DyACE: Online LLM-Driven Design of Perturbative Heuristics via Dynamic Co-evolution

Anonymous ACL submission

Abstract

Large language models (LLMs) have shown significant promise as code-generating agents for automated heuristic design (AHD). However, existing methods typically synthesize a fixed perturbative heuristic offline and reuse it throughout the optimization process, failing to adapt to the changing search demands of an evolving solution population. We propose DyACE, an online LLM-driven framework that couples heuristic optimization with solution-population evolution in a closed loop. At each search stage, DyACE performs short-horizon probing rollouts to evaluate operator behaviors on the current population and extract trajectory features that reveal search bottlenecks. A two-stage LLM module then diagnoses these signals and synthesizes executable perturbation operators tailored to the current search state. The generated operators are deployed for a finite evolutionary horizon, after which DyACE re-observes the updated population and revises its heuristic logic accordingly. Experiments on job shop scheduling, traveling salesman, and capacitated vehicle routing benchmarks show that DyACE improves over representative static LLM-based AHD baselines under a unified rollout-evaluation budget. Ablation studies further confirm that both online revision and rollout-derived state feedback are necessary for robust performance gains.

1 Introduction

Heuristic search remains a central approach for solving complex combinatorial optimization problems (Pearl, 1983). This work focuses on *perturbative heuristics*, which improve complete candidate solutions through iterative search operators (Burke et al., 2013; Gendreau et al., 2010). This family includes widely used paradigms such as Evolutionary Algorithms (Bäck et al., 1997; Eiben and Smith, 2015) and Local Search (Lourenço et al., 2003). The effectiveness of such solvers depends heavily on the operators used to transform the current solution population, yet designing these operators often requires substantial domain expertise and trial-and-error (Blum and Roli, 2003). The No Free Lunch theorem (Wolpert and Macready, 2002) formalizes this limitation at the level of problem classes: averaged uniformly over all possible objective functions, no optimizer is universally superior. Practical heuristic design must therefore exploit the structure of the target problem class rather than seek a single universal solver.

Automated Heuristic Design (AHD) aims to reduce this manual design burden by automatically constructing heuristic solvers (Burke et al., 2013; Stützle and López-Ibáñez, 2018). Traditional AHD has often used Genetic Programming (GP) (Koza, 1994; Langdon and Poli, 2013) to evolve symbolic rules or executable heuristic programs, including for scheduling and related optimization domains (Zhang et al., 2023). However, GP-based program search can suffer from a semantic gap between random program edits and meaningful perturbative behavior (O'Neill et al., 2010), limiting its ability to express high-level operator logic. Recent Large Language Models (LLMs) (Naveed et al., 2025) offer a different route: they can synthesize executable programs from natural-language specifications (Chen, 2021; Austin et al., 2021) and participate in executable design loops where generated artifacts are evaluated by an external environment and revised through language-based reasoning (Ma et al., 2023; Hemberg et al., 2024; Nasir et al., 2024).

These capabilities have led to a growing line of LLM-driven AHD methods. FunSearch (Romera-Paredes et al., 2023) couples LLM-generated programs with evolutionary selection, EoH (Liu et al., 2024b) uses LLMs to evolve heuristic code, ReEvo (Ye et al., 2024) incorporates natural-language reflection into heuristic evolution, and MCTS-AHD (Zheng et al., 2025) strengthens design-space explo-

ration with Monte Carlo tree search. Despite their differences, these methods largely follow a *search-then-deploy* protocol: they spend an offline design budget to synthesize or select a strong heuristic, and then freeze the resulting logic during the target solving process.

This protocol is restrictive for perturbative search. A solution population does not remain in a fixed state: early search often requires broad exploration, stagnant phases may require disruptive moves, and later search typically benefits from more conservative exploitation (Eiben and Smith, 2015; DaCosta et al., 2008). A fixed generated operator cannot explicitly react to changes in improvement trend, population diversity, stagnation, or exploitation pressure. Thus, the key limitation of search-then-deploy LLM-AHD is not merely insufficient offline exploration. Even stronger offline search, such as tree-search-based heuristic discovery, still commits to a static operator at deployment time and cannot revise the executable logic according to the evolving search trajectory.

We propose *Dynamic Algorithm Co-evolution (DyACE)*, an online LLM-driven framework that treats perturbative heuristic design as a non-stationary closed-loop control problem. Instead of asking the LLM to generate one final heuristic, DyACE keeps LLM-based code generation inside the solving process. It adopts a receding-horizon control perspective (DQ, 2009): at each decision stage, Look-Ahead Rollout Search executes short-horizon rollouts from the current population and extracts Search Trajectory Features describing improvement, diversity, operator precision, and operator impact. A Diagnosis Agent converts these execution-grounded signals into a natural-language diagnosis of the current search bottleneck, and a Coding Agent synthesizes a stage-specific executable operator. The selected operator is deployed for a limited horizon before DyACE observes the updated population and revises the heuristic logic again. Unlike adaptive operator selection methods that choose among predefined operators (DaCosta et al., 2008), DyACE generates and rewrites operator code conditioned on the observed trajectory.

We evaluate DyACE on three representative combinatorial optimization benchmarks: Job Shop Scheduling Problem (JSSP), Traveling Salesman Problem (TSP), and Capacitated Vehicle Routing Problem (CVRP). Under a unified rollout-evaluation budget, DyACE improves over representative static LLM-driven AHD baselines, including FunSearch, EoH, ReEvo, and MCTS-AHD. Ablation studies further show that online revision and rollout-derived state feedback are both necessary: revising code without execution-grounded trajectory information can destabilize search, while static deployment cannot respond to changing population states.

Our main contributions are as follows:

- We formulate LLM-driven perturbative heuristic design as a non-stationary closed-loop control problem, making explicit the mismatch between fixed generated heuristics and evolving search states.
- We introduce DyACE, an online framework that grounds LLM meta-reasoning in short-horizon execution feedback and decouples search-state diagnosis from executable operator synthesis.
- We provide an empirical evaluation on JSSP, TSP, and CVRP under a unified rollout-evaluation budget, including comparisons with representative static LLM-driven AHD baselines and ablations that isolate the effects of online revision and rollout-based grounding.

2 Problem Formulation

2.1 Perturbative Search

We study LLM-driven AHD for *perturbative heuristics*, where a solver improves complete candidate solutions through iterative search operators (Burke et al., 2013; Gendreau et al., 2010). This scope differs from constructive optimization, where solutions are built sequentially from partial decisions. In our setting, the LLM-generated artifact is executable operator code that transforms an existing solution population.

Let $I \sim \mathcal{D}$ be a problem instance with solution space $\mathcal{X}_I$ and cost function f_I. At generation t, the solver maintains a population $\mathcal{P}_t = \{x_1^{(t)}, \ldots, x_N^{(t)}\} \in \mathcal{X}_I^N$. A perturbative operator $h_t \in \Omega$ updates the population through

$$\mathcal{P}_{t+1} = \Phi_I(\mathcal{P}_t, h_t; \xi_t), \quad t = 0, \ldots, T-1, \quad (1)$$

where Φ_I includes selection and variation, and ξ_t captures stochasticity. The trajectory objective is the best solution found during search, $J_I(\mathcal{P}_{0:T}) = \min_{t,i} f_I(x_i^{(t)})$. Thus, performance depends not only on the operator itself, but also on when it is applied to the evolving population.

2.2 Search-Then-Deploy LLM-AHD

Existing LLM-driven AHD methods, including FunSearch (Romera-Paredes et al., 2023), EoH (Liu et al., 2024b), ReEvo (Ye et al., 2024), and MCTS-AHD (Zheng et al., 2025), usually follow a *search-then-deploy* protocol: they search over a code space $\mathcal{C}$ offline and then deploy the selected code as a fixed heuristic. Let $\Gamma : \mathcal{C} \to \Omega$ map generated code to an executable operator. The static objective is

$$c^* = \operatorname*{argmin}_{c \in \mathcal{C}} \mathbb{E}_{I \sim \mathcal{D}, \xi} \left[J_I(\mathrm{Rollout}_I(\mathcal{P}_0, \Gamma(c); \xi)) \right], \quad h_t \equiv \Gamma(c^*), \quad t = 0, \ldots, T-1. \tag{2}$$

This open-loop formulation is compact but restrictive for perturbative search. Early generations may require exploratory variation, stagnant phases may require disruptive moves, and late generations often benefit from conservative exploitation (Eiben and Smith, 2015). A fixed operator cannot explicitly react to these changing search states.

2.3 Online Heuristic Design as Non-stationary Bi-level Control

We instead formulate LLM-driven perturbative heuristic design as a non-stationary bi-level control problem. The lower level evolves the population through Eq. (1), while the upper level observes the current trajectory and generates the operator for the next decision horizon. Let $s_t = \phi(\mathcal{P}_t, r_t)$ denote Search Trajectory Features extracted from the current population and recent rollout records. An LLM-based meta-controller π_θ then produces the operator $h_t = \pi_\theta(s_t)$, yielding the closed-loop objective

$$\pi_\theta^* = \operatorname*{argmin}_{\pi_\theta} \mathbb{E}_{I \sim \mathcal{D}, \xi} \left[J_I(\mathcal{P}_{0:T}) \right], \quad \text{s.t.} \quad \mathcal{P}_{t+1} = \Phi_I(\mathcal{P}_t, \pi_\theta(s_t); \xi_t) \tag{3}$$

This formulation changes the LLM from an offline code generator into an execution-grounded meta-controller. Unlike bandit-based adaptive operator selection, which selects among predefined operators (DaCosta et al., 2008), DyACE synthesizes or revises executable operator code conditioned on the observed search trajectory.

