TOWARDS LLM-BASED ALGORITHM DESIGN FOR HETEROGENEOUS DATASETS: DECOUPLED CONFIGURATION POLICY AND PARAMETERIZED ALGORITHM

Anonymous authors
Paper under double-blind review

ABSTRACT

Automated Algorithm Design (AAD) based on Large Language Models (LLMs) has shown promising results, but existing methods often struggle to generalize across heterogeneous problem instances. Most of these methods either learn a single static algorithm to handle all cases or generate a finite set of algorithms to cover different types of instances. While these strategies can improve performance on seen data, they remain limited to a finite set of learned behaviors and may fail when encountering unseen or mixed cases that do not clearly belong to any known type. In this work, we aim to overcome this limitation by moving beyond selecting from a finite set of algorithms. We propose the decoupled configuration-policy and parameterized-algorithm (DCPPA) structure, in which the algorithm encodes behavioral differences as parameters, and the policy maps observable instance or state features to these parameters. DCPPA represents algorithmic behavior in a continuous, parameterized space, and enables the algorithm to capture gradual variations among diverse instances and smoothly adjust its behavior for cases not well covered by the training data. To construct DCPPA-structured algorithms, we develop TS-DCPPA, a two-stage pipeline. The first stage discovers complementary source algorithms and fuses them into a unified structure. With this structure fixed, the second stage employs a warm-start mechanism and learns a continuous parameter-tuning policy. Experiments on real-world problem instances demonstrate that TS-DCPPA achieves greater robustness and cross-distribution generalization than existing methods.

1 INTRODUCTION

Automated Algorithm Design (AAD) aims to discover high-performing components, structures, or entire algorithms while reducing manual trial-and-error and reliance on domain-specific expertise Ma et al. (2025). Large Language Models (LLMs) have recently made this search substantially more expressive by generating executable code directly, enabling AAD systems to explore rich program spaces beyond hand-crafted templates Liu et al. (2026b). This paradigm has produced strong results on problems such as circle packing Lange et al. (2025) and Boolean satisfiability Yu et al. (2025).

Yet most LLM-based AAD methods still return a single static algorithm. Specifically, they do not explicitly impose an instance-adaptive structure on the designed algorithm. To rapidly improve performance, they tend to emphasize the dominant regularities captured in algorithm design, which often results in static algorithms. The static algorithm is efficient to optimize and can work well when training and test instances share a dominant pattern Sim et al. (2025); Shi et al. (2026). Unfortunately, real-world scenarios often exhibit significant heterogeneity, and deployment distributions may deviate from training environments Zhang et al. (2025); Wang et al. (2026). This poses challenges to the robustness and generalization of existing methods.

In heterogeneous scenarios, different instance regimes can favor different heuristic behaviors or parameter settings Wu et al. (2024). When existing AAD methods are directly applied to such scenarios, their search processes can be biased toward designs that perform well only under a few regimes. The resulting algorithm, with limited dynamic adaptation capability, may struggle to accommodate highly diverse instances, leading to poor robustness and generalization Zhang et al.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

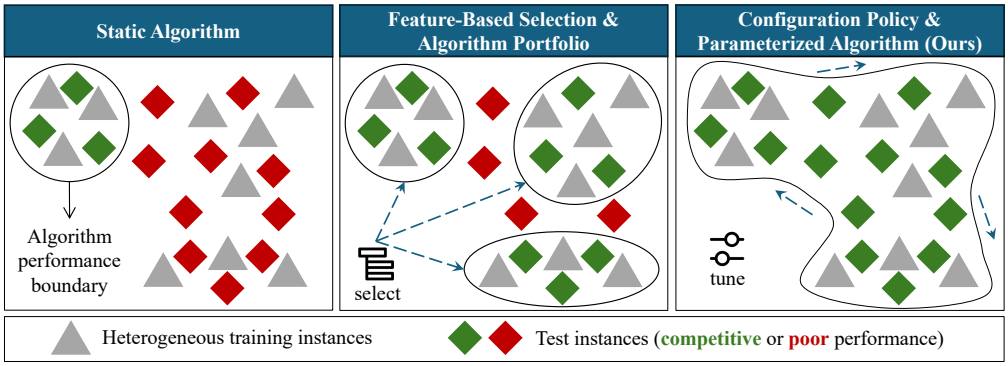

Figure 1: Comparison of static, selection-based, and parameterized algorithm design paradigms.

(2025); Yang et al. (2025). Although algorithm portfolio construction and feature-based algorithm selection methods have recently been proposed Yang et al. (2025); Zhang et al. (2025); Wang et al. (2026), they remain limited when a test instance lies between known categories, combines multiple patterns, or follows a shifted distribution. In such cases, the desired behavior may not correspond to any single learned algorithm, causing substantial performance degradation.

We propose replacing the static algorithm with a parameter-continuous, condition-aware algorithm. The key idea is to decouple the designed algorithm into two parts: a unified parameterized algorithm that exposes a structured behavior space, and a configuration policy that maps raw instance data or state signals to runtime parameters. As illustrated in Figure 1, the proposed idea enables the designed algorithm to better capture gradual variations across diverse instances and to handle cases that are not well covered by the training data. Moreover, we instantiate the idea with a two-stage LLM-based pipeline that first synthesizes a parameterized algorithm from complementary behaviors and then searches for a programmatic configuration policy with the parameterized algorithm fixed. The separate design reduces the algorithm search space and improves search efficiency.

Our contributions are highlighted as follows:

- We introduce the decoupled configuration-policy and parameterized-algorithm (DCPPA) structure, which turns LLM-designed algorithms from static programs or finite portfolios into configurable algorithm families.
- We develop TS-DCPPA, a two-stage design pipeline. Stage 1 discovers complementary algorithms and fuses them into a unified parameterized algorithm, validating that the fused algorithm under a corresponding parameter preset preserves behavioral fidelity to each source algorithm. Stage 2 searches for a condition-aware configuration program via a curriculum that initially restricts it to selecting validated parameter presets, and subsequently allows it to generate continuous parameter values.
- We evaluate TS-DCPPA on heterogeneous instances from the traveling salesman problem (TSP), online virtual machine placement (OVMP), and flexible job shop scheduling (FJSP). TS-DCPPA achieves the best average performance and delivers the best results under unseen distributions.

2 Method

2.1 DCPPA Formulation

Interfaces. The DCPPA structure separates condition-dependent control from the algorithm that makes the actual decision. Let $\mathcal{C}$ be the condition space, $\mathcal{Z}$ the downstream problem-input space, Θ the configuration space, and $\mathcal{A}$ the decision space. A configuration policy $\pi : \mathcal{C} \to \Theta$ maps an observed condition to a configuration, while a parameterized algorithm $\phi : \mathcal{Z} \times \Theta \to \mathcal{A}$ maps the problem input and configuration to a decision. Their composition is

$$h_{\phi,\pi}(z, c) = \phi\big(z, \pi(c)\big), \tag{1}$$

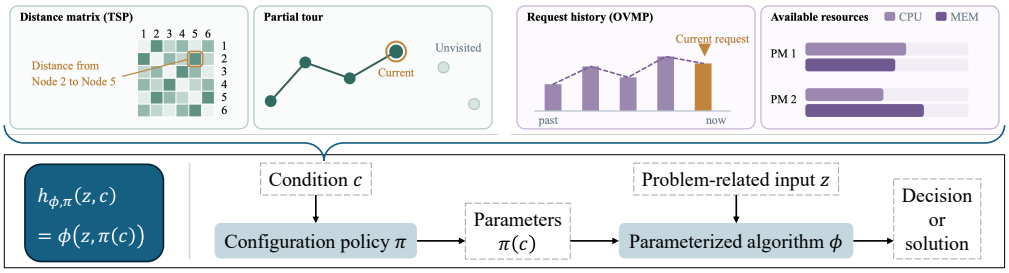

Figure 2: The DCPPA structure. The configuration policy π maps the observed condition c to runtime parameters, which configure the parameterized algorithm ϕ as it processes the problem input z.

where c denotes the condition available when the decision is made, such as instance data, the current solver state, or historical observations, and z denotes the task-specific input required by ϕ. The two inputs may overlap because the same current state can both inform the configuration and be consumed by the downstream decision rule.

