CALM: CO-EVOLUTION OF ALGORITHMS AND LANGUAGE MODEL FOR AUTOMATIC HEURISTIC DESIGN

Anonymous authorsPaper under double-blind review

ABSTRACT

Tackling complex optimization problems often relies on expert-designed heuristics, typically crafted through extensive trial and error. Recent advances demonstrate that large language models (LLMs), when integrated into well-designed evolutionary search frameworks, can autonomously discover high-performing heuristics at a fraction of the traditional cost. However, existing approaches predominantly rely on verbal guidance, i.e., manipulating the prompt generation process, to steer the evolution of heuristics, without adapting the underlying LLM. We propose a hybrid framework that combines verbal and numerical guidance, the latter achieved by fine-tuning the LLM via reinforcement learning (RL) based on the quality of generated heuristics. This joint optimization allows the LLM to co-evolve with the search process. Our method outperforms state-of-the-art (SOTA) baselines across various optimization tasks, running locally on a single 24GB GPU using a 7B model with INT4 quantization. It surpasses methods that rely solely on verbal guidance, even when those use significantly more powerful API-based models.

1 Introduction

Complex optimization problems are prevalent in real-world applications, including logistics (Duan et al., 2022; Tresca et al., 2022), scheduling (Mihoubi et al., 2021; Palacio et al., 2022), and transportation (Dahmani et al., 2024; Pereira et al., 2021). Traditionally, solving these problems relies heavily on manually crafting high-quality heuristics, a labor-intensive process requiring substantial expert knowledge. Given the limitations of this manual approach, Automatic Heuristic Design (AHD) emerged to streamline heuristic generation. Nevertheless, classic AHD approaches like Genetic Programming (GP) (Burke et al., 2009) still depend significantly on human-defined problem-specific components, limiting the search space and flexibility.

Recently, the advent of Large Language Models (LLMs) has introduced promising avenues for AHD by employing LLMs as heuristic generators and evolutionary computing (EC) techniques as a search framework. In this paradigm, heuristics generated by LLMs are iteratively evaluated through a predefined simulation framework, and superior heuristics inform subsequent generation prompts, thus creating a feedback-driven evolutionary loop (Liu et al., 2024a). Nevertheless, existing LLM-based AHD methods predominantly keep the underlying LLM untouched and merely guide heuristic evolution via textual prompt manipulations, referred to as "verbal gradients" (Ye et al., 2024). Consequently, these methods inherently neglect the opportunity of tuning and enhancing the generative capability of LLM based on the feedback from heuristic designs.

We propose Co-evolution of Algorithms and the Language Model (CALM) to capture this opportunity. CALM drastically differs from the state-of-the-art (SOTA) (Liu et al., 2024a; Ye et al., 2024; Dat et al., 2025; Zheng et al., 2025) by enabling the LLM to co-evolve alongside heuristic designs. This co-evolution is made possible by treating the heuristic generation process not only as a target of optimization but also as a rich source of training data. As heuristics are continually proposed, evaluated, and selected based on their performance, the evolutionary loop naturally produces abundant prompt-response-performance triplets. These data points are highly informative, as each heuristic's effectiveness provides an implicit signal about the utility of the underlying generation process. By using this signal as feedback for reinforcement learning (RL), we can fine-tune the LLM, thereby applying what we term "numerical gradients" to adapt the model itself. This co-evolution approach

060

061

062

063

064

065 066

067

068

069

071

072

073

074

075

076

077

078

079

080

081

082

083 084

085

087

880

090

091

092

094

096

098

099

100

101

102

103

104

105

106

107

Figure 1: Pipeline of existing LLM-based AHD methods (Romera-Paredes et al., 2024; Ye et al., 2024; Dat et al., 2025; Zheng et al., 2025) under a fixed LLM and our new approach CALM that enables the co-evolution of LLM in the iterative heuristic search process. New components are presented in bright colors.

unlocks a new dimension of adaptability, allowing the LLM to internalize characteristics of successful heuristics and improve its future generations.

CALM is one of the first LLM-based AHD frameworks that jointly optimize both the prompt generation process and the LLM model itself, overcoming the limitations of fixed-model approaches. For prompt generation, CALM introduces a suite of evolutionary operators, including fine-granularity mutation operators (injection and replacement) and a diversity-aware crossover operator, that promote meaningful and diverse heuristic variations while preserving structural coherence. Furthermore, a simple yet effective collapse mechanism is developed to help escape the local optima. For model improvement, CALM employs a memory-efficient RL algorithm GRPO (Shao et al., 2024) with a carefully designed reward function to enable efficient fine-tuning. Experimental results demonstrate that our new approach can discover heuristics that beat existing SOTA baselines (Liu et al., 2024a; Ye et al., 2024; Zheng et al., 2025), while running entirely on a local computer with a single 24GB GPU, in contrast to prior methods that depend heavily on commercial LLM APIs.

2 Related Work

As our approach centers on fine-tuning LLMs by RL for solving optimization problems, we review relevant literature in both RL and LLMs applied to optimization. Additional related topics, including LLMs for code generation and RL-based LLM fine-tuning, are discussed in Appendix B.

RL for Optimization Problems. Existing RL-based methods for optimization can be broadly categorized by the role the learned policy plays: (1) Instance-Level Solution Generator. Deep RL has been widely adopted to learn policies for solving specific optimization instances (Kwon et al., 2020; Pan et al., 2023; Bi et al., 2024). However, these methods differ fundamentally from LLM-based AHD methods, as they directly produce solutions rather than design the algorithms that generate them. The LLM-based AHD approach operates at a meta level, seeking to learn the algorithmic structure that produces solutions. This distinction also applies to the broader class of Neural Combinatorial Optimization (NCO) (Luo et al., 2024; Xiao et al., 2024; Sui et al., 2024; Zheng et al., 2023), where models are trained to directly solve instances. Moreover, NCO methods often require explicit adaptation to handle problem scales not seen during training, whereas our method generalizes more naturally to new scales. (2) Heuristic Generator. Some RL-based methods target meta-level search to discover heuristics instead of instance-level solutions. For example, AlphaDev (Mankowitz et al., 2023) learns to combine low-level operations to discover faster sorting algorithms, and Yi et al. (2022) searches for high-performing metaheuristics from predefined algorithmic components. While having similar goals, these approaches rely heavily on hand-engineered building blocks, akin to traditional AHD frameworks (Pillay and Qu, 2018; Sánchez-Díaz et al., 2021; Burke et al., 2009). In contrast, LLM-based method reduces manual intervention by leveraging LLMs to explore an open-ended heuristic space with minimal prior specification.

LLM for Optimization Problems. Studies in this area fall into two categories depending on how LLMs are employed: (1) *Instance-Level Solution Generator.* Several works (Abgaryan et al., 2024;

Jiang et al., 2024; Wu et al., 2024) prompt LLMs with instance-specific inputs for direct solution generation. LLM-based methods in this category focus on discovering reusable heuristics. Moreover, methods such as that proposed by Jiang et al. (2024) and Wu et al. (2024) keep LLM parameters frozen, and Abgaryan et al. (2024) fine-tune the model using supervised labels from an existing solver (Perron and Furnon, 2024). In contrast, our approach requires no imitation dataset, enabling its application to problems lacking established solvers. (2) Heuristic Generator. LLM-based AHD methods (Liu et al., 2023a; Chen et al., 2025; Romera-Paredes et al., 2024; Liu et al., 2024a; Ye et al., 2024; Liu et al., 2024b; Dat et al., 2025; Zheng et al., 2025; Novikov et al., 2025) repeatedly ingest information about the current elite heuristics—typically their natural-language descriptions, source code, and performance scores—and, via fixed prompt templates that mimic genetic operators, produce new candidate heuristics. Those candidates are then executed and evaluated, and the resulting feedback is fed back into the prompt, forming an evaluate-generate loop that continues until the evaluation budget is exhausted. Additionally, some recent studies have also explored incorporating strategies such as reduction techniques (Thach et al., 2025) and trajectory-based analysis (Yang et al., 2025), to further enhance AHD. However, prior work keeps the LLM static. Our approach improves this by continuously fine-tuning the LLM using prompt-response-performance tuples from the evolutionary process, enhancing future heuristic generation. Notably, there are concurrent explorations on fine-tuning LLMs for AHD (Surina et al., 2025; Liu et al., 2025). These studies provide valuable insights into how preference-based fine-tuning methods such as DPO (Rafailov et al., 2023) can improve heuristic discovery. Our work adopts a different approach by employing the score-based RL algorithm (Shao et al., 2024) to fine-tune LLMs for AHD, and further introduces specialized designs such as fine-granularity operators to enhance the fine-tuning process through prompt manipulation.

3 PRELIMINARY

3.1 LLM-BASED AHD

Let P be a problem with input space \mathcal{I} and solution space \mathcal{S} , and let a *heuristic* be a function $h: \mathcal{I} \to \mathcal{S}$. Given a training set $D \subset \mathcal{I}$ and an objective $f: \mathcal{S} \to \mathbb{R}$ (lower is better), the performance of a heuristic is $g(h) = \mathbb{E}_{x \in D}[-f(h(x))]$. Let \mathcal{H} denote the space of all feasible heuristics. The objective of AHD is to identify the optimal heuristic within this space, i.e., $h^* = \arg\max_{h \in \mathcal{H}} g(h)$.

LLM-based AHD is AHD where LLM serves as a heuristic generator. In practice, the LLM is charged with designing the core decision function of a solver. For example, on tasks like the Traveling Salesman Problem (TSP) or the Capacitated Vehicle Routing Problem (CVRP), an LLM-based AHD method might generate a function, which selects the next city to visit or constructs an edge-desirability matrix to guide solution search within an Ant Colony Optimization (ACO) framework.

3.2 GRPO

GRPO (Shao et al., 2024) is a recent RL algorithm that has proven effective in training LLMs, as evidenced by its application in models such as DeepSeek-R1. GRPO starts from an initial model π_{θ} and a reward function denoted by $r_{\phi}(q,o)$ that maps the prompt q and the generated response o to a scalar. At the beginning of each training round, it snapshots π_{θ} as a reference model $\pi_{\rm ref}$. Then, it split all task prompts into multiple batches. When training for each prompt batch \mathcal{D}_b , it first snapshots π_{θ} as $\pi_{\rm old}$. For each task prompt $q \in \mathcal{D}_b$, it samples a group of G responses $\{o_i\}_{i=1}^G \sim \pi_{\theta_{\rm old}}$ and computes rewards $\mathbf{r} = \{r_i = r_{\phi}(q,o_i)\}_{i=1}^G$ for each prompt-response pair. Subsequently, it computes the advantage $\hat{A}_{i,t}$ for each token t in response i as the normalized reward $(r_i - \text{mean}(\mathbf{r}))/\text{std}(\mathbf{r})$. The model parameters θ are updated by maximizing the following objective function:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{\left[q \sim \mathcal{Q}, \{o_i\} \sim \pi_{\theta_{\text{old}}}\right]} \\
\frac{1}{G} \sum_{i=1}^{G} \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \left\{ \min \left[\hat{r}_{i,t} \hat{A}_{i,t}, \operatorname{clip}\left(\hat{r}_{i,t}, 1 - \varepsilon, 1 + \varepsilon\right) \hat{A}_{i,t} \right] - \beta \, \mathbb{D}_{\text{KL}} \left[\pi_{\theta} \| \pi_{\text{ref}} \right] \right\}, \quad (1)$$

where ϵ and β are hyper-parameters, $\hat{r}_{i,t} = \pi_{\theta}(o_{i,t} \mid q, o_{i,< t})/\pi_{\theta}^{\text{old}}(o_{i,t} \mid q, o_{i,< t})$, and the KL divergence term is computed using an unbiased estimator (Schulman, 2020) with guaranteed positivity.

GRPO uses the group mean reward as a baseline to eliminate the need for an auxiliary value network, thereby reducing memory requirements. Additionally, the clipping mechanism combined with KL divergence regularization ensures stable and conservative updates.

4 METHODOLOGY

To explore the benefit of RL-based fine-tuning for discovering higher-quality heuristics in LLM-based AHD, we introduce CALM, a novel framework that integrates both verbal and numerical guidance in evolutionary heuristic search. As shown in Fig. 1, CALM maintains a pool of heuristics, each with its own idea, code, and performance. At every round, CALM draws a feasible evolutionary operator to produce a new prompt q. Subsequently, G responses are sampled from the local LLM π_{θ} , which are then evaluated. Based on the evaluation results, rewards are assigned to each response for GRPO to train the LLM, and new feasible heuristics are added to the pool. Consequently, CALM returns the best-so-far heuristic after running T rounds. Next, we elaborate on the critical techniques in CALM: prompt generation, collapse mechanism, and the reward function.

4.1 PROMPT GENERATION

CALM provides several evolutionary operators: injection, replacement, crossover, simplification, and initialization. Prompts are predominantly generated by the selected operator and heuristics sampled from maintained pools. The initialization operator is an exception, as it does not require heuristics from the pool. Next, we elaborate on the heuristic sampling method and operators¹.

Heuristic Sampling Method. The heuristic sampling approach varies for the crossover operator, details of which will be provided when introducing this operator. For the remaining operators, i.e., injection, replacement, and simplification, the heuristics are selected based on their performance rankings like (Liu et al., 2024a). Specifically, the probability of sampling a heuristic h is inversely proportional to its rank in the current pool (i.e., proportional to $1/\text{rank}_p(h)$). Heuristics ranked below a threshold, defined as the population size, are assigned a probability of zero.

Fine-Granularity Mutation Operators: Injection & Replacement. GRPO assigns an advantage score to each token based on the relative reward of the full response compared to others from the same prompt. This means each part of a heuristic is encouraged or penalized depending on the quality of the whole. However, heuristic performance can shift dramatically with changes to even a single sub-component, making uniform treatment of all parts—in terms of gradient direction—unreliable.

While cumulative gradient updates can correct misattributed rewards or penalties for the same token appearing in different responses, we aim to further boost this process. To this end, we introduce two novel operators that enable more precise control over heuristic variations. These operators encourage the LLM to retain more common parts while introducing meaningful modifications to the input heuristic (See Appendix E for examples). Consequently, GRPO is expected to more effectively identify the contribution of individual structural changes. The two newly designed operators are:

Injection. Given an existing heuristic, the injection operator prompts the LLM to incorporate a new component into it. Additionally, a concise description of the new component must be included in the response. All component descriptions are stored, and subsequent applications of the injection operator require the LLM to introduce components distinct from those previously saved, promoting diversity in generated heuristics. Unlike mutation operators in prior LLM-based AHD methods (Zheng et al., 2025; Liu et al., 2024a), which are fed with full heuristic codes: (1) Our approach uses compact summaries instead of full code, allowing more heuristics to fit within the LLM's context window; (2) Saved component descriptions are globally accessible and not limited to the currently sampled heuristics; (3) Prior methods often require entirely new heuristics, while our approach focuses on more granular modifications; (4) When the number of heuristics is below the population size, the sampling probability of the injection operator is increased to encourage exploration in the phase of population expansion.

