REFINING HYBRID GENETIC SEARCH FOR CVRP VIA REINFORCEMENT LEARNING-FINETUNED LLM

Anonymous authors
Paper under double-blind review

ABSTRACT

While large language models (LLMs) are emerging as automated heuristic designers for solving vehicle routing problems (VRPs), state-of-the-art approaches predominantly rely on massive, general-purpose models like GPT-4. This work challenges this paradigm by demonstrating that smaller, specialized LLMs, when finely tuned, can generate components that surpass expert-designed heuristics within advanced solvers. We introduce RFTHGS, a novel **R**einforcement learning (RL) framework for **F**ine-**T**uning a small LLM to produce high-performance crossover operators for the **H**ybrid **G**enetic **S**earch (HGS) solver to solve the capacitated vehicle routing problem (CVRP). Our methods utilizes a multi-tiered, curriculum-based reward function that progressively guides the LLM to first produce compilable code, then executable operators, and finally, components that exceed human expert-designed ones. Additionally, we introduce an operator caching mechanism to work in conjunction with the reward function, discouraging plagiarism and promoting diversity during training. Experimental results demonstrate that our fine-tuned LLM generates crossover operators which significantly outperform those designed by human experts in HGS. This performance advantage is consistent, holding from small-scale instances and generalizing to large-scale problems of up to 1000 nodes. Furthermore, RFTHGS surpasses leading neuro-combinatorial baselines, prompt-based methods, and commercial LLMs, including GPT-4o and GPT-4o-mini.

1 INTRODUCTION

Combinatorial Optimization Problems (COPs) represent a fundamental class of computational challenges that arise across diverse domains including supply chain management, logistics, scheduling, and network design (Bengio et al., 2021). These problems, characterized by their discrete decision variables and complex constraints, are often NP-hard in complexity, making exact solutions computationally intractable for large-scale instances. For decades, researchers have developed specialized algorithms, heuristics, and metaheuristics to approximate optimal solutions, yet these approaches typically require significant domain expertise and manual design efforts (Papadimitriou & Steiglitz, 1998). The emergence of large language models (LLMs), with their remarkable reasoning and pattern recognition capabilities, has introduced a transformative paradigm for tackling COPs. By leveraging their natural language processing and generative abilities, LLMs enable automated approaches that reduce reliance on manual algorithm design and expert intervention (Sun et al., 2024; Novikov et al., 2025). Investigating on leveraging LLMs to solve Vehicle Routing Problems (VRPs) represents one of the most cutting-edge frontiers in this field.

Initial studies explored the use of LLMs as end-to-end solvers for VRPs (Yang et al., 2024). However, these purely generative approaches often yield solutions that are substantially inferior to those from conventional or deep learning-based solvers and are frequently infeasible—a shortcoming attributed to the fact that LLMs are notoriously prone to hallucinations (Kalai et al., 2025). Consequently, a more promising direction is to integrate LLMs not as autonomous solvers but as intelligent operators within established optimization frameworks, such as evolutionary algorithms. In this hybrid paradigm, the LLM acts as a strategic generator or refiner within an iterative loop. For instance, Liu et al. (2024c) employ general-purpose LLMs to guide an evolutionary process, using in-context prompting to perform crossover and mutation. An alternative approach inverts this relationship, using evolutionary computation not as a framework for the LLM to power, but as a mechanism to guide

the LLM itself in generating increasingly effective heuristics. Representative works like EoH (Liu et al., 2024b) and ReEvo (Ye et al., 2024) iteratively refine LLM-generated heuristic through evolutionary selection. In contrast, another line of research prioritizes general-purpose frameworks for diverse VRP variants, favoring broader generalization at the cost of performance. Methods such as ARS (Li et al., 2025a), which leverage predefined structures to generate constraint-checking functions, and DRoC (Jiang et al., 2025), which utilizes retrieval to produce code that invokes external solvers like OR-Tools (Furnon & Perron, 2025), demonstrate improved generalization and robustness against code execution failures. Nonetheless, a significant performance gap remains with conventional and deep learning-based solvers, echoing doubts about the immediate application of existing methods to large-scale problems. Given that practical instances in the real scenario often rely on advanced solvers, a critical open question is whether we can finetune small LLMs to optimize key components within these solvers to achieve beyond expert performance, presenting a challenging yet promising research frontier.

We introduce RFTHGS, a reinforcement learning (RL) framework for fine-tuning a reasoning LLM with 14B parameters to autonomously generate effective crossover operators for the Hybrid Genetic Search (HGS) algorithm (Vidal, 2022), thereby enhancing its performance in solving large-scale Capacitated Vehicle Routing Problems (CVRP). Our RL approach utilizes solution quality as the key feedback signal, instantiated as a structured, tiered reward design. This reward is designed to guide the learning process through three progressive stages. First, from our observation that instruction-tuned models often fail to produce syntactically valid code, we reward the model with a reward for generating any compilable code. Second, an additional reward is granted if the operator code executes successfully without runtime errors or timeouts. Finally, for the compilable and executable code, the relative improvement in solution quality on a predefined set of CVRP instances—compared to a baseline expert-designed operator—translates into a linear reward (positive for outperforming the baseline, negative otherwise). It is important to emphasize that the CVRP instances serve exclusively to steer the feedback mechanism by evaluating operators generated by the LLM and are not provided as input to the model itself. To mitigate reward hacking and prevent the repeated generation of a single high-performing operator, we incorporate an operator buffer mechanism. This mechanism penalizes the model for producing duplicates, thereby explicitly incentivizing diversity in the discovered solutions. Through this iterative refinement process, our framework empowers the LLM-generated operators to ultimately exceed the performance of handcrafted operators designed by human experts. Extensive experiments verify that the LLM-generated crossover operator delivers substantial improvements over the expert-designed operator in HGS, achieving superior performance on both small and large-scale instances (up to 1,000 nodes) on real-world benchmark. Furthermore, it surpasses all leading neuro-combinatorial and prompt-based LLM baselines by a significant margin. This work provides, to the best of our knowledge, the first evidence that a small-scale reasoning LLM (14B parameters) can be fine-tuned via RL to produce critical components that exceed the performance of those in state-of-the-art, expert-designed solvers.

2 Related Work

2.1 On The Reasoning Ability of Large Language Models

The development of large language models (LLMs) with advanced reasoning capabilities has evolved through several key phases, beginning with Chain of Thought (CoT) prompting, which explicitly guides models to generate intermediate reasoning steps, significantly improving performance on tasks like arithmetic and commonsense reasoning (Plaat et al., 2024; Wei et al., 2022). This approach was further enhanced by inference-time strategies such as Self-Consistency (aggregating multiple reasoning paths) and Tree-of-Thoughts (ToT) (exploring branched reasoning trajectories), which reduce errors and improve robustness in multi-step problem-solving (Yao et al., 2023). A major shift occurred with the integration of reinforcement learning (RL) techniques, where models are trained using verifiable rewards (e.g., correct answers in mathematical problems or code execution results) to incentivize logical reasoning without relying solely on supervised fine-tuning (Xiang et al., 2025; Xu et al., 2025). For instance, DeepSeek-R1 (DeepSeek-AI et al., 2025) and OpenAI's o1 series (OpenAI et al., 2024) exemplify how RL-driven self-improvement and scaled inference-time compute enable deliberate, step-by-step reasoning. Additionally, hybrid methods such as Microsoft's rStar-Math (Guan et al., 2025) (which integrates Monte Carlo Tree Search for problem decomposition) and retrieval-augmented generation (RAG) (Lewis et al., 2020) combine the pattern

recognition capabilities of LLMs with external tools for rigorous symbolic operations. Recent advancements also focus on test-time training and outcome-based exploration to enhance adaptability and diversity in reasoning paths, while interpretability research aims to ensure faithful internal reasoning processes (Song et al., 2025). Despite progress, challenges such as hallucination, scalability, and generalisation persist, driving ongoing innovation in architectures and training paradigms (Shojaee et al., 2025). In contrast to the well-recognized success in solving math problems, training the LLM with reasoning capabilities to generate operators that can outperform the default expert-designed ones in advanced VRP solvers remains challenging and largely unexplored.

2.2 Solving CVRP With LLM In The Loop

The integration of LLMs into Vehicle Routing Problems (VRPs) has primarily advanced through prompting-based methodologies (Yang et al., 2024; Jiang et al., 2025; Liu et al., 2024c; Huang et al., 2024), which leverage the robust reasoning capabilities of state-of-the-art (SOTA) off-the-shelf LLMs (e.g., GPT-o3-mini). For instance, ARS (Li et al., 2025b) uses LLMs to automatically generate constraint-aware heuristics for solving complex vehicle routing problems by synthesizing natural language descriptions into executable code, which can construct heuristics for 90% of common VRP variants with different constraints. Similarly, Hercules (Wu et al., 2025) employs Core Abstraction Prompting (CAP) to derive high-performance heuristics by abstracting core components from elite solutions, though it remains dependent on powerful closed-source LLMs. Existing research is also investigating automatic heuristic design with LLM. Representative works like EoH (Liu et al., 2024b) and ReEvo (Ye et al., 2024) iteratively refine LLM-generated heuristics through evolutionary selection. Most recently, CALM (Huang et al., 2025) extends this paradigm by integrating reinforcement fine-tuning of the LLM into the evolutionary loop, allowing the model and its generated heuristics to co-evolve. In contrast, finetuning-based approaches for COP remain relatively sparse, often due to challenges like catastrophic forgetting, computational costs, and overfitting when adapting pre-trained models to specialized domains. While parameter-efficient methods like LoRA (Hu et al., 2021) mitigate some issues, fine-tuning small open-source LLMs (e.g., LLaMA (Touvron et al., 2023)) to generate operators for widely-used solvers like Hybrid Genetic Search (HGS) (Vidal, 2022) remains an open problem. Current efforts are mainly devoted to prompting, leaving a gap in developing specialized, lightweight models that can efficiently integrate with solver frameworks without relying on API-dependent, proprietary LLMs.