Parameterized behavior space. For a fixed ϕ, Θ indexes an algorithm family rather than a single heuristic. Changing $\theta = \pi(c)$ changes the behavior of the same executable mechanism. Continuous controls such as weights, thresholds, and penalties allow behavior to vary gradually across conditions. For example, in a constructive TSP heuristic, π can use the current route state to adjust the relative importance of distance, look-ahead cost, and spatial diversity at each decision step.

Static algorithms and algorithm selection are special cases of this formulation. A constant policy reduces Eq. (1) to a static algorithm. If Θ contains only discrete identifiers and ϕ dispatches to the corresponding candidate, the formulation reduces to portfolio selection. DCPPA instead targets a unified ϕ with shared computations and condition-dependent parameters, enabling intermediate behaviors that need not coincide with any member of a finite portfolio. Figure 2 summarizes this data flow.

2.2 TS-DCPPA Overview

Jointly searching for ϕ and π creates a nested program-design problem. A candidate edit may alter the algorithm family, its configuration policy, or their interaction, making performance changes difficult to attribute. TS-DCPPA therefore separates the search into two stages. Generally, Stage 1 constructs a parameterized behavior space, and Stage 2 learns a condition-to-configuration mapping within this fixed space. This decomposition is independent of the AAD backends used in either stage.

In our instantiation, Stage 1 uses EoH-S to discover complementary source algorithms and an LLM-based fusion procedure to construct and validate ϕ. Stage 2 uses EoH to search for π through a curriculum from source-preset selection to free parameter generation.

Let $\mathcal{S} = \{s_i\}_{i=1}^{m}$ be the training instances and $u(h, s_i)$ the evaluator utility of algorithm h on s_i, where smaller values indicate better performance. Figure 3 illustrates how the two stages construct a DCPPA-structured algorithm.

2.3 Stage 1: Constructing the Parameterized Algorithm

Complementary algorithm discovery. At a general level, Stage 1 seeks n algorithms such that each training instance can be handled well by at least one set member. This objective is expressed as the following multi-objective problem:

$$\mathcal{H}^* = \arg \min_{\substack{\mathcal{H} \subseteq \mathcal{D} \\ |\mathcal{H}|=n}} \left(u(\mathcal{H}, s_1), \ldots, u(\mathcal{H}, s_m) \right), \tag{2}$$

where $\mathcal{D}$ is the algorithm search space and $u(\mathcal{H}, s_i) = \min_{h \in \mathcal{H}} u(h, s_i)$. Unlike average-performance selection, Eq. (2) rewards an algorithm when it improves the best available result on some instances, even if it is not the strongest algorithm on average. The resulting set therefore provides complementary behavioral evidence for constructing a configurable algorithm family.

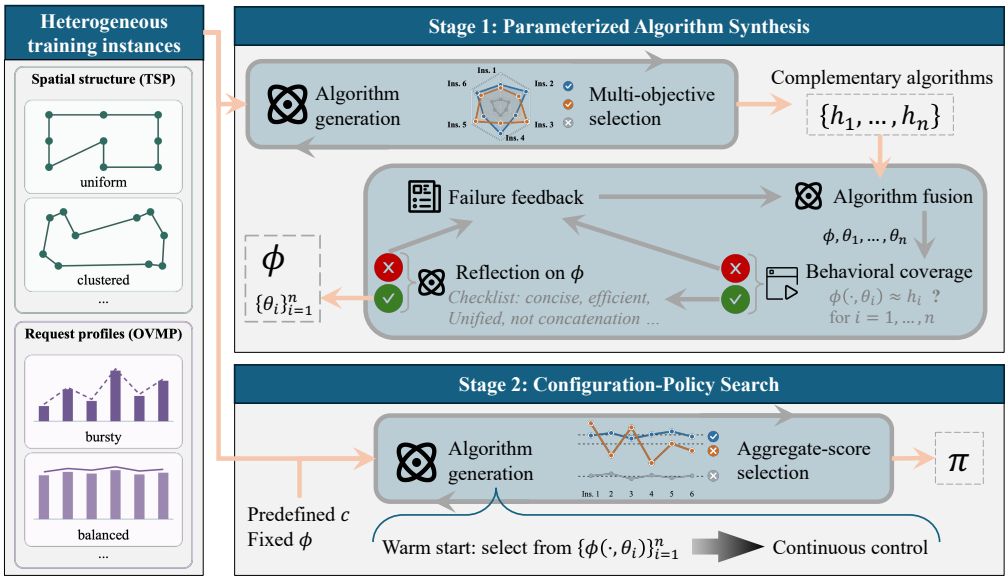

Figure 3: The TS-DCPPA pipeline. Stage 1 discovers complementary algorithms and fuses them into a validated parameterized algorithm. Stage 2 fixes the fused algorithm and searches for a configuration policy via a curriculum that transitions from discrete preset selection to continuous parameter tuning.

We instantiate this step with EoH-S Liu et al. (2026a), which applies linear scalarization to Eq. (2):

$$\mathcal{H}^* = \arg \min_{\substack{\mathcal{H} \subseteq \mathcal{D} \\ |\mathcal{H}|=n}} \sum_{i=1}^{m} u(\mathcal{H}, s_i). \tag{3}$$

It evolves algorithms through LLM-based crossover and mutation to approximate Eq. (3). Notably, the choice of EoH-S is not intrinsic to TS-DCPPA, as the subsequent fusion step requires only source programs with complementary instance-wise behavior. The survivor-selection details are provided in Appendix B.1.

Fusion into a parameterized algorithm. Given $\mathcal{H}^* = \{h_1, \dots, h_n\}$, an LLM synthesizes one parameterized algorithm ϕ and source-specific presets $\{\theta_i\}_{i=1}^n$. The synthesis preserves shared computations among the source algorithms and exposes their meaningful differences as parameters. It must not implement a dispatcher over copied source programs. Instead, the parameters modulate a unified computation, preferably through continuous controls.

At a general level, the fused algorithm should recover or approximate every source behavior under its corresponding preset:

$$\phi(\cdot, \theta_i) \approx h_i(\cdot), \qquad i = 1, \dots, n. \tag{4}$$

The source presets anchor the synthesized behavior space, while other parameter values may induce intermediate behaviors absent from $\mathcal{H}^*$. For example, TSP heuristics that prioritize distance, look-ahead, and diversity differently can share these scoring features while exposing their relative weights as parameters.

Validation and acceptance. We operationalize Eq. (4) through three checks. First, *code validity* verifies compilation, the required function name, and the task-input prefix. Second, *behavioral coverage* replays each source algorithm and its configured counterpart on the same validation inputs $\mathcal{X}_{\text{val}} = \{x_j\}_{j=1}^{N_{\text{val}}}$; execution errors are treated as validation failures. Using the task-specific output-equivalence relation $\equiv$, we compute

$$\text{sim}_i = \frac{1}{N_{\text{val}}} \sum_{j=1}^{N_{\text{val}}} \mathbb{I}\left[h_i(x_j) \equiv \phi(x_j, \theta_i)\right], \qquad \min_{1 \le i \le n} \text{sim}_i \ge \tau_{\text{sim}}. \tag{5}$$

The validation inputs are generated with LLM assistance and screened by executing the source algorithms before comparison. If Eq. (5) fails, the measured similarity and current preset are returned as revision feedback.

Third, after the two predefined checks pass, an *independent LLM review* assesses structural alignment using a fixed checklist. It checks interface and return semantics, shared computation, parameterized differences, and implementation simplicity. The review call returns only a list of substantive issues or an approval marker; a separate revision call generates code and presets when needed. This separation prevents the same response from assessing and revising its own candidate.

Code validity and behavioral coverage are mandatory gates. Before the soft round limit R_{soft}, LLM approval permits early stopping. At the limit, a candidate is accepted if both mandatory gates pass; otherwise, revision continues until they do. The validation hyperparameters are reported in Appendix B.2.

