and CodeJudge emphasizing syntactic and logical correctness through automated judging systems.

A.3 Additional Experiments

Table 8: Comparisons of different methods on CVRP metrics. **Bold** results indicate the best performance.

Method	n=200	n=500	n=1000
POMO	11.62	34.78	292.46
POMO+DAR	11.48	35.07	289.33
POMO+ReEvo	11.66$_{\pm 0.19}$	36.85$_{\pm 1.76}$	155.40$_{\pm 49.54}$
POMO+NeRM	**11.42**$_{\pm 0.36}$	**33.78**$_{\pm 0.59}$	**125.70**$_{\pm 48.56}$
LEHD	4.30	3.83	5.59
LEHD+DAR	5.34	5.30	7.16
LEHD+ReEvo	3.79$_{\pm 0.23}$	3.03$_{\pm 0.14}$	5.49$_{\pm 0.14}$
LEHD+NeRM	**3.07**$_{\pm 0.33}$	**2.98**$_{\pm 0.16}$	**5.25**$_{\pm 0.02}$

Capacitated Vehicle Routing Problem (CVRP). Table 8 presents the performance of POMO-based and LEHD-based algorithms on CVRP across different problem sizes. The results demonstrate that integrating ReEvo and NeRM significantly enhances heuristic effectiveness, enabling POMO (Kwon et al., 2020) and LEHD (Luo et al., 2023) to surpass the expert-designed attention-reshaping heuristic DAR (Wang et al., 2024b). This improvement underscores the advantage of incorporating both evolutionary and prompt-based methods in solving combinatorial optimization problems. Notably, POMO and LEHD combined with NeRM achieve the best results across all tested instances, highlighting the effectiveness of MoP in refining prompts and MoA in evolving heuristic code. The synergy between prompt refinement and structured code evolution not only enhances heuristic adaptability but also optimizes overall efficiency. These findings suggest that NeRM provides a more scalable and generalizable approach for solving CVRP, demonstrating superior performance over conventional heuristic design methods. More details of the evolution process can be found in Figures 5a and 5b. They show the evolution processes on CVRP 200 with “POMO+NeRM” and “LEHD+NeRM”, respectively. The horizontal axis represents the time required to reach the threshold, and the vertical axis represents the normalized performance between the best fitness value and the worst one. It can be seen that the proposed method converges faster.



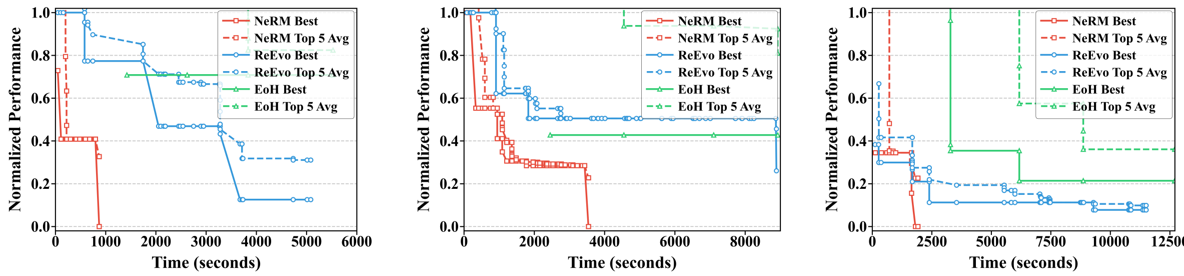
(a) Comparison of the POMO-based NeRM and ReEvo on the variation of fitness over time on the CVRP200.

(b) Comparison of the LEHD-based NeRM and ReEvo on the variation of fitness over time on the CVRP200.

Figure 5: Comparison of different problems of the variation of fitness over time on the CVRP200.

A.4 Additional Ablation Studies

Evolution Process on Different Problems. Figure 6a-6c shows the evolution processes on TSP 300, 400, 500, respectively.



(a) Comparison on the TSP300.

(b) Comparison on the TSP400.

(c) Comparison on the TSP500.

Figure 6: Comparison of different methods on the variation of fitness over time on problems of different sizes. The x-axis represents the running time, and the y-axis represents the normalized objective, which is obtained by normalizing the best and worst results through different methods.

In terms of efficiency of different heuristic evolution methods, we also visualize the running time required for different methods to achieve the threshold (i.e. fitness value) on TSP problems of different sizes. As shown in the Figure mentioned above, the horizontal axis represents different sizes of TSP problems, and the vertical axis represents the time required to reach the threshold. For the TSP problem of a specific problem size, ReEvo is initially run five times, with a maximum of 300 code generations per run. The fitness of the best code produced during these runs is taken as the threshold for that problem size. Subsequently, each method is rerun, and the runtime is recorded when the threshold fitness is achieved. To ensure fairness in the comparison, during the runtime, a serial process is used for code evaluation. Additionally, to mitigate the impact of network fluctuations, the time consumed by API invokes is excluded.

Different Training set sizes. Figure 7 illustrates the variation in prediction accuracy of the pre-trained code performance predictor as the number of training samples increases. For each training set size, we imposed a maximum of 300 epochs and incorporated an early stopping mechanism to prevent overfitting. The figure reveals that the prediction accuracy rises rapidly as the number of samples increases from 1000 to 3000. However, beyond 3000 samples, the marginal benefit of increasing the sample size gradually diminishes. Therefore, blindly expanding the training sample size to see as many sample pairs as possible to improve prediction efficiency is unnecessary.

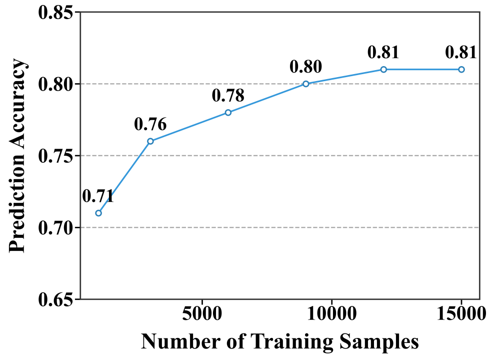


Figure 7: Accuracy of performance predictor varying with the number of training samples.

A.5 Generated Prompts and Heuristics

Best Prompt and Heuristic on Different Problems. This section presents the best prompts and corresponding heuristics evolved by NeRM on different problems, as shown in Figure 8- 11.

Example Evolution Process. Typical processes of the proposed flow, such init process, MoP, MoC and update process used in different problems, are shown in Table 9. EvoPrompt(**?**), LKH(**?**)

Table 9: Examples of the evolution Process on TSP, MKP, CVRP POMO, CVRP LEHD respectively.