Replacement. Given an existing heuristic, the replacement operator prompts the LLM to rewrite an existing component under a specific instruction. There are three distinct instructions, and each time the replacement operator is applied, one is randomly sampled for the given heuristic. While the

¹The complete algorithm and prompt details are in Appendix C and D, respectively, due to space limit.

"rewrite hyper-parameter" instruction is also present in prior studies (Liu et al., 2024a; Zheng et al., 2025), CALM introduces two novel instructions: (1) Rewrite an instance-independent decision rule as an instance-dependent one—to improve the heuristic's adaptability to varying problem contexts; (2) Rewrite a fragment that assigns equal or near-equal credit to all candidates as one that differentiates credit based on contextual performance—to encourage more effective prioritization and refined decision-making.

Diversity-Aware Crossover. To balance exploitation and exploration, each crossover invocation randomly chooses between (1) *performance-based*: sample both parents by performance rank; and (2) *diversity-based*: sample the first parent $h_{c,1}$ by performance rank and the second from all retained heuristics with probability inversely proportional to diversity rank (larger diversity is better). Specifically, let idea_token(·) denote the set of unique tokens in a heuristic's idea, the diversity is: $\operatorname{div}(h_{c,1},h) = |\operatorname{idea_token}(h) \setminus \operatorname{idea_token}(h_{c,1})|/|\operatorname{idea_token}(h)|$. This hybrid mechanism ensures that at least one parent heuristic is of high quality, while the second parent is either high-performing or structurally novel. The diversity-aware selection expands the evolutionary search space and leverages underutilized heuristics, potentially unlocking novel strategies that might otherwise be overlooked due to suboptimal early performance. More discussions are moved to Appendix F.

Simplification Operator. As heuristic structures grow increasingly complex through repeated applications of injection, crossover, and replacement, there is a risk of accumulating redundant or unnecessarily verbose components. The simplification operator counterbalances this tendency by prompting the LLM to produce a more concise and effective version of a given heuristic.

Initialization Operator. In cases where there is no heuristic in the pool (e.g., no initial/seeding function is provided), this operator is invoked to prompt the LLM to generate new heuristics.

4.2 COLLAPSE MECHANISM

Why to Collapse. A key reason LLM-based evolutionary heuristic search can succeed is that prompts containing better-performing heuristics tend to guide the LLM toward generating even stronger ones. This creates a self-reinforcing feedback loop, gradually evolving a population of increasingly effective heuristics. However, this process can also lead to inbreeding and premature convergence: over time, the population becomes dominated by minor variations of the current best-performing heuristic. When this state persists without meaningful breakthroughs, the search risks becoming trapped in a local optimum, a classic challenge in evolutionary computing (Eshelman, 1991).

How to Collapse. As a remedy, CALM introduces a proactive collapse mechanism that resets the search process when it detects stagnation, allowing the system to escape local optima and reinitiate meaningful exploration. Specifically, when the search has plateaued—characterized by a prolonged lack of performance improvement—we reset the population by discarding all heuristics except two: the original seed algorithm and the current best-performing heuristic. These two retained heuristics jointly serve as the seed algorithms for the new search process, grounding it in past progress while freeing it from the genetic redundancy accumulated in the previous population.

When to Collapse. Once the heuristic pool reaches its target population size, CALM begins tracking stagnation using a no-breakthrough counter c_n , initialized to zero. This counter records the number of consecutive prompt rounds—each involving G sampled responses—that fail to yield a globally superior heuristic. If any sampled heuristic in a round surpasses all previous ones in performance, c_n is reset to zero; otherwise, it increments by one.

To escape local optima, CALM introduces a probabilistic collapse mechanism based on this counter. At the end of each round, collapse is triggered if: $\mathrm{random}(0,1) < c_n \delta_0$ or $c_n \geq C$, where $\delta_0 \ll 1$ controls the rate at which collapse probability grows, and C is a hard cap ensuring collapse happens by the C-th stagnation step at the latest. To aid in hyperparameter selection, we further provide an analytical approximation for the expected number of rounds before collapse is triggered:

$$\mathbb{E}\left[c_n \mid \text{collapse}, C > \frac{1}{\delta_0}\right] \approx \sqrt{\frac{\pi}{2\delta_0}}.$$
 (2)

This collision of a rising-probability rule with a fixed maximum fosters a balance between giving the search plenty of room to improve and ensuring it doesn't stall infinitely. A detailed proof and discussion about the benefit of the mechanism can be found in Appendix G.

4.3 Design of Reward Function

The reward function assigns a score to each LLM-generated response, enabling the RL algorithm to update the LLM's parameters and progressively improve its outputs. In AHD, we aim for responses that yield feasible, novel, and high-performing heuristics. To guide this process, we adopt a *progressive* scoring scheme that assigns increasing scores across the following categories: (1) infeasible responses that fail to produce valid heuristics, (2) duplicate heuristics offering no new insights, (3) new heuristics, and (4) new high-performing heuristics.

For each invalid response, we assign a reward bounded below by a scalar $r_{\text{invalid}} \in (-1, 0)$. Rewards for valid heuristics are defined relative to this bound, ensuring that valid outputs always score higher.

For valid heuristics, performance serves as the primary learning signal. However, because the quality of the generated heuristic is influenced by the prompt—particularly its base heuristics—we avoid attributing full credit or blame to the LLM alone. Instead, we reward improvements relative to the best base heuristic in the prompt, ensuring that learning reflects meaningful gains rather than prompt bias. Specifically, let H denote the set of base heuristics used to construct prompt q, and $h_{\rm new}$ be the heuristic parsed from the LLM's output o. We define the top base heuristic as $h_{\rm t_base} = \arg\max_{h \in H} g(h)$, and measure relative performance via:

$$\Delta(h_{\text{new}}, h_{\text{t_base}}) = \text{clip}\left(\frac{|g(h_{\text{new}}) - g(h_{\text{t_base}})|}{\min\{|g(h_{\text{new}})|, |g(h_{\text{t_base}})|\}}, 0, 1\right). \tag{3}$$

Let $\alpha_1, \alpha_2 \in (0,1)$ and $\alpha_1 > \alpha_2$, the reward function $r_{\phi}(q, o \mid h_{\text{new}}, h_{\text{t base}})$ is then defined as:

$$r_{\phi}(q, o \mid h_{\text{new}}, h_{\text{t_base}}) = \begin{cases} \alpha_1 r_{\text{invalid}}, & \text{if } \exists h \in H \text{ s.t. } g(h) = g(h_{\text{new}}); \\ \alpha_2 r_{\text{invalid}} \cdot \Delta(h_{\text{new}}, h_{\text{t_base}}), & \text{if } g(h_{\text{new}}) < g(h_{\text{t_base}}); \\ 1 + \Delta(h_{\text{new}}, h_{\text{t_base}}), & \text{if } g(h_{\text{new}}) > g(h_{\text{t_base}}). \end{cases}$$
(4)

Under the reward function above, the reward is primarily determined by whether the new heuristic improves over the best base heuristic or not, with the relative performance gap further modulating the strength of the reward or penalty. When the generated heuristic is identical in performance to an existing base heuristic, a small but consistent reward ($\alpha_1 r_{\text{invalid}}$) is given to discourage trivial reproduction. If the new heuristic underperforms relative to the best base, a scaled negative reward is applied, while genuine improvements yield strictly positive rewards starting from 1.

5 EXPERIMENTS

Implementation Details of CALM. We build CALM on Unsloth (Daniel Han and team, 2023) and employ an INT4-quantized Qwen2.5-7B-Instruct model (Yang et al., 2024), fine-tuning just 1.15% of its weights. INT4 compression cuts memory usage up to $8\times$ versus FP32 but degrades precision. According to Yang et al. (2024), performance ranks as follows: GPT-4o-mini \approx Qwen2.5-Turbo > Qwen2.5-14B-Instruct > Qwen2.5-7B-Instruct > Qwen2.5-7B-Instruct-INT4. The 14B and 7B Instruct models share the same architecture, so the larger parameter count drives the 14B's edge over the 7B, while quantization further reduces the 7B's accuracy. Consequently, GPT-4o-mini—based baselines retain a clear advantage in raw accuracy over our lean, resource-efficient setup. More implementation details can be found in Appendix H

Optimization Tasks. Existing LLM-based methods can demonstrate near-optimal or optimal performance on some benchmark problems, such as TSP (Liu et al., 2024a; Ye et al., 2024; Zheng et al., 2025) (aided by ACO solvers) and knapsack problem (KP) (Zheng et al., 2025), leaving little room for further improvement. Therefore, we focus on tasks that remain challenging for LLM-based AHD as follows: Online Bin Packing (OBP) problem and TSP under step-by-step construction task, CVRP and Orienteering Problem (OP) under an ACO search framework. Detailed problem descriptions can be found in Appendix H.3

Baselines. To evaluate CALM, we compare its designed heuristics against the following baselines: (1) hand-crafted heuristics such as Best-Fit (Kenyon, 1995) for OBP, Greedy-Construct (GC) (Rosenkrantz et al., 1977) for TSP, and ACO (Blum, 2005) for CVRP and OP; (2) Nerual Combinatorial Optimization (NCO) methods including POMO (Kwon et al., 2020) and DeepACO (Ye et al., 2023); and (3) LLM-based AHD approaches like FunSearch (Romera-Paredes et al., 2024),

EoH (Liu et al., 2024a), ReEvo (Ye et al., 2024), HSEvo (Dat et al., 2025), OpenEvolve (Sharma, 2025), MCTS-AHD (Zheng et al., 2025), and EvoTune (Surina et al., 2025). Notably, AlphaEvolve (Novikov et al., 2025) does not release its official source code. OpenEvolve, developed by an independent group, is one of the most popular open-source reimplementations of AlphaEvolve. To ensure a fair comparison, we align CALM and all LLM-based AHD baselines with consistent settings, including shared seed heuristics, identical training datasets for evaluating heuristic performance, and comparable evaluation budgets—specifically, 1,000 heuristic evaluations for baselines and a fixed budget of 2,000 LLM queries for CALM across all tasks except OBP. Notably, prior AHD methods typically conduct 2,000 heuristic evaluations using over 4,000 queries for OBP, whereas CALM operates under a fixed budget of 2,000 queries.

5.1 OVERALL RESULTS

OBP. We train and evaluate CALM on the same dataset used by Zheng et al. (2025), which includes four training instances at varying scales and five testing instances spanning six different scales—two of which are out-of-domain and not seen during training. Results in Table 1 show that CALM consistently outperforms all baseline methods in terms of average optimality gap across the full test set. It can achieve superior performance on outof-domain and in-domain scales. Remarkably, CALM achieves a zero gap in set

Table 1: Average optimality gaps of heuristics for OBP over three runs. All methods are trained and evaluated on the same datasets as Zheng et al. (2025), with gaps measured relative to the lower bound by Martello and Toth (1990). Test sets whose scale matches the training distribution are underlined. Format: 1k_100 denotes instances with 1,000 items and a bin capacity of 100.

		Onlin	e Bin Pac	king (OBP	')		
Test sets	1k_100	1k_500	5k_100	5k_500	10k_100	10k_500	Avg.
Best Fit	4.77%	0.25%	4.31%	0.55%	4.05%	0.47%	2.40%
First Fit	5.02%	0.25%	4.65%	0.55%	4.36%	0.50%	2.56%
LLM-based AHD: GPT-40-mini (w/o. GRPO)							
FunSearch	2.45%	0.66%	1.30%	0.25%	1.05%	0.21%	0.99%
ЕоН	2.69%	0.25%	1.63%	0.53%	1.47%	0.45%	1.17%
ReEvo	3.94%	0.50%	2.72%	0.40%	2.39%	0.31%	1.71%
HSEvo	2.64%	1.07%	1.43%	0.32%	1.13%	0.21%	1.13%
OpenEvolve	4.84%	0.25%	4.28%	0.55%	4.07%	0.47%	2.41%
MCTS-AHD	2.45%	0.50%	1.06%	0.32%	0.74%	0.26%	0.89%
CALM (Ours)	2.78%	0.29%	0.83%	0.28%	0.50%	0.24%	0.82%
LLM-based AHD: Qwen2.5-7B-Instruct-INT4 (w/. GRPO)							
EvoTune	4.67%	0.25%	4.23%	0.55%	4.11%	0.60%	2.40%
CALM (Ours)	2.55%	0.00%	0.85%	0.17%	0.56%	0.14%	0.71%

1k_500, indicating exact optimal solutions at that scale.

TSP. CALM is trained on the same dataset used by Zheng et al. (2025): a training set of 64 TSP instances with N=50 nodes and three test sets of 1,000 instances each at N=50,100, and 200. As shown in Table 2, CALM-constructed heuristics outperform all LLM-based baselines on both out-of-domain test sets and achieve the second-best LLM-based result on the in-domain set. Notably, at the largest scale, CALM surpasses the NCO baseline POMO, which requires per-scale training.

CVRP. CALM is trained on 10 instances as in (Zheng et al., 2025) with N=50 nodes using the ACO framework, and evaluated on three test sets of 64 instances each at N=50,100, and 200, following the same generation protocol. During both training and testing, the number of ants and iterations is fixed to 30 and 100, respectively. As shown in Table 3, CALM consistently outperforms all LLM-based baselines across all test sets, including both the in-domain and out-of-domain ones.

Table 2: Performance on TSP, averaged over three runs. Methods are evaluated on three test sets of 1,000 instances each, using the same training and testing datasets as by Zheng et al. (2025). Indomain scales (i.i.d. to training) are underlined. Optimal tours are from LKH (Lin and Kernighan, 1973). Best LLM-based results are shaded, overall best in bold.

Traveling Salesman Problem (TSP)								
	N	=50	N:	=100	N=200			
Methods	Obj.↓	Gap↓	Obj.↓	Gap↓	Obj.↓	Gap↓		
Optimal	5.675	_	7.768	_	10.659	_		
GC	6.959	22.62%	9.706	24.94%	13.461	26.29%		
POMO	5.697	0.39%	8.001	3.01%	12.897	20.45%		
LLM-based AHD: GPT-3.5-turbo (w/o. GRPO)								
FunSearch	6.683	17.75%	9.240	18.95%	12.808	19.61%		
EoH	6.390	12.59%	8.930	14.96%	12.538	17.63%		
MCTS-AHD	6.346	11.82%	8.861	14.08%	12.418	16.51%		
LI	M-based	l AHD: GI	T-4o-mi	ini (w/o. G	RPO)			
FunSearch	6.357	12.00%	8.850	13.93%	12.372	15.54%		
EoH	6.394	12.67%	8.894	14.49%	12.437	16.68%		
OpenEvolve	6.281	10.68%	8.719	12.25%	12.148	13.96%		
MCTS-AHD	6.225	9.69%	8.684	11.79%	12.120	13.71%		
CALM (Ours)	6.273	10.54%	8.691	11.88%	12.104	13.56%		
LLM-based AHD: Qwen2.5-7B-Instruct-INT4 (w. GRPO)								
EvoTune	6.267	10.43%	8.777	12.99%	12.429	16.60%		
CALM (Ours)	6.244	10.04%	8.668	11.58%	12.088	13.41%		
		_						

OP. CALM is trained 5 OP instances with N=

50 nodes using the ACO framework and evaluated on three test sets of 64 instances each at N=

50, 100, and 200, following the generation protocol in HSEvo (Dat et al., 2025). Both training and testing use a fixed configuration of 20 ants and 50 iterations. As reported in Table 3, CALM consistently outperforms all other LLM-based baselines on the out-of-domain scales. As for the in-domain scale, it still outperforms EoH and the most recent approach, MCTS-AHD and EvoTune.