2.4 Stage 2: Learning the Configuration Policy

Search objective. Once ϕ is accepted, it remains fixed while we optimize the configuration policy:

$$\pi^* = \arg\min_{\pi} \sum_{i=1}^{m} u(h_{\phi,\pi}, s_i). \tag{6}$$

Fixing ϕ makes the feedback attributable to the policy. That is, a candidate edit changes how conditions are mapped to a known behavior space rather than changing the behavior space itself. The program-search backend is also replaceable. In this work, we instantiate the policy search with EoH Liu et al. (2024). Each policy-generation prompt contains the task description, a generic configuration-policy template, and the source code of the fixed ϕ, allowing the LLM to infer how the policy outputs affect the downstream decision mechanism.

Curriculum from source presets to free parameters. Stage 2 uses one evolutionary run with a phase-dependent task description. During the initial budget fraction ρ_{src}, the prompt provides the validated source presets. A candidate policy may choose among them according to the observed condition, but it must return one preset unchanged; it cannot modify values, combine presets, or generate new values. This warm-start phase restricts the initial search to validated behavioral anchors.

After the threshold determined by ρ_{src}, the prompt removes the preset table and allows policies to generate new parameter values rather than selecting from the source presets. This main phase extends the search from discrete anchors to the continuous or structured regions between them. The phase transition changes only the task description, and the EoH population and evaluation history are carried forward. The two phases therefore form one curriculum rather than two independent searches. The value of ρ_{src} is provided in Appendix B.3.

3 Experiments

We evaluate TS-DCPPA on three combinatorial optimization tasks, comparing its performance with LLM-based AAD baselines on heterogeneous and distribution-shifted instances. Ablations examine the contributions of source-algorithm discovery, fusion, warm start, and the two-stage decomposition.

3.1 Experimental Setup

3.1.1 Test Problems and Datasets

We consider the traveling salesman problem (TSP), online virtual machine placement (OVMP), and flexible job shop scheduling (FJSP). Test instances from sources represented in training are *in-distribution* (ID), including instances at different scales; those from unseen sources are *out-of-distribution* (OOD). Appendix A provides task definitions, dataset splits, algorithm interfaces, and seed heuristics.

For the TSP, we use TSPLIB Reinelt (1991), VLSI, and National instances. TSPLIB and VLSI provide the training and ID test instances, whereas National serves as the OOD test source.

For the OVMP, we use request sequences from Huawei (Huawei-East-1 Sheng et al. (2022)), Azure (AzurePublicDatasetV2 Cortez et al. (2017)), and a non-public proprietary source, denoted BD. Huawei and Azure provide the training and ID test sequences, whereas BD serves as the OOD test source.

For the FJSP, we use the Brandimarte Brandimarte (1993), Hurink Hurink et al. (1994), and Dauzère-Pérès and Paulli (DPP) Dauzère-Pérès & Paulli (1997) benchmark suites from FJSPLib. Brandimarte and Hurink provide the training and ID test instances, whereas DPP serves as the OOD test source.

3.1.2 Baselines and Performance Metric

We compare with six LLM-based AAD baselines: EoH Liu et al. (2024), ReEvo Ye et al. (2024), PartEvo Hu & Zhang (2025), HiFo Chen et al. (2026a), RefineEvo Wu et al. (2026), and the instance-adaptive InstSpecHH Zhang et al. (2025). The handcrafted seed heuristics are Nearest Neighbor for the TSP, First Fit for the OVMP, and Earliest Start Time for the FJSP. The main experiments use DeepSeek-V4-Flash-GA with temperature 0.8 and a budget of 1,000 candidate evaluations. Population sizes and evaluation time limits are given in Appendix A.3.

All methods use the same task-specific interface: selecting the next unvisited node for the TSP, placing each arriving request using request history and resource availability for the OVMP, and selecting a feasible operation–machine assignment for the FJSP.

For each run, we compute Gap, the mean instance-wise optimality gap over a test set $\mathcal{T}$, as

$$\mathrm{Gap} = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \frac{|f(h,t) - f_t^*|}{|f_t^*|} \times 100\%, \tag{7}$$

where $f(h,t)$ is the objective value of algorithm h on instance t. The reference f_t^* is the best-known tour length for the TSP, an offline bound computed with Gurobi Gurobi Optimization, LLC (2026) for the OVMP, and a benchmark lower bound on makespan for the FJSP. During search, the negative instance-wise gap serves as the performance score, so larger scores are better. We evaluate the algorithm from each of five runs on the complete test set and report the five-run mean, $\overline{\mathrm{Gap}}$.

3.2 Experimental Results

Table 1 reports $\overline{\mathrm{Gap}}$ (%), with equal weight for every instance in each ID/OOD partition. "Avg. Rank" averages the six column-wise ranks, including the seed heuristic. TS-DCPPA ranks first in all six columns, followed by HiFo with an average rank of 2.67.

A mean gap can be sensitive to a few poor results. Table 2 therefore reports the *mean per-instance rank*: methods are ranked on each instance by their five-run mean gap, ties receive average ranks, and these ranks are averaged within each partition. Lower values indicate more consistent performance; "Avg." is the mean across the six columns.

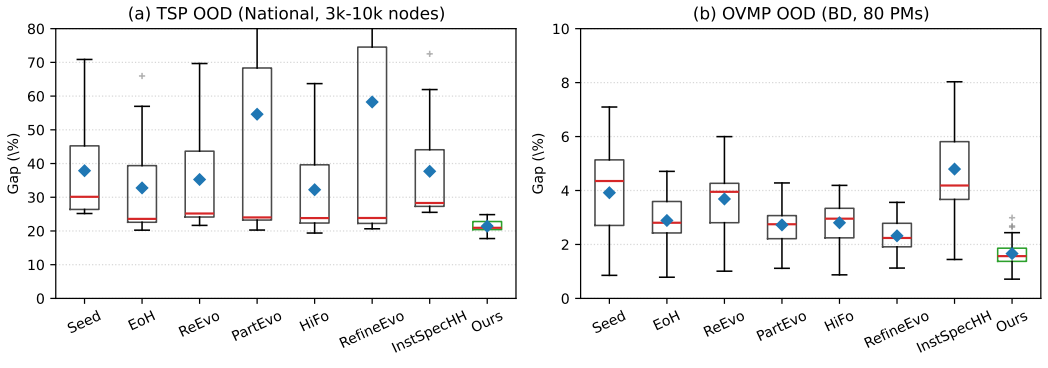

Figure 4: Instance-level gap on two OOD partitions of the main comparison: TSP National with 3k–10k nodes (left) and OVMP BD with 80 PMs (right). Boxes show the interquartile range, red lines the median, blue diamonds the mean, grey pluses outliers; y-axes are capped.

Table 1: Main comparison across the three tasks, reporting $\overline{\mathrm{Gap}}$ (%). ID and OOD denote test sources represented and unseen in training, respectively. Lower is better; **bold** and underline mark the best and second-best results using unrounded values. The seed heuristic is included in the ranking.

Method	TSP		OVMP		FJSP		Avg. Rank
	ID	OOD	ID	OOD	ID	OOD	
Seed heuristic	26.33	40.18	2.30	3.84	25.41	27.49	6.83
EoH	19.55	33.44	1.67	2.75	12.77	12.88	3.67
ReEvo	22.43	36.70	2.55	3.57	13.02	11.69	5.33
PartEvo	20.56	50.56	1.94	2.67	14.10	14.63	5.17
HiFo	20.13	33.13	1.48	2.73	11.71	11.27	2.67
RefineEvo	20.82	52.88	2.08	2.31	11.71	11.54	4.17
InstSpecHH	26.50	40.77	3.80	4.78	16.77	13.83	7.17
TS-DCPPA (Ours)	**18.49**	**20.99**	**1.03**	**1.65**	**11.07**	**8.89**	**1.00**

Table 2: Mean per-instance rank for the main comparison. Methods are ranked by each instance's five-run mean gap, with average ranks for ties. Lower is better; **bold** and underline mark the best and second-best results. The seed heuristic is included in the ranking.