Module	Metamorphosis on Prompts (MoP)	Metamorphosis on Algorithms (MoA)
Initial Population	Shown in Figure 12, 18, 24, 30	
Reflection	Shown in Figure 15, 21, 26, 32	Shown in Figure 13, 19, 25, 31
Evolution	Shown in Figure 16, 22, 28, 34	Shown in Figure 14, 20, 27, 33
Update	Shown in Figure 17, 23, 29, 35	

Best Prompt::
"""
Heuristics to estimate the goodness of including each edge in a solution, combining machine learning-driven tour initialization with population-based optimization, utilizing swarm intelligence and evolution strategies to diversify the solution pool, followed by adaptive large neighborhood search to focus on promising regions of the solution space, and ultimately refined through a recount heuristic, leveraging look-ahead mechanisms and machine learning-enhanced tour fragment recombination to iteratively converge towards an optimal solution that effectively balances distance minimization, route consolidation, and diversified exploration to robustly solve large-scale Traveling Salesman Problems.
"""

Best Algorithm:

```
def best_heuristic(distance_matrix: np.ndarray) -> np.ndarray:
    # Initialize the heuristics values with the distance values
    heuristics_values = np.copy(distance_matrix)
    # Simplify machine learning-driven tour initialization using centroid-based clustering
    centroid = np.mean(distance_matrix, axis=0)
    cluster_labels = np.argmin(distance_matrix, axis=0)
    # Update the heuristics values based on cluster assignment
    for i in range(distance_matrix.shape[0]):
        for j in range(distance_matrix.shape[1]):
            if i == cluster_labels[j]:
                heuristics_values[i, j] *= 0.9  # favor intra-cluster edges
    # Integrate city centrality measures using harmonic centrality
    city_centralities = 1 / np.sum(1 / (distance_matrix + 1), axis=0)
    # Update the heuristics values based on city centralities
    for i in range(distance_matrix.shape[0]):
        for j in range(distance_matrix.shape[1]):
            # favor edges between central cities and distant cities
            heuristics_values[i, j] *= (1 - city_centralities[i] * city_centralities[j])
    # Apply adaptive large neighborhood search using a simpler radius-based approach
    neighborhood_radius = int(distance_matrix.shape[0] * 0.2)
    for i in range(distance_matrix.shape[0]):
        neighborhood = np.argsort(distance_matrix[i])[:neighborhood_radius]
        for j in range(distance_matrix.shape[1]):
            if j in neighborhood:
                heuristics_values[i, j] *= 1.1  # favor edges in promising neighborhoods
    # Apply recount heuristic leveraging look-ahead mechanisms with a simplified score function
    for i in range(distance_matrix.shape[0]):
        for j in range(distance_matrix.shape[1]):
            look_ahead_distance = np.min(
                [distance_matrix[i, k] + distance_matrix[k, j] for k in range(distance_matrix.shape[0]) if
                 k != i and k != j])
            heuristics_values[i, j] /= (1 + look_ahead_distance / distance_matrix.shape[0])
    # Apply machine learning-enhanced tour fragment recombination with a simpler greedy strategy
    fragment_size = int(distance_matrix.shape[0] * 0.1)
    for _ in range(10):
        fragment_start = np.random.randint(0, distance_matrix.shape[0] - fragment_size)
        fragment = np.arange(fragment_start, fragment_start + fragment_size)
        fragment_distances = distance_matrix[fragment[:, None], fragment[None, :]]
        recombined_fragment = np.argmin(fragment_distances, axis=1)
        for i in range(fragment_size):
            heuristics_values[fragment[i], fragment[recombined_fragment[i]]] *= 1.1
    # Normalize the heuristics values
    heuristics_values = np.where(heuristics_values > 0, heuristics_values, 1e-6)  # avoid division by zero
    heuristics_values /= np.sum(heuristics_values)
    return heuristics_values
```

Figure 8: The best NeRM-generated prompt and corresponding heuristic for TSP_GLS.

Best Prompt::
"""
Utilizing probabilistic solution sampling techniques, this enhanced version of the Multiple Knapsack Problem (MKP) tackles the challenge of selecting item subsets for optimal prize maximization while adhering to complex, multi-dimensional weight limitations. By integrating advanced heuristic algorithms, we optimize the selection process through a strategic combination of metaheuristics and evolutionary computation, aiming to surpass conventional deterministic approaches and achieve superior solution quality and efficiency.
"""

Best Algorithm:
```
def best_heuristic(prize: np.ndarray, weight: np.ndarray) -> np.ndarray:
    # Enhanced heuristic function with adaptive randomness, balanced mutation, and dynamic thresholding for MKP
    # 1. Value-to-weight ratio
    # 2. Weight sparsity
    # 3. Proximity to average weight
    # 4. Adaptive randomness for diversity
    # 5. Normalization and thresholding for sparsity
    # 6. Mutation for dynamic thresholding
    # Calculate the value-to-weight ratio for each item
    value_to_weight_ratio = prize / weight.sum(axis=1)
    # Calculate the sparsity of each item based on its weight dimensions
    sparsity = 1 / (1 + np.sum(weight, axis=1))
    # Calculate the proximity to the average weight for each item
    average_weight = weight.mean(axis=1)
    proximity_to_average = np.abs(weight - average_weight[:, np.newaxis]).sum(axis=1)
    # Combine the factors into a single heuristic score
    combined_heuristic = value_to_weight_ratio * sparsity * (1 / proximity_to_average)
    # Add a random component to introduce diversity
    random_component = np.random.rand(prize.shape[0])
    random_component = random_component / np.sum(random_component)  # Normalize the random component
    # Adjust the combined heuristic to form a probability distribution
    combined_heuristic = (combined_heuristic + random_component) / np.sum(combined_heuristic + random_component)
    # Normalize the combined heuristic scores
    normalized_combined_heuristic = combined_heuristic / np.sum(combined_heuristic)
    # Mutation: dynamically adjust the threshold for sparsifying the heuristics
    initial_threshold = np.percentile(normalized_combined_heuristic, 25)  # Initial threshold
    mutation_rate = 0.05  # Mutation rate for threshold
    mutation = np.random.rand(prize.shape[0]) < mutation_rate
    threshold = np.where(mutation, np.percentile(normalized_combined_heuristic[mutation], 50), initial_threshold)
    # Sparsify the heuristics by setting elements below the dynamic threshold to zero
    heuristics = np.where(normalized_combined_heuristic >= threshold, normalized_combined_heuristic, 0)
    return heuristics
```

Figure 9: The best NeRM-generated prompt and corresponding heuristic for MKP_ACO.