3 Preliminary

HGS For Solving The Capacitated Vehicle Routing Problem (CVRP). The Hybrid Genetic Search (HGS) (Vidal, 2022) algorithm represents a state-of-the-art metaheuristic framework prominently applied to complex combinatorial optimization problems, particularly vehicle routing problems (VRPs). As an extension of the classical genetic algorithm, HGS distinguishes itself through a tight integration of population-based evolutionary search and intensive local improvement procedures. Its core mechanism involves maintaining a diverse population of solutions that are iteratively refined through a process of selection, crossover, and local search. Within this framework, the crossover operator is the primary mechanism for global exploration by recombining genetic material from parent solutions to generate novel offspring. This operator does not merely produce trivial combinations; rather, it constructs promising, high-quality solution skeletons that effectively inherit desirable attributes from both parents. These offspring solutions subsequently undergo rigorous local search, which acts upon the foundation laid by crossover to exploit the solution space locally and achieve feasibility and optimality. This synergistic interplay, where crossover provides a robust starting point for deep local exploitation, is a critical factor in the documented efficacy of HGS, enabling it to navigate the trade-off between exploration and exploitation effectively and consistently produce high-quality solutions for routing problems.

The Group Relative Policy Optimization (GRPO) Algorithm. GRPO (Shao et al., 2024) is an improved variant of Proximal Policy Optimization (PPO) (Schulman et al., 2017). The key innovation of GRPO lies in utilizing a normalized reward function to compute advantages, where the mean and variance are estimated through Monte-Carlo sampling (with sample size G) from the current policy $\pi_k(\cdot|x)$ at step k for each input (prompt) x. For given parameters ϵ, $\beta > 0$, and a reference policy π_{ref} (usually the base model), the GRPO objective optimization problem is formulated as:

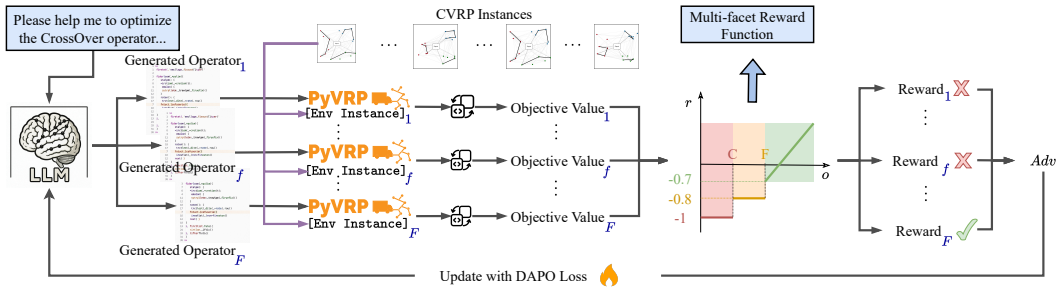


Figure 1: **The reinforcement learning pipeline of RFTHGS.** The framework iteratively optimizes an LLM to generate effective crossover operators for HGS. Each iteration consists of generating code from a structured prompt, evaluating the operator's performance on a validation set (using incremental compilation for speed), calculating a multi-faceted reward, and updating the LLM policy. The LLM only sees operator examples, not problem instances or the solver codebase.

$$\max_{\pi} \mathbb{E}_{y\sim\pi_k(\cdot|x)} \min\left[\frac{\pi(y|x)}{\pi_k(y|x)} A_{\pi_k}(x,y), \text{clip}\left(\frac{\pi(y|x)}{\pi_k(y|x)}, 1-\varepsilon, 1+\varepsilon\right) A_{\pi_k}(x,y)\right] - \beta \text{KL}(\pi||\pi_{\text{ref}}) \tag{1}$$

where KL denotes Kullback-Leibler divergence, and A_{π_k} represents GRPO advantage function:

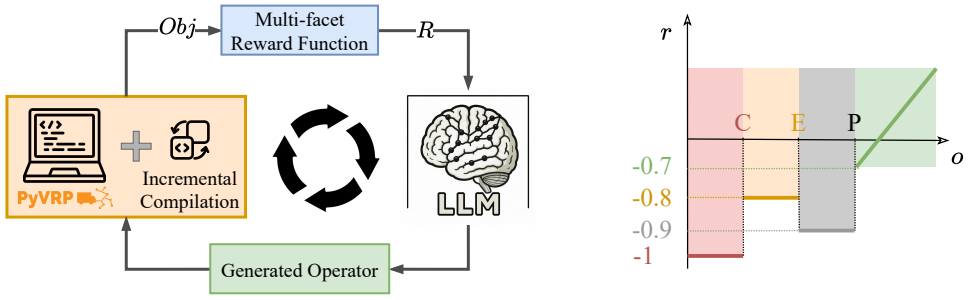
$$A_{\pi_k}(x,y_i) = \frac{r(x,y_i) - \mathbb{E}_{\pi_k} r(x,y_i)}{\sqrt{\mathbb{E}_{\pi_k}(r(x,y_i) - \mathbb{E}_{\pi_k} r(x,y_i))^2 + \varepsilon}} \simeq \frac{r(x,y_i) - \mu(\{r_\ell\})}{\sqrt{\sigma^2(\{r_\ell\}) + \varepsilon}}, \quad 1 \le \ell \le G \tag{2}$$

with the advantage estimated by sampling a "group" of size G for each input x, and μ and σ represent the empirical mean and standard deviation, respectively.

4 METHOD

We introduce RFTHGS, a reinforcement learning framework that fine-tunes large language models (LLMs) to generate crossover operators that outperform expert-designed ones in the Hybrid Genetic Search (HGS) solver (Vidal, 2022). The framework uses solution quality as the key reward signal to guide the LLM toward generating increasingly effective operators. As shown in Figure 1, RFTHGS is an iterative closed feedback loop. Each iteration consists of prompting the LLM to generate new operators, assessing their performance on a set of predefined CVRP instances, and then employing the feedback reward to refine the LLM via reinforcement learning.

Specifically, each iteration begins by constructing a few-shot CoT context (see Appendix A.3) that contains: (1) instructions specifying key properties (e.g., diversity and quality) and steps for generating a high-quality operator, and (2) examples of existing operators that illustrate the required structure and syntax. The LLM then generates new crossover operators based exclusively on this prompt, with no references to other modules in the HGS library or access to specific CVRP instances. For evaluation, each generated operator is integrated into the HGS library, and the code is recompiled to test on a fixed problem benchmark set. We employ incremental compilation to speed up this recompilation step. Finally, performance metrics such as compilability and improvement over baseline operators are combined into a multi-faceted reward, which is used to finetune the LLM via reinforcement learning. The RFTHGS framework automates the design of optimization operators, and demonstrates that small LLMs can evolve components that outperform human-designed ones in a state-of-the-art CVRP solver. We also verify that the RFTHGS framework is generic and can be applied to optimize other operators (or modules) within HGS, as demonstrated in Appendix A.6, where we apply RFTHGS to optimize the subpopulation operator as another example.



(a) (b)

Figure 2: (a) HGS as the environment for evaluating the quality of LLM-generated operators. We use the incremental compilation technique to boost the computation of objective values. (b) The multi-faceted reward function.

4.1 One-Step POMDP Modelling

We formulate the operator optimization as a one-step Partially Observable Markov Decision Process (One-Step POMDP), with formal definitions of state, action, reward and policy given as follows.

State: The state $X \in \mathcal{X}$[1] denotes a tokenized prompt received by the LLM, comprising both the task instructions and examples of the target operator to be optimized. To mitigate context overload, the input is restricted to the target operator itself (e.g., the crossover operator in HGS), rather than the entire library of the solver. This restriction results in a partially observable environment, as the model only perceives a subset of the full state space (i.e., the complete solver repository).

To enhance the versatility of the learning process, we maintain a buffer of few-shot examples containing operators generated by the LLM during training as well as those designed by human experts. At each iteration, we randomly sample examples from this buffer to construct the prompt. This approach enables the LLM to learn from and attempt to improve upon both its previous generations and expert-designed operators. Furthermore, it enriches the diversity of the initial states (i.e., prompts), which helps prevent overfitting and encourages broader exploration.

Action: The action (or response) $Y \sim p(\cdot|X \in \mathcal{X})$ generated by our operator-refining LLM is a sequence of tokens $Y \in \mathcal{Y}$ consisting of two main parts, where $p(\cdot|X)$ is the conditional probability from which the actions are sampled. The first part is a reasoning segment enclosed between the special tokens `<think>` and `</think>`, where it explains the steps and logic it plans to take for the optimization task. Following this, it outputs optimized version of the code for the target operator.

State Transition: The state terminates after the action is generated, since we only allow one round of optimization for the operator. Therefore, there are no state transitions, and the POMDP is one-step.

Reward: The reward $r \in \mathbb{R}$ is a scalar evaluating the quality of LLM-generated operators. Please refer to Section 4.2 below for details of the reward function.