3 The DyACE Framework

3.1 Framework Overview

DyACE implements the closed-loop policy in Eq. (3) by keeping LLM-based code generation inside the solving process. Given the current solution population $\mathcal{P}_t$, DyACE observes short-horizon search behavior, generates an executable perturbative operator h_t, deploys it for a limited horizon H, and then repeats the procedure from the updated population $\mathcal{P}_{t+H}$. The generated code is therefore a stage-specific control action rather than a fixed heuristic reused throughout the entire run.

As shown in Figure 1, DyACE consists of three modules. First, **Look-Ahead Rollout Search** creates short-horizon shadow rollouts from $\mathcal{P}_t$ and summarizes their trajectories into Search Trajectory Features s_t, together with rollout scores for candidate operators. These signals describe recent improvement trends, population diversity, operator success, and improvement magnitude. Second, **Decoupled LLM Meta-Reasoning** maps the rollout-derived state to code through two LLM calls: a Diagnosis Agent produces a verbal diagnosis of the current search failure mode and intended intervention, and a Coding Agent turns this diagnosis into executable operator code. Third, **Receding-Horizon Evolution** applies the selected operator for H generations, updates the population, and returns control to the rollout module for the next decision stage.

Together, these modules make DyACE an execution-conditioned operator-synthesis loop, with the following subsections detailing the rollout observation, LLM reasoning, and receding-horizon update steps.

3.2 Look-Ahead Rollout Search

Before invoking the LLM, DyACE constructs an execution-grounded observation of the current search state through look-ahead Monte Carlo rollouts. At decision stage t, the framework starts from the current population $\mathcal{P}_t$ and a candidate operator pool $\mathcal{A}_t = \{h_k\}_{k=1}^K$. For each candidate operator h_k, DyACE creates M independent shadow environments initialized with the same $\mathcal{P}_t$ and runs the operator for a short rollout horizon H_r. These shadow rollouts are not committed to the main search trajectory. Instead, they estimate how different executable operator logics behave if they are applied to the current population state.

Formally, let

$$\mathcal{R}_t^{k,m} = \left\{ \mathcal{P}_{t+\ell}^{(k,m)} \right\}_{\ell=0}^{H_r} \tag{4}$$

denote the m-th rollout trajectory generated by applying h_k from $\mathcal{P}_t$. DyACE summarizes the collec-

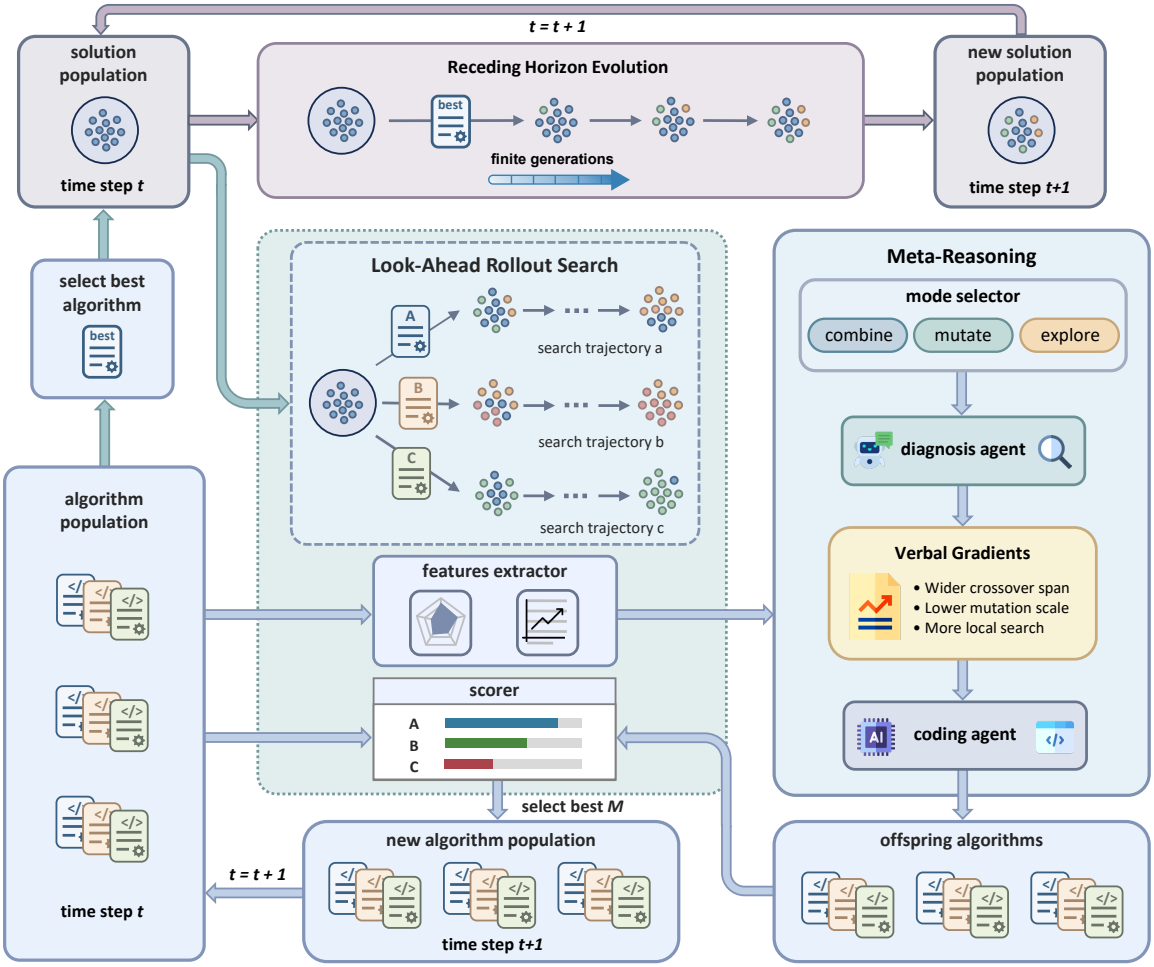

Figure 1: Overview of DyACE. At each decision stage, DyACE runs short-horizon rollouts from the current population, extracts Search Trajectory Features, converts them into verbal gradients through a Diagnosis Agent, and uses a Coding Agent to synthesize a stage-specific executable operator. The generated operator is deployed for a limited horizon before the loop observes the updated population and revises the heuristic logic.

tion of rollout trajectories into Search Trajectory Features

$$s_t = \phi\left(\mathcal{P}_t, \left\{\mathcal{R}_t^{k,m} \mid 1 \le k \le K,\ 1 \le m \le M\right\}\right), \tag{5}$$

which serves as the state representation passed to the LLM meta-reasoning module.

The feature groups are designed to describe both the solution population and the observed behavior of candidate operators. *Improvement trend* measures whether the best objective value keeps improving or has entered a plateau. *Population diversity* tracks the structural variety of the solution set and its rate of loss during the rollout. *Operator precision* records the fraction of offspring that improve over their parents, while *operator impact* measures the average improvement magnitude among successful updates. Together, these features convert raw execution traces into a compact, LLM-readable description of the current search phase, such as whether the population requires stronger exploration, escape from stagnation, or more conservative exploitation.

The same Monte Carlo rollouts also provide quantitative evidence for candidate comparison. Let $g(\mathcal{P})$ denote the optimality gap of the best solution in population $\mathcal{P}$. For each candidate operator h_k, DyACE computes its state-conditioned rollout score as

$$J_t(h_k) = \frac{1}{M}\sum_{m=1}^{M} g\left(\mathcal{P}_{t+H_r}^{(k,m)}\right), \tag{6}$$

where $\mathcal{P}_{t+H_r}^{(k,m)}$ is the population obtained by applying h_k in the m-th rollout. A lower $J_t(h_k)$ indicates better short-horizon performance from the current state. Because all candidates are evaluated from the same $\mathcal{P}_t$, this iso-state comparison reduces noise from different initial populations and gives the subsequent LLM generation step an execution-based

filter rather than relying only on natural-language plausibility.

3.3 Decoupled LLM Meta-Reasoning

Given the rollout-derived state from Section 3.2, DyACE separates search-state diagnosis from code synthesis. Directly prompting the LLM to write code from raw trajectory statistics is brittle: similar numeric patterns may call for different interventions under different population states, and unconstrained generation can modify irrelevant parts of the operator. DyACE therefore first converts rollout evidence into a natural-language diagnosis, and then uses this diagnosis to constrain executable code generation.

DyACE samples from three editing modes with preset probabilities to balance reuse and novelty in the operator pool. *Combine* provides two parent operators and asks for a hybrid that preserves compatible useful mechanisms from both. *Mutate* provides one parent and asks for a local revision of its code structure, control parameters, or search intensity while retaining the main strategy. *Explore* relaxes dependence on existing parents and encourages a structurally different operator, increasing the chance of introducing new search behavior.