Best Prompt::
""""
Develop a sophisticated solution algorithm for the Capacitated Vehicle Routing Problem (CVRP), targeting a minimization of route distances while ensuring vehicle capacities are adhered to strictly. Focus on the implementation of exact or heuristic methods that can effectively integrate constraint programming, dynamic window approaches, or multi-objective evolutionary algorithms to tackle complex routing challenges efficiently. Incorporate techniques like node partitioning, demand relaxation, and path decomposition to optimize route selection, with an emphasis on real-time adaptation and robust performance for dynamic problem instances.
""""

Best Algorithm:
```
def best_heuristic(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    n = distance_matrix.shape[0]
    # Normalize distance matrix
    max_distance = distance_matrix.max().item()
    normalized_distances = distance_matrix / max_distance
    # Normalize demands
    max_demand = demands.max().item()
    normalized_demands = demands / max_demand
    # Calculate the potential value for each edge using a balance heuristic
    # The heuristic now includes a dynamic weight based on the distance and demand
    # The weight increases with distance but is dampened by high demands
    distance_weight = 0.5
    demand_weight = 2.0
    penalty_threshold = 0.8  # Nodes with demand greater than 0.8 times the average demand are penalized
    average_demand = normalized_demands.mean()
    # Create a penalty term for high demand nodes
    penalty_term = torch.where(normalized_demands > penalty_threshold,
                               normalized_demands - average_demand,
                               torch.zeros_like(normalized_demands))
    # Introduce a node partitioning approach to separate nodes based on their demand
    # Higher demand nodes are more likely to be penalized or have lower potential
    partitioning_factor = 0.5
    partitioning = torch.where(normalized_demands > average_demand,
                               normalized_demands,
                               torch.zeros_like(normalized_demands))
    # Adjust potential values based on partitioning
    potential_values = (normalized_distances * (1 - partitioning) +
                        distance_weight * normalized_distances +
                        demand_weight * normalized_demands -
                        penalty_term * partitioning).abs()
    # Set the depot to 0 potential value
    potential_values[0, :] = 0
    potential_values[:, 0] = 0
    # Introduce a dynamic window approach to adapt weights based on current vehicle capacity
    # Assume vehicle_capacity is passed as an argument
    vehicle_capacity = demands.sum().item() / n
    dynamic_weight = 1.0 / (1.0 + vehicle_capacity)
    potential_values *= dynamic_weight
    return potential_values
```

Figure 10: The best NeRM-generated prompt and corresponding heuristic for CVRP_POMO.

```
Best Prompt::
"""
Revitalize eCVRP resolution through the synthesis of multi-criteria algorithms that optimize delivery time, route length, and load capacities simultaneously. Leverage mixed heuristic methodologies, evolutionary optimization techniques, advanced dynamic programming approaches, and strategic tabu search for adaptive, computationally efficient routing. Target an algorithmic ensemble that maximizes the synergy between service punctuality and economic efficiency in the dynamic distribution landscape.
"""
Best Algorithm:
def best_heuristic(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    total_capacity = demands.sum()
    normalized_demands = demands / total_capacity
    # Adjusting the heuristic function to balance variance by using a cap on the demand variance.
    demand_variance = (demands - normalized_demands.mean()).pow(2).mean()
    demand_variance_cap = torch.clamp(demand_variance, min=0, max=1)
    # Scaling the penalty based on the demand variance, but avoiding a penalty of 0 by capping
    # at a small positive value.
    penalty_factor = demand_variance_cap + 0.1
    # Introduce a scaling factor that adjusts the weight of distance to maintain scale invariance.
    scaling_factor = torch.sqrt(total_capacity)
    # Adjust the Z-score to account for the normalization of demands.
    z_scores = (demands - normalized_demands.mean()) / normalized_demands.std()
    outlier_penalty = torch.where(z_scores.abs() > 2, torch.tensor(10.0, dtype=distance_matrix.dtype),
                                  torch.tensor(1.0, dtype=distance_matrix.dtype))
    # Calculate heuristics by considering both demand and distance, and adjust by penalties and scaling factors.
    heuristic_matrix = -torch.mul(
        torch.mul(normalized_demands, distance_matrix),
        penalty_factor * outlier_penalty
    )
    heuristic_matrix = heuristic_matrix * scaling_factor
    # Apply a threshold to ensure heuristics are within a specific range, which helps in maintaining stability.
    threshold = torch.tensor(0.1, dtype=distance_matrix.dtype)
    heuristic_matrix = torch.clamp(heuristic_matrix, min=-threshold, max=threshold)
    return heuristic_matrix
```

Figure 11: The best NeRM-generated prompt and corresponding heuristic for CVRP_LEHD.

```
Init Prompt:
"""
Solving Traveling Salesman Problem (TSP) via guided local search. TSP requires finding the shortest path that visits all given nodes and returns to the starting node.
"""
Init Algorithm:
def seed_func(distance_matrix: np.ndarray) -> np.ndarray:
    return distance_matrix
```

Figure 12: Initialization of NeRM on TSP_GLS.

```
MoA Reflection:
"""---------------------------------------- Request ---------------------------------------------------------
Use reference algorithms, explain the reasons why the better algorithm is good.
Better Algorithm:
def heuristics(distance_matrix: np.ndarray) -> np.ndarray:
   # Initialize an array to store the prior indicators, filled with zeros
   prior_indicators = np.zeros_like(distance_matrix)
   # Calculate the average distance for normalization
   avg_distance = np.mean(distance_matrix)
   # Calculate the prior indicators for each edge
   # We assume that edges with shorter distances are less likely to be bad
   prior_indicators = 1 - (distance_matrix / (2 * avg_distance))
   # Set the diagonal elements to zero, as they represent the distance from a node to itself
   np.fill_diagonal(prior_indicators, 0)
   return prior_indicators

Worse Algorithm:
def heuristic(distance_matrix: np.ndarray) -> np.ndarray:
   return distance_matrix
"""
---------------------------------------- Response ---------------------------------------------------------
Algorithm Reflection:
Algorithm Reflection: Use averages instead of max values.
```

Figure 13: MoA reflection of NeRM on TSP_GLS.

```
MoA Revolution:
"""---------------------------------------- Request ---------------------------------------------------------
Use reference algorithm, assisting by reflect of previous round, generate a new algorithm.
Prompt: Solving Traveling Salesman Problem (TSP) via guided local search. TSP requires
finding the shortest path that visits all given nodes and returns to the starting node.
Algorithm:
def heuristics(distance_matrix: np.ndarray) -> np.ndarray:
   # Initialize an array to store the prior indicators, filled with zeros
   prior_indicators = np.zeros_like(distance_matrix)
   # Calculate the average distance for normalization
   avg_distance = np.mean(distance_matrix)
   # Calculate the prior indicators for each edge
   # We assume that edges with shorter distances are less likely to be bad
   prior_indicators = 1 - (distance_matrix / (2 * avg_distance))
   # Set the diagonal elements to zero, as they represent the distance from a node to itself
   np.fill_diagonal(prior_indicators, 0)
   return prior_indicators

Algorithm Reflection:
Use averages instead of max values.
Use max distance and introduce non-determinism.
Use averages instead of max values.
"""
---------------------------------------- Response ---------------------------------------------------------
Algorithm Revolution:
def evolved_algorithm(distance_matrix: np.ndarray) -> np.ndarray:
   # Initialize the heuristic matrix with the distance matrix
   heuristic_matrix = distance_matrix.copy()
   # Calculate the average distance of each edge
   avg_distance = np.mean(distance_matrix)
   # For each edge, calculate a penalty factor based on its distance
   # Edges with higher distances are penalized more
   for i in range(distance_matrix.shape[0]):
      for j in range(distance_matrix.shape[1]):
         # Calculate the penalty factor for the edge
         penalty_factor = (distance_matrix[i][j] - avg_distance) / avg_distance if avg_distance
!= 0 else 0
         # Update the heuristic value of the edge
         heuristic_matrix[i][j] = distance_matrix[i][j] + penalty_factor * distance_matrix[i][j]
   return heuristic_matrix
```

Figure 14: MoA revolution of NeRM on TSP_GLS.