Policy Network: The policy network is a base large language model, denoted by $\pi_\theta(\cdot|X),\ X \in \mathcal{X}$, with trainable parameters θ that parameterize the conditional probability distribution $p(\cdot|X)$ from which the optimized operator is sampled. In this work, we focus on relatively small LLMs (e.g., 14B parameters), which can be either a pretrained base model or an instruction-tuned variant.

4.2 The Multi-Faceted Reward Design With Anti-Plagiarism Cache

Building on the insight that carefully crafted, multi-faceted rewards are crucial for effective RL (Narvekar et al., 2020; Eppe et al., 2022; Huang et al., 2025), we developed a multi-tiered reward function to decompose the learning process. Particularly, the reward function follows a curriculum

[1]Here we use the symbol $X \in \mathcal{X}$ to represent the state to highlight that the RL task here is different from the conventional RL ones, where the initial state is the input to the LLM.

learning principle, guiding the LLM through progressive stages to evolve operators that exceed those designed by human experts. To ensure the robustness of this approach, we further introduce two key innovations: a mechanism to prevent reward hacking by deterring plagiarism of prompt examples, and a method to significantly accelerate the training process.

Anti-Plagiarism Cache With Abstract Syntax Tree. To mitigate reward hacking and encourage the exploration of unseen operators, we introduce a caching mechanism that leverages Abstract Syntax Trees (ASTs) to deter plagiarism. The AST provides a structured, hierarchical representation that abstracts away unnecessary syntactic details like punctuation and formatting to capture the essential logical structure of the generated operators. We cache the AST representations of all few-shot operator examples in the prompt. For each operator generated by the policy π_θ, its AST is compared against those in the cache. A penalty is invoked by the reward function if a substantial match is detected, indicating direct copying. This approach promotes diverse exploration by penalizing redundant operator generation.

HGS With Incremental Compilation As The Evaluator. We have to integrate each generated operator into the HGS library to evaluate its quality. This process inevitably requires recompiling the repository, which will incur prohibitive computational overhead, especially for large training batch sizes. Nonetheless, recompiling the entire library is unwarranted when only a single, small code snippet (the generated operator) is modified. To address this bottleneck, we employ an incremental compilation technique that selectively recompiles only the modified code and its dependencies, reducing recompilation time to approximately 25% of compiling the whole library and significantly accelerating the training speed.

Here we give the formulation of our three-stage reward function. First, the reward function assigns a reward of -0.8 for a syntactically correct and compilable operator to encourage a rapid transition from invalid code and improve exploration efficiency, or a penalty of -1 for invalid output. Upon achieving compilability, the function then assesses executability, penalizing runtime failures such as timeouts. An operator that executes successfully receives a reward of -0.7, independent of its solution quality. Finally, for executable operators, performance is evaluated on a predefined set of CVRP instances, with the reward quantified as the relative improvement over expert-designed benchmarks according to the following calculation:

$$r(o) = \begin{cases} -1 & o \notin \mathrm{C} \\ -0.8 & o \in \mathrm{C}, o \notin \mathrm{E} \\ -0.9 & o \in \mathrm{C}, o \in \mathrm{E}, o \in \mathrm{P} \\ \max\left(-0.7, [\phi^J_{\mathrm{HGS}}(o_{\mathrm{expert}}) - \phi^J_{\mathrm{HGS}}(o)]/\phi^J_{\mathrm{HGS}}(o_{\mathrm{expert}})\right) & o \in \mathrm{C}, o \in \mathrm{E}, o \notin \mathrm{P} \end{cases} \tag{3}$$

In this formulation, o represents the generated operator, while C, E, and P correspond to the sets of compilable, executable, and plagiarized code, respectively. For evaluation, the HGS library is recompiled to include the generated operator o. The performance metric $\phi^J_{\mathrm{HGS}}(\cdot)$ is then calculated as the average result on J random CVRP instances. To benchmark the effectiveness of our continuous reward design in Equation 3, we compare it with a discrete version where we employ a +1 reward if the generated operator outperforms the baseline operator in Table 3 (details in Appendix 5.4). We find that our continuous reward offers feedback proportional to performance gains, enabling sustained refinement and explaining its superior performance.

4.3 The Reinforcement Learning Algorithm

We use DAPO (Yu et al., 2025) as the reinforcement learning algorithm for training our operator refining network. Specifically, DAPO is an improved version of GRPO with four adjustments: 1). *Clip-Higher Mechanism*. Unlike GRPO following the original PPO setting where a unified clip ration is adopted for the positive and negative responses, DAPO decouples the clipping range into a higher upper bound ($\varepsilon_{\mathrm{high}}$) and a standard lower bound ($\varepsilon_{\mathrm{low}}$), allowing the policy to more aggressively increase probabilities for promising but initially low-likelihood tokens. This promotes greater exploration and diversity in generated responses, effectively preventing entropy collapse where the model becomes overly deterministic; 2). *Dynamic Sampling*. This strategy filters out prompt groups where all sampled responses are either all correct or all incorrect, as these yield zero advantage and provide no learning signal. By replacing them with new prompts that exhibit varied performance,

Table 1: **Performance comparison of baselines and our method for CVRPLIB across problem sizes.** Light gray columns indicate generalization to unseen problem sizes, while light gray rows represent generalization to higher iterations. The darker gray intersection areas highlight double generalization across both dimensions. Bold values denote best performance among all methods; asterisks (*) indicate that the results are unavailable.

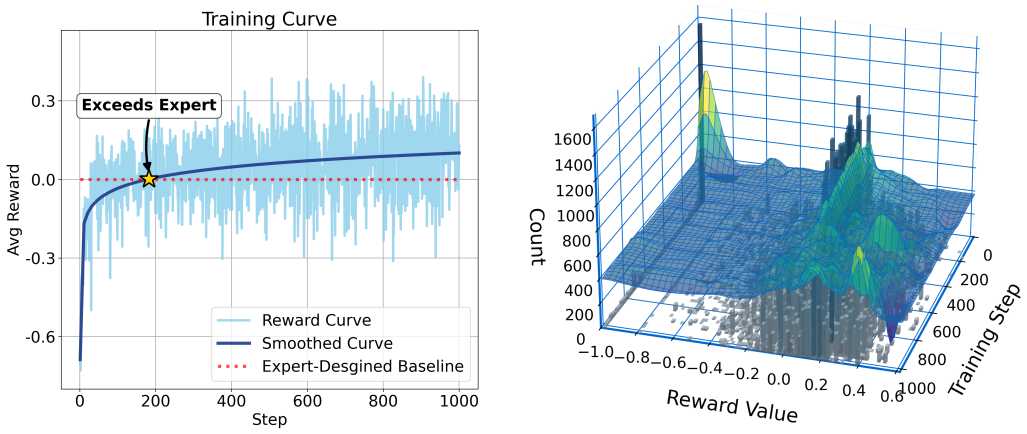
Methods	$n \in [100, 200)$		$n \in [200, 400)$		$n \in [400, 600)$		$n \in [600, 800)$		$n \in [800, 1000]$	
	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)
Conventional Solver										
HGS-PyVRP$_{800}$ (Wouda et al., 2024)	0.62	12.45	1.85	28.16	1.95	58.41	2.62	91.31	2.32	121.04
HGS-PyVRP$_{1000}$ (Wouda et al., 2024)	0.55	14.88	1.66	36.27	1.81	72.86	2.43	110.54	2.22	144.17
HGS-PyVRP$_{1200}$ (Wouda et al., 2024)	0.52	17.25	1.56	42.98	**1.69**	88.40	2.32	129.58	2.10	173.15
OR-Tools (Furnon & Perron, 2025)	4.26	88.83	5.05	172.07	4.98	296.53	6.71	416.95	4.65	532.32
LKH (Helsgaun, 2000)	1.42	191.12	1.97	252.23	2.85	432.36	3.65	599.41	3.31	545.85
NCO										
POMO (Kwon et al., 2020)	13.30	0.41	14.64	0.64	22.07	1.29	21.57	2.32	41.23	4.16
MTPOMO (Liu et al., 2024a)	6.50	0.98	8.79	1.02	16.58	1.89	26.56	2.81	28.19	4.33
MVMoE (Zhou et al., 2024)	5.46	0.80	8.14	1.56	13.26	2.85	16.59	4.17	18.40	6.25
RF-POMO (Berto et al., 2025)	5.67	0.54	7.07	1.15	10.29	1.97	12.28	2.86	13.31	4.48
RF-MoE-L (Berto et al., 2025)	7.15	0.85	7.67	1.58	10.76	2.79	15.15	3.95	15.70	5.84
AM (Kool et al., 2019)	200.75	0.30	204.59	0.58	253.98	1.12	301.08	1.51	280.49	2.01
DeepACO (Ye et al., 2023)	76.18	18.37	93.02	35.03	97.70	61.03	123.89	88.23	116.13	112.01
NeuroLKH (Xin et al., 2021)	1.96	1.02	*	*	*	*	*	*	*	*
NeuOpt (Ma et al., 2023)	3.51	*	*	*	*	*	*	*	*	*
Prompting-Based Method With LLM										
MCTS-AHD (Zheng et al., 2025)	18.51	4.35	19.07	10.06	18.40	22.50	28.51	37.60	19.70	55.61
ReEvo (Ye et al., 2024)	72.11	4.96	96.55	10.78	107.40	23.99	163.62	41.72	144.22	61.49
GPT4o$_{800}$ (Hurst et al., 2024)	0.62	12.1	1.85	29.3	1.95	59.4	2.62	91.7	2.32	119.2
GPT4o$_{1000}$ (Hurst et al., 2024)	0.55	15.0	1.66	36.1	1.81	73.4	2.43	111.2	2.22	143.7
GPT4o$_{1200}$ (Hurst et al., 2024)	0.52	17.7	1.56	43.1	1.69	87.3	2.32	130.5	2.10	172.8
GPT-o3$_{800}$ (Jaech et al., 2024)	0.62	11.96	1.85	30.20	1.95	58.94	2.62	91.39	2.32	119.04
GPT-o3$_{1000}$ (Jaech et al., 2024)	0.55	14.31	1.66	34.80	1.81	73.61	2.43	109.78	2.22	143.19
GPT-o3$_{1200}$ (Jaech et al., 2024)	0.52	17.65	1.56	43.53	1.69	87.89	2.32	130.71	2.10	172.93
GPT-o4-mini$_{800}$ (Jaech et al., 2024)	0.70	10.9	1.78	26.1	1.99	54.4	2.80	86.7	2.28	114.8
GPT-o4-mini$_{1000}$ (Jaech et al., 2024)	0.63	13.7	1.66	32.3	1.86	66.8	2.70	106.3	2.20	140.6
GPT-o4-mini$_{1200}$ (Jaech et al., 2024)	0.58	16.3	1.58	38.6	1.74	79.4	2.61	127.1	2.11	169.6
Ours										
RFTHGS$_{800}$	0.70	13.14	1.67	29.60	1.83	61.65	2.59	92.17	2.24	118.59
RFTHGS$_{1000}$	0.52	14.33	1.62	36.16	1.76	74.16	2.35	110.43	2.17	143.87
RFTHGS$_{1200}$	**0.46**	19.13	**1.55**	44.18	1.73	87.84	**2.26**	132.04	**2.09**	172.56