Under the sampled mode, the Diagnosis Agent receives the Search Trajectory Features s_t, rollout scores $\{J_t(h_k)\}_{k=1}^{K}$, and summaries of the relevant operators. It produces a verbal gradient v_t (Ye et al., 2024) that specifies the intended revision. In DyACE, this verbal gradient records the observed search bottleneck, the rollout evidence supporting it, and the operator-level change to be made. For example, low population diversity together with low operator precision may lead to a recommendation for stronger structural mixing, while stable but slow improvement may call for a more exploitation-oriented revision.

The Coding Agent then uses v_t, the sampled editing mode, the relevant parent operators, and the required operator interface to synthesize candidate code. The generated candidate is not accepted solely because it is plausible in text: it returns to rollout evaluation and pool selection before it can affect the main search. DyACE thus uses language reasoning to structure code generation while keeping execution feedback as the acceptance signal.

3.4 Receding-Horizon Algorithm Evolution

After meta-reasoning generates candidate operators, DyACE evaluates them through the same rollout mechanism used for state observation. The candidates are inserted into the operator pool and compared under the current population state using rollout scores. Based on this evaluation, DyACE selects an operator h_t for deployment. The selected operator is not treated as a final heuristic for the whole run; it is only used as the control action for the next deployment horizon.

Starting from $\mathcal{P}_t$, DyACE applies h_t for H generations:

$$\mathcal{P}_{t+\ell+1} = \Phi_I(\mathcal{P}_{t+\ell}, h_t; \xi_{t+\ell}), \quad \ell = 0, \ldots, H-1. \tag{7}$$

After this limited deployment, the population becomes $\mathcal{P}_{t+H}$. DyACE then returns to the rollout module, extracts new Search Trajectory Features from the updated population, and starts another round of meta-reasoning. The operator sequence is therefore adaptive over the solving trajectory: h_t is selected for the current horizon, while later horizons may use revised or entirely different operators.

The operator pool is maintained across decision stages: high-performing operators remain available as parents for later Combine or Mutate steps, and newly generated candidates enter the pool only after rollout evaluation. This pool-based update reuses useful code while still allowing later stages to revise operator logic as the population changes.

4 Experiments

4.1 Experimental Settings

4.1.1 Benchmarks and Datasets

We evaluate DyACE on three standard NP-hard Combinatorial Optimization Problems (COPs) to test its performance across different problem types.

Job Shop Scheduling Problem (JSSP). JSSP involves scheduling jobs on machines with precedence constraints to minimize the total completion time (Makespan). This problem represents a highly constrained discrete optimization challenge. We use the standard Taillard Benchmark (Taillard, 1993), selecting 16 instances ranging from medium (15×15) to large scale (100×20). These instances vary in complexity, making them a standard test for measuring the scalability of scheduling algorithms.

Traveling Salesman Problem (TSP). TSP aims to find the shortest Hamiltonian cycle visiting a set of cities exactly once. It tests whether the generated operators can handle permutation-structured search spaces over Euclidean instances. We select

6 classic instances from TSPLIB (Reinelt, 1991) with scales ranging from 51 to 200 nodes. These instances feature distinct node distributions, testing the robustness of the evolved algorithms in continuous Euclidean spaces.

Capacitated Vehicle Routing Problem (CVRP). CVRP extends TSP by adding vehicle capacity constraints, aiming to minimize travel distance while meeting customer demands. We employ the CMT dataset from CVRPLIB (Uchoa et al., 2017), selecting 6 instances with problem sizes between 50 and 199 nodes. This benchmark tests the algorithm's ability to balance objective optimization with strict feasibility constraints.

4.1.2 Compared Methods

To evaluate the efficacy of DyACE, we benchmark it against a diverse set of baselines, including GP, Gene Expression Programming (GEP) (Nie et al., 2013), heuristic dispatching rules (LSO, SPT/TWKR), and representative Deep Reinforcement Learning (DRL) models (Chen et al., 2022; Liu et al., 2024a; Zhang et al., 2020). Most importantly, we compare against representative LLM-based AHD frameworks: FunSearch (Romera-Paredes et al., 2023), EoH (Liu et al., 2024b), ReEvo (Ye et al., 2024), and MCTS-AHD (Zheng et al., 2025). These methods represent search-then-deploy algorithm discovery protocols and serve as the primary reference for validating our online, state-conditioned co-evolutionary approach.

4.1.3 Experimental Protocols.

To ensure a fair comparison, we enforce a unified computational budget of $B = 300$ algorithm evaluations and fix the solution population size at $N = 100$ for all methods. All LLM-based approaches use `GPT-4o-mini` as the reasoning engine. Since perturbative heuristic quality must be measured by executing the candidate operator on an evolving solution population, we treat rollout-based algorithm evaluation as the primary unit of computational budget; this controls the dominant evaluation cost shared by DyACE and the baselines. While search-then-deploy baselines use standard offline evolution, DyACE operates under an online co-evolution protocol. Detailed experimental settings and the computational-fairness analysis are provided in Appendix B.

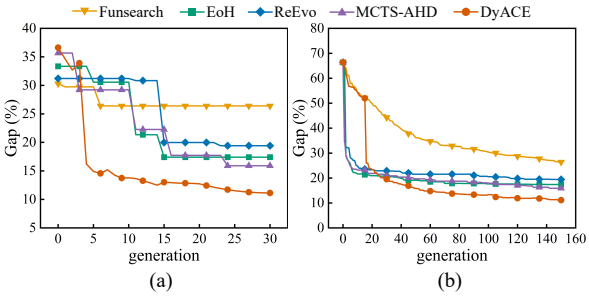

Figure 2: Search dynamics on JSSP instance TA71. (a) Best gap achieved by candidate heuristics during the algorithm-design process under the shared evaluation budget. (b) Best solution gap obtained during the final solution-evolution trajectory. Lower is better.

4.2 Main Results on Job Shop Scheduling (JSSP)

Table 1 reports the optimality gaps on the Taillard JSSP benchmark. DyACE obtains the lowest gap on all 16 evaluated instances, with an average gap of 15.30%. Among the static LLM-driven AHD baselines, MCTS-AHD is the strongest on average, reducing the gap to 19.58% compared with ReEvo at 21.04%, EoH at 21.40%, and FunSearch at 24.88%. This comparison indicates that strengthening offline heuristic search improves the static baseline, but still leaves a clear gap to online state-conditioned heuristic revision.

The per-instance results show the same pattern across problem scales. On medium-size instances such as TA21 and TA31, MCTS-AHD improves over earlier LLM-AHD baselines, but DyACE further reduces the gap from 18.70% to 15.23% on TA21 and from 18.99% to 15.14% on TA31. On larger instances, the benefit of online adaptation remains visible: on TA71 (100×20), DyACE reaches 11.13%, while the closest static LLM-AHD baseline, MCTS-AHD, remains at 15.90%. These results suggest that the advantage of DyACE is not only due to generating stronger heuristic code, but also to revising that code according to the evolving search state.

Figure 2 further compares the design and solving trajectories on TA71. In panel (a), search-then-deploy baselines, including FunSearch, EoH, ReEvo, and MCTS-AHD, improve candidate heuristics within an offline design process before final deployment. MCTS-AHD provides a stronger static trajectory than earlier evolutionary LLM-AHD baselines, reflecting the benefit of tree-search exploration, but it still commits to a fixed heuristic during deployment. DyACE instead keeps

Cases	Size	GP	GEP	DRL_Chen	DRL_Zhang	DRL_Liu	LSO	SPT/TWKR	FunSearch	EoH	ReEvo	MCTS-AHD	DyACE
TA01	15×15	25.67	25.67	38.99	16.41	21.20	58.98	35.17	13.81	14.13	11.78	10.56	**6.66**
TA02	15×15	25.80	19.45	31.75	24.12	14.55	41.40	23.63	13.26	11.98	10.85	11.82	**8.36**
TA11	20×15	28.89	34.05	35.08	32.20	29.11	63.30	38.98	21.23	20.27	20.78	19.68	**16.95**
TA12	20×15	30.87	26.70	29.11	32.04	23.77	59.99	44.04	18.95	19.17	16.75	18.14	**14.56**
TA21	20×20	27.28	27.22	30.63	37.15	27.71	61.21	34.35	20.95	21.07	19.98	18.70	**15.23**
TA22	20×20	28.69	23.81	25.94	31.37	20.25	57.63	31.94	22.19	21.75	20.31	21.19	**16.31**
TA31	30×15	30.78	29.20	35.03	45.41	29.08	40.48	38.04	23.24	19.73	21.71	18.99	**15.14**
TA32	30×15	34.53	34.30	37.78	33.86	23.49	47.65	40.81	27.64	25.95	25.34	25.62	**20.24**
TA41	30×20	42.19	36.11	26.73	33.02	34.56	43.29	44.54	30.03	29.13	26.83	25.89	**21.55**
TA42	30×20	34.38	34.90	42.59	37.53	35.42	59.83	45.22	31.34	30.82	26.90	26.90	**21.68**
TA51	50×15	30.54	32.90	36.30	30.40	30.72	39.28	36.52	25.91	21.23	21.92	19.35	**16.96**
TA52	50×15	21.41	20.61	27.39	21.23	27.87	34.80	30.19	25.44	19.67	26.67	18.69	**14.73**
TA61	50×20	28.49	26.99	26.67	27.41	23.71	46.03	30.82	33.23	27.86	24.02	23.61	**17.78**
TA62	50×20	26.73	29.77	29.38	29.73	23.98	46.99	36.81	37.50	23.91	25.86	25.90	**19.17**
TA71	100×20	15.39	14.90	15.68	18.08	15.10	23.61	22.71	26.39	17.40	19.42	15.90	**11.13**
TA72	100×20	11.48	8.57	27.81	9.92	15.85	28.82	22.58	26.98	18.34	17.47	12.33	**8.43**
Avg	–	27.70	26.57	31.05	28.74	24.77	47.08	34.77	24.88	21.40	21.04	19.58	**15.30**

Table 1: Optimality gaps on the Taillard JSSP benchmark. Lower is better.

candidate evaluation state-conditioned throughout the solving process, allowing the algorithm-design curve to keep responding to the current population state.