MoP Reflection:
""""-------------------------------------- Request --
Use reference prompts, explain the reasons why the better prompt is good.

Reference Prompt:
Better Prompt: Optimizing the Traveling Salesman Problem (TSP) using metaheuristic search techniques, specifically guided local search with iterative improvement, to efficiently explore the solution space and identify the minimum-distance Hamiltonian cycle that visits each node exactly once and returns to the origin.

Worse Prompt: Solving Traveling Salesman Problem (TSP) via guided local search. TSP requires finding the shortest path that visits all given nodes and returns to the starting node.
""""
-------------------------------------- Response --
Prompt Reflection:
Specify optimization strategies and techniques explicitly.

Figure 15: MoP reflection of NeRM on TSP_GLS.

MoP Revolution:
""""-------------------------------------- Request --
Use reference prompt and code, assisting by reflect of previous round, generate a new prompt.
Prompt: Solving Traveling Salesman Problem (TSP) via guided local search. TSP requires finding the shortest path thatvisits all given nodes and returns to the starting node.

Best code:
```
def init_heuristic(distance_matrix: np.ndarray) -> np.ndarray:
   # Initialize the heuristic matrix with the distance matrix
   heuristic_matrix = distance_matrix.copy()
   # Calculate the average distance of each edge
   avg_distance = np.mean(distance_matrix)
   # For each edge, calculate a penalty factor based on its distance
   # Edges with higher distances are penalized more
   for i in range(distance_matrix.shape[0]):
      for j in range(distance_matrix.shape[1]):
         # Calculate the penalty factor for the edge
         penalty_factor = (distance_matrix[i][j] - avg_distance) / avg_distance if avg_distance != 0
else 0
         # Update the heuristic value of the edge
         heuristic_matrix[i][j] = distance_matrix[i][j] + penalty_factor * distance_matrix[i][j]
   return heuristic_matrix
```

Prompt Reflection:
Specify optimization strategies and techniques explicitly.
Specify key techniques and optimization strategies clearly.
Explicitness, conciseness, and clarity improve descriptions.
""""
-------------------------------------- Response --
Prompt Revolution:
Emphasizing adaptive search trajectories and perturbation strategies, the proposed method leverages a hybrid metaheuristic framework integrating guided local search with iterative improvement, incorporating dynamic parameter tuning and stochastic perturbations to exhaustively explore the TSP solution space and effectively converge on a minimum-distance Hamiltonian cycle.

Figure 16: MoP revolution of NeRM on TSP_GLS.

Evolution Update:

Prompt:
Emphasizing adaptive search trajectories and perturbation strategies, the proposed method leverages a hybrid metaheuristic framework integrating guided local search with iterative improvement, incorporating dynamic parameter tuning and stochastic perturbations to exhaustively explore the TSP solution space and effectively converge on a minimum-distance Hamiltonian cycle.

Seed Algorithm:
```
def new_seed_func(distance_matrix: np.ndarray) -> np.ndarray:
    # Initialize the heuristic matrix with the distance matrix
    heuristic_matrix = distance_matrix.copy()
    # Calculate the average distance of each edge
    avg_distance = np.mean(distance_matrix)
    # For each edge, calculate a penalty factor based on its distance
    # Edges with higher distances are penalized more
    for i in range(distance_matrix.shape[0]):
        for j in range(distance_matrix.shape[1]):
            # Calculate the penalty factor for the edge
            penalty_factor = (distance_matrix[i][j] - avg_distance) / avg_distance if avg_distance != 0
else 0
            # Update the heuristic value of the edge
            heuristic_matrix[i][j] = distance_matrix[i][j] + penalty_factor * distance_matrix[i][j]
    return heuristic_matrix
```

Figure 17: Evolution udpate of NeRM on TSP_GLS.

Init Prompt:
"""
Solving Multiple Knapsack Problems (MKP) through stochastic solution sampling based on "heuristics". MKP involves selecting a subset of items to maximize the total prize collected, subject to multi-dimensional maximum weight constraints.
"""

Init Algorithm:
```
def seed_func(prize: np.ndarray, weight: np.ndarray) -> np.ndarray:
    return prize / np.sum(weight, axis=1)
```

Figure 18: Initialization of NeRM on MKP_ACO.

MoA Reflection:
```
"""------------------------------------------ Request ---------------------------------------------------------
Use reference algorithms, explain the reasons why the better algorithm is good.
def heuristics(prize: np.ndarray, weight: np.ndarray) -> np.ndarray:
    # Calculate the prize-to-weight ratio for each item
    prize_to_weight_ratio = prize / weight.sum(axis=1)
    # Normalize the prize-to-weight ratio to sum to 1
    normalized_prize_to_weight_ratio = prize_to_weight_ratio / prize_to_weight_ratio.sum()
    # Use a probabilistic sampling mechanism to select items
    heuristics = np.random.choice(normalized_prize_to_weight_ratio,
size=normalized_prize_to_weight_ratio.shape[0],
                                                        p=normalized_prize_to_weight_ratio)
    return heuristics

Worse Algorithm:
def heuristics(prize: np.ndarray, weight: np.ndarray) -> np.ndarray:
    return prize / np.sum(weight, axis=1)
"""
------------------------------------------ Response ---------------------------------------------------------
```
Algorithm Reflection:
Algorithm Reflection: Use of sorting, explicit optimization goal.

Figure 19: MoA reflection of NeRM on MKP_ACO.