Table 3: Performance of ACO-based heuristics on CVRP and OP, averaged over three runs. All methods are evaluated on three test sets of 64 randomly generated instances each, following the setup in (Zheng et al., 2025) and (Dat et al., 2025), respectively. Optimal solutions are approximated using DeepACO with significantly more ants and iterations than those in the baseline configurations.

			CV	'RP					()P		
	N	=50	N=	=100	N=	200	N:	=50	N=	=100	N=	200
Methods	Obj.↓	Gap↓	Obj.↓	Gap↓	Obj.↓	Gap↓	Obj.↑	Gap↓	Obj.↑	Gap↓	Obj.↑	Gap↓
Optimal ACO	8.888 18.581	- 109.05%	14.932 30.107	- 101.63%	27.159 37.590	- 40.69%	19.867 13.354	- 32.69%	36.392 24.131	- 33.69%	63.380 37.586	- 40.69%
	LLM-based AHD: GPT-40-mini (w/o. GRPO)											
ЕоН	9.894	11.32%	16.953	13.54%	30.314	11.62%	13.388	32.61%	24.154	33.63%	37.319	41.12%
ReEvo	9.558	7.54%	16.350	9.50%	29.219	7.58%	15.103	23.98%	30.523	16.13%	53.807	15.10%
HSEvo	9.431	6.11%	16.396	9.81%	29.520	8.69%	15.082	24.08%	30.454	16.32%	53.260	15.97%
OpenEvolve	10.077	13.37%	17.418	16.65%	31.190	14.84%	14.314	27.95%	28.336	22.13%	48.576	23.36%
MCTS-AHD	9.372	5.44%	15.974	6.98%	28.434	4.70%	14.847	25.27%	30.163	17.12%	53.024	16.34%
CALM (Ours)	9.404	5.81%	16.046	7.46%	28.713	5.72%	15.017	24.41%	30.294	16.76%	53.098	16.22%
			LLM-be	ased AHD:	Qwen2.5-	7B-Instru	ct-INT4 (w/. GRPO)			
EvoTune	9.405	5.82%	15.975	6.98%	28.823	6.13%	15.053	24.23%	29.743	18.27%	50.499	20.32%
CALM (Ours)	9.228	3.83%	15.745	5.44%	28.230	3.95%	15.054	24.22%	30.778	15.43%	55.406	12.58%

5.2 DISCUSSION

Efficacy of our verbal gradient. For each problem instance, we further evaluate the design of our verbal gradient in isolation (i.e., without GRPO) by (1) switching the backend to the GPT-4o-mini API, (2) setting G=1, and (3) using T=4000 for OBP and T=2000 for all other tasks—matching the query budgets of prior LLM-based AHD methods. As shown in Tables 1–3, this API-based variant of CALM delivers performance on par with or superior to the recent MCTS-AHD approach: it achieves the lowest optimality gaps on the $5k_100$ and $10k_100$ OBP datasets and ranks second on average across all OBP test sets, matches MCTS-AHD and outperforms all other baselines on every CVRP test set, consistently surpasses MCTS-AHD on all OP instances, and closely tracks MCTS-AHD on TSP at N=50 and 100 while outperforming all non-MCTS baselines at those scales and even surpassing MCTS-AHD at N=200. These results demonstrate that, even without RL or advanced techniques such as reflection (Ye et al., 2024; Dat et al., 2025) and tree search (Zheng et al., 2025), CALM's verbal guidance mechanism remains highly effective, placing the API-based CALM firmly within the top tier of existing LLM-based AHD methods.

Power of RL. We have tested the performance of CALM without the GRPO algorithm and under many ablation settings. As shown in Table 4, results demonstrate that disabling the GRPO module causes the largest drop in performance across near all ablations. In other words, *The reinforcement-learning component has the most significant impact on overall performance among all ablation settings.* Morever, as illustrated in Table 1~3, with GRPO and our custom reward, the Qwen2.5-7B-Instruct-INT4-derived heuristic not only closes the gap but actually outperforms the GPT-4o-mini-based heuristic. We have also visualized the training curve in Figure 2. Results show CALM's heuristics lag early—likely due to GPT-4o-mini's head start—but as GRPO adapts the LLM, its heuristics converge and outperform all baselines. This suggests the transformative power of RL in enhancing AHD.

Impact of reward design. Our feasible-response reward allocates credit by comparing each generated heuristic against its parent(s), rather than attributing full reward or blame solely to the LLM. We evaluate two alternative schemes (keeping the infeasible-response penalty unchanged): (i) performance-based reward, where a feasible heuristic receives a positive reward proportional to its performance relative to the seed algorithm; and (ii) the $\{0.5\,r_{\rm rand},\,1\}$ -improvement reward, which assigns reward 1 if the new heuristic outperforms all parent or baseline heuristics, and $0.5\,r_{\rm rand}$ otherwise. Both alternatives remove the trivial-reproduction penalty and mitigate the performance bias present in Equation (4). As Table 4 demonstrates, neither variant beats our original design: the performance-based scheme underperforms even the no-RL baseline on the OP problem, while

the $\{0.5\,r_{\rm rand},\,1\}$ -improvement strategy delivers closer but still inferior results compared to our proposed reward function. This confirms the effectiveness of our original reward design.

Impact of collapse. We examine the impact of the collapse mechanism by analyzing the heuristics produced by CALM both without collapse and under various hyperparameter configurations that influence when collapse is triggered. As shown in Table 4, incorporating the collapse mechanism generally enhances the heuristic search process. An exception arises in the configuration with the strictest tolerance for not discovering a breakthrough heuristic (i.e., when $\delta_0 = 0.005$ and C = 15). A detailed analysis of the evolutionary trajectory under this setting reveals a significantly reduced number of breakthroughs. In one run on the OP problem, no breakthrough heuristic was identified after the 132nd LLM query. These findings suggest that setting a reasonable tolerance for the absence of breakthroughs—balancing patience with the benefits of early stopping—is important for supporting a more effective evolution.

Impact of operators. We evaluate each operator's contribution by measuring CALM's performance with that operator removed (Table 4). Results show that all operators positively impact heuristic quality. Crossover, injection, and replacement are similarly critical—removing any one notably degrades performance in either OBP or OP. Among all, removing simplification causes the largest drop in both tasks, likely because it uniquely reduces redundancy and curbs complexity, counterbalancing other operators that tend to increase

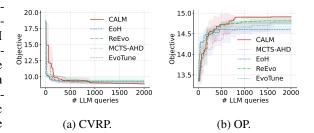


Figure 2: Objective score of the best heuristic in training averaged over 3 runs (std. dev. shaded).

Table 4: Optimality gap under ablation settings for problem OBP and OP averaged over three runs.

Method	OBP	OP
CALM (local, w/ GRPO)	0.71%	17.41%
CALM (API, w/o GRPO)	0.82%	19.13%
RL-based Fine-tuning		
local, w/o GRPO	1.78%	19.89%
$rew \in \{0.5r_{invalid}, 1\}$	1.04%	17.44%
rew=performance	1.24%	21.30%
Collapse Mechanism		
w/o Collapse	0.98%	19.57%
$\delta_0 = 0.0005, C = 15$	0.77%	18.31%
$\delta_0 = 0.005, C = 15$	1.93%	27.22%
$\delta_0 = 0.0005, C = \infty$	0.96%	19.50%
$\delta_0 = 0.005, C = \infty$	0.98%	18.38%
Operators		
w/o diversity	1.05%	19.44%
w/o crossover	0.88%	18.49%
w/o injection	1.11%	18.68%
w/o replacement	1.20%	17.57%
w/o simplification	1.35%	19.45%

heuristic length. Moreover, when crossover is applied without diversity-based selection—using only performance-based sampling—CALM performs worse than with no crossover at all, highlighting the importance of diversity awareness in the most-used operator.

Additional Experimental Results. Due to space constraints, additional experimental results are presented in Appendix I, including a detailed breakdown of running time, the effects of fine-tuning and foundational model choices, performance on more challenging OBP instances, scaling behavior, statistical significance (p-values), sensitivity analyses of reward-function hyperparameters, and the set of elite heuristics discovered.

6 CONCLUSION

This paper introduces CALM, the first framework to marry prompt evolution with on-the-fly LLM adaptation for AHD, freeing it from the constraints of fixed-model approaches. Running entirely on a single 24 GB GPU with a compact foundation model, CALM autonomously uncovers heuristics that outmatch SOTA API-based baselines across various challenging optimization scenarios. Moreover, even without the power of RL, CALM matches or exceeds prior best results using the same LLM API, demonstrating the potency of our verbal-gradient designs. In the future, we expect that scaling CALM's paradigm to larger models and extended post-training could further push the frontier of automated algorithm discovery.

7 ETHICS STATEMENT

This work complies with the ICLR Code of Ethics. The research did not involve human participants or animal experimentation. All datasets employed were obtained and used in accordance with relevant licensing and usage policies, ensuring no infringement of privacy. No personally identifiable information was processed, and no experiments were conducted that could pose privacy or security risks. Throughout the study, we have taken deliberate steps to mitigate biases and avoid discriminatory outcomes. We are committed to transparency, reproducibility, and integrity in both our methodology and reporting.

8 REPRODUCIBILITY STATEMENT

We have provided all information necessary to reproduce the main experimental results of this work, sufficient to support its central claims and conclusions. In detail, the complete algorithm is provided in Appendix C, the prompts used (including the system prompt, operator prompts, and task descriptions) are detailed in Appendix D, the experimental settings are described in Section 5 and further elaborated in Appendix H, and the full source code, including the discovered heuristics, is included in the supplementary material.

REFERENCES

- Jiahui Duan, Xialiang Tong, Fei Ni, Zhenan He, Lei Chen, and Mingxuan Yuan. A data-driven column generation algorithm for bin packing problem in manufacturing industry. *arXiv* preprint *arXiv*:2202.12466, 2022.
- Giulia Tresca, Graziana Cavone, Raffaele Carli, Antonio Cerviotti, and Mariagrazia Dotoli. Automating bin packing: A layer building matheuristics for cost effective logistics. *IEEE Transactions on Automation Science and Engineering*, 19(3):1599–1613, 2022.
- Bachir Mihoubi, Brahim Bouzouia, and Mehdi Gaham. Reactive scheduling approach for solving a realistic flexible job shop scheduling problem. *International journal of production research*, 59 (19):5790–5808, 2021.
- Jessica Coto Palacio, Yailen Martínez Jiménez, Leander Schietgat, Bart Van Doninck, and Ann Nowé. A q-learning algorithm for flexible job shop scheduling in a real-world manufacturing scenario. *Procedia CIRP*, 106:227–232, 2022.
- Nadia Dahmani, Ines Sbai, Takwa Tlili, and Saoussen Krichen. On solving the 21-cvrp using an adaptive chemical reaction algorithm: postal transportation real-case. *International Journal of System Assurance Engineering and Management*, pages 1–25, 2024.
- Rafael HM Pereira, Marcus Saraiva, Daniel Herszenhut, Carlos Kaue Vieira Braga, and Matthew Wigginton Conway. r5r: rapid realistic routing on multimodal transport networks with r 5 in r. *Findings*, 2021.
- Edmund K Burke, Mathew R Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R Woodward. Exploring hyper-heuristic methodologies with genetic programming. *Computational intelligence: Collaboration, fusion and emergence*, pages 177–201, 2009.
- Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning*, pages 32201–32223. PMLR, 2024a.
- Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, 2024.
- Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. Hsevo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26931–26938, 2025.

- Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. In *International Conference on Machine Learning*. PMLR, 2025.
 - Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
 - Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.
 - Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.
 - Yuxin Pan, Yize Chen, and Fangzhen Lin. Adjustable robust reinforcement learning for online 3d bin packing. *Advances in Neural Information Processing Systems*, 36:51926–51954, 2023.
 - Jieyi Bi, Yining Ma, Jianan Zhou, Wen Song, Zhiguang Cao, Yaoxin Wu, and Jie Zhang. Learning to handle complex constraints for vehicle routing problems. *Advances in Neural Information Processing Systems*, 37:93479–93509, 2024.
 - Fu Luo, Xi Lin, Zhenkun Wang, Xialiang Tong, Mingxuan Yuan, and Qingfu Zhang. Self-improved learning for scalable neural combinatorial optimization. *arXiv preprint arXiv:2403.19561*, 2024.
 - Pei Xiao, Zizhen Zhang, Jinbiao Chen, Jiahai Wang, and Zhenzhen Zhang. Neural combinatorial optimization for robust routing problem with uncertain travel times. *Advances in Neural Information Processing Systems*, 37:134841–134867, 2024.
 - Jingyan Sui, Shizhe Ding, Boyang Xia, Ruizhi Liu, and Dongbo Bu. Neuralgls: learning to guide local search with graph convolutional network for the traveling salesman problem. *Neural Computing and Applications*, 36(17):9687–9706, 2024.
 - Zhi Zheng, Shunyu Yao, Genghui Li, Linxi Han, and Zhenkun Wang. Pareto improver: Learning improvement heuristics for multi-objective route planning. *IEEE Transactions on Intelligent Transportation Systems*, 25(1):1033–1043, 2023.
 - Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.
 - Wenjie Yi, Rong Qu, Licheng Jiao, and Ben Niu. Automated design of metaheuristics using reinforcement learning within a novel general search framework. *IEEE Transactions on Evolutionary Computation*, 27(4):1072–1084, 2022.
 - Nelishia Pillay and Rong Qu. Hyper-heuristics: theory and applications. Springer, 2018.
 - Xavier Sánchez-Díaz, José Carlos Ortiz-Bayliss, Ivan Amaya, Jorge M Cruz-Duarte, Santiago Enrique Conant-Pablos, and Hugo Terashima-Marín. A feature-independent hyper-heuristic approach for solving the knapsack problem. *Applied Sciences*, 11(21):10209, 2021.
 - Henrik Abgaryan, Ararat Harutyunyan, and Tristan Cazenave. Llms can schedule. *arXiv preprint* arXiv:2408.06993, 2024.
 - Xia Jiang, Yaoxin Wu, Yuan Wang, and Yingqian Zhang. Unco: Towards unifying neural combinatorial optimization through large language model. *arXiv preprint arXiv:2408.12214*, 2024.
 - Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. Netllm: Adapting large language models for networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 661–678, 2024.