Panel (b) shows the corresponding solution trajectory. The static baselines tend to flatten once their deployed heuristic exhausts its useful search behavior, whereas DyACE continues to reduce the solution gap through periodic operator revision. This comparison connects the aggregate result in Table 1 to the mechanism of DyACE: the gain over MCTS-AHD is not merely due to a larger offline search space, but to online execution-grounded code revision.

4.3 Generalization to Routing Problems

To examine whether the same online code-revision mechanism transfers beyond scheduling, we further evaluate DyACE on two routing benchmarks: TSP and CVRP. For both domains, we use a restricted perturbative search space with only crossover and mutation operators. This setting limits the influence of hand-engineered local search routines and makes the comparison focus on the quality of LLM-generated operator logic.

4.3.1 Traveling Salesman Problem (TSP)

Table 2 reports the optimality gaps on six TSPLIB instances. DyACE achieves the lowest or tied-lowest gap on all instances, reaching zero gap on five instances and retaining the smallest gap on the largest instance, kroB200. The static LLM-AHD baselines are competitive on smaller instances, but their performance becomes less stable as the instance size increases.

The detailed results show two useful patterns. First, the strongest static baselines can already solve some smaller instances: ReEvo and DyACE both reach 0.00% on eil51 and st70, while MCTS-AHD also reaches 0.00% on st70. Second, as the instances become larger, the gap between static deployment and online revision becomes clearer. On rd100, bier127, and u159, DyACE reaches 0.00%, while ReEvo and MCTS-AHD retain nonzero gaps. On kroB200, DyACE obtains a gap of 6.38%, compared with 16.19% for ReEvo and 16.48% for MCTS-AHD. These results suggest that online operator revision is useful even when the operator space is restricted to basic permutation variation mechanisms.

Method	eil51	st70	rd100	bier127	u159	kroB200
FunSearch	2.35	9.93	50.83	61.98	157.13	218.22
EoH	0.47	3.70	13.65	5.25	3.09	19.90
ReEvo	**0.00**	**0.00**	0.61	1.71	2.58	16.19
MCTS-AHD	2.35	**0.00**	15.28	3.54	2.41	16.48
DyACE	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**6.38**

Table 2: Optimality gaps on TSPLIB instances using only crossover and mutation operators. Lower is better.

4.3.2 Capacitated Vehicle Routing Problem (CVRP)

Table 3 summarizes the results on the CVRPLIB CMT instances. CVRP introduces capacity constraints in addition to route-length minimization, making the generated operators responsible for balancing objective improvement and feasibility preservation. DyACE obtains the lowest gap on all six instances, while the strongest static LLM-AHD baseline varies across instances.

The CVRP results further separate online revision from stronger offline exploration. MCTS-AHD is competitive on CMT3, where it is the best static baseline with a 16.51% gap, but DyACE fur-

Method	CMT1	CMT2	CMT3	CMT11	CMT4	CMT5
FunSearch	25.58	47.33	74.85	128.46	134.47	166.82
EoH	17.04	33.49	19.15	33.01	20.62	168.84
ReEvo	12.57	41.27	17.12	21.46	18.95	12.96
MCTS-AHD	20.46	33.52	16.51	22.57	18.97	21.61
DyACE	**3.17**	**7.57**	**12.21**	**15.76**	**16.14**	**9.19**

Table 3: Optimality gaps on CVRPLIB CMT instances using only crossover and mutation operators. Lower is better.

Method	**TA21** (20×20)	**TA51** (50×15)	**TA71** (100×20)
DyACE-w/o-Both	20.95	26.41	20.20
DyACE-Blind	17.36	21.45	19.60
DyACE-Static	16.63	19.17	17.94
DyACE	**15.23**	**16.96**	**11.13**

Table 4: Ablation study on JSSP instances. Values are optimality gaps; lower is better.

ther reduces the gap to 12.21%. On CMT2, DyACE reaches 7.57%, while all static baselines remain above 33%. On CMT5, ReEvo is the strongest static baseline at 12.96%, but DyACE still improves the gap to 9.19%.

Overall, the routing experiments support the same conclusion as the JSSP results. Static LLM-AHD baselines can be competitive on particular instances, and tree-search-based offline exploration helps in some cases, but no fixed heuristic remains consistently strongest across the routing benchmarks. DyACE instead revises operator code as the solution population evolves, which is especially useful when the search must balance exploration, exploitation, and feasibility constraints under a restricted operator interface.

4.4 Ablation Study

We conduct an ablation study on three JSSP instances to isolate the two main components of DyACE: online receding-horizon revision and rollout-derived Search Trajectory Features. We compare the full method with three variants. *DyACE-Static* removes the online revision loop after heuristic design, so the selected operator is deployed as a fixed heuristic. *DyACE-Blind* keeps the online revision loop but removes Search Trajectory Features, forcing the LLM to revise operators without structured execution feedback from look-ahead rollouts. *DyACE-w/o-Both* removes both components.

Table 4 shows that both components contribute to performance. Compared with DyACE-Static, the full method reduces the gap from 16.63% to 15.23% on TA21, from 19.17% to 16.96% on TA51, and from 17.94% to 11.13% on TA71. The improvement grows with instance scale, suggesting that deployment-time operator revision is especially useful when the search trajectory becomes longer and more non-stationary.

The comparison with DyACE-Blind highlights the role of rollout-grounded state information. Although DyACE-Blind can revise operators online, it performs worse than DyACE-Static on TA71 (19.60% vs. 17.94%). This result suggests that online revision without structured trajectory feedback can introduce poorly targeted code changes, particularly on larger instances. In contrast, DyACE conditions each revision on Search Trajectory Features extracted from look-ahead rollouts, which helps the LLM distinguish whether the current population requires stronger exploration, stagnation escape, or more conservative exploitation.

Finally, DyACE-w/o-Both gives the weakest results across all three instances, showing that neither static heuristic deployment nor ungrounded online revision is sufficient on its own. The full DyACE framework performs best because it combines the two ingredients: it keeps the LLM in the solving loop, and it grounds each code revision in execution feedback from the current population state.

5 Conclusion

This paper studied LLM-driven automated heuristic design as an online, execution-grounded process. Rather than treating the LLM as a one-shot generator of a fixed heuristic before deployment, we framed perturbative heuristic design as non-stationary operator synthesis over an evolving solution population. We introduced DyACE, which formulates this setting as non-stationary bi-level control: it extracts Search Trajectory Features from short-horizon rollouts, converts them into language-based diagnoses, and synthesizes stage-specific executable operators for receding-horizon deployment. This keeps LLM code generation inside the solving loop while grounding each revision in execution feedback.

Experiments on JSSP, TSP, and CVRP show that DyACE improves over search-then-deploy LLM-AHD baselines, including MCTS-AHD. Ablations show that stable gains require both online revision and rollout-derived state features. These findings suggest that LLM-based algorithm design should account for execution-time revision, not only code generation.

Limitations

Scope of Heuristic Design. This work focuses on LLM-driven design of *perturbative heuristics*, where generated operators modify complete candidate solutions during iterative search. The conclusions therefore do not directly extend to constructive heuristics, exact optimization algorithms, or heavily engineered domain-specific solvers. For TSP and CVRP, we intentionally restrict the operator interface to crossover and mutation operators to isolate the quality of generated perturbative logic, rather than comparing against full-featured routing solvers with specialized local search and repair components.

Cross-instance Reuse. DyACE performs online operator revision for each target instance using execution feedback from the current solution population. This design is useful for adapting to instance-specific search dynamics, but the present study does not systematically evaluate how generated operators, trajectory diagnoses, or revision strategies transfer across unseen instances. A natural extension is to cache and reuse successful operator logic or verbal diagnoses across related instances, reducing repeated design effort while preserving the online adaptation mechanism.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

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

André Biedenkapp, H Furkan Bozkurt, Theresa Eimer, Frank Hutter, and Marius Lindauer. 2020. Dynamic algorithm configuration: Foundation of a new meta-algorithmic framework. In *ECAI 2020*, pages 427–434. IOS Press.

Christian Blum and Andrea Roli. 2003. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308.

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

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ruiqi Chen, Wenxin Li, and Hongbing Yang. 2022. A deep reinforcement learning framework based on an attention mechanism and disjunctive graph embedding for the job-shop scheduling problem. *IEEE Transactions on Industrial Informatics*, 19(2):1322–1331.

Pietro A Consoli, Yi Mei, Leandro L Minku, and Xin Yao. 2016. Dynamic selection of evolutionary operators based on online learning and fitness landscape analysis. *Soft Computing*, 20(10):3889–3914.