MoA Revolution:

```
"""-------------------------------------- Request -----------------------------------------------------------
Use reference algorithm, assisting by reflect of previous round, generate a new algorithm.
Solving Multiple Knapsack Problems (MKP) through stochastic solution sampling based on
"heuristics". MKP involves selecting a subset of items to maximize the total prize collected,
subject to multi-dimensional maximum weight constraints.
Algorithm:
def heuristics(prize: np.ndarray, weight: np.ndarray) -> np.ndarray:
   # Calculate the prize-to-weight ratio for each item
   prize_to_weight_ratio = prize / weight.sum(axis=1)
   # Normalize the prize-to-weight ratio to sum to 1
   normalized_prize_to_weight_ratio = prize_to_weight_ratio / prize_to_weight_ratio.sum()
   # Use a probabilistic sampling mechanism to select items
   heuristics = np.random.choice(normalized_prize_to_weight_ratio,
        size=normalized_prize_to_weight_ratio.shape[0],
        p=normalized_prize_to_weight_ratio)
   return heuristics

Algorithm Reflection:
Use averages instead of max values.
Prioritization of higher value items, simplicity in heuristic implementation.
simple heuristic, sort by prize, index-based allocation.
"""
-------------------------------------- Response -----------------------------------------------------------
```

Algorithm Revolution:

```
def evolved_algorithm(prize: np.ndarray, weight: np.ndarray) -> np.ndarray:
   # Calculate the prize-to-weight ratio for each item
   prize_to_weight_ratio = prize / weight.sum(axis=1)
   # Normalize the prize-to-weight ratio to sum to 1
   normalized_ratio = prize_to_weight_ratio / prize_to_weight_ratio.sum()
   # Incorporate adaptive learning mechanism: adjust the heuristic based on past
performance
   # This is a simplified example of an adaptive learning mechanism, which could be more
complex
   # in a real-world scenario.
   # Assume we have a learning rate and an exploration factor
   learning_rate = 0.1
   exploration_factor = 0.2
   # Initialize the heuristic array with normalized ratio
   heuristics = np.zeros_like(prize_to_weight_ratio)
   # Iterate over each item to calculate its heuristic value
   for i in range(len(prize)):
      # Calculate a heuristic value that combines exploration and exploitation
      heuristics[i] = (1 - exploration_factor) * normalized_ratio[i] + exploration_factor *
np.random.rand()
   # Apply dimensionality reduction to simplify the problem space
   # This could be a PCA or any other suitable dimensionality reduction technique
   # For simplicity, we'll just take the top-k items based on prize-to-weight ratio
   k = int(0.1 * len(prize))  # Example of selecting top 10% of items
   top_k_indices = np.argsort(prize_to_weight_ratio)[-k:]
   heuristics[top_k_indices] *= 2  # Increase the heuristic of top-k items
   return heuristics
```

Figure 20: MoA revolution of NeRM on MKP_ACO.

MoP Reflection:
""""------------------------------------- Request ---
Use reference prompts, explain the reasons why the better prompt is good.

Reference Prompt:
Better Prompt: Enhancing MKP resolution via an advanced heuristic-based probabilistic sampling mechanism. This approach strategically selects items based on a dynamic prize-to-weight ratio ranking, integrating a novel indexing system for efficient subset sampling. The algorithm sharpens its search by prioritizing high-value items, adeptly balancing the trade-offs between exploration and exploitation. It incorporates adaptive learning algorithms and employs sophisticated dimensionality reduction techniques to refine load distribution. This ensures a targeted optimization process that respects weight and capacity constraints, yielding a more precise and effective solution strategy.

Worse Prompt: Solving Multiple Knapsack Problems (MKP) through stochastic solution sampling based on "heuristics". MKP involves selecting a subset of items to maximize the total prize collected, subject to multi-dimensional maximum weight constraints.
""""
------------------------------------- Response ---
Prompt Reflection:
Give detailed, specific, and technical specification.

Figure 21: MoP reflection of NeRM on MKP_ACO.

MoP Revolution:
""""------------------------------------- Request ---
Use reference prompt and code, assisting by reflect of previous round, generate a new prompt.
Prompt: Solving Traveling Salesman Problem (TSP) via guided local search. TSP requires finding the shortest path that
visits all given nodes and returns to the starting node.

Best code:
```
def heuristic(distance_matrix: np.ndarray) -> np.ndarray:
   # Initialize the heuristic matrix with the distance matrix
   heuristic_matrix = distance_matrix.copy()
   # Calculate the average distance of each edge
   avg_distance = np.mean(distance_matrix)
   # For each edge, calculate a penalty factor based on its distance
   # Edges with higher distances are penalized more
   for i in range(distance_matrix.shape[0]):
      for j in range(distance_matrix.shape[1]):
         # Calculate the penalty factor for the edge
         penalty_factor = (distance_matrix[i][j] - avg_distance) / avg_distance if avg_distance != 0
else 0
         # Update the heuristic value of the edge
         heuristic_matrix[i][j] = distance_matrix[i][j] + penalty_factor * distance_matrix[i][j]
   return heuristic_matrix
```

Prompt Reflection:
Misleading Aspects: Avoid vague terms like "novel," "robust," or "strategic" without detail.
Algorithmic Insights: Clearly define the heuristic's purpose and its impact on the solution process.
""""
------------------------------------- Response ---
Prompt Revolution:
Solving the Multiple Knapsack Problem (MKP) using a stochastic solution sampling heuristic that leverages the prize-to-weight ratio for item selection. This heuristic employs a modified greedy algorithm, which sorts items by their prize-to-weight ratio in descending order. The algorithm iteratively selects items that maximize the total prize while respecting the multi-dimensional weight constraints of each knapsack. It also incorporates a balancing mechanism that dynamically adjusts the selection criteria based on the remaining capacity of each knapsack, ensuring optimal load distribution without exceeding weight limitations.

Figure 22: MoP revolution of NeRM on MKP_ACO.

Evolution Update:

Prompt:
Solving the Multiple Knapsack Problem (MKP) using a stochastic solution sampling heuristic that leverages the prize-to-weight ratio for item selection. This heuristic employs a modified greedy algorithm, which sorts items by their prize-to-weight ratio in descending order. The algorithm iteratively selects items that maximize the total prize while respecting the multi-dimensional weight constraints of each knapsack. It also incorporates a balancing mechanism that dynamically adjusts the selection criteria based on the remaining capacity of each knapsack, ensuring optimal load distribution without exceeding weight limitations.

Seed Algorithm:
```
def new_seed_func(prize: np.ndarray, weight: np.ndarray) -> np.ndarray:
    # Calculate the prize-to-weight ratio for each item
    prize_to_weight_ratio = prize / weight.sum(axis=1)
    # Sort items by their prize-to-weight ratio in descending order
    sorted_indices = np.argsort(prize_to_weight_ratio)[::-1]
    # Initialize an array to store the heuristics value for each item
    heuristics = np.zeros_like(prize)
    # Iterate over the sorted indices and update the heuristics array
    for i in sorted_indices:
        # Calculate the cumulative weight for the current item
        cumulative_weight = weight[i].sum()
        # Update the heuristics value for the current item
        heuristics[i] = cumulative_weight / prize[i]
        # Break if the cumulative weight exceeds the maximum allowed weight
        if cumulative_weight >= 1:
            break
    return heuristics
```

Figure 23: Evolution udpate of NeRM on MKP_ACO.