- Laurent Perron and Vincent Furnon. Or-tools, 2024. URL https://developers.google.com/optimization/.
 - Fei Liu, Xialiang Tong, Mingxuan Yuan, and Qingfu Zhang. Algorithm evolution using large language model. *arXiv preprint arXiv:2311.15249*, 2023a.
 - Zijie Chen, Zhanchao Zhou, Yu Lu, Renjun Xu, Lili Pan, and Zhenzhong Lan. Qube: Enhancing automatic heuristic design via quality-uncertainty balanced evolution, 2025. URL https://arxiv.org/abs/2412.20694.
 - Fei Liu, Rui Zhang, Zhuoliang Xie, Rui Sun, Kai Li, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Llm4ad: A platform for algorithm design with large language model. *arXiv preprint arXiv:2412.17287*, 2024b.
 - Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. Technical report, Google DeepMind, 05 2025. URL https://storage.googleapis.com/deepmind-media/DeepMind.com/Blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/AlphaEvolve.pdf.
 - Nguyen Thach, Aida Riahifar, Nathan Huynh, and Hau Chan. Redahd: Reduction-based end-to-end automatic heuristic design with large language models. *arXiv preprint arXiv:2505.20242*, 2025.
 - Xianliang Yang, Ling Zhang, Haolong Qian, Lei Song, and Jiang Bian. Heuragenix: Leveraging llms for solving complex combinatorial optimization challenges. *arXiv preprint arXiv:2506.15196*, 2025.
 - Anja Surina, Amin Mansouri, Lars Quaedvlieg, Amal Seddas, Maryna Viazovska, Emmanuel Abbe, and Caglar Gulcehre. Algorithm discovery with llms: Evolutionary search meets reinforcement learning. *arXiv preprint arXiv:2504.05108*, 2025.
 - Fei Liu, Rui Zhang, Xi Lin, Zhichao Lu, and Qingfu Zhang. Fine-tuning large language model for automated algorithm design. *arXiv preprint arXiv:2507.10614*, 2025.
 - Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741, 2023.
 - John Schulman. Approximating kl divergence, 2020. URL http://joschu.net/blog/kl-approx.html. Accessed: 2025-05-11.
 - Larry J Eshelman. Preventing premature convergence in genetic algorithms by preventing incest. In *Proceedings of Fourth International Conference on Genetic Algorithms*, 1991, 1991.
 - Michael Han Daniel Han and Unsloth team. Unsloth, 2023. URL http://github.com/unslothai/unsloth.
 - An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
 - Claire Kenyon. *Best-fit bin-packing with random order*. PhD thesis, Laboratoire de l'informatique du parallélisme, 1995.
 - Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977.
 - Christian Blum. Ant colony optimization: Introduction and recent trends. *Physics of Life reviews*, 2 (4):353–373, 2005.
 - Haoran Ye, Jiarui Wang, Zhiguang Cao, Helan Liang, and Yong Li. Deepaco: Neural-enhanced ant systems for combinatorial optimization. *Advances in neural information processing systems*, 36: 43706–43728, 2023.

- Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github.com/codelion/openevolve.
 - Silvano Martello and Paolo Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete applied mathematics*, 28(1):59–70, 1990.
 - Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.
 - Nafis Tanveer Islam, Joseph Khoury, Andrew Seong, Mohammad Bahrami Karkevandi, Gonzalo De La Torre Parra, Elias Bou-Harb, and Peyman Najafirad. Llm-powered code vulnerability repair with reinforcement learning and semantic reward. *arXiv preprint arXiv:2401.03374*, 2024.
 - Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. Code less, align more: Efficient llm fine-tuning for code generation with data pruning. *arXiv preprint arXiv:2407.05040*, 2024.
 - Junqiao Wang, Zeng Zhang, Yangfan He, Yuyang Song, Tianyu Shi, Yuchen Li, Hengyuan Xu, Kunyu Wu, Guangwu Qian, Qiuwu Chen, et al. Enhancing code llms with reinforcement learning in code generation. *arXiv preprint arXiv:2412.20367*, 2024.
 - Wei Shen and Chuheng Zhang. Policy filtration in rlhf to fine-tune llm for code generation. *arXiv* preprint arXiv:2409.06957, 2024.
 - Junjie Li, Aseem Sangalay, Cheng Cheng, Yuan Tian, and Jinqiu Yang. Fine tuning large language model for secure code generation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 86–90, 2024.
 - Zeyuan Ma, Hongshu Guo, Jiacheng Chen, Guojun Peng, Zhiguang Cao, Yining Ma, and Yue-Jiao Gong. Llamoco: Instruction tuning of large language models for optimization code generation. *arXiv* preprint arXiv:2403.01131, 2024.
 - John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
 - Yao Zhao, Rishabh Joshi, Tianqi Liu, Misha Khalman, Mohammad Saleh, and Peter J Liu. Slic-hf: Sequence likelihood calibration with human feedback. *arXiv preprint arXiv:2305.10425*, 2023.
 - Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu. Statistical rejection sampling improves preference optimization. *arXiv preprint arXiv:2309.06657*, 2023b.

AP	PENDIX
A	The Use of LLMs
В	Extended Discussion about Related Work
C	Complete Algorithm
D	Prompts Used in CALM
E	Example Prompt-Response Pairs for Injection and Replacement
F	Discussion about the Diversity-Aware Crossover Operator
G	More Details For the Collapse Mechanism
	G.1 Proof of Equation (2)
	G.2 Goodness
Н	More Experimental Details
	H.1 Implementation Details
	H.2 Baseline Implementations
	H.3 Description of Problems in Experiments
I	More Experimental Results
	I.1 Breakdown of CALM's Time Consumption
	I.2 Impact of Fine-tuning on the Foundational Model
	I.3 Impact of the Foundational LLM
	I.4 More Results on Harder OBP Instances
	I.5 Scaling Behavior
	I.6 P-values for Significance
	I.7 Sensitivity to the Hyperparameters in the Reward Function
	I.8 Generated Heurisitcs
J	Limitations
K	Broader Impact
L	License

A THE USE OF LLMS

LLM serves as a heuristic generator in our core method, as all LLM-based AHD frameworks do. The idea of the method was originally created and implemented by human. Additionally, the LLM was employed as a tool to refine and polish the writing.

B EXTENDED DISCUSSION ABOUT RELATED WORK

LLM for Code Generation. Recent work has explored improving LLMs' code generation capabilities through post-training (Islam et al., 2024; Tsai et al., 2024; Wang et al., 2024; Shen and Zhang, 2024; Li et al., 2024). For example, Islam et al. (2024) employ RL and semantic feedback to repair vulnerabilities, while Wang et al. (2024) demonstrate RL's effectiveness in enhancing code quality. Despite surface similarities, our task differs drastically: in code generation, objectives often prioritize pass rates (Shen and Zhang, 2024; Wang et al., 2024; Tsai et al., 2024) or safety (Li et al., 2024; Islam et al., 2024), whereas our goal is to maximize heuristic performance. Moreover, in code generation, fine-tuning aims to produce a generally stronger model, while in our case, both the model tuning and prompt evolution serve a singular goal—improving the quality of generated heuristics.

Notably, LLaMoCo (Ma et al., 2024) trains LLMs for optimization by fine-tuning on curated prompt—code pairs and enabling direct code generation for new problems. Its training data is derived from established sources such as papers, competitions, and benchmarks. By contrast, CALM adapts LLMs using prompts and responses generated dynamically during the evolutionary process, allowing problem-specific adaptation without external data. A promising future direction is to combine the supervised training of LLaMoCo as a first stage with CALM's reinforcement learning as a second stage for adaptive optimization.

RL for LLM Fine-tuning. Reinforcement learning is a central technique for fine-tuning large language models, with the RLHF paradigm commonly relying on Proximal Policy Optimization (PPO) (Schulman et al., 2017) to iteratively refine model outputs based on human feedback. Building on this, Group Relative Policy Optimization (GRPO) (Shao et al., 2024) simplifies training by removing the need for a separate value network, instead estimating baselines over groups of candidate completions—leading to improved sample efficiency and stability. Other alternatives such as Direct Preference Optimization (DPO), SLiC-HF (Zhao et al., 2023), and Rejection Sampling Optimization (RSO) (Liu et al., 2023b) offer off-policy mechanisms that further reduce computational burden. While we do not aim to develop new fine-tuning algorithms, our method integrates GRPO within the broader co-evolution framework to adapt the LLM in tandem with heuristic evolution. We specifically adopt GRPO because it requires only a scalar signal per prompt-response pair (in contrast to preference-based signals), making it suitable for our setting. Moreover, we implement fine-tuning using Unsloth (Daniel Han and team, 2023), a GPU-efficient open-source framework that enables fast, memory-light training even on single consumer-grade GPUs—making our method especially practical and accessible for researchers with limited hardware resources.

C COMPLETE ALGORITHM

The complete algorithm body is shown in Algorithm 1.

```
810
            Algorithm 1: CALM
811
            Input: LLM \pi_{\theta}, Evaluation environment g, number G of responses to be sampled for one
812
                          prompt, maximum round number T, Population size L_{\rm p}, Sampling weight w for each
813
                         operator, Hyperparameter \delta_0 and C that control the collapse mechanism, set of seed
814
                         heuristic \mathcal{H}_{\text{seed}} (set to be \emptyset if not given any seed heuristic).
815
            Initialize collapse counter t_c = -1, best heuristic h^* = \text{null}, best performance g^* = -\infty,
816
              heuristic pool \mathcal{H}_{\text{pool}} = \mathcal{H}_{\text{seed}}, w_i = \boldsymbol{w}_{\text{injection}};
817
            for t = 1, \dots, T do
818
                  Operator base OPs \leftarrow \{Initialization\};
819
                  if |\mathcal{H}_{\text{pool}}| \geq 1 then
                       OPs \leftarrow \{Injection, Replacement, Crossover, Simplification\};
820
                  end
821
                  if |\mathcal{H}_{\text{pool}}| \geq 2 then
822
                       OPs \leftarrow OPs \cup \{Crossover\};
823
                  end
824
                  if |\mathcal{H}_{\text{pool}}| < L_{\text{p}} then
825
                        \boldsymbol{w}_{\text{Injection}} \leftarrow \max(\boldsymbol{w});
                  else
827
                       \boldsymbol{w}_{\text{Injection}} \leftarrow w_i;
828
                  \mathcal{H}_{\text{base}} \leftarrow \emptyset, op \leftarrow Draw an operator from OPs with the probability proportional to w;
829
                  if op \neq Initialization then
                        h_{c,1} \leftarrow \text{Draw} an heuristic from top-L_p-performing heuristics in \mathcal{H}_{pool}, where the
830
                         sampling probability of an heuristic h is proportional to 1/\operatorname{rank}_p(h) and \operatorname{rank}_p(h) is
831
                         the heuristic's performance rank;
832
                        \mathcal{H}_{\text{base}} \leftarrow \mathcal{H}_{\text{base}} \cup \{h\};
833
                        if op = Crossover then
834
                             if random(0,1) \leq 0.5 then
835
                                   h_{c,2} \leftarrowDraw a heuristic from the population by performance rank as sampling
836
                                     h_{c,1};
837
                             else
838
                                   Calculate diversity metric
                                    \operatorname{div}(h_{c,1},h) = \frac{|\operatorname{idea\_token}(h) \setminus \operatorname{idea\_token}(h_{c,1})|}{|\operatorname{idea\_token}(h)|}, \forall h \in \mathcal{H}_{\text{pool}};
839
                                                                       |idea\_token(h)|
840
                                   h_{c,2} \leftarrowDraw a heuristic from the pool by diversity rank where the sampling
841
                                     probability is proportional to 1/\text{rank}_d(h) (a larger diversity value yields a
                                     higher probability);
843
                              \mathcal{H}_{\text{base}} \leftarrow \mathcal{H}_{\text{base}} \cup \{h_{c,2}\};
844
                        end
845
846
                  q \leftarrow Generate prompt by the operator op and base heuristics \mathcal{H}_{\text{base}};
847
                  \mathcal{O} \leftarrow \text{Sample } G \text{ responses from } \pi_{\theta} \text{ for } q;
                  \mathcal{H}_{\text{feasible}}, \hat{r}_{\mathcal{O}} \leftarrow \text{Try extracting a feasible heuristic from each response } o \in \mathcal{O} \text{ and assign}
848
                   reward to each response following Section 4.3;
849
                  \theta \leftarrow Update the LLM by GRPO that optimizes Equation 1 with (q, \mathcal{O}, \hat{r}_{\mathcal{O}});
850
                  \mathcal{H}_{\text{pool}} \leftarrow \mathcal{H}_{\text{pool}} \cup \mathcal{H}_{\{\rceil \dashv f \rangle \mid \updownarrow \rceil};
851
                  h^* \leftarrow \arg\max_{h \in \mathcal{H}_{\text{pool}}} g(h);
852
                  if g(h^*) = g^* and |\mathcal{H}_{\mathrm{pool}}| \geq L_{\mathrm{p}} then
853
                        /* If the population is full, the counter for collapse
854
                               starts.
855
                        t_c \leftarrow \max(t_c, 0) + 1;
856
857
                       g^* = (h^*), t_c \leftarrow \min(t_c, 0);
858
                  if random(0,1) \leq \delta_0 t_c or t_c \geq C then
859
                        \mathcal{H}_{\text{base}} \leftarrow \{h^*\} \cup \mathcal{H}_{\text{seed}}, t_c \leftarrow -1; /* \text{ Collapse}
                                                                                                                                                     */
860
                  end
861
            end
862
             Return:h^*
```

865

866

867 868

870

871

872

873

874

875

876

877

878

879

882

883

885

887

889

890 891 892

893 894

895

897

899

900

901

902

903

904

905 906

907

908

909

910 911

912 913 914

915 916

Searching superior heuristics on the {problem.name} problem in an evolutionary manner through conversation between User and Assistant. In this problem, {problem.description} The User provides existing algorithms and requests a new one. ## Your Task You should first present a concise conceptual description, followed by a complete code implementation. * The description must: * Be enclosed with a double brace and starts with "The idea of the algorithm is to". * Ensure it is self-contained, insightful, and creatively original. * Not reference or rely on any prior ideas or existing code. * The code must: * Strictly follow the input-output variable names and types used in the provided implementation. * Be a single Python function formatted within Python code blocks. * Exclude any usage examples. * Ensure the algorithm is deterministic. * Avoid introducing unnecessary, arbitrarily-tuned hyperparameters; any parameters used should be essential and systematically derived from the input. Overall, your response should be like: {{The idea of the algorithm is to (sepcific description here)}} python

Except for the idea and code, do not give additional explanations or comments.

Figure 3: Template of the system prompt.

D PROMPTS USED IN CALM

your code here

System Prompt. The system prompt is generated by inserting the name and description into the template shown in Figure 3. The specific prompt used for each problem can be found in Table 5.