Method	TSP		OVMP		FJSP		Avg.
	ID	OOD	ID	OOD	ID	OOD	
Seed heuristic	6.98	6.68	6.21	6.40	7.02	7.56	6.81
EoH	2.99	2.95	3.90	4.05	4.32	5.14	3.89
ReEvo	5.28	5.32	5.37	6.04	4.27	3.83	5.02
PartEvo	3.86	4.74	4.20	3.96	5.16	6.50	4.74
HiFo	3.60	2.53	3.31	4.15	3.48	3.39	3.41
RefineEvo	4.15	5.32	4.23	2.83	3.08	3.22	3.81
InstSpecHH	7.12	6.68	6.84	7.12	5.84	5.31	6.49
TS-DCPPA (Ours)	**2.01**	**1.79**	**1.96**	**1.44**	**2.82**	**1.06**	**1.85**

TS-DCPPA also has the lowest mean per-instance rank in every column, averaging 1.85 versus 3.41 for HiFo. It ranks strictly first on 357 of 614 test instances (58.1%), although the win rate varies across partitions and is 37.8% on FJSP-ID.

Compared with the strongest LLM-based baseline in each column, TS-DCPPA reduces the TSP gaps by 1.07 and 12.13 percentage points on ID and OOD instances, respectively. The corresponding reductions are 0.45 and 0.66 on the OVMP, and 0.64 and 2.38 on the FJSP. The largest reduction occurs on TSP-OOD. InstSpecHH remains close to the TSP seed and performs worse than the OVMP seed; on the FJSP, it improves on the seed but has the highest ID gap among the learned methods.

Figure 4 shows the OOD instance-level gap distributions for the main comparison. On TSP National with 3k–10k nodes, all LLM-based baselines exhibit long upper tails driven by hard instances, while TS-DCPPA compresses both the median and the tail. On OVMP BD with 80 PMs, TS-DCPPA also shows the lowest median and the tightest interquartile range. Appendix D.4 additionally reports the ID counterparts and the ablation and DCPPA-baseline groups, along with paired Wilcoxon tests with Holm correction; in the main comparison, all adjusted p-values against TS-DCPPA are below 0.05.

Additional experiments evaluate homogeneous synthetic instances (Appendix D.2) and a second LLM backend (Appendix D.3). Appendix E illustrates the generated parameterized algorithms and configuration policies.

3.3 Ablation Studies

We consider four variants of TS-DCPPA: (i) **w/o fusion**: Stage 1 retains EoH-S, but Stage 2 learns a policy that selects a source algorithm at each decision step without fusion; (ii) **w/o complementarity**: EoH replaces EoH-S in Stage 1, while Stage 2 remains unchanged; (iii) **w/o warm start**: Stage 2

Table 3: Ablation results, reporting $\overline{\text{Gap}}$ (%) on the same ID/OOD partitions as Table 1. Lower is better; **bold** and underline mark the best and second-best results.

Method	TSP		OVMP		FJSP		Avg. Rank
	ID	OOD	ID	OOD	ID	OOD	
w/o fusion	19.59	33.46	2.38	2.14	13.43	14.41	4.00
w/o complementarity	19.38	33.54	2.81	3.65	12.14	12.10	3.83
w/o warm start	18.57	**20.72**	1.47	1.78	11.49	9.18	1.83
EoH-DCPPA	21.39	35.96	1.56	2.13	13.52	14.40	4.17
TS-DCPPA (Ours)	**18.49**	20.99	**1.03**	**1.65**	**11.07**	**8.89**	**1.17**

Table 4: Mean per-instance rank for the ablation study, computed as in Table 2. Lower is better; **bold** and underline mark the best and second-best results.

Method	TSP		OVMP		FJSP		Avg.
	ID	OOD	ID	OOD	ID	OOD	
w/o fusion	3.17	3.16	3.70	3.11	3.49	4.33	3.49
w/o complementarity	3.05	3.53	4.39	4.68	2.96	3.11	3.62
w/o warm start	2.28	**1.84**	2.36	2.28	2.62	1.72	2.19
EoH-DCPPA	4.42	4.37	2.72	3.41	3.62	4.33	3.81
TS-DCPPA (Ours)	**2.08**	2.11	**1.82**	**1.52**	**2.30**	**1.50**	**1.89**

searches for a configuration policy without the initial source-preset selection phase; (iv) **EoH-DCPPA**: EoH generates a DCPPA-structured algorithm end-to-end in a single stage.

In Table 3, TS-DCPPA achieves the best average rank (1.17), with the lowest gap in five columns and the second lowest on TSP-OOD. Removing fusion or complementarity increases the gap in every column, yielding average ranks of 4.00 and 3.83, respectively. Removing warm start is less detrimental (average rank 1.83): it slightly improves TSP-OOD but increases OVMP-ID and FJSP-ID gaps by 0.44 and 0.42 percentage points. EoH-DCPPA has the highest average rank (4.17), supporting the two-stage search over end-to-end generation under the same evaluation budget.

Table 4 gives the same top two methods by mean per-instance rank: TS-DCPPA (1.89) and w/o warm start (2.19). Their ordering reverses only on TSP-OOD. Removing fusion and complementarity yields averages of 3.49 and 3.62, reversing their order in the gap-based comparison.

The paired tests in Appendix D.4 show significant differences from TS-DCPPA for w/o fusion, w/o complementarity, and EoH-DCPPA on every partition. For w/o warm start, the differences are significant only on OVMP-ID, OVMP-OOD, and FJSP-ID. Warm start therefore provides a less uniform benefit than fusion or complementary source discovery.

Appendix D.1 extends the end-to-end comparison to all five static LLM-based AAD baselines under the DCPPA structural constraint.

4 Related Work

4.1 LLM-Based AAD

LLM-based AAD leverages the code-generation and semantic-reasoning capabilities of LLMs to discover heuristics and algorithms as executable code. A common paradigm embeds the LLM as a variation operator in an iterative loop: candidates are generated, evaluated by a task-specific evaluator, and then revised or recombined according to performance feedback. FunSearch Romera-Paredes et al. (2024) and EoH Liu et al. (2024) are representative early methods. Subsequent work improves the search loop through refined prompt engineering Ye et al. (2024), niching and alternative search frameworks Hu & Zhang (2025); Zheng et al. (2025); van Stein & Bäck (2025), and multi-objective settings Yao et al. (2025). Other efforts focus on reinforcement learning-based model adaptation

and fine-tuning Surina et al. (2025); Liu et al. (2025), problem reduction before search Thach et al. (2025), and co-evolution of training sets and algorithms Li et al. (2025).

4.2 INSTANCE-ADAPTIVE AAD

The need to adapt algorithmic behavior to individual instances has long been studied outside the LLM setting. Algorithm selection maps each instance to one algorithm from a finite portfolio Rice (1976); Xu et al. (2008); algorithm configuration optimizes the parameters of a predefined parameterized algorithm Hutter et al. (2009); hyper-heuristics select or generate low-level heuristics Burke et al. (2013). In these formulations, the configurable structure, whether a heuristic pool or a parameterized target algorithm, is specified manually before optimization begins. Recent LLM-based methods introduce instance-dependent control along similar lines. InstSpecHH Zhang et al. (2025) designs specialized heuristics over feature-defined subclasses; DASH Wang et al. (2026) and EoH-S Liu et al. (2026a) generate diverse algorithms without predefined subclasses, but EoH-S does not specify how to select from the resulting set on a new instance; DyCA Hu et al. (2026) follows the same complementary-set paradigm and dynamically clusters instances from their behavioral responses to generated algorithms, designing one specialist per cluster; and HeurAgenix Yang et al. (2025) uses an LLM as a runtime selector.

These methods still operate over a finite set of algorithms or invoke the LLM while solving. In contrast, TS-DCPPA fuses complementary source algorithms into a unified parameterized algorithm and learns a policy that maps raw instance or state information to continuous parameters; the deployed program makes no LLM calls. EoH-S and EoH are replaceable search backends in the current implementation. Appendix C provides an extended discussion.

5 CONCLUSION, LIMITATION, AND FUTURE WORK

Conclusion. This paper has proposed the DCPPA structure to address the robustness and generalization challenges of LLM-based AAD under heterogeneous scenarios. Instead of a single static algorithm or the selection of a finite portfolio, DCPPA represents a parameterized algorithm controlled by a condition-aware configuration policy. This structure allows the generated algorithm to adapt its behavior continuously according to instance data or state information. Based on this structure, we have developed TS-DCPPA, an efficient two-stage design pipeline that first synthesizes a unified parameterized algorithm from complementary algorithms and then searches for a programmatic configuration policy with the parameterized algorithm fixed. This decomposition reduces the difficulty of jointly designing the algorithm family and its control logic. Experiments have shown that TS-DCPPA achieves stronger average performance and more robust generalization than representative LLM-based AAD baselines across heterogeneous and distribution-shifted settings, and it further attains the best rank on homogeneous synthetic instances, indicating that the two-stage search remains sample-efficient even when instance adaptation brings little marginal benefit. The ablation studies have further verified the effectiveness of the two-stage pipeline and the algorithm fusion.