Init Prompt:
""""""

Init Population:
Assisting in solving Capacitated Vehicle Routing Problem (CVRP) with some prior heuristics. CVRP requires finding the shortest path that visits all given nodes and returns to the starting node. Each node has a demand and each vehicle has a capacity. The total demand of the nodes visited by a vehicle cannot exceed the vehicle capacity. When the total demand exceeds the vehicle capacity, the vehicle must return to the starting node.
""""""

Init Algorithm:
```
def seed_func(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    return torch.zeros_like(distance_matrix)func(distance_matrix: np.ndarray) -> np.ndarray:
    return distance_matrix
```

Figure 24: Initialization of NeRM on CVRP_POMO.

```
MoA Reflection:
"""-------------------------------------- Request --------------------------------------------------------
Use reference algorithms, explain the reasons why the better algorithm is good.
def heuristics(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    n = distance_matrix.shape[0]
    total_capacity = demands.sum()
    normalized_demands = demands / total_capacity
    # Initialize the heuristics matrix with zeros
    heuristics = torch.zeros_like(distance_matrix)
    # Calculate the heuristic value for each edge
    for i in range(n):
        for j in range(n):
            if i != j:
                # Demand-driven edge assessment
                demand_difference = normalized_demands[i] - normalized_demands[j]
                # Capacity-based heuristic adjustment
                heuristics[i, j] = -abs(demand_difference)
    return heuristics

Worse Algorithm:
def heuristics(prize: np.ndarray, weight: np.ndarray) -> np.ndarray:
    return prize / np.sum(weight, axis=1)
"""
-------------------------------------- Response --------------------------------------------------------
Algorithm Reflection:
1. Normalize appropriately.
2. Combine demand and distance.
3. Weight factors for balance.
4. Adjust for depot prominence.
```

Figure 25: MoA reflection of NeRM on CVRP_POMO.

```
MoP Reflection:
"""-------------------------------------- Request --------------------------------------------------------
Use reference prompts, explain the reasons why the better prompt is good.

Reference Prompt:
Better Prompt: Optimizing the Capacitated Vehicle Routing Problem (CVRP) by integrating
advanced heuristic algorithms. This approach involves identifying efficient paths that
encompass all nodes, ensuring each vehicle's load does not surpass its capacity constraints.
The emphasis is on the strategic allocation of vehicle capacities, dynamic route adjustments
based on node demands, and the integration of metaheuristics for improved solution quality.
Techniques such as Tabu Search, Genetic Algorithms, and Ant Colony Optimization are
employed to enhance the solution's robustness and effectiveness, ultimately yielding optimized
routing strategies for complex logistics scenarios.

Worse Prompt: Assisting in solving Capacitated Vehicle Routing Problem (CVRP) with some
prior heuristics. CVRP requiresfinding the shortest path that visits all given nodes and returns
to the starting node. Each node has a demand and each vehicle has a capacity. The total
demand of the nodes visited by a vehicle cannot exceed the vehicle capacity. When the total
demand exceeds the vehicle capacity, the vehicle must return to the starting node.
"""
-------------------------------------- Response --------------------------------------------------------
Prompt Reflection:
Specify heuristic computation details, focus on demand and capacity interaction, clarify
optimization objectives.
```

Figure 26: MoP reflection of NeRM on CVRP_POMO.

MoA Revolution:

```
"""-------------------------------------- Request ---------------------------------------------------------
Use reference algorithm, assisting by reflect of previous round, generate a new algorithm.
Assisting in solving Capacitated Vehicle Routing Problem (CVRP) with some prior heuristics.
CVRP requires finding the shortest path that visits all given nodes and returns to the starting
node. Each node has a demand and each vehicle has a capacity. The total demand of the
nodes visited by a vehicle cannot exceed the vehicle capacity. When the total
demand exceeds the vehicle capacity, the vehicle must return to the starting node.
Algorithm:
def heuristics(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    n = distance_matrix.shape[0]
    total_capacity = demands.sum()
    normalized_demands = demands / total_capacity
    # Initialize the heuristics matrix with zeros
    heuristics = torch.zeros_like(distance_matrix)
    # Calculate the heuristic value for each edge
    for i in range(n):
        for j in range(n):
            if i != j:
                # Demand-driven edge assessment
                demand_difference = normalized_demands[i] - normalized_demands[j]
                # Capacity-based heuristic adjustment
                heuristics[i, j] = -abs(demand_difference)
    return heuristics

Algorithm Reflection:
1. Normalize appropriately.
2. Combine demand and distance.
3. Weight factors for balance.
4. Adjust for depot prominence.
"""
-------------------------------------- Response ---------------------------------------------------------
```

Algorithm Revolution:

```
def evolved_algorithm(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    n = distance_matrix.shape[0]
    # Initialize the heuristic matrix with zeros
    heuristic_matrix = torch.zeros_like(distance_matrix)
    # Calculate the cumulative demand from the depot to each customer
    cumulative_demand = torch.cumsum(demands, dim=0)
    # Calculate the heuristics based on the distance matrix and cumulative demand
    for i in range(n):
        for j in range(n):
            if i != j:
                # Calculate the difference in cumulative demand from the depot to j
                # and from i to j
                demand_diff = cumulative_demand[j] - cumulative_demand[i]
                # Use the distance as a weight
                heuristic_value = -distance_matrix[i, j] + demand_diff
            else:
                # No cost to go back to the depot
                heuristic_value = 0
            # Update the heuristic matrix
            heuristic_matrix[i, j] = heuristic_value
    return heuristic_matrix
```

Figure 27: MoA revolution of NeRM on CVRP_POMO.

MoP Revolution:
"""-- Request ---
Use reference prompt and code, assisting by reflect of previous round, generate a new prompt.
Prompt: Assisting in solving Capacitated Vehicle Routing Problem (CVRP) with some prior heuristics. CVRP requires finding the shortest path that visits all given nodes and returns to the starting node. Each node has a demand and each vehicle has a capacity. The total demand of the nodes visited by a vehicle cannot exceed the vehicle capacity. When the total demand exceeds the vehicle capacity, the vehicle must return to the starting node.

Best code:
```
def heuristics(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    n = distance_matrix.shape[0]
    # Initialize the heuristic matrix with zeros
    heuristic_matrix = torch.zeros_like(distance_matrix)
    # Calculate the cumulative demand from the depot to each customer
    cumulative_demand = torch.cumsum(demands, dim=0)
    # Calculate the heuristics based on the distance matrix and cumulative demand
    for i in range(n):
        for j in range(n):
            if i != j:
                # Calculate the difference in cumulative demand from the depot to j
                # and from i to j
                demand_diff = cumulative_demand[j] - cumulative_demand[i]
                # Use the distance as a weight
                heuristic_value = -distance_matrix[i, j] + demand_diff
            else:
                # No cost to go back to the depot
                heuristic_value = 0
            # Update the heuristic matrix
            heuristic_matrix[i, j] = heuristic_value
    return heuristic_matrix
```

Prompt Reflection:
Specify heuristic computation details, focus on demand and capacity interaction, clarify optimization objectives.
"""
-- Response ---
Prompt Revolution:
Develop a demand-driven heuristic for solving the Capacitated Vehicle Routing Problem (CVRP) that incorporates a clear allocation strategy for vehicle capacities, taking into account the specific demands of each node. The heuristic must ensure that vehicle loads remain within predefined capacity constraints while optimizing overall efficiency. It should dynamically adjust vehicle capacities and routes in response to varying node demands. This algorithm should use Tabu Search and Genetic Algorithms to manage the vehicle routing effectively, prioritizing high-demand nodes and evaluating potential routes based on maximum demand and route efficiency. The goal is to create a scalable and robust routing strategy that adapts to fluctuating demands and total capacities, ensuring an optimized solution that balances demand-driven assessments with capacity constraints.