Injection Prompt. The template used to generate injection prompts is shown in Figure 4. In the prompt template, the algorithm details are generated by the given heuristics and the prompt template in Figure 8. The description of the most recent injected components is created by (1) parsing the string wrapped within "The new component ... has been introduced", (2) globally saving the historical new components, and (3) picking the last 10 new components to be used.

Replacement Prompt. The replacement prompt is created by the template, some predefined component Paris shown in Figure 5, and the algorithm detail template shown in Figure 8.

Crossover Prompt. The crossover prompt is generated by the template shown in Figure 6 and the algorithm detail template shown in Figure 8.

Simplification Prompt. The simplification prompt is created by the template shown in Figure 7 and the algorithm detail template shown in Figure 8.

Initialization Prompt. The initialization prompt is created by the template shown in Figure 9. The algorithm template is a function signature.

E EXAMPLE PROMPT-RESPONSE PAIRS FOR INJECTION AND REPLACEMENT

The example prompt-response pairs with concrete explanation for the modification on heuristics is shown in Figure 10, 11, 12, and 13.

Inject a novel, meaningful component into the following algorithm. The component may be self-devised or inspired by ideas from other domains or problems.

{algorithm_details(given_heuristics)}

Use a concise noun phrase to describe the new component in the responded idea like "The new component ... has been introduced.". Exclude the following components that have already been explored: {description of most recent injected components}

Figure 4: Template of the injection prompt.

For the following algorithm, identify {old_component} and rewrite it to {new_component}.

{algorithm_details(given_heuristics)}

old_component	new_component
a fixed, instance-independent decision rule	an instance-dependent rule that derives its value from the current observation
a key hyper-parameter expressed as either a constant literal or a stationary variable	a more principled constant justified by theory or practice
a fragment that assigns equal or near-equal credits to multiple elements	a fragment where credits are deterministically and reasonably differentiated

Figure 5: Template of the replacement prompt.

Please generate a new algorithm that is motivated by the following algorithms but performs better on any same instance.

{algorithm_details(given_heuristics)}

Figure 6: Template of the crossover prompt.

Please create a simplified and more elegant version of an algorithm by distilling and refining the core ideas from the following:

{algorithm_details(given_heuristics)}

Figure 7: Template of the simplification prompt.

```
## Algorithm k

* Performance: {heuristic.performance} (Rank {heurisitc.rank})

* Idea: {heuristic.idea} {heuristic.unit}

* Code:

```python
{heuristic.code}

...
...
```

Figure 8: Template of algorithm details.

Be very creative and inventive. Generate an efficient algorithm following the template below:

{algorithm\_template}

Figure 9: Template of the initialization prompt.

Table 5: Information of each problem used in prompts

	OBP	CVRP	OP	TSP
Name	Online Bin Packing	Capacitated Vehicle Routing	Orienteering	Traveling Salesman
Unit	percent of the gap to the lower bound	units of travel distance	units of collected reward	length units of the tour
Description	Items arrive sequentially and must be placed immediately into bins only if they fit within the remaining capacity. The objective is to minimize the number of bins used.	A fleet of vehicles with limited carrying capacity must deliver goods to a set of geographically distributed customers with known demands, minimizing the total travel distance while ensuring that no vehicle exceeds its capacity. The goal is to design a heuristic function that estimates the desirability of moving between customers, to be used within an Ant Colony Optimization (ACO) algorithm.	An agent must visit a subset of locations, each offering a reward, within a maximum travel budget. The objective is to maximize the total collected reward while adhering to the travel constraint. The goal is to design a heuristic function that estimates the desirability of moving between locations, to be used within an Ant Colony Optimization (ACO) algorithm.	The algorithm must find a tour that visits each node exactly once and returns to the start node. The objective is to minimize the length of the tour.

```
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
 Prompt
1037
 Inject a novel, meaningful component into the following algorithm. The component may be self-
1038
 devised or inspired by ideas from other domains or problems.
1039
1040
 ## Algorithm 1
1041
 * Performance: 12.578 units of collected reward (Rank: 1)
1042
 * Idea: The idea of the algorithm is to return the ratio of the prize and the inter-node distance
1043
 * Code:
1044
           ```python
1045
           # The idea of the algorithm is to return the ratio of the prize and the inter-node distance
1046
1047
           import numpy as np
1048
1049
           def heuristics(prize: np.ndarray, distance: np.ndarray, maxlen: float) -> np.ndarray:
1050
             return prize[np.newaxis, :] / distance
1051
1052
1053
           Use a concise noun phrase to describe the new component in the responded idea like "The new
1054
           component ... has been introduced.".
1055
1056
                                                              Inject a new decay component, and the
1057
                                                              performance is improved by 4.21%!
          Response
1058
1059
           {{The idea of the algorithm is to return the ratio of the prize and the inter-node distance. The new
1060
           component incorporating a distance decay factor has been introduced.}}
1061
           ```python
1062
 import numpy as np
1063
1064
 def heuristics(prize: np.ndarray, distance: np.ndarray, maxlen: float) -> np.ndarray:
1065
 distance_decay = np.exp(-distance / maxlen)
1066
 return prize[np.newaxis, :] / distance * distance_decay
1067
```

Figure 10: Example of improvement made by injection operator.

```
1080
1081
1082
1083
1084
1086
 Prompt
1087
1088
 For the following algorithm, identify a fragment that assigns equal or near-equal credits to multiple
1089
 elements and rewrite it to a fragment where credits are deterministically and reasonably
 differentiated.
1090
1091
 ## Algorithm 1
1092
 * Performance: 13.826 units of collected reward (Rank: 1)
1093
 * Idea: The idea of the algorithm is to return the ratio of the prize and the inter-node distance, while
1094
 incorporating a distance decay factor that uses a novel hybrid decay. This hybrid decay combines
1095
 elements of the initial exponential and linear decays, starting with an initial exponential decay that
1096
 quickly reduces the influence of distant nodes, followed by a linear decay that gradually reduces
 the influence further. This hybrid approach enhances the exploration-exploitation trade-off,
1097
 providing a balance between visiting close nodes and exploring more distant ones, thereby
1098
 optimizing the balance.
1099
 * Code:
1100
                       ```python
1101
                       import numpy as np
1102
1103
                       def heuristics(prize: np.ndarray, distance: np.ndarray, maxlen: float) -> np.ndarray:
                           # Initial exponential decay followed by a linear decay
1104
                           distance_{decay} = np.exp(-distance / (0.4 * maxlen)) * (1 - 2 * (distance - 0.4 * maxlen) / (0.6 * maxlen)) / (0.6 * maxlen) / (0.6 * maxle
1105
                       maxlen))
1106
                           distance_decay[distance > 0.4 * maxlen] = 0 # Ensure linear decay only applies to close nodes
1107
                           return prize[np.newaxis, :] / distance * distance_decay
1108
1109
                                                                                                                     Treat edge pairs differently by a
1110
                                                                                                                     distance threshold, and the
1111
                                                                                                                     performance is improved by 0.39%!
                    Response
1112
                       {{The idea of the algorithm is to return the ratio of the prize and the inter-node distance, while
1113
                       incorporating a hybrid decay that enhances the exploration-exploitation trade-off.}}
1114
                       ```python
1115
 import numpy as np
1116
1117
 def heuristics(prize: np.ndarray, distance: np.ndarray, maxlen: float) -> np.ndarray:
 # Initial exponential decay followed by a linear decay
1118
 initial_exponential_decay = np.exp(-distance / (0.4 * maxlen))
1119
 linear_decay = 1 - 2 * (distance - 0.4 * maxlen) / (0.6 * maxlen)
1120
 linear_decay[distance > 0.4 * maxlen] = 1 # Ensure linear decay only applies to close nodes
1121
1122
 # Hybrid decay combining initial exponential and linear decay
1123
 hybrid_decay = initial_exponential_decay * linear_decay
1124
 return prize[np.newaxis, :] / distance * hybrid_decay
```

Figure 11: Example of improvement made by replacing an equal-credit rule operator.

1127 1128

```
1134
 Prompt
1135
 For the following algorithm, identify a key hyper-parameter expressed as either a constant literal or
1136
 a stationary variable and rewrite it to a more principled constant justified by theory or practice.
1137
 ## Algorithm 1
1138
 * Performance: 7.103 length units of the tour (Rank: 1)
1139
 * Idea: The idea of the algorithm is to select the next node to visit from the unvisited nodes by
1140
 calculating a score that considers the direct distance to the current node, the average distance to
 the unvisited nodes, the standard deviation of these distances, and the distance to the destination
1141
 node
1142
 * Code:
1143
                          ```python
                          import numpy as np
1144
1145
                          def select_next_node(current_node: int, destination_node: int, unvisited_nodes: set, distance_matrix:
1146
                          np.ndarray) -> int:
1147
                            c1, c2, c3, c4 = 0.4, 0.3, 0.2, 0.1
                            scores = \{\}
1148
1149
                            for node in unvisited_nodes:
1150
                               distances = [distance\_matrix[current\_node][node], \\ distance\_matrix[destination\_node][node]]
1151
                               average distance to current = np.mean(distances)
                               std_dev_distance_to_current = np.std(distances)
1152
1153
1154
                                 c1 * distance_matrix[current_node][node]
                                 - c2 * average_distance_to_current
1155
                                 + c3 * std_dev_distance_to_current
1156
                                 - c4 * distance_matrix[destination_node][node]
1157
1158
                               scores[node] = score
1159
                            next node = min(scores, key=scores.get)
1160
                            return next_node
1161
                                                                             Change one hyperparameter, and the
1162
                                                                             performance is improved by 8.29%!
                         Response
1163
1164
                          {{The idea of the algorithm is to select the next node to visit from the unvisited nodes by
                          calculating a score that considers the direct distance to the current node, the average distance to
1165
                          the unvisited nodes, the standard deviation of these distances, and the distance to the destination
1166
                          node.}}
1167
                          ```python
 import numpy as np
1169
1170
 def select_next_node(current_node: int, destination_node: int, unvisited_nodes: set, distance_matrix:
1171
 np.ndarray) -> int:
 c1, c2, c3, c4 = 0.5, 0.3, 0.2, 0.1
1172
 scores = \{\}
1173
1174
 for node in unvisited_nodes:
1175
 distances = [distance_matrix[current_node][node], distance_matrix[destination_node][node]]
 average distance to current = np.mean(distances)
1176
 std_dev_distance_to_current = np.std(distances)
1177
1178
 c1 * distance_matrix[current_node][node]
1179
 - c2 * average_distance_to_current
1180
 + c3 * std_dev_distance_to_current
1181
 - c4 * distance_matrix[destination_node][node]
1182
 scores[node] = score
1183
1184
 next_node = min(scores, key=scores.get)
1185
 return next_node
1186
1187
```

Figure 12: Example of improvement made by replacing a hyperparameter.