Luis DaCosta, Alvaro Fialho, Marc Schoenauer, and Michèle Sebag. 2008. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 913–920.

Rawlings JB Mayne DQ. 2009. Model predictive control: theory and design.

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

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*.

Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michele Sebag. 2010. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 60(1):25–64.

Michel Gendreau, Jean-Yves Potvin, and 1 others. 2010. *Handbook of metaheuristics*, volume 2. Springer.

Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. 2024. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21.

Holger H Hoos and Thomas Stützle. 2018. Stochastic local search. In *Handbook of approximation algorithms and metaheuristics*, pages 297–307. Chapman and Hall/CRC.

Jin Huang, Xinyu Li, Liang Gao, Qihao Liu, and Yue Teng. 2024. Automatic programming via large language models with population self-evolution for dynamic job shop scheduling problem. *arXiv preprint arXiv:2410.22657*.

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

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

John R Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112.

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

Kefan Li, Yuan Yuan, Hongyue Yu, Tingyu Guo, and Shijie Cao. 2025. Cocoevo: Co-evolution of programs and test cases to enhance code generation. *IEEE Transactions on Evolutionary Computation*.

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

Chien-Liang Liu, Chun-Jan Tseng, and Po-Hao Weng. 2024a. Dynamic job-shop scheduling via graph attention networks and deep reinforcement learning. *IEEE Transactions on Industrial Informatics*, 20(6):8662–8672.

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

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

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

Helena R Lourenço, Olivier C Martin, and Thomas Stützle. 2003. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*.

Yi Mei, Qi Chen, Andrew Lensen, Bing Xue, and Mengjie Zhang. 2022. Explainable artificial intelligence by genetic programming: A survey. *IEEE Transactions on Evolutionary Computation*, 27(3):621–641.

Muhammad Umair Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. 2024. Llmatic: neural architecture search via large language models and quality diversity optimization. In *proceedings of the Genetic and Evolutionary Computation Conference*, pages 1110–1118.

Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2025. A comprehensive overview of large language models. *ACM Transactions on Intelligent Systems and Technology*, 16(5):1–72.

Li Nie, Liang Gao, Peigen Li, and Xinyu Shao. 2013. Reactive scheduling in a job shop where jobs arrive over time. *Computers & Industrial Engineering*, 66(2):389–405.

Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. 2010. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3):339–363.

Judea Pearl. 1983. Heuristics: intelligent search strategies for computer problem solving.

Jiyuan Pei, Jialin Liu, and Yi Mei. 2024. Learning from offline and online experiences: A hybrid adaptive operator selection framework. In *Proceedings of the genetic and evolutionary computation conference*, pages 1017–1025.

Jiyuan Pei, Yi Mei, Jialin Liu, Mengjie Zhang, and Xin Yao. 2025. Adaptive operator selection for metaheuristics: A survey. *IEEE Transactions on Artificial Intelligence*.

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

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

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Kumar, Emilien Dupont, Francisco Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2023. Mathematical discoveries from program search with large language models. *Nature*, 625.

Thomas Stützle and Manuel López-Ibáñez. 2018. Automated design of metaheuristic algorithms. In *Handbook of metaheuristics*, pages 541–579. Springer.

Eric Taillard. 1993. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285.

Dirk Thierens. 2005. An adaptive pursuit strategy for allocating operator probabilities. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1539–1546.

Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. 2017. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858.

Niki Van Stein and Thomas Bck. 2024. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*.

David H Wolpert and William G Macready. 2002. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.

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

Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Xu Chi. 2020. Learning to dispatch for job shop scheduling via deep reinforcement learning. *Advances in neural information processing systems*, 33:1621–1632.

Fangfang Zhang, Yi Mei, Su Nguyen, and Mengjie Zhang. 2023. Survey on genetic programming and machine learning techniques for heuristic design in job shop scheduling. *IEEE Transactions on Evolutionary Computation*, 28(1):147–167.

Qi Zhao, Qiqi Duan, Bai Yan, Shi Cheng, and Yuhui Shi. 2023. Automated design of metaheuristic algorithms: A survey. *arXiv preprint arXiv:2303.06532*.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. 2025. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. *arXiv preprint arXiv:2501.08603*.

A Related Work

A.1 LLMs for Executable Code Generation and Algorithm Design

LLMs have been applied beyond text generation to produce executable programs that are evaluated by external environments and iteratively revised through language-based feedback (Ma et al., 2023; Hemberg et al., 2024; Nasir et al., 2024). Yang et al. (2023) show that LLMs can act as black-box optimizers by generating and refining candidate solutions from natural-language descriptions of past performance. PromptBreeder (Fernando et al., 2023) extends this to self-referential prompt evolution, while Liu et al. (2024c) demonstrate LLMs as evolutionary optimizers for combinatorial problems. These executable design loops motivate LLM-driven AHD, where the generated artifact is algorithmic code whose quality is judged by running it on optimization instances.

A common branch of LLM-driven AHD uses the LLM as a proposal or variation mechanism within an external search process. FunSearch (Romera-Paredes et al., 2023) pairs an LLM with an island-based evolutionary framework to discover programs with strong empirical or mathematical scores. EoH (Liu et al., 2024b) uses LLMs to generate and recombine heuristic code across a population, while ReEvo (Ye et al., 2024) introduces verbal reflection, where the LLM receives natural-language summaries of relative performance and uses them to guide code revision. Related LLM-based automatic programming has also been studied for dynamic job shop scheduling (Huang et al., 2024). LLaMEA (Van Stein and Bck, 2024) automates the generation of full metaheuristic algorithms. More recent work explores program-test co-evolution for code generation (Li et al., 2025) or co-evolution of algorithms and the language model itself (Huang et al., 2025). MCTS-AHD (Zheng et al., 2025) replaces evolutionary search with Monte Carlo tree search to improve exploration coverage across the heuristic design space.

For heuristic synthesis, most of these LLM-driven AHD methods still follow a search-then-deploy structure: an offline design phase produces a heuristic, which is then applied without further revision during the target solving process. DyACE departs from this protocol by keeping the LLM in a closed loop with the evolving solution population.

A.2 Automated Heuristic Design

AHD encompasses a range of approaches for automatically constructing or configuring heuristic solvers (Stützle and López-Ibáñez, 2018). Algorithm configuration methods such as ParamILS (Hutter et al., 2009) and irace (López-Ibáñez et al., 2016) tune parameters within fixed algorithmic frameworks but do not generate new operator logic. Hyper-heuristics (Burke et al., 2013; Pillay and Qu, 2018) operate at a higher level, selecting or generating low-level heuristics from a predefined set. GP (Koza, 1994; Langdon and Poli, 2013) goes further by evolving executable heuristic programs from scratch, and has been widely applied to scheduling and routing domains (Zhang et al., 2023). However, GP-based search can produce syntactically valid but behaviorally weak heuristics due to the semantic gap between random program edits and meaningful search behavior (O'Neill et al., 2010; Mei et al., 2022), and the resulting programs are often difficult to interpret (Mei et al., 2022). LLMs can partially alleviate this issue by generating semantically coherent code and using natural-language reasoning to guide program search, as described in the previous subsection.

A.3 Perturbative Heuristics and Search Dynamics

Perturbative heuristics form a broad family of methods that improve complete candidate solutions through iterative search operators (Gendreau et al., 2010; Blum and Roli, 2003). This family includes stochastic local search (Hoos and Stützle, 2018), iterated local search (Lourenço et al., 2003), and population-based evolutionary algorithms (Eiben and Smith, 2015). Classic examples of perturbative operators include k-opt moves for TSP (Lin and Kernighan, 1973) and crossover and mutation operators for scheduling problems (Bäck et al., 1997). The effectiveness of these methods depends critically on the operators applied at each stage of the search: early phases typically benefit from broad exploration to avoid premature convergence, while later phases require more focused exploitation of promising regions (Eiben and Smith, 2015). This non-stationarity means that a single fixed operator will be mismatched to at least some phases of the trajectory, motivating the need for adaptive operator control.

A.4 Adaptive Operator Selection

Adaptive operator selection (AOS) addresses the problem of dynamically choosing among a fixed set of operators based on online performance feedback (DaCosta et al., 2008; Pei et al., 2025). Classical AOS methods formulate operator selection as a multi-armed bandit problem, assigning credit to operators based on the fitness improvements they produce (DaCosta et al., 2008; Thierens, 2005; Fialho et al., 2010). Later work showed that richer state information can improve selection quality: fitness landscape features extracted during search can be used to condition operator choice on the current search geometry (Consoli et al., 2016). More recent AOS work combines offline and online experience to improve selection policies (Pei et al., 2024).

A related line, Dynamic Algorithm Configuration (DAC) (Biedenkapp et al., 2020), adjusts algorithm parameters at runtime rather than selecting perturbative operators. Both AOS and DAC assume a predefined action space: AOS chooses among existing operators, while DAC changes configurations within a fixed algorithmic framework. DyACE differs in that the LLM generates, combines, and rewrites operator code in response to Search Trajectory Features extracted from short-horizon rollouts. The action space is therefore operator synthesis rather than operator selection or parameter control, making DyACE a generative extension of online operator control.