DAPO ensures every training batch contains meaningful gradients, improving training efficiency and stability without sacrificing throughput. However, in our paper, we deprecate this design as the reward signal in our case is continuous, specifying a wide range of situations from uncompilable code to superior performance gain against the baseline operators that all contribute useful learning signals for the LLM to learn; 3). *Token-Level Policy Gradient Loss*. Unlike GRPO, which averages losses at the response level, DAPO calculates and aggregates the loss over all tokens in the batch before averaging. This ensures each token's contribution to the gradient is weighted equally, providing more precise updates for long reasoning chains and better reinforcing correct steps in lengthy responses; 4). *Overlong Reward Shaping*. To address the issue of truncated lengthy responses that may contain valid reasoning, DAPO employs two strategies: Overlong Filtering excludes these responses from training updates to avoid misleading penalties, and Soft Overlong Punishment applies a gradual, length-dependent penalty beyond a certain token threshold to encourage conciseness without harshly punishing correct but verbose reasoning. The pseudo-code of our algorithm is shown in Algorithm 1.

5 Experiments

5.1 Experiment Settings And Baselines

We give the details of the configurations of our RFTHGS algorithm. Specifically, we initialize the policy with Qwen-14B reasoning LLM (Yang et al., 2025). For DAPO, we follow its optimal set-



(a) (b)

Figure 3: **Training dynamics of the RFTHGS framework.** (a) Average reward per step, showing stable convergence. (b) Evolution of the reward distribution, illustrating the effectiveness of the multi-faceted reward function in guiding the learning process.

tings reported in the original paper with $\varepsilon_{\text{high}} = 0.28$ and $\varepsilon_{\text{low}} = 0.2$. The batch size is set to 16 and the rollout group size is 16. Therefore, the policy model will generate 256 crossover operators for each step. For calculating reward during training, we use a fixed set of 30 CVRP instances sampled from the CVRPLIB X instances (Uchoa et al., 2017), restricting the selection to those with at most 400 nodes. During the testing phase, we sample 16 operators and report the performance of the best one. The final evaluation of our method is performed on the CVRPLIB benchmark, which encompasses a wide range of instance sizes from small scales to industry-level scales (up to 1000 nodes). We benchmark RFTHGS against a variety of baselines on CVRPLIB X instances (Uchoa et al., 2017). These include the state-of-the-art conventional solvers, neuro-combinatorial techniques, and prompting strategies that utilize commercial LLMs such as the GPT-4 series. To ensure an equitable comparison for the LLM-based approaches, we consistently sample 16 operators and select the best one for each. Further details on the baselines are available in Table 1. Detailed breakdown of the computational resources required for RL training is provided in Appendix A.7

5.2 Performance On CVRPLIB

Table 1 compares RFTHGS against a diverse set of baselines on CVRPLIB instances, including conventional heuristics, neuro-combinatorial methods, and prompting techniques that utilize commercial LLMs such as the flagship GPT-4o series. The results demonstrate that RFTHGS outperforms all baseline methods by a substantial margin. This superior performance is underscored by its exceptional generalization capability to large-scale problems unseen during training. Notably, although trained exclusively on instances with $n < 400$, our approach generalizes effectively to instances of up to $n = 1000$, which are more than twice the size of the largest training instances. This validates the potential of refining advanced solvers via learned components for complex combinatorial optimization problems. The code comparison between human-expert designed operators and our LLM-optimized operators is presented in Appendix A.8, highlighting the key modifications introduced by the LLM that contribute to improved performance.

Another key observation is that our RFTHGS framework enables a 14B-parameter LLM to outperform trillion-parameter GPT reasoning models (GPT-4o, GPT-o3, GPT-o4-mini). This advantage is demonstrated through both the quality of the modifications and their practical efficacy. As shown in Table 2, our model achieves a perfect successful compilation rate of 16/16, substantially exceeding the rates of the GPT

Table 2: **Successful compilation rate**.

Successful Compilation Rate			
GPT-4o	GPT-o3	GPT-o4-mini	RFTHGS-14B
3/16	9/16	3/16	**16/16**

Table 3: **Ablation study on reward design.** FRTHGS$_d$ is the 14B LLM trained with reward in Equation 4. FRTHGS$_c$ is the 14B LLM trained with reward in Equation 3. Shaded areas are generalization results.

	$n \in [100, 200)$		$n \in [200, 400)$		$n \in [400, 600)$		$n \in [600, 800)$		$n \in [800, 1000]$	
Methods	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)
HGS	**0.62**	12.45	1.85	28.16	1.95	58.41	2.62	91.31	2.32	121.04
FRTHGS$_d$	0.83	11.75	1.78	28.89	1.92	58.12	2.64	94.30	2.30	120.28
FRTHGS$_c$	0.70	13.14	**1.67**	29.60	**1.83**	61.65	**2.59**	92.17	**2.24**	118.59

models (3/16, 9/16, and 3/16, respectively). Crucially, while the GPT models often introduce numerous modifications, these changes consistently fail to improve performance. This is evident in Table 1, where the crossover operators modified by these GPT models exhibit performance identical to the original, unmodified operator, confirming that no functionally helpful modifications were made. In contrast, our RFTHGS-guided model produces targeted, effective modifications that yield consistent performance gains. This demonstrates that specialized fine-tuning for a specific task is more effective than using a general-purpose model of a much larger scale.

5.3 Learning Pattern Analysis

Figure 3a presents the learning curve of the RFTHGS framework, demonstrating stable and monotonic convergence. The average reward increases smoothly without significant oscillations, indicating a well-structured learning landscape with the effective design of our reward function. A critical inflexion occurs around step 200, where the generated operator surpasses the expert-designed baseline, marking the transition from learning executable operators to discovering superior heuristics. Beyond this intersection point, the curve continues to show consistent improvement, ultimately achieving substantially higher performance. This smooth progression shows that RFTHGS can effectively guide the LLM in generating increasingly sophisticated crossover operators. Figure 3b reveals the underlying learning patterns through the dynamics of the reward distribution. The heat map exhibits a clear curriculum learning pattern that precisely echoes our multi-tiered reward design. Initially, the density concentrates at lower rewards as the model masters syntactic correctness and compilability. Subsequently, the distribution shifts toward intermediate rewards, corresponding to the phase where operators become executable and yield valid solutions. Finally, the density center progressively migrates to the highest reward region, indicating the refinement toward operators that consistently outperform human-designed ones. This tri-phasic progression validates the effectiveness of our hierarchical reward structure in decomposing the complex operator design task into manageable learning stages.

5.4 Generalization Performance On Iterations

The generalization capability of RFTHGS is further assessed across two joint dimensions, i.e., iteration count and problem size. Regarding iteration generalization, models trained with an 800-iteration budget (RFTHGS$_{800}$) maintain robust performance when evaluated at higher budgets of 1000 and 1200 iterations, consistently outperforming expert-designed baselines. This indicates that the optimized operators retain their efficacy beyond their training configuration. In terms of problem size, although trained exclusively on instances with $n < 400$, RFTHGS generalizes effectively to significantly larger problems (up to $n = 1000$). This joint generalization underscores the robustness and strong out-of-distribution scalability of our method. The results are shown in the last two rows (grey areas) of Table 1.

6 Conclusion

This paper introduces RFTHGS, a reinforcement learning framework that optimizes operators in the Hybrid Genetic Search (HGS) solver for solving the Capacitated Vehicle Routing Problem (CVRP). By fine-tuning with domain-specific rewards, we demonstrate that specialized small LLMs can surpass large general and deep thinking ones like GPT-4o, GPT-o4-mini, and GPT-o3 with trillions of parameters. Our core innovation is a novel RL-based fine-tuning paradigm guided by solution quality, featuring a multi-tiered reward mechanism with anti-plagiarism caching for progressive learning.