Figure 28: MoP revolution of NeRM on CVRP_POMO.

Evolution Update:

Prompt:
Develop a demand-driven heuristic for solving the Capacitated Vehicle Routing Problem (CVRP) that incorporates a clear allocation strategy for vehicle capacities, taking into account the specific demands of each node. The heuristic must ensure that vehicle loads remain within predefined capacity constraints while optimizing overall efficiency. It should dynamically adjust vehicle capacities and routes in response to varying node demands. This algorithm should use Tabu Search and Genetic Algorithms to manage the vehicle routing effectively, prioritizing high-demand nodes and evaluating potential routes based on maximum demand and route efficiency. The goal is to create a scalable and robust routing strategy that adapts to fluctuating demands and total capacities, ensuring an optimized solution that balances demand-driven assessments with capacity constraints.

Seed Algorithm:
```
def new_seed_func(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    n = distance_matrix.shape[0]
    # Initialize the heuristic matrix with zeros
    heuristic_matrix = torch.zeros_like(distance_matrix)
    # Calculate the cumulative demand from the depot to each customer
    cumulative_demand = torch.cumsum(demands, dim=0)
    # Calculate the heuristics based on the distance matrix and cumulative demand
    for i in range(n):
        for j in range(n):
            if i != j:
                # Calculate the difference in cumulative demand from the depot to j
                # and from i to j
                demand_diff = cumulative_demand[j] - cumulative_demand[i]
                # Use the distance as a weight
                heuristic_value = -distance_matrix[i, j] + demand_diff
            else:
                # No cost to go back to the depot
                heuristic_value = 0
            # Update the heuristic matrix
            heuristic_matrix[i, j] = heuristic_value
    return heuristic_matrix
```

Figure 29: Evolution udpate of NeRM on CVRP_POMO.

Init Prompt:
""""""

Init Population:
Assisting in solving Capacitated Vehicle Routing Problem (CVRP) with some prior heuristics. CVRP requires finding the shortest path that visits all given nodes and returns to the starting node. Each node has a demand and each vehicle has a capacity. The total demand of the nodes visited by a vehicle cannot exceed the vehicle capacity. When the total demand exceeds the vehicle capacity, the vehicle must return to the starting node.
""""""

Init Algorithm:
```
def seed_func(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    return torch.zeros_like(distance_matrix)
```

Figure 30: Initialization of NeRM on CVRP_LEHD.

MoA Reflection:

```
"""-------------------------------------- Request ---------------------------------------------------------
Use reference algorithms, explain the reasons why the better algorithm is good.
def heuristics(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    dist_mat = distance_matrix.numpy()
    demands = demands.numpy()
    # Create an initial heuristic value of 0 for the diagonal (no movement)
    heuristics = np.zeros_like(dist_mat)
    # Calculate the sum of demands for each vehicle
    total_demand = demands.sum()
    # Calculate the heuristic for each edge
    for i in range(len(demands)):
        for j in range(len(demands)):
            if i != j:
                # Calculate the demand that would be served if we went to node j
                new_demand = total_demand - demands[j]
                if demands[i] <= total_demand - demands[j]:
                    # If the new demand is within the vehicle capacity, calculate the heuristic
                    heuristics[i, j] = dist_mat[i, j] * (1 - new_demand / total_demand)
    return torch.from_numpy(heuristics)

Worse Algorithm:
def heuristics(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    return torch.zeros_like(distance_matrix)
"""
-------------------------------------- Response ---------------------------------------------------------
```

Algorithm Reflection:

1. Use vectorized operations for efficiency.
2. Incorporate demand and distance in a meaningful way.
3. Avoid unnecessary loops.

Figure 31: MoA reflection of NeRM on CVRP_LEHD.

MoP Reflection:

"""-------------------------------------- Request ---

Use reference prompts, explain the reasons why the better prompt is good.

Reference Prompt:

Better Prompt: Revitalize eCVRP resolution through the synthesis of multi-criteria algorithms that optimize delivery time, route length, and load capacities simultaneously. Leverage mixed heuristic methodologies, evolutionary optimization techniques, advanced dynamic programming approaches, and strategic tabu search for adaptive, computationally efficient routing. Target an algorithmic ensemble that maximizes the synergy between service punctuality and economic efficiency in the dynamic distribution landscape.

Worse Prompt: Assisting in solving Capacitated Vehicle Routing Problem (CVRP) with some prior heuristics. CVRP requiresfinding the shortest path that visits all given nodes and returns to the starting node. Each node has a demand and each vehicle has a capacity. The total demand of the nodes visited by a vehicle cannot exceed the vehicle capacity. When the total demand exceeds the vehicle capacity, the vehicle must return to the starting node.

"""

-------------------------------------- Response ---

Prompt Reflection:

Detailed, innovative, multi-faceted approach.

Figure 32: MoP reflection of NeRM on CVRP_LEHD.

MoA Revolution:

```
"""------------------------------------- Request ---------------------------------------------------------
Use reference algorithm, assisting by reflect of previous round, generate a new algorithm.
Assisting in solving Capacitated Vehicle Routing Problem (CVRP) with some prior heuristics.
CVRP requires finding the shortest path that visits all given nodes and returns to the starting
node. Each node has a demand and each vehicle has a capacity. The total demand of the
nodes visited by a vehicle cannot exceed the vehicle capacity. When the
total demand exceeds the vehicle capacity, the vehicle must return to the starting node.
Algorithm:
def heuristics(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    dist_mat = distance_matrix.numpy()
    demands = demands.numpy()
    heuristics = np.zeros_like(dist_mat)
    # Calculate the sum of demands for each vehicle
    total_demand = demands.sum()
    # Calculate the heuristic for each edge
    for i in range(len(demands)):
        for j in range(len(demands)):
            if i != j:
                new_demand = total_demand - demands[j]
                if demands[i] <= total_demand - demands[j]:
                    heuristics[i, j] = dist_mat[i, j] * (1 - new_demand / total_demand)
    return torch.from_numpy(heuristics)

Algorithm Reflection:
1. Use vectorized operations for efficiency.
2. Incorporate demand and distance in a meaningful way.
3. Avoid unnecessary loops.
"""
------------------------------------- Response ---------------------------------------------------------
Algorithm Revolution:
def evolved_algorithm(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    total_capacity = demands.sum()
    load_factor = demands / total_capacity
    demand_normalized = demands / demands.sum()
    heuristic_values = -distance_matrix + demand_normalized.unsqueeze(1) *
demands.unsqueeze(0)
    # Introduce a penalty for high load factor to encourage load balancing
    penalty_factor = 0.1
    high_load_penalty = penalty_factor * torch.clamp(load_factor, min=1.0) *
torch.clamp(demand_normalized, min=0.5)
    heuristic_values -= high_load_penalty
    # Introduce a discount factor for high savings to encourage exploration
    discount_factor = 0.05
    high_savings_penalty = discount_factor * torch.clamp(torch.abs(heuristic_values),
max=0.5)
    heuristic_values -= high_savings_penalty
    # Introduce a dynamic adjustment for exploration and exploitation
    exploration_factor = torch.rand_like(load_factor)
    exploitation_factor = 1 - exploration_factor
    exploration_factor = torch.clamp(exploration_factor, min=0.5, max=1.0)
    exploitation_factor = torch.clamp(exploitation_factor, min=0.5, max=1.0)
    # Combine exploration and exploitation factors
    heuristic_values *= exploration_factor.unsqueeze(0) * exploration_factor.unsqueeze(1)
    heuristic_values *= exploitation_factor.unsqueeze(0) * exploitation_factor.unsqueeze(1)
    # Adapt heuristics with nuanced scaling for penalties, discounts, and dynamic conditions
    dynamic_factor = load_factor ** 0.5
    dynamic_factor = torch.clamp(dynamic_factor, min=1.0)
    heuristic_values *= dynamic_factor.unsqueeze(0) * dynamic_factor.unsqueeze(1)
    return heuristic_values
```