B Evaluation Protocol and Computational Fairness

B.1 Shared Experimental Setup

We executed experiments on a single node equipped with an AMD EPYC 9654 CPU and employed `GPT-4o-mini` as the underlying reasoning engine for all LLM-based methods. For the traditional GP and GEP baselines, we followed the experimental setup described in (Huang et al., 2024). To ensure a fair comparison at the solution level, we standardized the lower-level configuration across all methods. Specifically, the solution population size was fixed at $N = 100$, and we initialized these populations randomly to guarantee an unbiased starting point. The universal objective for all methods was to minimize the optimality gap of the best-found solution. An anonymized implementation is available at https://anonymous.4open.science/r/DyACE-0952.

B.2 Algorithm Space and Operator Interface

All LLM-based methods are evaluated under the same task-specific operator interface. The generated artifact is executable perturbative operator code, while the outer solver wrapper is fixed across methods. In particular, the LLM is not allowed to change the population size, objective function, instance parser, budget counter, rollout horizon, or final evaluation metric. This design makes the comparison focus on the quality of the generated operator logic rather than on differences in surrounding solver engineering.

The operator space is defined separately for each problem domain. For the Job Shop Scheduling Problem (JSSP), candidate heuristics are generated within a Memetic Algorithm template consisting of three operator slots: CROSSOVER, MUTATION, and LOCALSEARCH. This space allows the generated heuristic to combine population-level exploration with local refinement, which is important for scheduling instances with strong precedence and machine-usage constraints. For the Traveling Salesman Problem (TSP) and Capacitated Vehicle Routing Problem (CVRP), we use a stricter pure Evolutionary Algorithm template with only CROSSOVER and MUTATION. We exclude auxiliary local search operators in these two domains so that the evaluation depends more directly on the evolved perturbative logic rather than on a strong hand-engineered improvement routine.

The same interface is used by DyACE and all static baselines, including FunSearch, EoH, ReEvo, and MCTS-AHD. A generated operator must follow the required function signature, consume the current population or parent solutions provided by the solver wrapper, and return candidate offspring or an updated population in the problem-specific representation. The wrapper then evaluates the resulting population with the same objective function and rollout protocol for all methods.

We also apply the same validation wrapper to all generated programs. Code that fails to compile, violates the required function signature, or produces invalid outputs when detected is rejected and does not enter final deployment. We do not introduce method-specific repair rules or additional domain-specific post-processing for any baseline. This keeps the implementation protocol aligned across methods while avoiding claims about a universal feasibility-repair mechanism.

B.3 Rollout-based Evaluation Budget

The main computational cost in this work comes from evaluating executable perturbative heuristics, rather than from counting how many pieces of code are generated. Perturbative heuristics operate on complete candidate solutions and improve them through repeated search operators (Burke et al., 2013; Gendreau et al., 2010). Their quality is therefore trajectory-dependent: a candidate operator cannot be reliably judged from its textual form, from a syntactic inspection of the generated code, or from a small number of isolated local moves. Its effect becomes meaningful only after it is executed on an actual solution population, where improvement, diversity loss, stagnation, and exploitation behavior unfold over multiple generations.

This property is not introduced by DyACE; it is inherent to perturbative heuristic design. Automated heuristic design must assign scores to candidate heuristic logic (Stützle and López-Ibáñez, 2018; Zhao et al., 2023), and for perturbative operators this score is obtained by running the operator inside a solver trajectory. As a result, the expensive operation is executing generated heuristic code on a population and observing the resulting search trajectory. This also limits the feasibility of large-scale generalization training in this setting: training a perturbative heuristic generator over many large instances would require repeatedly evaluating many candidate operators through population-level rollouts, rather than collecting cheap labels from static examples. We therefore focus on instance-specific perturbative heuristic design under a controlled execution budget.

We use rollout-based algorithm evaluation as the basic unit of this budget. In one such evaluation, a candidate heuristic is executed on the current or initial solution population for a fixed rollout horizon, and its quality is measured by the resulting best-found optimality gap. This unit is shared by all compared LLM-based AHD methods. For search-then-deploy baselines, each candidate heuristic is evaluated by running it through a full evolutionary rollout before the best discovered heuristic is selected for final deployment. For DyACE, candidate heuristics are evaluated from the current population state through look-ahead rollouts, so that the evaluation reflects the non-stationary search stage in which the heuristic may be deployed.

For this reason, we use a unified rollout-evaluation budget of $B = 300$ algorithm evaluations for every method on each problem instance. This budget controls the number of candidate heuristics that can be assessed through execution, which is the dominant cost in perturbative heuristic design. It also avoids a misleading comparison based only on outer-loop generation counts: one algorithm-design generation may contain different numbers of candidate evaluations across methods, and therefore does not correspond to the same computational cost.

We do not use raw wall-clock time as the primary optimization objective because runtime is not a monotonic proxy for solution quality in perturbative search. A fast operator may terminate quickly while making little improvement to the solution population, whereas a slower operator may perform more useful local modifications and lead to a better final gap. The goal of our evaluation is therefore quality-first: all methods are compared under the same rollout-evaluation budget and the same final metric, while loose per-evaluation time caps are used only as safeguards against pathological candidates that run excessively long.

This protocol is also important for comparing DyACE with static LLM-AHD baselines. Static baselines evaluate each candidate heuristic through a full 150-generation rollout. DyACE instead uses a 30-generation short-horizon rollout for each online candidate evaluation. Thus, DyACE does not receive hidden extra computation from its online adaptation loop; under the same budget $B = 300$, its per-evaluation rollout horizon is only one fifth of the full rollout used by the static baselines. The online budget is spent on short, state-conditioned evaluations from the current population, not on additional full-horizon evaluations. Consequently, the online adaptation loop does not make DyACE more expensive in the dominant rollout-evaluation cost; it makes each candidate assessment shorter while preserving the execution feedback needed for perturbative heuristic design.

B.4 Protocols for Static Baselines

All static LLM-AHD baselines follow a search-then-deploy evaluation protocol. They first use the design budget to search for a candidate heuristic, and then freeze the best discovered heuristic for final deployment over the 150-generation solution evolution. To isolate the effect of the algorithm-design protocol rather than the choice of language model, all LLM-based methods in our experiments, including DyACE and all static baselines, use the

same backbone model, `GPT-4o-mini`. This protocol separates offline heuristic design from online solution evolution, in contrast to DyACE, which revises the heuristic during the solving trajectory.

FunSearch, EoH, and ReEvo are implemented as offline evolutionary search baselines. They use the same task-specific operator interface as DyACE, and every LLM call in these baselines is made to `GPT-4o-mini`. The design budget is fixed to $B = 300$ candidate heuristic evaluations for each problem instance. Each candidate heuristic is evaluated by executing it on the solution population for a full 150-generation rollout, and the resulting best-found optimality gap is used as its score. After the offline search finishes, the best candidate is selected as the final heuristic and is deployed without further revision.

MCTS-AHD is configured differently because it searches the heuristic design space through Monte Carlo tree search rather than an evolutionary framework. It also uses `GPT-4o-mini` for all LLM-generated candidate heuristics. We initialize its root with 4 candidate heuristic nodes and set $k = 2$ for tree expansion. The same budget $B = 300$ is used as the core constraint, where one budget unit corresponds to one evaluated candidate heuristic in the search tree. Thus, the budget controls the number of heuristic evaluations performed by the tree search, not the number of evolutionary generations. Each evaluated node is mapped to executable heuristic code through the same operator interface and is scored by the same 150-generation full rollout used for the other static baselines.

This setup ensures that all static baselines receive the same number of candidate heuristic evaluations and use the same final deployment horizon. The difference lies only in how they allocate the design budget: FunSearch, EoH, and ReEvo allocate it through evolutionary search, whereas MCTS-AHD allocates it through tree search.

B.5 DyACE Online Protocol

DyACE uses the same overall rollout-evaluation budget as the static baselines, but allocates it online along the solving trajectory. It also uses `GPT-4o-mini` as the LLM backbone and the same task-specific operator interface. The key difference is that DyACE evaluates and revises candidate operators from the current solution population, rather than completing all heuristic design before deployment.

We set the algorithm population size to $M = 5$ and run DyACE for 30 algorithm-design steps. At each step, DyACE first evaluates 5 existing or parent operators through look-ahead rollout search from the current population state. It then generates 5 new candidate heuristics through the Diagnosis Agent and Coding Agent, and evaluates these candidates with the same rollout-based scoring procedure. Therefore, each algorithm-design step uses 10 candidate heuristic evaluations, and the total evaluation count is

$$30 \times (5 + 5) = 300,$$

which exactly matches the budget $B = 300$ used by the static baselines.

The parent evaluations in DyACE are not hidden extra computation: they are included in the same $B = 300$ rollout-evaluation budget. These re-evaluations are needed because the usefulness of a perturbative operator depends on the current population state. An operator that was effective in an early exploratory phase may be less useful after diversity decreases, while an exploitation-oriented operator may become more useful near convergence. DyACE therefore spends part of the shared budget on state-conditioned parent evaluation before generating and selecting the next operator.

Each DyACE candidate evaluation uses a 30-generation short-horizon rollout. After one algorithm-design step, the selected operator is deployed to advance the main solution population by 5 generations. Repeating this process for 30 steps yields the same 150-generation solution horizon used by the static baselines. Thus, DyACE changes how the evaluation budget is allocated over time, but it does not extend the final solution-evolution horizon.