Extensive experiments on CVRPLIB benchmarks confirm that the crossover operator generated by our method demonstrates superior performance over the expert-designed operator within the HGS framework, achieving substantial improvements, particularly on large-scale instances with up to 1,000 nodes. To our knowledge, this is the first work to show that a small, fine-tuned LLM can generate operators that exceed expert-crafted components in a leading combinatorial optimization solver. In future, we will try to evolve more operators inside HGS and solve more types of VRPs.

7 Ethics Statement

This study involves no personal data, human subjects, or other sensitive content and therefore presents no obvious ethical concerns. The only potential risk lies in the fact that operators generated by LLMs may contain bugs which, if deployed without thorough validation, could cause losses.

8 Reproducibility Statement

To ensure the reproducibility of our work, we have provided comprehensive experimental details throughout this paper. Section 5.1 presents complete experimental configurations and environment specifications, while the Appendix A.3 includes detailed prompts. These materials provide sufficient information for independent reproduction of our experimental results. Furthermore, we will open-source our full code base and model weights to further improve reproducibility.

References

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

Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, André Hottung, Niels Wouda, Leon Lan, Junyoung Park, Kevin Tierney, and Jinkyoo Park. Routefinder: Towards foundation models for vehicle routing problems, 2025. URL https://arxiv.org/abs/2406.15007.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu,

Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Manfred Eppe, Christian Gumbsch, Matthias Kerzel, Phuong DH Nguyen, Martin V Butz, and Stefan Wermter. Intelligent problem-solving as integrated hierarchical reinforcement learning. *Nature Machine Intelligence*, 4(1):11–20, 2022.

Vincent Furnon and Laurent Perron. Or-tools routing library, 2025. URL https://developers.google.com/optimization/routing/.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*, 2025.

Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European journal of operational research*, 126(1):106–130, 2000.

J. Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *ArXiv*, abs/2106.09685, 2021. URL https://api.semanticscholar.org/CorpusID:235458009.

Sen Huang, Kaixiang Yang, Sheng Qi, and Rui Wang. When large language model meets optimization. *Swarm and Evolutionary Computation*, 90:101663, 2024.

Ziyao Huang, Weiwei Wu, Kui Wu, Jianping Wang, and Wei-Bin Lee. Calm: Co-evolution of algorithms and language model for automatic heuristic design, 2025. URL https://arxiv.org/abs/2505.12285.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

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

Adam Tauman Kalai, Ofir Nachum, Santosh S. Vempala, and Edwin Zhang. Why language models hallucinate, 2025. URL https://arxiv.org/abs/2509.04664.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ByxBFsRqYm.

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.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

Kai Li, Fei Liu, Zhenkun Wang, Xialiang Tong, Xiongwei Han, Mingxuan Yuan, and Qingfu Zhang. ARS: Automatic routing solver with large language models. *arXiv preprint arXiv:2502.15359*, 2025a.

Kai Li, Fei Liu, Zhenkun Wang, Xialiang Tong, Xiongwei Han, Mingxuan Yuan, and Qingfu Zhang. Ars: Automatic routing solver with large language models, 2025b. URL https://arxiv.org/abs/2502.15359.

Fei Liu, Xi Lin, Zhenkun Wang, Qingfu Zhang, Tong Xialiang, and Mingxuan Yuan. Multi-task learning for routing problem with cross-problem zero-shot generalization. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 1898–1908, 2024a.

Fei Liu, Xialiang Tong, 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 *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024b.

Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. Large language models as evolutionary optimizers. In *2024 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, 2024c. doi: 10.1109/CEC60901.2024.10611913.

Yining Ma, Zhiguang Cao, and Yeow Meng Chee. Learning to search feasible and infeasible regions of routing problems with flexible neural k-opt. *Advances in Neural Information Processing Systems*, 36:49555–49578, 2023.

Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(181):1–50, 2020.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg

Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. Openai o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

Aske Plaat, Annie Wong, Suzan Verberne, Joost Broekens, Niki van Stein, and Thomas Bäck. Reasoning with large language models, a survey. *CoRR*, 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity, 2025. URL https://arxiv.org/abs/2506.06941.

Yuda Song, Julia Kempe, and Remi Munos. Outcome-based exploration for llm reasoning, 2025. URL https://arxiv.org/abs/2509.06941.

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

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aur'elien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971, 2023. URL https://api.semanticscholar.org/CorpusID:257219404.

Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858, 2017. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2016.08.012. URL https://www.sciencedirect.com/science/article/pii/S0377221716306270.

Thibaut Vidal. Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood. *Computers & Operations Research*, 140:105643, 2022.

vndee. llm-sandbox: Lightweight and portable LLM sandbox runtime. https://github.com/vndee/llm-sandbox, 2025.

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

Niels A. Wouda, Leon Lan, and Wouter Kool. PyVRP: a high-performance VRP solver package. *INFORMS Journal on Computing*, 36(4):943–955, 2024. doi: 10.1287/ijoc.2023.0055. URL https://doi.org/10.1287/ijoc.2023.0055.

Xuan Wu, Di Wang, Chunguo Wu, Lijie Wen, Chunyan Miao, Yubin Xiao, and You Zhou. Efficient heuristics generation for solving combinatorial optimization problems using large language models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, pp. 3228–3239, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714542. doi: 10.1145/3711896.3736923. URL https://doi.org/10.1145/3711896.3736923.

Violet Xiang, Charlie Snell, Kanishk Gandhi, Alon Albalak, Anikait Singh, Chase Blagden, Duy Phung, Rafael Rafailov, Nathan Lile, Dakota Mahan, et al. Towards system 2 reasoning in llms: Learning how to think with meta chain-of-thought. *arXiv preprint arXiv:2501.04682*, 2025.

Liang Xin, Wen Song, Zhiguang Cao, and Jie Zhang. Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 7472–7483. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/3d863b367aa379f71c7afc0c9cdca41d-Paper.pdf.

Fengli Xu, Qianyue Hao, Zefang Zong, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, Tianjian Ouyang, Fanjin Meng, et al. Towards large reasoning models: A survey of reinforced reasoning with large language models. *arXiv preprint arXiv:2501.09686*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Helan Liang, and Yong Li. DeepACO: Neural-enhanced ant systems for combinatorial optimization. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=cd5D1DD923.

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 A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 43571–43608. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/4ced59d480e07d290b6f29fc8798f195-Paper-Conference.pdf.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=Do1OdZzYHr.

Jianan Zhou, Zhiguang Cao, Yaoxin Wu, Wen Song, Yining Ma, Jie Zhang, and Chi Xu. Mvmoe: Multi-task vehicle routing solver with mixture-of-experts. *Proceedings of Machine Learning Research*, 235:61804–61824, 2024.

Table 4: **Ablation study on reward design.** FRTHGS$_d$ is the 14B LLM trained with reward in Equation 4. FRTHGS$_c$ is the 14B LLM trained with reward in Equation 3. Shaded areas are generalization results.

	$n \in [100, 200)$		$n \in [200, 400)$		$n \in [400, 600)$		$n \in [600, 800)$		$n \in [800, 1000]$	
Methods	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)	Gap% (↓)	Time (s)
HGS	**0.62**	12.45	1.85	28.16	1.95	58.41	2.62	91.31	2.32	121.04
FRTHGS$_d$	0.83	11.75	1.78	28.89	1.92	58.12	2.64	94.30	2.30	120.28
FRTHGS$_c$	0.70	13.14	**1.67**	29.60	**1.83**	61.65	**2.59**	92.17	**2.24**	118.59

A Appendix

A.1 Ablation Studies on Reward Design

To benchmark the effectiveness of our continuous reward design (Equation 3), we compare it with a discrete version defined as follows:

$$r_d(o) = \begin{cases} -1 & o \notin \mathrm{C} \\ -0.8 & o \in \mathrm{C}, o \notin \mathrm{E} \\ -0.9 & o \in \mathrm{C}, o \in \mathrm{E}, o \in \mathrm{P} \\ 0 & o \in \mathrm{C}, o \in \mathrm{E}, o \notin \mathrm{P}, \phi^J_{\mathrm{HGS}}(o_{\mathrm{expert}}) < \phi^J_{\mathrm{HGS}}(o) \\ 1 & o \in \mathrm{C}, o \in \mathrm{E}, o \notin \mathrm{P}, \phi^J_{\mathrm{HGS}}(o_{\mathrm{expert}}) > \phi^J_{\mathrm{HGS}}(o) \end{cases} \tag{4}$$

Table 4 demonstrates the clear advantage of our continuous reward design. FRTHGS$_c$ consistently outperforms the discrete-reward variant FRTHGS$_d$, particularly on larger problem sizes. The discrete reward's binary nature (0 or 1) provides limited guidance. Once an operator beats the baseline, the gradient vanishes as all improvements receive the same reward, and the advantage is thus 0 (Equation 2). In contrast, our continuous reward offers feedback proportional to performance gains, enabling sustained refinement and explaining its superior performance.

A.2 The Use of Large Language Models (LLMs)

We clarify that all intellectual contributions in this work, from initial idea conception and algorithm design to experimental implementation and result validation, were conducted exclusively by the human authors. While we employed LLMs during the manuscript preparation phase to refine language expression and improve readability, their role was strictly limited to linguistic polishing. The manuscript's structure, core arguments, and all substantive content were determined entirely by the human authors, with LLMs serving merely as an auxiliary tool for enhancing clarity and grammatical accuracy, similar to traditional proofreading services.