Limitation and Future Work. This work still has several limitations. The effectiveness of TS-DCPPA depends on the quality of the complementary algorithms discovered in the first stage and on whether the fusion process can expose their useful behavioral differences as parameters. In addition, the current two-stage pipeline improves search efficiency by fixing the parameterized algorithm before policy search, but this separation may also limit the joint optimization of the parameterized algorithm and the configuration policy. Future work may develop stronger strategies for algorithm fusion and new methods for constructing parameterized algorithms with better behavioral coverage. Another promising direction is to use multiple well-performing DCPPA-structured algorithms as the initial population and then build a bilevel optimization method to optimize them, with the upper level optimizing the parameterized algorithm and the lower level optimizing the configuration policy.

AI USE STATEMENT

In this work, we used generative AI tools to assist in implementing methods, specifically to help write parts of the experimental code and to reproduce baseline methods. We have not used generative

AI tools to generate synthetic datasets, help develop theoretical models or conceptual frameworks, propose or refine research hypotheses, design or provide feedback on research methodology or experiments, translate the manuscript, clean or reformat datasets, or interpret experimental results. The remaining tasks with required disclosure, namely formulating mathematical claims, providing ingredients for or writing proofs, and supporting qualitative or thematic data analysis, are not applicable to this work. Additionally, we used generative AI tools to draft parts of the manuscript, edit the paper to improve readability, suggest the paper structure, title, and keywords, create or edit software code beyond method implementation (e.g., utility and plotting scripts), and support brainstorming and information sourcing. We have reviewed all AI-assisted work: AI-generated or AI-assisted code was verified and tested for correctness by the authors, and AI-assisted text was cross-checked against the original references to ensure factual and citation accuracy. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Principles and Practice of Constraint Programming (CP)*, volume 5732 of *Lecture Notes in Computer Science*, pp. 142–157. Springer, 2009.

Paolo Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41(3):157–183, 1993.

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

Vaggos Chatziafratis, Ishani Karmarkar, Yingxi Li, and Ellen Vitercik. Accelerating data-driven algorithm selection for combinatorial partitioning problems. *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Chentong Chen, Mengyuan Zhong, Ye Fan, Jialong Shi, and Jianyong Sun. HiFo-prompt: Prompting with hindsight and foresight for LLM-based automatic heuristic design. In *International Conference on Learning Representations (ICLR)*, 2026a.

Chentong Chen, Mengyuan Zhong, Ye Fan, Jialong Shi, and Jianyong Sun. TIDE: Tuning-Integrated dynamic evolution for LLM-based automated heuristic design. *arXiv preprint arXiv:2601.21239*, 2026b.

Hongzheng Chen, Yingheng Wang, Yaohui Cai, Hins Hu, Jiajie Li, Shirley Huang, Chenhui Deng, Rongjian Liang, Shufeng Kong, Haoxing Ren, et al. HeuriGym: An agentic benchmark for llm-crafted heuristics in combinatorial optimization. *arXiv preprint arXiv:2506.07972*, 2025.

Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pp. 153–167, 2017.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. HSEvo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 39, pp. 26931–26938, 2025.

Stéphane Dauzère-Pérès and Jan Paulli. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations Research*, 70 (0):281–306, 1997.

Ruibo Duan, Yuxin Liu, Xinyao Dong, and Chenglin Fan. EALG: Evolutionary adversarial generation of language model-guided generators for combinatorial optimization. *arXiv preprint arXiv:2506.02594*, 2025.

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A service for black-box optimization. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 1487–1495, 2017.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL https://www.gurobi.com.

Qinglong Hu and Qingfu Zhang. Partition to Evolve: Niching-enhanced evolution with LLMs for automated algorithm discovery. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Qinglong Hu, Qingfu Zhang, Fei Liu, Xialiang Tong, Kun Mao, and Mingxuan Yuan. Beyond average performance: Dynamic instance clustering and specialized algorithm design in LLM-assisted evolutionary search. *arXiv preprint arXiv:2608.03129*, 2026.

Johann Hurink, Bernd Jurisch, and Monika Thole. Tabu search for the job-shop scheduling problem with multi-purpose machines. *Operations-Research-Spektrum*, 15(4):205–215, 1994.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization (LION)*, volume 6683 of *Lecture Notes in Computer Science*, pp. 507–523. Springer, 2011.

Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC–instance-specific algorithm configuration. In *European Conference on Artificial Intelligence (ECAI)*, pp. 751–756, 2010.

Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, pp. 149–190. Springer, 2016.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

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

Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. AutoFolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning (ICML)*, pp. 32201–32223. PMLR, 2024.

Fei Liu, Rui Zhang, Xi Lin, Zhichao Lu, and Qingfu Zhang. Fine-tuning large language model for automated algorithm design. *arXiv preprint arXiv:2507.10614*, 2025.

Fei Liu, Yilu Liu, Qingfu Zhang, Xialiang Tong, and Mingxuan Yuan. EoH-S: Evolution of heuristic set using llms for automated heuristic design. In *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, 2026a.

Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Zhe Zhao, Xialiang Tong, Kun Mao, Zhichao Lu, Zhenkun Wang, et al. A systematic survey on large language models for algorithm design. *ACM Computing Surveys*, 2026b.

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

Zeyuan Ma, Hongshu Guo, Yue-Jiao Gong, Jun Zhang, and Kay Chen Tan. Toward automated algorithm design: A survey and practical guide to meta-black-box-optimization. *IEEE Transactions on Evolutionary Computation*, 2025.

Gerhard Reinelt. TSPLIB—A traveling salesman problem library. *ORSA Journal on Computing*, 3 (4):376–384, 1991.

John R. Rice. The algorithm selection problem. In *Advances in Computers*, volume 15, pp. 65–118. Elsevier, 1976.

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

Junjie Sheng, Shengliang Cai, Haochuan Cui, Wenhao Li, Yun Hua, Bo Jin, Wenli Zhou, Yiqiu Hu, Lei Zhu, Qian Peng, Hongyuan Zha, and Xiangfeng Wang. VMAgent: A practical virtual machine scheduling platform. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 5944–5947. International Joint Conferences on Artificial Intelligence Organization, 7 2022.

Yiding Shi, Jianan Zhou, Wen Song, Jieyi Bi, Yaoxin Wu, Zhiguang Cao, and Jie Zhang. Generalizable heuristic generation through LLMs with meta-optimization. In *International Conference on Learning Representations (ICLR)*, 2026.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K. Reddy. LLM-SR: Scientific equation discovery via programming with large language models. In *International Conference on Learning Representations (ICLR)*, 2025.

Kevin Sim, Quentin Renau, and Emma Hart. Beyond the hype: Benchmarking LLM-evolved heuristics for bin packing. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pp. 386–402. Springer, 2025.

Anja Surina, Amin Mansouri, Lars C.P.M. Quaedvlieg, Amal Seddas, Maryna Viazovska, Emmanuel Abbe, and Caglar Gulcehre. Algorithm discovery with LLMs: Evolutionary search meets reinforcement learning. In *Conference on Language Modeling (COLM)*, 2025.

Nguyen Thach, Aida Riahifar, Nathan Huynh, and Hau Chan. RedAHD: Reduction-based end-to-end automatic heuristic design with large language models. *arXiv preprint arXiv:2505.20242*, 2025.

Niki van Stein and Thomas Bäck. LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29(2): 331–345, 2025.

Rongzheng Wang, Yihong Huang, Muquan Li, Jiakai Li, Di Liang, Bob Simons, Pei Ke, Shuang Liang, and Ke Qin. Rethinking LLM-driven heuristic design: Generating efficient and specialized solvers via dynamics-aware optimization. *arXiv preprint arXiv:2601.20868*, 2026.