```
1188
1189
1190
 Prompt
1191
1192
 For the following algorithm, identify a fixed, instance-independent decision rule and rewrite it to an
1193
 instance-dependent rule that derives its value from the current observation.
1194
1195
 ## Algorithm 1
1196
 * Performance: 3.052 percent of the gap to the lower bound (Rank: 1)
1197
 * Idea: The idea of the algorithm is to score each available bin by integrating the remaining
1198
 capacity, item size, and a penalty for overfilled bins. This algorithm introduces a novel adaptive
1199
 scaling factor based on the geometric mean of the remaining capacity and the maximum capacity.
 It also uses a quintic penalty term to better penalize overfilled bins and a quadratic scaling to
1201
 stabilize the score for bins with very small remaining capacity.
1202
 * Code:
           ```python
1203
           import numpy as np
1205
1206
           def step(item_size: float, remaining_capacity: np.ndarray) -> np.ndarray:
1207
             max bin cap = np.max(remaining capacity)
             geometric_mean = remaining_capacity * max_bin_cap / (remaining_capacity + max_bin_cap)
1208
             scale_factor = np.sqrt(geometric_mean / max_bin_cap)
1209
             quintic_penalty = remaining_capacity**5 / (item_size**5)
1210
1211
             score = (remaining capacity - max bin cap)**2 / item size + quintic penalty
             score[remaining capacity > item size] = -score[remaining capacity > item size]
1212
             score[1:] -= score[:-1]
1213
             score *= scale factor
1214
             return score
1215
1216
1217
                                                               Replace a static threshold by the
1218
                                                               dynamic item size, and the
                                                               performance is improved by 7.79%!
          Response
1219
1220
           {{The idea of the algorithm is to score each available bin by integrating the remaining capacity,
1221
```

{{The idea of the algorithm is to score each available bin by integrating the remaining capacity, item size, and a penalty for overfilled bins. This algorithm uses a deterministic rule that derives its value from the current input instance by calculating the geometric mean of the remaining capacity and the maximum capacity as the scale factor and adjusting the score based on the item size and remaining capacity.}}
'``python import numpy as np

def step(item_size: float, remaining_capacity: np.ndarray) -> np.ndarray:
 max_bin_cap = np.max(remaining_capacity)
 geometric_mean = np.sqrt(remaining_capacity * max_bin_cap)
 scale_factor = geometric_mean / max_bin_cap
 score = (remaining_capacity - item_size) / item_size + remaining_capacity**5 / (item_size**5)
 score[remaining_capacity > item_size] = -score[remaining_capacity > item_size]
 score[1:] -= score[:-1]
 score *= scale_factor
 return score

Figure 13: Example of improvement made by replacing a instance-independent decision rule.

F DISCUSSION ABOUT THE DIVERSITY-AWARE CROSSOVER OPERATOR

Notably, Zheng et al. (2025) allowed heuristic selection beyond the top-performing population, offering greater exploration flexibility, though without explicitly modeling diversity. In contrast, Dat et al. (2025) emphasized the role of diversity in heuristic evolution but did not integrate it into crossover and operated within a fixed-size population. Therefore, CALM's crossover operator complements prior work by explicitly incorporating diversity into the crossover process.

G MORE DETAILS FOR THE COLLAPSE MECHANISM

G.1 Proof of Equation (2)

Let c_n be the stagnation counter just before collapse. Under the collapse mechanism with per-round hazard

$$p_k = k \, \delta_0, \qquad k = 1, 2, \dots$$

the probability of surviving beyond k rounds is

$$\Pr[c_n > k] = \prod_{i=1}^k (1 - i \, \delta_0),$$

which vanishes for $k \geq \lfloor 1/\delta_0 \rfloor$.

By definition,

$$\mathbb{E}[c_n] = \sum_{k=1}^{\infty} k \Pr[c_n = k].$$

Introduce the nonnegative array

$$a_{j,k} = \begin{cases} \Pr[c_n = k], & k \ge j \ge 1, \\ 0, & \text{otherwise.} \end{cases}$$

Then

$$\sum_{k=1}^{\infty} k \Pr[c_n = k] = \sum_{k=1}^{\infty} \sum_{j=1}^{k} \Pr[c_n = k] = \sum_{k=1}^{\infty} \sum_{j=1}^{\infty} a_{j,k}.$$

Since $a_{i,k} \ge 0$, Tonelli's theorem allows swapping the sums:

$$\sum_{k=1}^{\infty} \sum_{j=1}^{\infty} a_{j,k} = \sum_{j=1}^{\infty} \sum_{k=1}^{\infty} a_{j,k} = \sum_{j=1}^{\infty} \sum_{k=j}^{\infty} \Pr[c_n = k] = \sum_{j=1}^{\infty} \Pr[c_n \ge j] = \sum_{j=0}^{\infty} \Pr[c_n > j].$$

Hence the tail-sum identity

$$\mathbb{E}[c_n] = \sum_{j=0}^{\infty} \Pr[c_n > j].$$

For $\delta_0 \ll 1$ we approximate the finite product by exponentiating its logarithm, using the Maclaurin expansion

$$\ln(1-x) = -\sum_{m=1}^{\infty} \frac{x^m}{m} = -x - \frac{x^2}{2} - \dots, \quad |x| < 1,$$

with $x = i\delta_0$. Truncating at the linear term gives

$$\sum_{i=1}^{k} \ln(1 - i\,\delta_0) \approx -\sum_{i=1}^{k} i\,\delta_0 = -\frac{\delta_0}{2}\,k(k+1) \approx -\frac{\delta_0}{2}\,k^2,$$

so

$$\Pr[c_n > k] \approx \exp(-\frac{\delta_0}{2}k^2).$$

Substituting into the tail-sum and replacing the discrete sum by an integral yields

$$\mathbb{E}[c_n] \approx \sum_{k=0}^{\infty} e^{-\frac{\delta_0}{2}k^2} \approx \int_0^{\infty} e^{-\frac{\delta_0}{2}x^2} dx = \sqrt{\frac{\pi}{2\delta_0}},$$

which establishes Equation (2).

G.2 GOODNESS

Following this reset, the search effectively starts anew, but with a strategic advantage: it builds upon the best insights discovered so far. Importantly, during the early stage of repopulation, the system temporarily relaxes selection constraints. New heuristics generated via injection, replacement, or crossover are allowed into the population regardless of performance, as long as the total number of heuristics remains below the target population size. This gives structurally novel but potentially suboptimal components the opportunity to propagate and evolve—something not feasible under normal selection pressure, where only top-performing heuristics are retained and processed further.

TT 1/4

H MORE EXPERIMENTAL DETAILS

H.1 IMPLEMENTATION DETAILS

We build CALM on Unsloth (Daniel Han and team, 2023), with two modifications: raising the learning rate to 5×10^{-5} for faster adaptation and sampling G=4 responses per prompt to enable more evolutionary steps under a fixed query budget.

We set the initial collapse growth rate to $\delta_0=0.0005$ (max threshold C=25), cap training at T=500 rounds, and assign operator sampling probabilities in the ratio 1:1:2:4 for simplification, injection, modification, and crossover, respectively. Each heuristic is evaluated within 60 s (Zheng et al., 2025). All experiments ran on a 24 GB NVIDIA A30 GPU with an Intel Xeon Gold 5220R CPU.

For invalid responses, the maximum reward $r_{\rm invalid}$ is set to -0.75. Furthermore, we apply a hierarchy of failure modes, assigning progressively higher (i.e., less negative) rewards to increasingly plausible but still unacceptable outputs. These modes include: (1) omission of a required idea (reward: $r_7 = -1.0$); (2) missing code block ($r_6 = -0.95$); (3) improperly formatted function ($r_5 = -0.9$); (4) runtime errors or time budget violations ($r_4 = -0.85$); and (5) detection of randomness in the heuristic ($r_3 = -0.75$)², which incurs the mildest penalty among infeasible cases.

Under this configuration, the average running time of CALM for the OBP, CVRP, OP, and TSP is about 6.8, 7.2, 5.3, and 5.5 hours, respectively, for T=500 steps. However, it is important to note that the actual running time for a single trial may vary considerably due to the stochastic nature of the LLM and the potentially large number of heuristics generated, each requiring time-intensive evaluation.

H.2 BASELINE IMPLEMENTATIONS

The source code, training dataset, and test dataset for AlphaEvolve (Novikov et al., 2025) are not available. Therefore, we use OpenEvolve (Sharma, 2025) as the baseline, which is the most popular open-source implementation of AlphaEvolve.

In its original implementation, EvoTune (Surina et al., 2025) requires approximately 80GB of GPU memory to conduct experiments on LLMs with fewer than 7B parameters, which exceeds the computational resources available to us. By contrast, our CALM method could operate on a single GPU with 24GB of memory. To ensure a fair comparison, we re-implemented EvoTune within the same Unsloth (Daniel Han and team, 2023) framework, following its official source code, so that it can be executed on the same Owen model under identical GPU constraints.

Besides, ReEvo (Ye et al., 2024) and its follow-up approach HSEvo (Dat et al., 2025) can stop at a very early stage in evolution as found by Zheng et al. (2025). Thus, the results of them on TSP are not reported. For the OP and CVRP tasks, OpenEvolve (Sharma, 2025) failed to discover improved heuristics beyond the early stages of heuristics search. As a result, its training curve is omitted from Figure 2 for clarity.

²Randomized heuristics are excluded in the experiments because their stochastic behavior substantially increases evaluation cost and noise. To enforce determinism, CALM penalizes responses that invoke randomness (e.g., usage of random, np.random, etc.). The framework could support randomized heuristics by relaxing this constraint, though evaluation overhead would increase.

Table 6: Breakdown of time consumption in CALM (with detailed wall-clock time).

	Inference	Evaluation	Training
CVRP	73.29% (4.059 h)	16.11% (0.893 h)	10.60% (0.587 h)
OBP	78.59% (5.749 h)	11.10% (0.812 h)	10.31% (0.754 h)
OP	82.12% (4.281 h)	9.77% (0.510 h)	8.11% (0.423 h)
TSP	83.23% (6.457 h)	7.20% (0.559 h)	9.57% (0.742 h)

H.3 DESCRIPTION OF PROBLEMS IN EXPERIMENTS

Online Bin Packing (OBP). A sequence of items of varying sizes arrives one by one. Each bin has a fixed capacity. Upon arrival of an item, the algorithm must immediately assign it to an existing bin that has enough remaining space or open a new bin. The goal is to minimize the total number of bins used. The input of the heuristic is the size of the current item and the remaining capacities of the bins. The output of the heuristic is the priority score of each observed bin, where the feasible bin with the highest score will be selected to accommodate the item.

Traveling Salesman Problem (TSP) under Step-by-Step Construction. Given a set of locations with pairwise travel distances, the objective is to construct a tour that starts at one location, visits each other location exactly once, and returns to the start. At each step the heuristic must choose the next unvisited location based solely on the information gathered so far. The aim is to keep the total travel distance as small as possible.

Capacitated Vehicle Routing Problem (CVRP) under ACO. A fleet of vehicles with identical load capacity must serve a set of customers, each with a known demand, and all vehicles start and end at a central depot. Under the Ant Colony Optimization framework, many artificial "ants" build routes by moving from customer to customer. Each ant's choice of next customer is guided by a combination of pheromone trails—updated based on previous high-quality solutions—and heuristic scores provided by the LLM. The goal is to serve all customers while minimizing the total distance traveled and respecting vehicle capacity limits.

Orienteering Problem (OP) under ACO. Starting from a given location (and possibly ending at the same or another specified location), an agent may visit a subset of available sites, each offering a reward, subject to an overall travel budget. Within the ACO framework, ants construct candidate paths by choosing which site to visit next based on pheromone levels and LLM-generated heuristic scores that estimate the benefit of each edge under the reward-and-budget trade-off. The aim is to collect as much reward as possible without exceeding the travel budget.

I More Experimental Results

I.1 Breakdown of CALM's Time Consumption

We break down the total CALM running time into three components: (i) Inference—the time taken by the LLM to generate responses³; (ii) Evaluation—the time spent parsing each heuristic and validating its feasibility and performance; and (iii) Training—the time required to compute the loss and update the LLM parameters. The time spent on each component across different tasks is summarized in Table 6.

These results show that inference is the dominant time cost in CALM. Despite parallelizing heuristic evaluations across the training dataset, evaluation still requires more time than model training for most tasks. In other words, employing the fine-tuning algorithm in the LLM-based AHD introduces a minimal time overhead.

³We additionally quantified the potential overhead of gradient computation during inference, as training in CALM includes rollouts. Since gradient computation is tightly integrated into PyTorch, we compared two runs: (i) inference with gradients enabled and (ii) inference with torch.no_grad() to disable gradient computation. Across all tasks, the extra cost was consistently below 0.25% of the pure inference time, which is negligible and does not affect the breakdown reported in Table 6.

Table 7: Average scores of fine-tuned models. Lower is better for CVRP (\downarrow) and higher is better for OP (\uparrow).

	Step=0	Step=100	Step=200	Step=300	Step=400	Step=500
$CVRP(\downarrow)$ $OP(\uparrow)$	66.356	32.403	40.860	40.451	49.699	32.403
	11.956	25.025	25.025	25.025	12.228	25.025

Table 8: Feasibility ratio of fine-tuned models.

	Step=0	Step=100	Step=200	Step=300	Step=400	Step=500
CVRP	10.00%	100.00%	83.33%	62.50%	71.43%	100.00%
OP	26.32%	100.00%	100.00%	100.00%	45.45%	100.00%

I.2 IMPACT OF FINE-TUNING ON THE FOUNDATIONAL MODEL

To investigate the performance of the fine-tuned model, we conducted additional experiments. Specifically, we saved a snapshot of the LLM every 100 training steps during the evolutionary process of CALM. For each snapshot, we used the same prompt, which instructs the LLM to generate an improved variant of the seed algorithm for a given task. For each snapshot and task prompt, we repeatedly sampled responses until five feasible outputs capable of producing valid heuristic code for the task were obtained. We then recorded the following metrics: (i) the average score of the five heuristics, where the score for each heuristic is calculated as the performance averaged over the test scales reported in our manuscript; and (ii) the feasibility ratio, defined as the number of feasible responses (fixed at five) divided by the total number of samples required to obtain them. We focused on the snapshots generated during the run for CVRP that yielded the best heuristic among all three runs. For these LLM snapshots, we evaluated them on both CVRP and OP. Results are shown in Table 7 and 8.

Key observations are as follows:

- Both the average score of the discovered heuristics and the feasibility ratio of the responses improve significantly after fine-tuning. For example, in the CVRP task, the feasibility ratio increases from 10% to 100%, and the average score decreases by more than 50% after fine-tuning. This demonstrates the effectiveness of CALM in enhancing the LLM's capability.
- Although the LLM is fine-tuned on data generated from AHD for the CVRP task, substantial
 improvements in both the average score and feasibility ratio are also observed for the OP
 task. This suggests that the improvements gained through fine-tuning on CALM-generated
 data generalize beyond the in-domain task and can benefit other related tasks.
- Beginning at step 200, both the average score and feasibility ratio fluctuate during the training process. Notably, the heuristics achieving the best scores (20.088 for CVRP and 25.252 for OP) were discovered by LLM snapshots saved at 300 and 400 training steps, respectively. Interestingly, these snapshots also exhibit the lowest feasibility ratios and non-leading average scores for the respective tasks. This indicates that an LLM capable of producing an exceptional heuristic may not be the most stable in generating feasible responses or in producing consistently high-quality heuristics on average. In other words, an LLM capable of occasional breakthroughs may exhibit erratic behavior—illustrating the notion that genius can verge on madness.

I.3 IMPACT OF THE FOUNDATIONAL LLM

We have added additional experiments by replacing the foundational model with (i) a SOTA reasoning LLM o4-mini and (ii) another open-source compact model Llama-3.1-8B-Instruct-Int4. Results are as shown in Table 9 and 10.

The SOTA reasoning LLM o4-mini effectively identifies superior heuristics under the CALM framework (w/o GRPO), achieving notable performance improvements—approximately 15.5% with the

Table 9: Optimality gaps on OBP (Qwen, Llama, and o4-mini) with CALM.

	1k_100	1k_500	5k_100	5k_500	10k_100	10k_500	Avg.
Qwen+GRPO	2.55%	0.00%	0.85%	0.17%	0.56%	0.14%	0.71%
Llama+GRPO	2.98%	0.00%	0.96%	0.10%	0.54%	0.10%	0.78%
o4-mini	2.29%	0.00%	0.85%	0.10%	0.34%	0.02%	0.60%

Table 10: Objective scores and optimality gaps on OP (Qwen & Llama) with CALM.

	N=50	N=100	N=200
Qwen	15.054 (24.22%)	30.778 (15.43%)	55.406 (12.58%)
Llama	15.038 (24.31%)	30.599 (15.92%)	54.593 (13.86%)

o4-mini model—though this advantage comes with over twice the inference time compared to Qwen+GRPO. Despite this trade-off, using locally deployed, compact models remains competitive, particularly when the time budget for search is limited. Additionally, heuristics identified by the Llama model show strong performance and generalizability, outperforming all other methods at certain scales (5k_500 and 10k_500) in OBP and surpassing all baseline methods in OP at N=100 and 200, while maintaining comparable results at all scales. Moreover, removing GRPO significantly reduces average optimality gaps, by 34.33% in OBP and 17.91% in OP, further highlighting the robustness of the proposed method.

I.4 More Results on Harder OBP Instances

Smaller-scale instances of OBP are more challenging in the online setting, as each decision has a larger impact and variance is higher, making them a stricter robustness test for heuristics. Thus, we further evaluated CALM's performance on OBP with smaller problem scales. Specifically, we generated 10 Weibull-distributed instances for each of the following training scales (in the format n_capacity): 100_100, 100_500, 300_100, 300_500, 500_100, and 500_500. For evaluation, 50 instances were generated for each scale. CALM was equipped with Llama-3.1-8B-Instruct-INT4. For comparison, we included MCTS-AHD, the SOTA LLM-based AHD method that achieved the best performance on the smallest scale in Table 9.

The results show that CALM+Llama achieves a lower average optimality gap than MCTS-AHD, even when the latter is paired with a more powerful LLM. CALM underperforms only at the 500_500 scale. In addition, the standard deviation of the average optimality gap is smaller for CALM (0.03%) compared to MCTS-AHD (0.21%).

I.5 SCALING BEHAVIOR

We conducted additional experiments to evaluate the scaling behavior of CALM on OP, using an increased training budget of 2500 steps. Results are shown in Table 12. The key findings are as follows:

- With a substantially larger evaluation budget, CALM is able to discover heuristics that outperform those found with only 500 training steps, as shown in the table below.
- Without LLM fine-tuning, CALM is unable to consistently discover new, superior heuristics at early stages of training. In one instance, no better heuristic was found beyond step 256.
- Evaluation of fine-tuned model snapshots at different training steps shows that after several
 hundred steps, performance fluctuates and does not always improve monotonically. Nevertheless, the fine-tuned models consistently outperform the untuned baseline. This suggests
 that, in later stages, fine-tuning may not significantly enhance the LLM's capabilities but
 instead introduces variation to the LLM for heuristic generation. This variation may help
 maintain the LLM's performance while increasing the diversity of the heuristic population.

Table 11: Optimality gaps on OBP with smaller scales.

	100_100	100_500	300_100	300_500	500_100	500_500	Avg.
MCTS-AHD (GPT-4o-mini)	6.97%	1.39%	5.67%	0.57%	5.20%	0.63%	3.40%
CALM (Llama-3.1-8B-Instruct-INT4)	6.80%	1.39%	5.61%	0.57%	5.06%	0.64%	3.35%

Table 12: Objective scores and optimality gaps of CALM (w/ GRPO) on OP under different search budgets.

	N=50	N=100	N=200
#LLM Queries=2,000	15.054 (24.22%)	30.778 (15.43%)	55.406 (12.58%)
#LLM Queries=10,000	15.201 (23.49%)	31.153 (14.40%)	56.432 (10.96%)

Overall, these results indicate that CALM exhibits favorable scaling behavior under larger training budgets.

I.6 P-VALUES FOR SIGNIFICANCE

To further highlight the superiority of CALM, we compare its performance against the state-of-the-art LLM-based AHD method MCTS-AHD (Zheng et al., 2025) across ten independent runs on two representative tasks: the TSP (step-by-step construction) and the OP (ACO-based). For fairness, we adopt the exact same dataset as used by Zheng et al. (2025). The per-run performance of MCTS-AHD on the TSP task is directly obtained from their appendix, while for the OP task we obtain the results using the official implementation under the same evaluation environment as CALM. The results are summarized in Tab. 13. The small p-values further confirm with high confidence that CALM consistently outperforms MCTS-AHD on both tasks.

I.7 SENSITIVITY TO THE HYPERPARAMETERS IN THE REWARD FUNCTION

Our reward design is guided by a fundamental principle: rewards should increase progressively with the quality of the generated response. This principle is illustrated by the high-level cases in Figure 1. To differentiate between these cases, we introduce several hyperparameters (e.g., α_1 , α_2 , and $r_{\rm invalid}$). To evaluate the robustness of this principle, we conducted an additional ablation study where CALM's reward function was instantiated under the following settings:

 The original implementation with a relatively even reward distribution, as described in Section H.1.

• Random sampling of all hyperparameters under the progressive-guiding constraint, i.e., $1>\alpha_1>\alpha_2>0>r_3=r_{\rm invalid}>r_4>r_5>r_6>-1$.

• The same α_1 , α_2 , and r_{invalid} values as in Section H.1, but with all invalid responses uniformly assigned r_{invalid} .

The results are presented in Table 14.

Across all settings, heuristics derived from these reward designs remain highly competitive at every scale. In particular, when invalid responses are assigned a unified reward, CALM achieves the best performance at N=50, surpassing all methods reported in Table 3. The slight performance gap between CALM with randomly sampled hyperparameters and the other two implementations likely results from uneven reward spacing across neighboring cases. Overall, these findings indicate that CALM's effectiveness is not tied to precise numerical values in reward shaping but instead depends on adherence to the underlying principle of progressive reward allocation.

I.8 GENERATED HEURISITCS

Table 13: Performance comparison of CALM and MCTS-AHD on TSP (step-by-step construction) and OP (ACO-based) tasks over ten runs. "avg." represents the average and "std." the standard deviation. p-values are calculated using single-tailed t-tests.

Methods	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10	avg.	std.	p-value
	$TSP(\downarrow)$												
MCTS-AHD (GPT-4o-mini)	6.452	6.447	6.284	6.386	6.316	6.372	6.480	6.480	6.259	6.388	6.386	0.080	
CALM (Qwen2.5-7B)	6.220	6.217	6.213	6.205	6.221	6.174	6.213	6.219	6.224	6.222	6.213	0.015	0.0000012315
	OP(↑)												
MCTS-AHD (GPT-4o-mini)	14.668	14.910	14.786	14.602	14.738	14.724	14.642	14.826	14.722	14.690	14.731	0.091	
CALM (Qwen2.5-7B)	14.876	14.822	14.951	14.798	14.844	14.669	14.850	14.878	14.880	14.746	14.831	0.079	0.00831786

Table 14: Ablation study of reward parameter choices under CALM's reward design on the OP task.

Settings	N=50	N=100	N=200
Original reward configuration (Sec. H.1)	15.054 (24.22%)	30.780 (15.43%)	55.406 (12.58%)
Original reward configuration (Sec. H.1) but the same reward for all invalid responses	15.059(24.20%)	30.744(12.52%)	55.341(12.68%)
Randomly sampled reward parameters under progressive-guiding constraint	15.046 (24.26%)	30.613 (15.88%)	55.165 (12.96%)