This online protocol preserves computational fairness while enabling adaptation. All candidate evaluations are counted under the same budget, and each DyACE evaluation uses a shorter rollout horizon than the 150-generation full rollout used by static LLM-AHD baselines. DyACE therefore does not receive additional hidden evaluations; instead, it trades full-horizon offline evaluations for short, state-conditioned online evaluations.

B.6 Time Caps and Wall-clock Considerations

We use per-evaluation time caps as practical safeguards against pathological candidate heuristics. For static LLM-AHD baselines, including FunSearch, EoH, ReEvo, and MCTS-AHD, each candidate evaluation is capped at 5 minutes because the

candidate is assessed through a 150-generation full rollout. For DyACE, each candidate evaluation is capped at 2 minutes because it uses a 30-generation short-horizon rollout. If a candidate exceeds its time cap, the candidate is discarded and does not enter the operator pool or final selection.

These caps are not used as an additional optimization objective. The objective of our evaluation is the final optimality gap under a fixed rollout-evaluation budget. We do not add a runtime penalty to the candidate score because runtime is not a monotonic proxy for perturbative heuristic quality. A fast operator can terminate quickly while producing little improvement, whereas a slower operator can perform more useful structural modifications and lead to a better final gap. The usefulness of a perturbative operator also depends on the current search state, including population diversity, stagnation, and exploitation pressure.

The time caps therefore serve a limited but important role: they prevent failed or inefficient generated code from dominating wall-clock time, while leaving the comparison governed by the same rollout-evaluation budget and the same final quality metric. This is consistent with the quality-first evaluation protocol used throughout the paper. In our implementation, all compared methods generally complete within two hours under these caps.

C Prompt Design Details

To ensure reproducibility, we provide the core prompt templates used in the Decoupled Meta-Reasoning module. The interaction involves two distinct agents: the Diagnosis Agent and the Coding Agent.

C.1 Diagnosis Agent Prompt

The Diagnosis Agent receives the Search Trajectory Features and is tasked with generating Verbal Gradients. It does not generate code but analyzes the search status.

Combine Prompt for Diagnosis Agent

You are an expert in {problem type} optimization algorithms, tasked with analyzing the advantages of the parent algorithms and propose a fusion optimization method.

Algorithm 1 : {parent1}
search trajectory on solution population:
{parent1_feature}

Algorithm 2 : {parent2}
search trajectory on solution population:
{parent2_feature}

Conduct a thorough analysis and propose optimization suggestions:
1. Compare their search trajectory performance.
2. Identify each algorithm's key strengths and weaknesses based on the search trajectory and parameter settings.
3. Propose specific directions for combining the two parent strategies.

Must output in the following format:
<analysis> Detailed analysis </analysis>
<direction> Specific Improvement Directions </direction>

Prompt 1: The combine mode prompt used for the Diagnosis Agent.

Mutate Prompt for Diagnosis Agent

You are an expert in {problem type} optimization algorithms. Please analyze the parent algorithm and provide specific improvement directions.

Parent algorithm : {parent_algorithm}
search trajectory on solution population:
{parent_feature}

Conduct a thorough analysis and propose optimization suggestions:
1. Analyse parent algorithm's search trajectory performance.
2. Identify the main strengths and weaknesses of the parent algorithm based on the search trajectory and parameter settings.
3. Propose specific local optimization directions that make targeted adjustments to the parent algorithm.

Must output in the following format:
<analysis> Detailed analysis </analysis>
<direction> Specific Improvement Directions </direction>

Prompt 2: The mutate mode prompt used for the Diagnosis Agent.

Explore Prompt for Diagnosis Agent

You are an expert in {problem type} optimization algorithms, tasked with analyzing a parent algorithm and proposing innovative, distinct optimization approaches.

Parent algorithm : {parent_algorithm}
search trajectory on solution population: {parent_feature}

Conduct a thorough analysis and propose optimization suggestions:
1. Analyse parent algorithm's search trajectory performance.
2. Identify the main strengths and weaknesses of the parent algorithm based on the search trajectory and parameter settings.
3. Propose specific exploratory algorithm optimization directions that differ substantially from the parent.

Must output in the following format:
<analysis> Detailed analysis </analysis>
<direction> Specific Improvement Directions </direction>

Prompt 3: The explore mode prompt used for the Diagnosis Agent.

C.2 Coding Agent Prompt

The Coding Agent use the Verbal Gradients from the Diagnosis Agent to synthesize the executable Python code for the next heuristic operator.

Initialize Prompt for Coding Agent

You are a senior Python algorithm engineer tasked with generating directly executable algorithms for the {problem_type}.

Seed Algorithm : {parent}
search trajectory on solution population: {parent_feature}

Output format: Only three sections, output text directly:
Description is wrapped in <description>...</description>
Parameters are wrapped in <parameter>...</parameter>
Code is wrapped in <code>...</code>

<description>
(First line: concise Algorithm Title)
(Then 2–3 short paragraphs explaining the main ideas.)
</description>
<parameter>
{parameter_to_evolve}
</parameter>
<code>
{algorithm_to_evolve}
</code>

Prompt 4: The initialize mode prompt used for the Coding Agent.

Combine Prompt for Coding Agent

You are a senior Python metaheuristic engineer. Task: Combine the advantages of two parent optimization algorithms to produce a new directly executed algorithm code.

Algorithm 1 : {parent1}
search trajectory on solution population: {parent1_feature}

Algorithm 2 : {parent2}
search trajectory on solution population: {parent2_feature}

{optimization_direction}

Output format: Only three sections, output text directly
Description is wrapped in <description>...</description>
Parameters are wrapped in <parameter>...</parameter>
Code is wrapped in <code>...</code>

<description>
(First line: concise Algorithm Title)
(Then 2–3 short paragraphs explaining the main ideas.)
</description>
<parameter>
{parameter_to_evolve}

</parameter>
<code>
{algorithm_to_evolve}
</code>

Prompt 5: The combine mode prompt used for the Coding Agent.

Mutate Prompt for Coding Agent

You are a senior Python metaheuristic engineer. Task: Make incremental improvements to the reference algorithm, outputting a directly executable new algorithm.

Algorithm to improve : {parent}
search trajectory on solution population: {parent_feature}

{optimization_direction}

Output format: Only three sections, output text directly
Description is wrapped in <description>...</description>
Parameters are wrapped in <parameter>...</parameter>
Code is wrapped in <code>...</code>

<description>
(First line: concise Algorithm Title)
(Then 2–3 short paragraphs explaining the main ideas.)
</description>
<parameter>
{parameter_to_evolve}
</parameter>
<code>
{algorithm_to_evolve}
</code>

Prompt 6: The mutate mode prompt used for the Coding Agent.

Explore Prompt for Coding Agent

You are a senior Python metaheuristic engineer. Task: Propose a completely new JSSP optimization algorithm with different ideas from the reference algorithm.

Reference Algorithm : {parent}
search trajectory on solution population: {parent_feature}

{optimization_direction}

Output format: Only three sections, output text directly
Description is wrapped in <description>...</description>
Parameters are wrapped in <parameter>...</parameter>
Code is wrapped in <code>...</code>

<description>
(First line: concise Algorithm Title)
(Then 2–3 short paragraphs explaining the main ideas.)
</description>
<parameter>
{parameter_to_evolve}
</parameter>
<code>
{algorithm_to_evolve}
</code>

Prompt 7: The explore mode prompt used for the Coding Agent.

D Case Study: Evolutionary Trajectory on JSSP Instance ta51

To illustrate the connection between landscape dynamics and DyACE's decisions, we analyze the evolutionary trajectory on the JSSP instance ta51 (50×15). Figure 3 illustrates the convergence curve, which correlates directly with the specific algorithmic evolutions detailed below.

D.1 Initial Exploration and Adaptive Divergence

The search process begins with a rapid initial descent led by Algorithm S8 in Generation 0. This algorithm establishes a robust baseline by combining standard Order Crossover (OX) with a balanced mutation scheme that utilizes both random swaps and inversions. However, the system quickly encounters a resistance level where improvements slow down. By Generation 5, the Diagnosis Agent identifies that the current search suffers from low population diversity. In response, DyACE synthesizes Algorithm S26, which introduces a diversity-aware adaptive control mechanism. This algorithm adjusts crossover rates based on a diversity thresh-