Xingyu Wu, Yan Zhong, Jibin Wu, Bingbing Jiang, and Kay Chen Tan. Large language model-enhanced algorithm selection: Towards comprehensive algorithm representation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 5235–5244, 2024.

Yang Wu, Junran Pan, Yifan Zhang, Ning Xu, Fanshuo Zeng, and Jian Cheng. RefineEvo: Planning-guided heuristic evolution with bidirectional experience. In *International Conference on Machine Learning (ICML)*, 2026.

Zhuoliang Xie, Fei Liu, Zhenkun Wang, and Qingfu Zhang. LLM-driven neighborhood search for efficient heuristic design. In *IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8. IEEE, 2025.

Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 24, pp. 210–216, 2010.

Xianliang Yang, Ling Zhang, Haolong Qian, Lei Song, and Jiang Bian. HeurAgenix: Leveraging LLMs for solving complex combinatorial optimization challenges. *arXiv preprint arXiv:2506.15196*, 2025.

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

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. *Advances in Neural Information Processing Systems (NeurIPS)*, 37:43571–43608, 2024.

Cunxi Yu, Rongjian Liang, Chia-Tung Ho, and Haoxing Ren. Autonomous code evolution meets NP-completeness. *arXiv preprint arXiv:2509.07367*, 2025.

Shaofeng Zhang, Shengcai Liu, Ning Lu, Jiahao Wu, Ji Liu, Yew-Soon Ong, and Ke Tang. LLM-driven instance-specific heuristic generation and selection. *arXiv preprint arXiv:2506.00490*, 2025.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *International Conference on Machine Learning (ICML)*, 2025.

This is an appendix for "Towards LLM-Based Automated Algorithm Design for Heterogeneous Datasets: Decoupled Configuration Policy and Parameterized Algorithm".

APPENDIX CONTENTS

- Appendix A: Additional Experimental Details
  - Appendix A.1: Tasks and Dataset Splits
  - Appendix A.2: Algorithm Interfaces and Seed Heuristics
  - Appendix A.3: Computational Settings
  - Appendix A.4: Assets and Licenses
- Appendix B: TS-DCPPA Implementation Details
  - Appendix B.1: Complementary-Algorithm Discovery and Stage Transition
  - Appendix B.2: Parameterized-Algorithm Fusion and Validation
  - Appendix B.3: Configuration-Policy Search
- Appendix C: Extended Discussion of Related Work
- Appendix D: More Experiments
  - Appendix D.1: DCPPA-Constrained Baselines
  - Appendix D.2: Experiments on Homogeneous Scenarios
  - Appendix D.3: Sensitivity to the LLM Backend
  - Appendix D.4: Per-Instance Behavior
- Appendix E: Case Studies
  - Appendix E.1: TSP Case Study
  - Appendix E.2: OVMP Case Study
- Appendix F: Prompts
  - Appendix F.1: Task Descriptions and Algorithm Templates
  - Appendix F.2: Parameterized-Algorithm Fusion Prompts
  - Appendix F.3: Stage 2 Configuration-Policy Prompts

A ADDITIONAL EXPERIMENTAL DETAILS

A.1 TASKS AND DATASET SPLITS

A.1.1 TSP

The TSP minimizes the total length of a Hamiltonian cycle. We use 77 EUC_2D and 4 CEIL_2D instances from TSPLIB Reinelt (1991), excluding `linhp318` because the constructive interface does not support fixed-edge constraints. VLSI[1] contains 102 EUC_2D instances from industrial circuit-design problems, and National[2] contains 27 EUC_2D country-level instances. We retain instances with fewer than 10,000 nodes. Training uses 10 randomly sampled TSPLIB and 10 VLSI instances, all with fewer than 3,000 nodes. The remaining eligible TSPLIB and VLSI instances form the ID test set, and National forms the OOD test set. Each source is split into $[0, 3000)$ and $[3000, 10000)$ node ranges.

A.1.2 OVMP

The OVMP maximizes the number of sequentially accepted virtual-machine requests subject to physical-machine (PM) resource constraints. The request sources are Huawei-East-1 Sheng et al. (2022), AzurePublicDatasetV2 Cortez et al. (2017), and the non-public BD traces. Each PM has two NUMA nodes and 80 CPU cores in total; memory capacity is 180 GB for Huawei, 320 GB for Azure, and 240 GB for BD, divided equally between the nodes. Training uses 10 Huawei and 10

[1] `https://www.math.uwaterloo.ca/tsp/vlsi/index.html`

[2] `https://www.math.uwaterloo.ca/tsp/world/countries.html`

Azure subsequences on 50 PMs, with each episode ending when the next request cannot be placed. Each ID source provides 50 test instances at each of two scales, 50 and 80 PMs. The OOD set contains 30 BD traces with more than 10,000 requests, each evaluated at both scales. Requests above a memory-demand threshold span both NUMA nodes; other requests occupy one node.

A.1.3 FJSP

The FJSP minimizes makespan by assigning each operation to an eligible machine while respecting job precedence. Processing times depend on the machine. We use benchmark suites from FJSPLib[3]: Brandimarte Brandimarte (1993) and Hurink Hurink et al. (1994) supply training and ID test instances, while DPP Dauzère-Pérès & Paulli (1997) supplies OOD instances. Training uses 4 Brandimarte and 16 Hurink instances (4 edata, 7 rdata, and 5 vdata). The ID test set contains the remaining 11 Brandimarte, 62 edata, 59 rdata, and 61 vdata instances; the OOD set contains all 18 DPP instances.

A.2 Algorithm Interfaces and Seed Heuristics

Each handcrafted seed initializes the AAD methods and provides a reference baseline. For the TSP, generated algorithms construct a tour from node 0 by selecting one unvisited node at each step. Nearest Neighbor chooses the closest unvisited node and returns to node 0 after visiting all nodes.

For the OVMP, generated algorithms use request history and current resource availability to place each arriving request. First Fit scans PMs and their NUMA nodes in index order and chooses the first feasible location. A request spanning both nodes requires each node to accommodate half its CPU and memory demand.

For the FJSP, generated algorithms select a feasible operation–machine assignment at each step. Earliest Start Time minimizes the later of the operation's ready time and the machine's availability time. Ties are broken by processing time, operation index, and machine index.

A.3 Computational Settings

Experiments run on a Linux server with an Intel(R) Xeon(R) Platinum 8457C CPU, 64 logical CPU cores, and 247 GiB memory. No GPU is used. LLM calls use a remote service; candidate algorithms are evaluated locally on 20 training instances. Baseline searches use population size 10 and evaluation time limits of 30 seconds for the TSP and FJSP and 60 seconds for the OVMP. In the heterogeneous experiments, two-stage methods use population sizes 5 and 10 and budgets of 500 evaluations per stage. Their Stage 1/Stage 2 time limits are 22/44 seconds for the TSP and FJSP and 45/90 seconds for the OVMP.

A.4 Assets and Licenses

Table 5 lists the external assets and licenses. EoH, ReEvo, and PartEvo use the LLM4AD implementations; HiFo and RefineEvo use their public repositories. We reimplement InstSpecHH following Zhang et al. (2025). Public datasets come from the listed sources. The BD traces cannot be redistributed due to confidentiality restrictions.

B TS-DCPPA Implementation Details

EoH-S and EoH implement the Stage 1 and Stage 2 searches, respectively. The task interface specifies execution constraints, including whether stochastic behavior or persistent state is permitted.

B.1 Complementary-Algorithm Discovery and Stage Transition

EoH-S uses reverse per-instance ranks during search: candidates are ranked from worst to best, so larger ranks indicate better performance. The candidate with the highest mean rank is retained first. Subsequent candidates are added greedily by their increase in the sum of the best per-instance ranks

[3] https://scheduleopt.github.io/benchmarks/fjsplib

Table 5: External assets used in this work with their sources and licenses. Reported licenses are the ones stated by the corresponding repositories or dataset websites at the time of writing; readers should consult the primary sources for the most up-to-date terms.