A.3 Prompt Template

```
# ROLE: Expert C++ Optimization Engineer for Vehicle Routing Problems

You are a senior C++ optimization engineer with expertise in algorithmic
    optimization, particularly for Vehicle Routing Problems (VRP). Your
    task is to analyze and improve the selective_route_exchange.cpp file'
    s crossover algorithm.

## TASK OVERVIEW
You are given the file selective_route_exchange.cpp (full listing below).
Your goal is to make ONE small, reliable modification that tends to
    create
children with better penalised cost (Solution quality ↑) while keeping
    runtime
and interface intact.

## THINKING PROCESS REQUIREMENTS
```

```
1. First, thoroughly analyze the current implementation to understand:
   - The algorithm's purpose and workflow
   - Key decision points and heuristics
   - Performance bottlenecks or optimization opportunities
   - Any constraints that must be preserved

2. Generate at least 3 different modification approaches, evaluating each
    on:
   - Potential improvement to solution quality
   - Impact on runtime performance
   - Compatibility with existing code
   - Risk of introducing bugs or side effects

3. For your chosen modification:
   - Justify why it's likely to improve solution quality
   - Verify it maintains the function signature and behavior
   - Double-check for compatibility with the rest of the codebase
   - Consider edge cases and verify robustness

############################################################
## HARD RULES (Mandatory Verification Checklist)

1. ☐ Keep the function signature and namespace exactly the same:
   pyvrp::crossover::selectiveRouteExchange(...)

2. ☐ The file must still compile under C++17 with the current #include
    lines.
   You may NOT remove #include directives.

3. ☐ Do not change any public headers, class interfaces, or external
    behaviour
   except for the improved offspring quality.

4. ☐ DO NOT fabricate or use non-existent or unmentioned attributes or
    methods.
   Verify every method you use exists in the provided code or
       documentation.

5. ☐ Wrap the code you output with '''cpp and '''.

6. ☐ Mark ALL your modifications with clear "// MODIFY: XXX" comments
    explaining the change.

7. ☐ You must make at least one modification; DO NOT copy the original
    code.

8. ☐ Before finalizing, double-check that your modification:
   - Does not introduce new parameters
   - Does not change the function's contract
   - Is focused on improving solution quality, not runtime
   - Is fully compatible with the existing codebase
   - Uses only documented methods and attributes

############################################################
## DELIVERABLES (strict):

A. ≤ 2-sentence summary of the optimization idea, clearly explaining how
    it improves solution quality.

B. Output the FULL C++ code with your modifications. Mark all changes
    with "// MODIFY: XXX" comments.

C. Brief explanation of your verification process and why you're
    confident the modification will:
   - Improve solution quality
```

```
   - Maintain compatibility with the existing codebase
   - Not significantly impact runtime performance

############################################################
## SCORING AND EVALUATION

We will benchmark on a fixed random seed over several CVRP instances.
Your patch should reduce the average optimal gap in ≥90% of the instances
     without
increasing total runtime by >3%.

Key considerations for high-quality solutions:
- More efficient route structures (fewer vehicles, shorter routes)
- Better client assignment to routes based on spatial relationships
- Improved handling of capacity constraints
- Preservation of high-quality route segments during crossover
- Better diversity in the generated offspring

############################################################

## selective_route_exchange.cpp
```cpp
{code}
```

Extra Information:
DOMAIN KNOWLEDGE: CVRP AND CROSSOVER OPERATIONS

The Selective Route Exchange is a crossover operation for the Capacitated
     Vehicle Routing Problem (CVRP). The algorithm:
1. Selects routes from two parent solutions
2. Exchanges these routes to create offspring
3. Aims to preserve beneficial route structures while creating new
    combinations

Key Optimization Areas to Consider:
- Route selection strategy (which routes to exchange)
- Client-to-route assignment decisions
- Proximity/distance calculations between routes or clients
- Handling of capacity constraints
- Diversity generation in offspring solutions

Essential Fields and Methods for CVRP Crossover

ProblemData Key Methods:
- `numLocations()` - Returns `size_t` total number of locations (depots +
     clients)
- `numClients()` - Returns `size_t` number of client locations
- `centroid()` - Returns `std::pair<double, double>` center of all client
     locations
- `client(idx)` - Returns `ProblemData::Client` with coordinates (x, y)

Route Key Methods:
- `centroid()` - Returns `std::pair<double, double>` center of route's
    client locations
- `vehicleType()` - Returns `VehicleType` (size_t) vehicle type index
- `begin()` / `end()` - Iterator support for visiting clients in route
- `size()` - Returns `size_t` number of clients in route
- `visits()` - Returns `std::vector<Client>` all client indices in route
    order

Route Construction:
- `Route(data, visits, vehicleType)` - Constructor taking `std::vector<
    Client>` visits and vehicle type
```

```
**Client Iteration:**
- Routes are iterable containers of `Client` (size_t) indices
- Use range-based for loops: `for (Client c : route)` to access all
    clients in route
- Client coordinates: `data.client(c).x`, `data.client(c).y`
```

A.4 Example Output With Reasoning

```
<think>
Okay, let's dive into this code. The task is to improve the
    selective_route_exchange.cpp's crossover algorithm to get better
    penalized costs without changing the function signature or runtime
    much.

First, I need to understand what the current code does. The selective
    route exchange crossover selects routes from two parents based on
    their angular orientation relative to the data's centroid. It sorts
    the routes by ascending polar angle, then exchanges routes starting
    at given indices. The goal is to create offspring that combine
    beneficial route structures from both parents.

Looking at the current implementation, the routes are sorted using a
    comparator that compares the absolute value of their angles. The
    comment says this is to cluster routes with similar directions. But
    maybe there's a better way to sort them. For instance, sorting by the
     actual angle (not absolute) might group routes that are in similar
    directions more effectively, leading to better route alignment and
    potentially better solution quality.

Wait, the current code uses std::abs(angleA) < std::abs(angleB) for
    comparison. But if we sort by the actual angle, routes that are in
    the same direction (like 30 degrees and 330 degrees) would be far
    apart in the sorted list. However, if we sort by angle in a circular
    manner (like considering angles modulo 360), that could cluster them
    better. But the current code uses absolute value, which might not
    account for directional similarity properly.

So, perhaps changing the comparator in the sortByAscAngle function from
    absolute angles to the actual angles would lead to better clustering.
     For example, routes with angles close to each other, even if they're
     on opposite ends of the circle, might still be better grouped if
    they are in similar directions when considering the centroid. Wait,
    the centroid is for all clients, so the route's angle is the
    direction of the route's centroid relative to the data's centroid.

If we sort by the actual angle instead of absolute value, routes that are
     in similar directions (like 0 degrees and 45 degrees) would be
    closer together. However, routes with angles like 170 and -170 (which
     have absolute values of 170 and 170) would be considered equal, but
    their actual angles are very different. So sorting by absolute value
    might not be the best approach.

Alternatively, maybe we should sort the routes by their actual angle,
    considering them in a circular way. But how to handle that in a
    comparator? For example, placing routes with angles near each other,
    but accounting for circular continuity. However, that's more complex
    and might not be feasible with a simple comparator.

But the current code uses absolute angle values. Let's think: if the
    angle is computed as the direction from the data's centroid to the
    route's centroid, sorting by absolute value might not be the best.
    For example, a route with an angle of 0 degrees (directly along the
    data's x-axis) and a route with 90 degrees (along y-axis) are not
```

similar. But if we sort by the actual angle, they'll be ordered in a way that might group routes with similar directions.

Wait, maybe the current approach is a mistake. Changing the comparator from absolute angle to the actual angle might lead to better route clusters. For instance, if two routes are in similar directions (like 30 and 35 degrees), they'll be grouped together, which might lead to better combinations during crossover.

So the modification would be in the sortByAscAngle function. The comparator currently uses std::abs(angleA) < std::abs(angleB). Changing that to compare angleA < angleB would sort routes by their actual angle, which could lead to better alignment of similar-direction routes.

This seems like a small change that could improve the clustering, leading to more effective exchanges between similar routes. This could result in offspring that preserve good route structures from both parents, thus improving the penalized cost.

I need to make sure that this change doesn't break anything else. The comparator is used to sort the routes. Changing the comparison from absolute angles to actual angles would affect the order, but the algorithm relies on adjacent routes in the sorted list having similar directions. If the comparator is fixed to sort by actual angles, that should be more accurate.

So the plan is to modify the lambda in the sortByAscAngle function from comparing absolute angles to comparing the angles directly. That's a simple code change with minimal risk of bugs. It's confined to the comparator and doesn't alter other parts of the code. The rest of the logic remains the same, ensuring compatibility.
</think>

A. The modification changes the route sorting strategy from absolute angle comparison to actual angle comparison, enabling better clustering of routes with similar directional alignment relative to the data's centroid. This increases the likelihood of exchanging similar, well-aligned routes, which can produce offspring with more efficient route structures.