```
1591
      The idea of the algorithm is to refine the scoring mechanism by
1592
          introducing logarithmic adjustments and a novel scoring component
1593
          that captures the logarithmic relationship between the remaining
          capacity and the square of the item size, and an adjusted
1594
          logarithmic density term that provides a more refined scoring
1595
          mechanism. This new algorithm aims to enhance the accuracy of bin
1596
          suitability assessment by adding a component that adjusts the score
1597
          based on the logarithmic difference between the remaining capacity
1598
          and the maximum bin capacity. The algorithm also simplifies the
          scoring steps to make it more elegant and efficient.
1599
1600
1601
      import numpy as np
1602
1603
      def step(item_size: float, remaining_capacity: np.ndarray) -> np.ndarray:
1604
          max_bin_cap = max(remaining_capacity)
          bin_density = np.sum(remaining_capacity) / (item_size *
1605
              len(remaining_capacity))
1606
          log_adj = np.log(remaining_capacity + 1) / np.log(max_bin_cap + 1)
1607
           score = (remaining_capacity - max_bin_cap)**2 / item_size +
              remaining_capacity**2 / (item_size**2) + remaining_capacity**2 /
              (item_size**3) + bin_density * remaining_capacity
1609
1610
          score[remaining_capacity > item_size] = -score[remaining_capacity >
1611
              item_size]
1612
          score[1:] -= score[:-1]
1613
1614
          score *= log_adj
          score += log_adj * remaining_capacity
1615
1616
          score *= log_adj
1617
          new_component = remaining_capacity / (item_size - remaining_capacity
1618
              + 1)
1619
          score += new_component
```

```
1620
           new_component = remaining_capacity * np.log(remaining_capacity + 1)
1621
               / (item_size * np.log(max_bin_cap + 1)) * (1 -
1622
              remaining_capacity / item_size)
1623
           score += new_component
1624
           new_adjustment = (remaining_capacity / item_size) * log_adj
1625
           score += new_adjustment
1626
1627
           remaining_capacity_adjusted = remaining_capacity / item_size
           score += np.log(remaining_capacity_adjusted + 1) /
              np.log(max_bin_cap + 1)
1629
1630
           new_component = (remaining_capacity - 1) / (item_size -
1631
              remaining_capacity + 1) * log_adj / np.log(max_bin_cap + 1)
1632
           score += new_component
1633
           new_component = log_adj * remaining_capacity / (item_size -
1634
              remaining_capacity)
1635
           score += new_component
1636
1637
          new_component = remaining_capacity * np.log(remaining_capacity + 1)
1638
              / (item_size**2) * (1 - remaining_capacity / item_size)
           score += new_component
1639
1640
          return score
1641
```

Heuristic 1: OBP, by CALM (local, w/ GRPO)

```
1644
1645
      The idea of the algorithm is to introduce the "Bin Utilization
1646
          Diminution" component, which assesses the degree of bin usage
1647
          throughout the sequence of placements and introduces a diminishing
1648
          incentive for overpopulating any particular bin beyond a certain
          threshold. This encourages a more even distribution of item
1649
          placements across all bins, thereby reducing the risk of reaching
1650
          capacity too quickly in any single bin, helping to extend the
1651
          lifespan and utility of each bin in the packing process. By
1652
          dynamically adjusting the fit score to favor items that contribute
1653
          to a balanced utilization, the algorithm aims to enhance overall bin
          efficiency and minimize the total bin count.
1654
1655
1656
      import numpy as np
1657
1658
      def step(item_size: float, remaining_capacity: np.ndarray) -> np.ndarray:
          avg_item_size = np.mean(item_size) if item_size > 0 else 1.0
1659
          adaptive_factor = avg_item_size / np.maximum(remaining_capacity,
1660
              1e-10)
1661
           fit_score = np.maximum(remaining_capacity - item_size, 0) /
               (remaining_capacity + 1e-10)
1663
           fit_score[remaining_capacity < item_size] = -np.inf
1664
1665
           sustainability_score = (remaining_capacity - avg_item_size) ** 2
1666
           sustainability_score[remaining_capacity < item_size] = np.inf
1667
1668
          historical_fit_scores = np.cumsum(fit_score)
          normalized_historical_fit_scores = historical_fit_scores /
1669
              (np.max(historical_fit_scores) + 1e-10)
1670
1671
          combined_scores = (
1672
               (0.5 * fit_score * adaptive_factor) +
1673
               (0.3 / (sustainability_score + 1e-10))
               (0.2 * normalized_historical_fit_scores)
```

```
1674
1675
1676
          differentiation_factor = 1 / (1 + np.arange(len(remaining_capacity))
1677
          combined scores *= differentiation factor
1678
1679
           cumulative_fit_impact = np.cumsum(fit_score) / (np.arange(1,
1680
              len(remaining_capacity) + 1) + 1)
1681
           cumulative_fit_adjustment = np.maximum(fit_score -
1682
              cumulative_fit_impact, 0)
1683
          combined_scores += 0.4 * cumulative_fit_adjustment
1684
1685
           temporal_utilization_metric = np.arange(len(remaining_capacity)) /
1686
               (np.maximum(remaining_capacity, 1e-10) + 1e-10)
          combined_scores *= (1 + temporal_utilization_metric)
1687
1688
           sequential_elasticity = np.exp(-np.arange(len(remaining_capacity)) /
1689
               (np.mean(np.maximum(remaining_capacity, 1e-10)) + 1e-10))
1690
          combined_scores *= sequential_elasticity
1691
1692
           size_factor = 1 + (item_size / (np.sum(item_size) + 1e-10))
1693
           # New Component: Bin Utilization Diminution
1694
          overutilization_penalty = np.maximum(0, np.cumsum(item_size) /
1695
              (np.maximum(np.cumsum(remaining_capacity), 1e-10) + 1e-10) - 1)
1696
           combined_scores -= 0.3 * overutilization_penalty # Encourage even
              distribution across bins
1697
1698
           # Eventual Capacity Influence
1699
          eventual_capacity_score = np.log(np.maximum(np.arange(1,
1700
              len(remaining_capacity) + 1), 1)) /
1701
              (np.maximum(remaining_capacity, 1e-10) + 1e-10)
          combined_scores -= 0.3 * eventual_capacity_score # Penalize bins
1702
              that don't contribute to optimal utilization
1703
1704
          distinct_scores = combined_scores * size_factor
1705
1706
          return distinct_scores
1707
```

Heuristic 2: OBP, by CALM (API, w/o GRPO)

```
1710
      ....
1711
      The idea of the algorithm is to further refine the savings potential
1712
          calculation by emphasizing a more adaptive balance factor that is
          influenced by the current instance's capacity utilization and the
1713
          diversity of capacity usage across the routing problem. By
1714
          leveraging a more sophisticated adaptive balance factor and reducing
1715
          the complexity of the penalty factor, we ensure that nodes that are
1716
          too close to each other are penalized appropriately without overly
          compounding the impact. This simplified yet adaptive approach allows
1717
          for a nuanced exploration of the solution space, enhancing the ACO
1718
          algorithm's ability to converge to high-quality solutions while
1719
          maintaining a balance between exploration and exploitation.
1720
          Additionally, we introduce a clustering-based adjustment factor that
1721
          captures the overall network connectivity and adjusts the savings
1722
          potential accordingly, leading to more robust and flexible routing
          plans.
1723
      ....
1724
1725
      import numpy as np
1726
1727
      def advanced_heuristics_v7(distance_matrix: np.ndarray, coordinates:
         np.ndarray, demands: np.ndarray, capacity: int) -> np.ndarray:
```

```
1728
           capacity_prob = demands / capacity
1729
          distance_reciprocal = 1 / distance_matrix
1730
          proximity_factor = np.linalg.norm(coordinates[:, np.newaxis, :] -
1731
              coordinates[np.newaxis, :, :], axis=2)
          proximity_factor /= np.max(proximity_factor) # Normalize between 0
1732
              and 1
1733
          proximity_factor = 1 - proximity_factor # Invert for higher penalty
1734
              as proximity increases
1735
           remaining_demands = capacity - demands
1736
           future_savings = (remaining_demands[:, np.newaxis] *
1737
              remaining_demands) / (distance_matrix * (remaining_demands[:,
1738
              np.newaxis] + remaining_demands))
1739
          capacity_ratio = remaining_demands / capacity
1740
          proximity_savings = proximity_factor * capacity_ratio
1741
           # Cluster-based proximity adaptive savings potential
1742
          cluster_savings = np.zeros_like(distance_matrix)
1743
          cluster_distance = np.sum(distance_matrix, axis=1) /
1744
              np.linalg.norm(capacity_prob - 1, ord=1)
1745
          cluster_adj_factor = (remaining_demands[:, np.newaxis] *
              remaining_demands * cluster_distance ** 3.5) / (distance_matrix
1746
              * (remaining_demands[:, np.newaxis] + remaining_demands))
1747
1748
           # Adaptive balance factor adjusted based on remaining capacity and
1749
              cluster adjustment
1750
          balance_factor = np.min([1, 0.975 + 0.05 * capacity_prob.mean() +
              0.03 * cluster_adj_factor.mean() + 0.005 *
1751
              np.var(capacity_prob)])
1752
1753
           # Penalty factor that heavily penalizes nodes that are too close to
1754
              each other, focusing on the proximity to the next node
1755
          penalty_factor = proximity_factor ** 3
1756
           # Combine all components
1757
          probability = distance_reciprocal * capacity_prob * proximity_factor
1758
              * future_savings * proximity_savings * cluster_adj_factor * (1 -
1759
              balance_factor + proximity_savings * balance_factor) * (1 -
1760
              penalty_factor) * (1 + cluster_adj_factor * 0.6)
1761
          return probability
1762
```

Heuristic 3: CVRP, by CALM(local, w/ GRPO)