Figure 33: MoA revolution of NeRM on CVRP_LEHD.

MoP Revolution:

```
"""------------------------------------- Request ---------------------------------------------------------
Use reference prompt and code, assisting by reflect of previous round, generate a new prompt.
Prompt: Solving Traveling Salesman Problem (TSP) via guided local search. TSP requires finding
the shortest path that visits all given nodes and returns to the starting node.

Best code:
def heuristics(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    total_capacity = demands.sum()
    load_factor = demands / total_capacity
    demand_normalized = demands / demands.sum()
    heuristic_values = -distance_matrix + demand_normalized.unsqueeze(1) *
demands.unsqueeze(0)
    # Introduce a penalty for high load factor to encourage load balancing
    penalty_factor = 0.1
    high_load_penalty = penalty_factor * torch.clamp(load_factor, min=1.0) *
torch.clamp(demand_normalized, min=0.5)
    heuristic_values -= high_load_penalty
    # Introduce a discount factor for high savings to encourage exploration
    discount_factor = 0.05
    high_savings_penalty = discount_factor * torch.clamp(torch.abs(heuristic_values), max=0.5)
    heuristic_values -= high_savings_penalty
    # Introduce a dynamic adjustment for exploration and exploitation
    exploration_factor = torch.rand_like(load_factor)
    exploitation_factor = 1 - exploration_factor
    exploration_factor = torch.clamp(exploration_factor, min=0.5, max=1.0)
    exploitation_factor = torch.clamp(exploitation_factor, min=0.5, max=1.0)
    # Combine exploration and exploitation factors
    heuristic_values *= exploration_factor.unsqueeze(0) * exploration_factor.unsqueeze(1)
    heuristic_values *= exploitation_factor.unsqueeze(0) * exploitation_factor.unsqueeze(1)
    # Adapt heuristics with nuanced scaling for penalties, discounts, and dynamic conditions
    dynamic_factor = load_factor ** 0.5
    dynamic_factor = torch.clamp(dynamic_factor, min=1.0)
    heuristic_values *= dynamic_factor.unsqueeze(0) * dynamic_factor.unsqueeze(1)
    return heuristic_values

Prompt Reflection:
Detailed, innovative, multi-faceted approach.
"""
------------------------------------- Response -------------------------------------------------------
```

Prompt Revolution:

Design a hybrid optimization algorithm for the Capacitated Vehicle Routing Problem (CVRP) that combines dynamic programming for route length minimization, advanced heuristics for demand and capacity management, and tabu search for strategic exploration. The algorithm should integrate evolutionary optimization techniques to refine routing and balance cost and timeliness. The objective is to achieve optimal load balancing and resource utilization, ensuring punctuality and economic efficiency in a dynamic distribution environment. The algorithm must adapt to changing conditions, effectively managing fluctuations in demand and vehicle capacity, and demonstrate robustness in solving CVRP instances with varying complexities.

Figure 34: MoP revolution of NeRM on CVRP_LEHD.

Evolution Update:

Prompt:

Design a hybrid optimization algorithm for the Capacitated Vehicle Routing Problem (CVRP) that combines dynamic programming for route length minimization, advanced heuristics for demand and capacity management, and tabu search for strategic exploration. The algorithm should integrate evolutionary optimization techniques to refine routing and balance cost and timeliness. The objective is to achieve optimal load balancing and resource utilization, ensuring punctuality and economic efficiency in a dynamic distribution environment. The algorithm must adapt to changing conditions, effectively managing fluctuations in demand and vehicle capacity, and demonstrate robustness in solving CVRP instances with varying complexities.

Seed Algorithm:

```
def new_seed_func(distance_matrix: torch.Tensor, demands: torch.Tensor) -> torch.Tensor:
    total_capacity = demands.sum()
    load_factor = demands / total_capacity
    demand_normalized = demands / demands.sum()
    heuristic_values = -distance_matrix + demand_normalized.unsqueeze(1) *
demands.unsqueeze(0)
    # Introduce a penalty for high load factor to encourage load balancing
    penalty_factor = 0.1
    high_load_penalty = penalty_factor * torch.clamp(load_factor, min=1.0) *
torch.clamp(demand_normalized, min=0.5)
    heuristic_values -= high_load_penalty
    # Introduce a discount factor for high savings to encourage exploration
    discount_factor = 0.05
    high_savings_penalty = discount_factor * torch.clamp(torch.abs(heuristic_values), max=0.5)
    heuristic_values -= high_savings_penalty
    # Introduce a dynamic adjustment for exploration and exploitation
    exploration_factor = torch.rand_like(load_factor)
    exploitation_factor = 1 - exploration_factor
    exploration_factor = torch.clamp(exploration_factor, min=0.5, max=1.0)
    exploitation_factor = torch.clamp(exploitation_factor, min=0.5, max=1.0)
    # Combine exploration and exploitation factors
    heuristic_values *= exploration_factor.unsqueeze(0) * exploration_factor.unsqueeze(1)
    heuristic_values *= exploitation_factor.unsqueeze(0) * exploitation_factor.unsqueeze(1)
    # Adapt heuristics with nuanced scaling for penalties, discounts, and dynamic conditions
    dynamic_factor = load_factor ** 0.5
    dynamic_factor = torch.clamp(dynamic_factor, min=1.0)
    heuristic_values *= dynamic_factor.unsqueeze(0) * dynamic_factor.unsqueeze(1)
    return heuristic_values
```

Figure 35: Evolution udpate of NeRM on CVRP_LEHD.