Category	Asset	Source	License
Method	EoH	https://github.com/Optima-CityU/llm4ad	MIT License
	ReEvo	https://github.com/Optima-CityU/llm4ad	MIT License
	PartEvo	https://github.com/Optima-CityU/llm4ad	MIT License
	HiFo	https://github.com/Challenger-XJTU/HiFo-Prompt	MIT License
	RefineEvo	https://github.com/samwu-learn/RefineEvo	MIT License
	InstSpecHH	Reimplemented by us	N/A
Dataset	TSPLIB	http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/	Academic use
	VLSI	https://www.math.uwaterloo.ca/tsp/vlsi/index.html	Academic use
	National	https://www.math.uwaterloo.ca/tsp/world/countries.html	Academic use
	Huawei-East-1	https://github.com/mail-ecnu/VMAgent	MIT License
	AzurePublicDatasetV2	https://github.com/Azure/AzurePublicDataset	CC BY 4.0
	BD	Proprietary, non-public	N/A
	Brandimarte, Hurink, DPP	https://scheduleopt.github.io/benchmarks/fjsplib	Academic use

covered by the selected set. If no candidate improves coverage, the candidate with the next-highest mean rank fills the slot. This gives a rank-based approximation to Eq. (3).

At the end of Stage 1, the complementary source set is passed to the fusion procedure. The resulting validated parameterized algorithm and source presets are then used to initialize Stage 2.

B.2 Parameterized-Algorithm Fusion and Validation

The initial fusion call and every revision call jointly produce the complete parameterized algorithm and one preset for each source algorithm. If a returned preset is absent or malformed, a separate parameter-generation call reconstructs it for the corresponding source algorithm. The resulting candidate is then compiled and checked for the required function name and task-input prefix.

Behavioral coverage is evaluated on 10 independently generated input sequences with 10 consecutive calls per sequence, giving $N_{\mathrm{val}} = 100$ task-specific inputs. Function state is retained within each sequence and reset between sequences. The generated inputs and source-algorithm outputs are cached and reused across fusion rounds. For each source algorithm, the configured candidate must achieve a behavioral similarity of at least $\tau_{\mathrm{sim}} = 0.9$. A failed comparison returns the measured similarity and current preset as revision feedback.

After code validity and behavioral coverage pass, a separate LLM review call evaluates structural alignment and returns only an issue list or an approval marker. Revision is performed by a subsequent LLM call. Early stopping requires the unchanged candidate to receive two consecutive approvals, providing a consistency check across successive review calls. We use a soft alignment-round limit of $R_{\mathrm{soft}} = 20$; after this limit, a candidate is accepted once code validity and behavioral coverage pass, while a candidate that fails either mandatory gate continues to be revised.

B.3 Configuration-Policy Search

The accepted parameterized algorithm remains fixed throughout Stage 2. Every policy-generation prompt contains the task description, a generic configuration-policy template, and the source code of the fixed algorithm. During the first $\rho_{\mathrm{src}} = 0.2$ fraction of the Stage 2 evaluation budget, the prompt additionally provides the validated source presets and requires the policy to return one preset unchanged. For the remaining evaluations, the preset table is omitted and the policy may generate new parameter values rather than selecting from the source presets.

The phase transition changes only the task description supplied to EoH. The population and evaluation history are retained, so the two phases form one continuous search rather than two independent runs.

C Extended Discussion of Related Work

LLM-based AAD. LLM-based AAD generates heuristics and algorithms as executable code, with the LLM acting as a semantic variation operator inside an iterative evaluate-and-revise loop. Beyond

the representative FunSearch and EoH, the search procedure has been strengthened through reflective prompt engineering Ye et al. (2024), diversity-aware harmony search and niching strategies Dat et al. (2025); Hu & Zhang (2025), neighborhood search Xie et al. (2025), the LLaMEA evolutionary framework van Stein & Bäck (2025), and open-ended sample-efficient program evolution Lange et al. (2025). Orthogonal directions include multi-objective heuristic evolution Yao et al. (2025), reinforcement-learning-based model adaptation and fine-tuning Surina et al. (2025); Liu et al. (2025), problem reduction before search Thach et al. (2025), and adversarial or co-evolutionary construction of training instances and test cases Duan et al. (2025); Li et al. (2025). Recent benchmarks further evaluate LLM-crafted heuristics across combinatorial tasks Chen et al. (2025). Another line decouples structural generation from numerical-constant optimization and refines constants within a fixed code skeleton Chen et al. (2026b); Shojaee et al. (2025). Unlike these methods, which calibrate constants within a fixed backbone, TS-DCPPA explicitly constructs a condition-to-configuration structure.

Algorithm selection. The algorithm selection problem Rice (1976); Kotthoff (2016), asks for a mapping from problem instances to algorithms drawn from a finite candidate set. Practical systems learn this mapping from instance features. SATzilla Xu et al. (2008) predicts algorithm performance with regression models, ISAC Kadioglu et al. (2010) clusters instances and assigns the best algorithm of each cluster, and AutoFolio Lindauer et al. (2015) configures the selection pipeline itself. Related work also considers LLM-enhanced feature representations Wu et al. (2024) and learning-theoretic guarantees for data-driven algorithm selection Chatziafratis et al. (2025). These systems select one complete algorithm from a portfolio that exists beforehand and cannot express behaviors absent from that portfolio.

Algorithm configuration. Algorithm configuration starts from a single parameterized target algorithm and optimizes its parameters, typically categorical and numerical, on a set of training instances. General-purpose configurators include local-search methods Hutter et al. (2009), sequential model-based optimization Hutter et al. (2011), iterated racing López-Ibáñez et al. (2016), and evolutionary approaches Ansótegui et al. (2009). The Google Vizier service Golovin et al. (2017) operationalizes black-box optimization and parameter tuning at scale. Hydra Xu et al. (2010) bridges configuration and selection by iteratively configuring complementary solvers into a portfolio. The common premise is that the target algorithm and its configuration space are specified manually; configuration then decides how to set parameters rather than what can be configured.

Hyper-heuristics. Hyper-heuristics operate on a space of low-level heuristics rather than directly on the solution space and either select a heuristic or generate a new one Burke et al. (2013). Selection hyper-heuristics output a heuristic identifier from a finite set, whereas generation hyper-heuristics compose new heuristics from existing components. Both typically assume the low-level heuristic set is provided in advance.

Instance-adaptive AAD with LLMs. Recent LLM-based AAD combines algorithm generation with instance-dependent control. InstSpecHH Zhang et al. (2025) partitions instances with handcrafted features and designs one heuristic per subclass; DASH Wang et al. (2026) co-optimizes solver mechanisms and runtime schedules and retrieves group-specialized solvers through profiled library retrieval; EoH-S Liu et al. (2026a) evolves a set of performance-complementary algorithms without a predefined subclass structure and without specifying a selector for new instances; DyCA Hu et al. (2026) follows the same complementary-set paradigm and dynamically clusters instances from their response vectors across generated algorithms to design one specialist per cluster; HeurAgenix Yang et al. (2025) replaces the handcrafted selector with an LLM and reselects a heuristic while solving. These methods either retain a finite portfolio as the deployed behavior or invoke the LLM at solve time. TS-DCPPA instead constructs the configurable space automatically through fusion and deploys a program that maps conditions to continuous parameters without LLM calls.

D MORE EXPERIMENTS

D.1 DCPPA-CONSTRAINED BASELINES

We extend EoH-DCPPA to ReEvo, PartEvo, HiFo, and RefineEvo. Each baseline directly generates a DCPPA-structured algorithm under the same structural constraint, evaluation budget, and task interface. These variants test whether the structure alone accounts for the gains of TS-DCPPA.

TS-DCPPA has the lowest gap in all six columns of Table 6, with average rank 1.00 versus 3.17 for the strongest constrained baseline, ReEvo-DCPPA. Relative to the best competitor in each column, its TSP gap reductions are 1.26 and 12.27 percentage points on ID and OOD instances, respectively. The corresponding reductions are 0.53 and 0.48 on the OVMP, and 0.42 and 2.50 on the FJSP. These results support the two-stage search beyond directly imposing the DCPPA structure.

Table 6: DCPPA-constrained baselines, reporting $\overline{\mathrm{Gap}}$ (%) on the main ID/OOD partitions. Lower is better; **bold** and underline mark the best and second-best results. The seed heuristic is included in the ranking.