```
1765
1766
      The idea of the algorithm is to refine the credit allocation process in
          the vehicle routing problem by implementing a deterministic
1767
          weighting mechanism that assigns distinct credits to customers based
1768
          on their delivery demands, individual distance factors, and their
1769
          influence on overall routing efficiency, thus ensuring that credits
1770
          reflect meaningful differences without redundancy.
1771
1772
      import numpy as np
1773
      from sklearn.cluster import DBSCAN
1774
1775
      def heuristics(distance_matrix: np.ndarray, coordinates: np.ndarray,
1776
          demands: np.ndarray, capacity: int) -> np.ndarray:
          num_customers = demands.shape[0]
1777
          cumulative_penalty = np.zeros(num_customers)
1778
1779
           # Calculate baseline scores from demand to distance with added
1780
              urgency weighting
           urgency_weight = np.linspace(1, 1.5, num_customers)
1781
          base_score = (demands * urgency_weight) / (distance_matrix + 1e-5)
```

```
1782
          base_score[np.isnan(base_score)] = 0
1783
1784
           # Set penalties for exceeding capacity based on cumulative demands
1785
           for i in range(num customers):
               current_demand = demands[i]
1786
               cumulative_penalty[i] = max(0, current_demand - capacity)
1787
1788
           # Normalize distances to emphasize closer customers to refine scoring
1789
           normalized_distance_score = 1 / (np.clip(distance_matrix, 1e-5,
1790
              None) ** 2.5)
1791
           # Calculate effective capacity utilization adjustment
1792
           effective_capacity_utilization = np.clip((capacity - demands) /
1793
              capacity, 0, 1)
1794
           # Historical performance adjustments
1795
          historical_performance_factor = np.zeros(num_customers)
1796
           for i in range(num_customers):
1797
               historical_performance_factor[i] = np.mean([base_score[j] for j
1798
                   in range(num_customers) if distance_matrix[i][j] < 10 and j</pre>
1799
                   ! = i1)
1800
           # Spatial clustering mechanism
1801
           clustering_model = DBSCAN(eps=5, min_samples=2).fit(coordinates)
1802
           labels = clustering_model.labels_
1803
           cluster_scores = np.zeros(num_customers)
1804
           # Calculate cluster-based scores with deterministic differentiation
1805
           for cluster_id in set(labels):
1806
               if cluster_id != -1: # Ignore noise points
                   cluster_indices = np.where(labels == cluster_id)[0]
1808
                   total_demand = demands[cluster_indices].sum()
1809
                   for idx in cluster_indices:
                       # Implement differentiated scoring based on demand,
1810
                           ensuring non-equal credits
1811
                       cluster_demand_factor = (demands[idx] / total_demand) if
1812
                           total_demand > 0 else 0
1813
                       distance_weight = 1 / (1 + distance_matrix[idx].min())
1814
                           # Closer customers get more weight
                       cluster_scores[idx] = cluster_demand_factor *
1815
                           distance_weight # Mix demand and distance
1816
1817
           # New resilience score based on historical demand variability
1818
           demand_variability = np.std(demands)
1819
           resilience_score = 1 / (1 + demand_variability)
1820
           # Compose final scores combining all elements including the new
1821
              resilience score
1822
           final_scores = base_score * normalized_distance_score *
1823
              effective_capacity_utilization * (1 +
              historical_performance_factor + cluster_scores) *
              resilience_score
1825
1826
           return final_scores
1827
```

Heuristic 4: CVRP, by CALM (API, w/o GRPO)

1829 1830 1831

1832

1833

1834

```
The idea of the algorithm is to refine the exploration-expemy exploitation trade-off by introducing a sinusoidal decay that incorporates a sinusoidal penalty with a sinusoidal smoothness adjustment. This adjustment helps to smooth the preference for both recent and distant nodes, leading to a more balanced and improved performance.
```

```
1836
1837
1838
      import numpy as np
1839
      def enhanced_heuristics(prize: np.ndarray, distance: np.ndarray, maxlen:
1840
          float) -> np.ndarray:
1841
           # Exponential decay for immediate high Subscription nodes
1842
           exp_ratio = np.exp(prize[np.newaxis, :] / distance - maxlen)
1843
1844
           # Logarithmic scaling for exploration
           log_ratio = np.log(prize[np.newaxis, :] + 1) / distance
1845
1846
           # Sinusoidal decay for recent nodes with a sinusoidal smoothness
1847
              adiustment
1848
           sinusoidal_penalty = 0.5 * (1 + np.sin(np.pi * distance / (maxlen +
              1))) * (distance / maxlen) * maxlen
1849
1850
           # Combined ratio
1851
           combined_ratio = exp_ratio * log_ratio * (1 - sinusoidal_penalty)
1852
1853
           # Ensure the ratio is non-negative
1854
           combined_ratio[combined_ratio < 0] = 0</pre>
1855
           return combined_ratio
1856
```

Heuristic 5: OP, by CALM (local, w/ GRPO)

```
1859
       . . . .
1860
      The idea of the algorithm is to introduce a novel component called
1861
          "reward fluctuation sensitivity" which adjusts the desirability of
1862
          each location based on the variability of rewards over time. This
1863
          component accounts for the possibility that rewards may change or
1864
          fluctuate due to external factors, thereby allowing the agent to
          prioritize locations not only by their current rewards but also by
1865
          the potential volatility of those rewards. This sensitivity is
1866
          integrated into the existing framework, allowing for a more dynamic
1867
          response to the changing landscape of rewards, ultimately enhancing
1868
          the agent\'s decision-making process and route optimization.
1869
1870
       import numpy as np
1871
1872
      def heuristics (prize: np.ndarray, distance: np.ndarray, maxlen: float)
1873
          -> np.ndarray:
1874
           adjusted_distance = distance + 1e-10 # Avoid division by zero
          potential_reward = np.zeros_like(prize)
1875
1876
           for i in range(len(prize)):
1877
               reachable_indices = np.where(distance[i] <= maxlen)[0]</pre>
1878
               potential_reward[i] = np.sum(prize[reachable_indices]) if
                   reachable_indices.size > 0 else 0
1879
1880
           reward_hist_factor = potential_reward / (1 + np.sum(prize)) # Shape
1881
              reward based on historical performance
1882
           reward_decay = np.exp(-adjusted_distance / maxlen) # Decay effect
1883
              for distant rewards
1884
          proximity_factor = (maxlen - adjusted_distance) ** 4 # Further
1885
              enhance proximity impact with quartic distance
1886
          proximity_factor[proximity_factor < 0] = 0</pre>
1887
1888
           tiered_adjustment = (prize / (adjusted_distance + 1e-10)) ** 2
1889
              Classify rewards into categories for tiering
```

```
1890
           # Reward volatility assessment component
1891
           volatility_factor = np.zeros_like(prize)
1892
           for i in range(len(prize)):
1893
               historical_rewards = prize[np.where(distance[i] <= maxlen)[0]]</pre>
               if historical_rewards.size > 1:
1894
                   volatility_factor[i] = np.std(historical_rewards) /
1895
                       np.mean(historical_rewards) # Coefficient of variation
1896
1897
           # Risk-reward analysis component
1898
           variability_factor = np.zeros_like(prize)
           for i in range(len(prize)):
1899
               historical_rewards = prize[np.where(distance[i] <= maxlen)[0]]</pre>
1900
               if historical_rewards.size:
1901
                   variability_factor[i] = np.mean(historical_rewards) -
1902
                       np.std(historical_rewards) # Basic differentiation
1903
           final_heuristic = (reward_hist_factor * reward_decay *
1904
              proximity_factor *
1905
                               tiered_adjustment) / (1 + volatility_factor +
1906
                                   variability_factor + 1e-10)
1907
           return final heuristic
1908
```

Heuristic 6: OP, by CALM (API, w/o GRPO)

```
1910
1911
1912
      The idea of the algorithm is to select the next node by optimizing a
1913
          heuristic that considers the distance to the current node, the
          average distance to unvisited nodes, the variance of distances to
1914
          the current node from the unvisited nodes, the entropy of distances
1915
          to the destination node from each of the unvisited nodes, the
1916
          average distance from the destination node to each of the unvisited
1917
          nodes, the current node's distance to the destination node, and the
1918
          standard deviation of the overall tour distances. This proposed
          algorithm aims to introduce a new term that captures the deviation
1919
          of the current node from the average tour length and balances it
1920
          with the entropy term to reduce the overall tour length.
1921
          Additionally, this method assigns more weight to the standard
1922
          deviation of the distances from the destination node to each of the
1923
          unvisited nodes, which helps in reducing the variability of
          distances and thus leading to more consistent and shorter tour
1924
          lengths.
1925
      ....
1926
1927
      import numpy as np
1928
      def select_next_node(current_node: int, destination_node: int,
1929
          unvisited_nodes: set, distance_matrix: np.ndarray) -> int:
1930
          scores = {}
1931
1932
          for node in unvisited_nodes:
              all_distances = [distance_matrix[node][i] for i in
1933
                   unvisited_nodes if i != node]
1934
               average_distance = np.mean(all_distances)
1935
               standard_deviation = np.std(all_distances)
1936
               variance_of_distances = np.var([distance_matrix[current_node][i]
1937
                   for i in unvisited_nodes if i != node])
1938
               entropy_of_distances =
                   -np.sum(np.log2([distance_matrix[destination_node][i] for i
1939
                   in unvisited_nodes if i != node]) / len(unvisited_nodes))
1940
               average_distance_to_destination =
1941
                  np.mean([distance_matrix[destination_node][i] for i in
1942
                   unvisited_nodes if i != node])
1943
               score = (
```

```
1944
                   0.6 * distance_matrix[current_node][node]
1945
                   - 0.4 * average_distance
1946
                   + 0.3 * standard_deviation
1947
                   - 0.2 * entropy_of_distances
                   - 0.1 * distance_matrix[destination_node][node]
1948
                   - 0.08 * variance_of_distances
1949
                   - 0.05 * average_distance_to_destination
1950
                   - 0.01 * (np.mean([distance_matrix[current_node][i] for i in
1951
                       unvisited_nodes]) - average_distance)
1952
                   - 0.005 * entropy_of_distances
                   - 0.008 * distance_matrix[current_node][node] *
1953
                       distance_matrix[node][destination_node]
1954
                   - 0.006 * standard_deviation *
1955
                       distance_matrix[node] [destination_node]
1956
               )
               scores[node] = score
1957
1958
           next_node = min(scores, key=scores.get)
1959
           return next_node
1960
```

Heuristic 7: TSP, by CALM (local, w/ GRPO)

```
1963
1964
      The idea of the algorithm is to select the next node to visit from the
1965
          unvisited nodes, incorporating a novel component of dynamic path
1966
          optimization feedback. The new component analyzes previous decision
1967
          points in the tour to determine the effectiveness of the routes
          taken, adjusting future node selection to favor pathways that have
1968
          historically resulted in lower overall traversal costs. This method
1969
          not only enhances the algorithm's ability to learn from its own
1970
          experiences but also promotes the selection of routes that align
1971
          with optimal connectivity patterns established during the tour.
1972
1973
      import numpy as np
1974
1975
      def select_next_node(current_node: int, destination_node: int,
1976
          unvisited_nodes: set, distance_matrix: np.ndarray) -> int:
1977
          threshold = 0.7
          c1, c2, c3, c4, c5 = 0.4, 0.3, 0.2, 0.1, 0.1
1978
          scores = {}
1979
1980
          for node in unvisited_nodes:
1981
               all_distances = [distance_matrix[node][i] for i in
1982
                   unvisited_nodes if i != node]
               average_distance_to_unvisited = np.mean(all_distances)
1983
               std_dev_distance_to_unvisited = np.std(all_distances)
1984
1985
               # New component: consider dynamic path optimization feedback
               feedback_paths = [distance_matrix[i][node] for i in
                   range(len(distance_matrix)) if i not in unvisited_nodes and
1987
                   distance_matrix[current_node][i] < threshold]</pre>
1988
               average_feedback_distance = np.mean(feedback_paths) if
1989
                   feedback_paths else 0
1990
1991
               score = (
                   c1 * distance_matrix[current_node][node]
1992
                   - c2 * average_distance_to_unvisited
1993
                   + c3 * std_dev_distance_to_unvisited
1994
                   - c4 * distance_matrix[destination_node][node]
1995
                   + c5 * average_feedback_distance
1996
1997
               scores[node] = score
```

```
next_node = min(scores, key=scores.get)
return next_node
```

Heuristic 8: TSP, by CALM (API, w/o GRPO)

J LIMITATIONS

 A current limitation of our method is that the evolution of the LLM during the heuristic discovery process depends heavily on performance signals derived from heuristics present in the prompt and response. As a result, trajectories that do not contain explicit heuristics (e.g., the response from a reflection prompt may contain the thoughts only) in either component provide no reward signal, limiting the LLM's ability to learn from such cases.

Another limitation is that we currently evaluate our method, CALM, using a compact LLM on a single 24GB GPU. This restriction is primarily due to limited computational resources and the high cost associated with high-accuracy, full-parameter fine-tuning on larger models. While this setup demonstrates the feasibility of our approach in a resource-constrained environment, further evaluation on larger-scale models and infrastructure would be valuable for understanding the method's full potential and scalability.

In future work, we aim to address these limitations by (1) exploring mechanisms for adapting the LLM in the absence of explicit performance feedback, enabling more effective use of reinforcement learning, and (2) extending evaluations to more powerful models and settings. These directions may allow for better integration with techniques such as reflection (Ye et al., 2024; Dat et al., 2025), which have shown promise in enhancing LLM-based automated heuristic discovery.

K Broader Impact

The CALM framework stands to greatly accelerate the pace of innovation in algorithm design by seamlessly integrating prompt engineering and on-the-fly model adaptation. By enabling state-of-the-art heuristic discovery on a single 24 GB GPU, CALM democratizes access to cutting-edge Automatic Heuristic Design. This empowers research groups, startups, and educational institutions with limited compute budgets to explore and deploy high-performance solutions in domains such as logistics, scheduling, and resource allocation.

L LICENSE

The licenses and URLs of baselines, models, and softwares are summarized in Table 15.

Table 15: A summary of licenses.

Resources	Type	License	URL
Unsloth	Code	Apache-2.0 License	https://github.com/unslothai/unsloth
Qwen2.5	Model	Apache-2.0 License	https://huggingface.co/Qwen/Qwen 2.5-7B-Instruct
LKH3	Code	Available for academic research use	http://webhotel4.ruc.dk/~keld/re search/LKH-3/
OR-Tools	Code	MIT License	https://developers.google.com/op timization/pack/knapsack?hl=zh-cn
POMO	Code	Available online	https://github.com/yd-kwon/POMO/tree/master
DeepACO	Code	MIT License	https://github.com/henry-yeh/Dee pACO
Funsearch	Code	Apache License	https://github.com/google-deepmind/funsearch
ЕоН	Code	MIT License	https://github.com/FeiLiu36/EoH/tree/main
ReEvo	Code	MIT License	https://github.com/ai4co/reevo
HSEvo	Code	Available online	https://github.com/datphamvn/HSE vo
MCTS-AHD	Code	MIT License	https://github.com/zz1358m/MCTS-A HD-master
EvoTune	Code	MIT License	https://github.com/CLAIRE-Labo/EvoTune
OpenEvolve	Code	Apach-2.0 License	https://github.com/codelion/openevolve