B. ```cpp
#include "selective_route_exchange.h"

#include "DynamicBitset.h"

#include <cmath>
#include <vector>

using Client = size_t;
using Clients = std::vector<Client>;
using Route = pyvrp::Route;
using Routes = std::vector<Route>;

namespace
{
// Angle of the given route w.r.t. the centroid of all client locations.
double routeAngle(pyvrp::ProblemData const &data, Route const &route)
{
    auto const [dataX, dataY] = data.centroid();
    auto const [routeX, routeY] = route.centroid();
    return std::atan2(routeY - dataY, routeX - dataX);
}

Routes sortByAscAngle(pyvrp::ProblemData const &data, Routes routes)

```
{
    // MODIFY: Sort routes by actual angle instead of absolute angle
    // to better cluster routes with similar directional alignment.
    auto cmp = [&data](Route const &a, Route const &b) {
        double angleA = routeAngle(data, a);
        double angleB = routeAngle(data, b);
        return angleA < angleB;
    };

    std::sort(routes.begin(), routes.end(), cmp);
    return routes;
}
}  // namespace

pyvrp::Solution pyvrp::crossover::selectiveRouteExchange(
    std::pair<Solution const *, Solution const *> const &parents,
    ProblemData const &data,
    CostEvaluator const &costEvaluator,
    std::pair<size_t, size_t> const &startIndices,
    size_t const numMovedRoutes)
{
    // We create two candidate offsprings, both based on parent A:
    // Let A and B denote the set of customers selected from parents A
        and B
    // Ac and Bc denote the complements: the customers not selected
    // Let v denote union and ^ intersection
    // Parent A: A v Ac
    // Parent B: B v Bc

    // Offspring 1:
    // B and Ac\B, remainder A\B unplanned
    // (note B v (Ac\B) v (A\B) = B v ((Ac v A)\B) = B v Bc = all)
    // Note Ac\B = (A v B)c

    // Offspring 2:
    // A^B and Ac, remainder A\B unplanned
    // (note A^B v Ac v A\B = (A^B v A\B) v Ac = A v Ac = all)

    auto startA = startIndices.first;
    auto startB = startIndices.second;

    size_t nRoutesA = parents.first->numRoutes();
    size_t nRoutesB = parents.second->numRoutes();

    if (startA >= nRoutesA)
        throw std::invalid_argument("Expected startA < nRoutesA.");

    if (startB >= nRoutesB)
        throw std::invalid_argument("Expected startB < nRoutesB.");

    if (numMovedRoutes < 1 || numMovedRoutes > std::min(nRoutesA,
        nRoutesB))
    {
        auto msg = "Expected numMovedRoutes in [1, min(nRoutesA, nRoutesB
            )]";
        throw std::invalid_argument(msg);
    }

    // Sort parents' routes by (ascending) polar angle.
    auto const routesA = sortByAscAngle(data, parents.first->routes());
    auto const routesB = sortByAscAngle(data, parents.second->routes());

    DynamicBitset selectedA(data.numLocations());
    DynamicBitset selectedB(data.numLocations());
```

```
    // Routes are sorted on polar angle, so selecting adjacent routes in
        both
    // parents should result in a large overlap when the start indices
        are
    // close to each other.
    for (size_t r = 0; r < numMovedRoutes; r++)
    {
        for (Client c : routesA[(startA + r) % nRoutesA])
            selectedA[c] = true;

        for (Client c : routesB[(startB + r) % nRoutesB])
            selectedB[c] = true;
    }

    // For the selection, we want to minimize |A\B| as these need
        replanning
    while (true)
    {
        // Difference for moving 'left' in parent A
        int differenceALeft = 0;

        for (Client c : routesA[(startA - 1 + nRoutesA) % nRoutesA])
            differenceALeft += !selectedB[c];

        for (Client c : routesA[(startA + numMovedRoutes - 1) % nRoutesA
            ])
            differenceALeft -= !selectedB[c];

        // Difference for moving 'right' in parent A
        int differenceARight = 0;

        for (Client c : routesA[(startA + numMovedRoutes) % nRoutesA])
            differenceARight += !selectedB[c];

        for (Client c : routesA[startA])
            differenceARight -= !selectedB[c];

        // Difference for moving 'left' in parent B
        int differenceBLeft = 0;

        for (Client c : routesB[(startB - 1 + numMovedRoutes) % nRoutesB
            ])
            differenceBLeft += selectedA[c];

        for (Client c : routesB[(startB - 1 + nRoutesB) % nRoutesB])
            differenceBLeft -= selectedA[c];

        // Difference for moving 'right' in parent B
        int differenceBRight = 0;

        for (Client c : routesB[startB])
            differenceBRight += selectedA[c];

        for (Client c : routesB[(startB + numMovedRoutes) % nRoutesB])
            differenceBRight -= selectedA[c];

        int const bestDifference = std::min({differenceALeft,
                                             differenceARight,
                                             differenceBLeft,
                                             differenceBRight});

        if (bestDifference >= 0)  // there are no further improving moves
            break;

        if (bestDifference == differenceALeft)
```

```
        {
            for (Client c : routesA[(startA + numMovedRoutes - 1) %
                nRoutesA])
                selectedA[c] = false;

            startA = (startA - 1 + nRoutesA) % nRoutesA;
            for (Client c : routesA[startA])
                selectedA[c] = true;
        }
        else if (bestDifference == differenceARight)
        {
            for (Client c : routesA[startA])
                selectedA[c] = false;

            startA = (startA + 1) % nRoutesA;
            for (Client c : routesA[(startA + numMovedRoutes - 1) %
                nRoutesA])
                selectedA[c] = true;
        }
        else if (bestDifference == differenceBLeft)
        {
            for (Client c : routesB[(startB + numMovedRoutes - 1) %
                nRoutesB])
                selectedB[c] = false;

            startB = (startB - 1 + nRoutesB) % nRoutesB;
            for (Client c : routesB[startB])
                selectedB[c] = true;
        }
        else if (bestDifference == differenceBRight)
        {
            for (Client c : routesB[startB])
                selectedB[c] = false;

            startB = (startB + 1) % nRoutesB;
            for (Client c : routesB[(startB + numMovedRoutes - 1) %
                nRoutesB])
                selectedB[c] = true;
        }
    }

    // Identify differences between route sets
    auto const selectedBNotA = selectedB & ~selectedA;

    std::vector<Clients> visits1(nRoutesA);
    std::vector<Clients> visits2(nRoutesA);

    // Replace selected routes from parent A with routes from parent B
    for (size_t r = 0; r < numMovedRoutes; r++)
    {
        size_t indexA = (startA + r) % nRoutesA;
        size_t indexB = (startB + r) % nRoutesB;

        for (Client c : routesB[indexB])
        {
            visits1[indexA].push_back(c);  // c in B

            if (!selectedBNotA[c])
                visits2[indexA].push_back(c);  // c in A^B
        }
    }

    // Move routes from parent A that are kept
    for (size_t r = numMovedRoutes; r < nRoutesA; r++)
    {
```

```
        size_t indexA = (startA + r) % nRoutesA;

        for (Client c : routesA[indexA])
        {
            if (!selectedBNotA[c])
                visits1[indexA].push_back(c);  // c in Ac\B

            visits2[indexA].push_back(c);  // c in Ac
        }
    }

    // Turn visits back into routes.
    Routes routes1;
    Routes routes2;
    for (size_t r = 0; r < nRoutesA; r++)
    {
        if (!visits1[r].empty())
            routes1.emplace_back(data, visits1[r], routesA[r].vehicleType
                ());

        if (!visits2[r].empty())
            routes2.emplace_back(data, visits2[r], routesA[r].vehicleType
                ());
    }

    auto const sol1 = Solution(data, routes1);
    auto const sol2 = Solution(data, routes2);

    auto const cost1 = costEvaluator.penalisedCost(sol1);
    auto const cost2 = costEvaluator.penalisedCost(sol2);
    return cost1 < cost2 ? sol1 : sol2;
}
```

C. This modification changes the route sorting comparator from absolute angle comparison to actual angle comparison. This should better cluster routes with similar directional orientation relative to the data centroid, improving the likelihood of exchanging routes that are structurally compatible. The change only affects the route sorting strategy and does not alter any other part of the algorithm's logic, ensuring compatibility with existing code. Since it's just a comparator change, it does not impact runtime performance. Comparing actual angles instead of absolute angles is a more accurate measure of directional similarity, which should lead to better solution quality.

A.5 RFTHGS Pseudo Code

Algorithm 1: RFTHGS: Reinforcement Finetuning For Refining HGS

Input: Initial policy $\pi_{\theta_{\text{old}}}$, full instance set $\mathcal{I}$, instance batch size B, clipping parameters $\varepsilon_{\text{lower}}, \varepsilon_{\text{upper}}$, group size G
Output: Optimized policy π_θ
Initialize $\theta \leftarrow \theta_{\text{old}}$;
foreach *iteration* $1, 2, \ldots, N$ **do**
  Draw a random subset $\mathcal{B} \subset \mathcal{I}$ with $|\mathcal{B}| = B$;
  foreach *instance* $I \in \mathcal{B}$ **do**
    `// Step 1: Generate operators with LLM`
    Construct prompt q for CrossOver operator optimization;
    Sample G operators $\{o_1, \ldots, o_G\} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$;
    `// Step 2: Evaluate with PyVRP`
    foreach *operator* o_i **do**
      Run PyVRP solver with o_i substituted on instance set $\mathcal{B}$;
      Obtain objective value and compute reward r_i;
    end
    `// Step 3: Compute advantages`
    $\mathbf{r} = (r_1, r_2, \ldots, r_G)$
    Normalised reward for each operator: $\hat{A}_{i,t} = \dfrac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r})}$;
    `// Step 4: Update policy with DAPO`
    foreach *token position* t **do**
      $r_t(\theta) \leftarrow \dfrac{\pi_\theta(o_t \mid q, o_{<t})}{\pi_{\theta_{\text{old}}}(o_t \mid q, o_{<t})}$;
      $L_{\text{policy}} \leftarrow \mathbb{E}\big[\min\big(r_t(\theta)\,\hat{A}_t,\ \text{clip}\big(r_t(\theta), 1-\varepsilon_{\text{lower}}, 1+\varepsilon_{\text{upper}}\big)\,\hat{A}_t\big)\big]$;
    end
    $\theta \leftarrow \theta + \alpha \nabla_\theta L_{\text{policy}}$;
  end
end

A.6 Optimizing other operators using RFTHGS

The RFTHGS framework is not only suitable for optimizing the crossover operator but can also be applied to optimize other operators (or modules) influencing solution quality within HGS. In this section, we demonstrate the application of RFTHGS to optimize the `subpopulation` (`subp`) module in HGS. The primary function of this module is to manage population control by determining which individuals to eliminate based on cost and diversity metrics, ensuring the population size remains within defined thresholds.