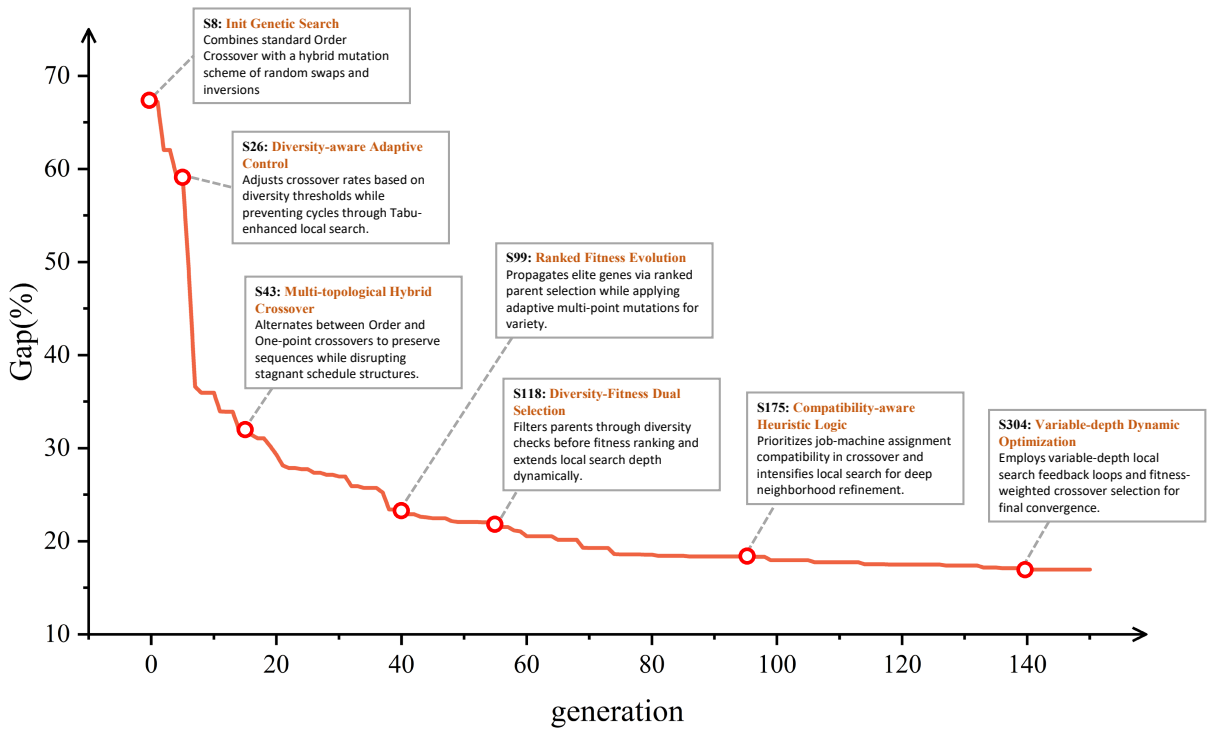

Figure 3: Case study of algorithmic evolution on JSSP instance ta51. The markers indicate key transitions in heuristic logic synthesized to address specific search stages.

old of 0.0917 and incorporates Tabu Search logic into the local search phase to prevent the population from cycling back to previously evaluated schedules. Additionally, it adopts a more aggressive multi-operation swap mutation that modifies three job positions simultaneously to force exploration into new regions of the solution space.

D.2 Structural Hybridization and Selection Optimization

As the evolution progresses to Generation 15, the population faces a structural bottleneck where standard operators fail to locate new promising regions. To address this, DyACE transitions to Algorithm S43, which features a Multi-topological Hybrid Crossover. Instead of relying on a single method, this algorithm probabilistically selects between Order Crossover and One-Point Crossover. This specific combination allows the solver to alternate between preserving job sequences and disrupting stagnant schedule structures, effectively helping the population escape local optima.

By Generation 40, the Meta-Controller shifts focus toward the quality of gene propagation by introducing Algorithm S99. This algorithm abandons random parent selection in favor of a Ranked Selection mechanism, which ensures that higher-performing parents are more likely to contribute to the next generation. To prevent premature convergence caused by this selection pressure, the algorithm increases the mutation rate to 0.25 and utilizes adaptive multi-point adjustments. This trend of refining the selection logic culminates in Generation 55 with Algorithm S118, which implements a Diversity-Fitness Dual Selection. This two-layer approach filters parents based on their diversity contribution before applying fitness ranking. Furthermore, it enhances the local search by dynamically extending the search depth if consecutive improvements are detected, allowing the solver to adapt its intensity to the local landscape geometry.

D.3 Domain Awareness and Final Convergence

A significant turning point in the trajectory occurs in Generation 95 with the synthesis of Algorithm S175. Landscape analysis at this stage reveals that generic genetic operators are no longer yielding significant gains. Consequently, DyACE generates a Compatibility-aware Heuristic Crossover designed to prioritize job-machine assignment compatibility. By shifting from simple permutations to a logic that understands the physical constraints of the Job Shop Scheduling Problem, the algorithm achieves more precise structural optimizations. To complement this domain-specific logic, the local

Cases	Size	Random	LPT	SPT	STPT	MPSR	DRL-Liu	GP	GEP	FunSearch	EoH	ReEvo	MCTS-AHD	DyACE
DMU03	20×15	40.13	68.14	32.92	54.96	25.78	20.94	29.62	33.69	18.67	18.13	18.24	18.16	**11.64**
DMU04	20×15	45.71	51.63	32.67	73.92	25.70	24.43	27.61	31.10	21.43	23.86	16.22	21.69	**11.95**
DMU08	20×20	32.62	42.75	47.87	39.87	25.44	28.54	19.26	26.19	22.33	23.15	17.79	23.28	**10.51**
DMU09	20×20	32.41	45.89	38.52	51.68	25.13	21.38	35.71	33.76	20.86	24.42	20.86	17.53	**12.26**
DMU13	30×15	48.08	51.59	30.75	41.46	29.29	27.90	29.45	30.73	26.32	26.27	22.41	23.69	**17.93**
DMU14	30×15	56.33	64.73	35.03	41.75	24.87	21.51	26.37	24.13	19.15	20.98	19.18	20.57	**16.62**
DMU18	30×20	38.55	51.14	62.10	42.56	30.15	24.87	22.16	27.91	30.00	31.84	24.51	23.60	**17.74**
DMU19	30×20	37.31	53.58	36.04	38.08	30.84	28.37	23.83	39.20	34.53	34.90	27.52	29.75	**21.86**
DMU23	40×15	27.42	50.92	33.89	39.70	15.32	12.25	15.49	19.86	16.26	16.77	15.00	13.20	**8.08**
DMU24	40×15	30.77	39.50	18.40	41.89	15.28	14.44	19.62	17.43	17.51	13.90	14.20	15.40	**11.53**
DMU28	40×20	43.58	56.05	39.77	64.05	26.32	26.77	28.24	30.90	32.33	25.49	23.40	24.79	**18.50**
DMU29	40×20	40.74	57.45	39.95	63.93	30.19	24.15	32.94	32.68	33.40	26.52	29.40	31.64	**19.66**
DMU33	50×15	20.29	53.26	28.51	33.22	9.67	12.74	6.65	6.16	9.39	9.55	9.90	6.69	**6.08**
DMU34	50×15	39.70	48.39	30.47	43.73	18.09	16.69	17.49	16.60	21.43	18.01	13.69	20.11	**8.99**
DMU38	50×20	34.52	58.43	39.23	49.75	33.10	27.34	27.20	31.30	33.00	29.56	26.71	25.47	**22.02**
DMU39	50×20	40.89	48.15	32.10	55.00	20.98	17.90	20.78	23.96	25.54	20.38	21.32	21.60	**16.95**
Mean	–	37.28	52.34	35.62	47.93	23.54	21.33	23.02	25.52	23.89	22.73	20.02	21.07	**14.52**

Table 5: Additional optimality gaps on DMU JSSP instances. Lower is better. The mean is computed over the 16 listed instances.

search iterations are doubled to 100, focusing on intense neighborhood refinement to capitalize on high-quality solutions.

The evolution concludes with Algorithm S304 in Generation 140, which establishes a final refinement phase to secure global optimality. This algorithm employs a Variable-depth Dynamic Optimization logic that uses a feedback loop to adjust local search iterations based on previous improvement success. It also utilizes a hybrid crossover that dynamically toggles between single-point and two-point methods based on parental fitness. This highly adaptive configuration provides the precision necessary for the final convergence stage, successfully navigating the remaining complexity of the ta51 instance and reaching the final reported schedule/gap.

E Additional Results

E.1 Additional JSSP Results

Table 5 reports additional JSSP results on 16 DMU benchmark instances. In addition to the LLM-based AHD baselines used in the main experiments, we include Random and several lightweight dispatching-rule baselines, including LPT, SPT, STPT, and MPSR, together with DRL-Liu, GP, and GEP. All methods are evaluated using optimality gap as the metric, where lower is better. DyACE follows the same online rollout-evaluation protocol described in Appendix B.5.

Across the DMU instances, DyACE obtains the lowest gap on all 16 cases and achieves the best mean gap among the compared methods. The gains are especially visible on medium-scale instances such as DMU03, DMU08, DMU23, and DMU29, where static LLM-AHD baselines remain substantially above DyACE. These additional results support the main JSSP findings on a separate benchmark family.

Method	kroA100	eil101	ch130	ch150	rat195	d198
FunSearch	60.30	39.11	106.07	112.91	74.09	82.04
EoH	19.94	17.65	59.59	4.96	12.44	41.77
ReEvo	16.05	12.40	5.68	6.28	10.33	2.62
MCTS-AHD	4.82	3.82	8.36	8.03	5.90	39.63
DyACE	**0.23**	**2.54**	**1.08**	**1.16**	**1.21**	**0.55**

Table 6: Additional routing results on TSPLIB instances. Values are optimality gaps; lower is better.

E.2 Additional Routing Problem Results

Table 6 reports additional routing results on six TSPLIB instances. All methods are evaluated under the same restricted perturbative operator interface used in the main text, with only crossover and mutation operators. DyACE obtains the lowest gap on all six instances, while the strongest static baseline varies across instances. These results provide additional evidence that online operator revision remains useful for routing problems beyond the instances reported in the main text.