To rigorously test the generalizability of our framework, we applied RFTHGS to the Capacitated Vehicle Routing Problem with Time Windows (CVRPTW) variant. It is important to emphasize that the configuration of the RFTHGS framework remained consistent with the experiments described in the main manuscript, where the POMDP formulation, training algorithm, base model, and reward calculation were identical.

Experimental Setup For training, we generate a dataset comprising 50 randomly generated CVRPTW instances with problem sizes ranging from 100 to 300 customers. For the testing phase, we generated 100 new random instances for each problem size, extending up to 1000 customers.

Results and Analysis The performance of the RFTHGS-optimized `subp` operator on CVRPTW instances is summarized in Table 5. The performance is reported as the relative gap to the original HGS, calculated as $(Cost_{RFTHGS} - Cost_{HGS})/Cost_{HGS}$. A negative value indicates that the RFTHGS-optimized operator outperforms the baseline HGS.

Table 5: Performance of the optimized subpopulation operator on CVRPTW instances. Values represent the relative gap (%) compared to the baseline HGS.

instance size	**100**	**200**	**300**	**500**	**1000**
HGS-subp (%)	0.000	0.000	0.000	0.000	0.000
RFTHGS-subp (%)	-0.088	-0.213	-0.299	-0.289	-0.224

As shown in Table 5, the optimized `subp` operator consistently outperforms the standard HGS across all problem sizes. Furthermore, the operator demonstrates strong generalization capabilities, maintaining superior performance on much larger problem sizes (500 and 1000) that were not seen during training, achieving relative gaps of -0.289% and -0.224%, respectively.

Mechanism of Improvement An analysis of the code generated by RFTHGS reveals that the enhanced `subp` operator modifies the fitness evaluation logic. The LLM introduced a third ranking metric based on the *number of routes* in a solution. This new metric is weighted and combined with the existing cost and diversity rankings to determine individual survival during parent selection.

This modification proves effective because it introduces a selective pressure that favors solutions utilizing fewer vehicles. By biasing the population towards more compact configurations, the algorithm indirectly promotes lower total costs, as fewer routes inherently reduce the cumulative distance associated with frequent returns to the depot. These results confirm that RFTHGS is a framework capable of optimizing diverse components within mature solvers across different problem variants.

A.7 Training Cost Analysis

We provide a breakdown of the computational resources for the RL training of our 14B model. The total training duration was approximately 98 GPU hours per device. Analyzing the cost per training step reveals a total duration of roughly 660 seconds, which is composed of 360 seconds for LLM training (inference, backward propagation, and updates) and 300 seconds for operator evaluation (execution and reward calculation).

Since reward computation constitutes a significant portion of the training cycle, future implementations can mitigate this bottleneck by adopting fully asynchronous training paradigms and utilizing lightweight sandbox environments, such as `llm-sandbox` vndee (2025). These tools enable the isolation of code execution on CPU-only nodes, allowing for massive parallelization and reduced GPU idle time.

A.8 Comparison between human-expert and LLM-optimized operators

Crossover

```
# human expert

...

Routes sortByAscAngle(pyvrp::ProblemData const &data, Routes routes)
{
    auto cmp = [&data](Route const &a, Route const &b)
    { return routeAngle(data, a) < routeAngle(data, b); };

    std::sort(routes.begin(), routes.end(), cmp);
    return routes;
}

...


# llm optimized

...
```

```
// Angle of the given route w.r.t. the centroid of all client locations.
// Adjusted to [0, 2π) range for better circular proximity handling
double routeAngle(pyvrp::ProblemData const &data, Route const &route)
{
    auto const [dataX, dataY] = data.centroid();
    auto const [routeX, routeY] = route.centroid();
    double angle = std::atan2(routeY - dataY, routeX - dataX);
    if (angle < 0) angle += 2 * M_PI;  // MODIFY: Adjust angle to [0, 2π)
        range
    return angle;
}

...
```

Subpopulation

```
# human expert

...

    std::vector<size_t> byCost(size());
    std::iota(byCost.begin(), byCost.end(), 0);
    std::stable_sort(
        byCost.begin(),
        byCost.end(),
        [&](size_t a, size_t b)
        {
            return costEvaluator.penalisedCost(*items_[a].solution)
                   < costEvaluator.penalisedCost(*items_[b].solution);
        });
    // clang-format on

    std::vector<std::pair<double, size_t>> diversity;
    for (size_t costRank = 0; costRank != size(); costRank++)
    {
        auto const dist = items_[byCost[costRank]].avgDistanceClosest();
        diversity.emplace_back(-dist, costRank);  // higher is better
    }

    std::stable_sort(diversity.begin(), diversity.end());

    auto const popSize = static_cast<double>(size());
    auto const numElite = std::min(params.numElite, size());
    auto const divWeight = 1 - numElite / popSize;

    for (size_t divRank = 0; divRank != size(); divRank++)
    {
        auto const costRank = diversity[divRank].second;
        auto const idx = byCost[costRank];
        items_[idx].fitness = (costRank + divWeight * divRank) / (2 *
            popSize);
    }
}

...


# llm optimized
...
    /*
        ------------------------------------------------------------------------

     * STEP 2: compute diversity rank (modified: normalize diversity
        values)
```

```
     * ---------------------------------------------------------------------
         */
    std::vector<std::pair<double, size_t>> diversity;
    for (size_t costRank = 0; costRank != size(); costRank++)
    {
        auto const dist = items_[byCost[costRank]].avgDistanceClosest();
        // MODIFY: Normalize diversity values between lbDiversity and
            ubDiversity
        auto normalizedDist = std::clamp(dist, params.lbDiversity, params
            .ubDiversity);
        diversity.emplace_back(-normalizedDist, costRank); // higher is
            better
    }
    std::stable_sort(diversity.begin(), diversity.end());

    /*
        ---------------------------------------------------------------------

     * STEP 3 (NEW): rank individuals by number of routes
                          *
     *              (fewer routes  -> better -> lower rank)
                        *
     * ---------------------------------------------------------------------
         */
    // MODIFY: Added route-count ranking to capture vehicle-usage quality
        .
    std::vector<size_t> byRoutes(size());
    std::iota(byRoutes.begin(), byRoutes.end(), 0);
    std::stable_sort(
        byRoutes.begin(),
        byRoutes.end(),
        [&](size_t a, size_t b)
        {
            return items_[a].solution->numRoutes()
                   < items_[b].solution->numRoutes();
        });

    // routeRankOfIdx[i] = rank of individual i in the route ranking.
    std::vector<size_t> routeRankOfIdx(size());
    for (size_t rank = 0; rank != byRoutes.size(); ++rank)
        routeRankOfIdx[byRoutes[rank]] = rank;

    /*
        ---------------------------------------------------------------------

     * STEP 4: combine the three ranks into the biased fitness
     * ---------------------------------------------------------------------
         */
    auto const popSize  = static_cast<double>(size());
    auto const numElite = std::min(params.numElite, size());

    // MODIFY: Adjust weights based on normalized diversity values
    std::vector<double> normalizedDiversities;
    for (const auto& [negDist, costRank] : diversity) {
        normalizedDiversities.push_back(-negDist);
    }
    double avgDiversity = std::accumulate(normalizedDiversities.begin(),
        normalizedDiversities.end(), 0.0) / normalizedDiversities.size();
    double diversityFactor = (avgDiversity - params.lbDiversity) / (
        params.ubDiversity - params.lbDiversity);
    auto baseDivWeight = 1.0 - numElite / popSize;
    auto baseRouteWeight = 0.5 * (1.0 - numElite / popSize);
    auto divWeight = baseDivWeight * (1.0 - diversityFactor) +
        baseDivWeight * 0.5 * diversityFactor;
    auto routeWeight = baseRouteWeight * diversityFactor;
```

```
    for (size_t divRank = 0; divRank != size(); divRank++)
    {
        auto const costRank  = diversity[divRank].second;
        auto const idx       = byCost[costRank];
        auto const routeRank = routeRankOfIdx[idx];

        // MODIFY: fitness now blends cost, diversity and route-count
            with dynamic weights.
        auto const denom =
            (1.0 /*cost*/ + divWeight + routeWeight) * popSize;

        items_[idx].fitness =
            (costRank + divWeight * divRank + routeWeight * routeRank) /
                denom;
    }
}

...
```