Method	TSP		OVMP		FJSP		Avg. Rank
	ID	OOD	ID	OOD	ID	OOD	
Seed heuristic	26.33	40.18	2.30	3.84	25.41	27.49	6.83
EoH-DCPPA	21.39	35.96	1.56	2.13	13.52	14.40	3.33
ReEvo-DCPPA	19.74	33.27	2.22	2.86	11.48	11.40	3.17
PartEvo-DCPPA	22.88	37.22	1.96	3.04	13.93	16.03	5.50
HiFo-DCPPA	22.11	34.93	1.61	2.39	12.32	15.79	3.67
RefineEvo-DCPPA	29.85	39.72	1.65	2.45	11.53	13.58	4.50
TS-DCPPA (Ours)	**18.49**	**20.99**	**1.03**	**1.65**	**11.07**	**8.89**	**1.00**

TS-DCPPA also has the lowest mean per-instance rank in every column of Table 7, averaging 1.77 versus 3.25 for ReEvo-DCPPA. The FJSP-ID margin is smaller: TS-DCPPA scores 2.75, followed by RefineEvo-DCPPA at 2.89 and ReEvo-DCPPA at 3.02. The paired tests in Appendix D.4 find no significant difference against these two baselines on FJSP-ID.

Table 7: Mean per-instance rank for the DCPPA-constrained baselines, computed as in Table 2. Lower is better; **bold** and underline mark the best and second-best results. The seed heuristic is included in the ranking.

Method	TSP		OVMP		FJSP		Avg.
	ID	OOD	ID	OOD	ID	OOD	
Seed heuristic	6.06	6.00	5.84	6.26	6.27	6.33	6.13
EoH-DCPPA	3.64	3.79	3.10	2.57	4.58	4.39	3.68
ReEvo-DCPPA	2.23	2.16	4.85	5.12	3.02	2.11	3.25
PartEvo-DCPPA	4.86	5.16	5.10	5.72	4.58	5.33	5.13
HiFo-DCPPA	4.06	3.58	3.23	3.32	3.92	5.39	3.92
RefineEvo-DCPPA	5.54	5.74	3.78	3.48	2.89	3.39	4.13
TS-DCPPA (Ours)	**1.60**	**1.58**	**2.09**	**1.53**	**2.75**	**1.06**	**1.77**

D.2 EXPERIMENTS ON HOMOGENEOUS SCENARIOS

We also compare the methods on homogeneous synthetic TSP and OVMP instances, where training and test data share a common distribution. This setting tests whether TS-DCPPA remains competitive under the same evaluation budget when source heterogeneity is absent.

TS-DCPPA attains the lowest gap at every scale in Table 8. Averaged equally over the three TSP scales, its gap is 15.78%, compared with 24.47% for Nearest Neighbor and 16.46% for the strongest static baseline, RefineEvo. The corresponding OVMP averages are 0.66%, 2.19% for First Fit, and

0.73% for EoH. Thus, the gains extend to these homogeneous settings, although the margins over the strongest baselines are smaller.

Table 8: Results on homogeneous synthetic TSP and OVMP instances, reporting $\overline{\text{Gap}}$ (%) at three scales per task. All columns are ID. Lower is better; **bold** and underline mark the best and second-best results. The seed heuristic is included in the ranking.

Method	TSP			OVMP			Avg. Rank
	n=100	n=1000	n=10000	20 PMs	50 PMs	80 PMs	
Seed heuristic	24.78	24.78	23.86	2.44	2.23	1.90	7.00
EoH	<u>14.87</u>	16.88	<u>17.84</u>	<u>1.26</u>	<u>0.57</u>	<u>0.36</u>	<u>2.33</u>
ReEvo	15.10	<u>16.44</u>	18.01	1.44	0.74	0.45	4.00
PartEvo	17.09	19.65	20.33	1.29	0.77	0.56	5.33
HiFo	15.50	17.21	18.32	1.32	0.78	0.47	5.17
RefineEvo	14.92	16.46	17.99	1.31	0.68	0.40	3.17
TS-DCPPA (Ours)	**14.27**	**15.49**	**17.57**	**1.12**	**0.53**	**0.32**	**1.00**

The mean per-instance ranks in Table 9 give the same conclusion. TS-DCPPA ranks best at every scale, with task-wise averages of 2.39 on the TSP and 2.59 on the OVMP, compared with 3.14 and 3.04 for EoH, the strongest baseline under this metric.

Table 9: Mean per-instance rank for the homogeneous experiments, computed as in Table 2. Lower is better; **bold** and underline mark the best and second-best results. The seed heuristic is included in the ranking.

Method	TSP			OVMP			Avg. rank
	n=100	n=1000	n=10000	20 PMs	50 PMs	80 PMs	
Seed heuristic	6.22	6.98	7.00	5.67	6.69	6.82	6.56
EoH	<u>3.24</u>	3.54	<u>2.62</u>	3.92	<u>2.67</u>	<u>2.53</u>	<u>3.09</u>
ReEvo	3.52	<u>2.76</u>	3.19	4.35	4.03	3.81	3.61
PartEvo	4.90	5.94	6.00	3.62	4.17	5.37	5.00
HiFo	3.68	4.16	3.88	<u>3.59</u>	4.39	3.87	3.93
RefineEvo	3.30	2.96	2.94	3.76	3.63	3.33	3.32
TS-DCPPA (Ours)	**3.14**	**1.66**	**2.38**	**3.09**	**2.42**	**2.27**	**2.49**

D.3 Sensitivity to the LLM Backend

We repeat the heterogeneous TSP and OVMP comparisons with Doubao-Seed-2.1-Turbo, including the five static baselines, the seed heuristics, and TS-DCPPA. The evaluation budget and test sets match the DeepSeek-V4-Flash-GA experiments.

Under Doubao-Seed-2.1-Turbo, TS-DCPPA again has the lowest gap in all four columns (Table 10), with average rank 1.00 versus 2.75 for EoH. Relative to DeepSeek-V4-Flash-GA, its gap changes by less than 1 percentage point on each TSP partition and by 0.34 and 0.03 on OVMP-ID and OVMP-OOD. The advantage therefore holds for both tested backends.

TS-DCPPA also has the lowest mean per-instance rank in every column (Table 11), averaging 1.51 versus 3.46 for HiFo. Its advantage under the second backend is therefore visible in both gap magnitude and per-instance ranking.

D.4 Per-Instance Behavior

Figures 5–7 compare the five-run mean gaps of individual instances on four partitions: TSPLIB and National with $[3000, 10000)$ nodes, and Huawei and BD with 80 PMs. These cover ID and OOD sources for both tasks. The main comparison shows the clearest separation on the OOD partitions; the ablation and constrained-baseline plots show how this pattern changes across method variants.

Table 10: Heterogeneous TSP and OVMP results with Doubao-Seed-2.1-Turbo, reporting $\overline{\text{Gap}}$ (%) on the main ID/OOD partitions. Lower is better; **bold** and underline mark the best and second-best results. The seed heuristic is included in the ranking.

Method	TSP		OVMP		Avg. Rank
	ID	OOD	ID	OOD	
Seed heuristic	26.33	40.18	2.30	3.84	7.00
EoH	19.51	34.04	1.41	2.50	2.75
ReEvo	20.29	34.21	1.62	2.42	3.75
PartEvo	22.24	36.87	1.58	2.67	5.50
HiFo	21.51	34.42	1.22	1.99	3.25
RefineEvo	19.73	34.69	2.03	2.63	4.75
TS-DCPPA (Ours)	**17.71**	**20.46**	**0.69**	**1.68**	**1.00**

Table 11: Mean per-instance rank with Doubao-Seed-2.1-Turbo, computed as in Table 2. Lower is better; **bold** and underline mark the best and second-best results. The seed heuristic is included in the ranking.

Method	TSP		OVMP		Avg. rank
	ID	OOD	ID	OOD	
Seed heuristic	6.46	6.32	6.21	6.37	6.34
EoH	2.94	2.95	3.90	4.36	3.54
ReEvo	3.70	3.42	3.50	3.79	3.60
PartEvo	5.35	5.84	4.54	4.97	5.18
HiFo	4.77	4.00	2.95	2.11	3.46
RefineEvo	3.11	3.84	5.51	5.04	4.38
TS-DCPPA (Ours)	**1.65**	**1.63**	**1.40**	**1.36**	**1.51**

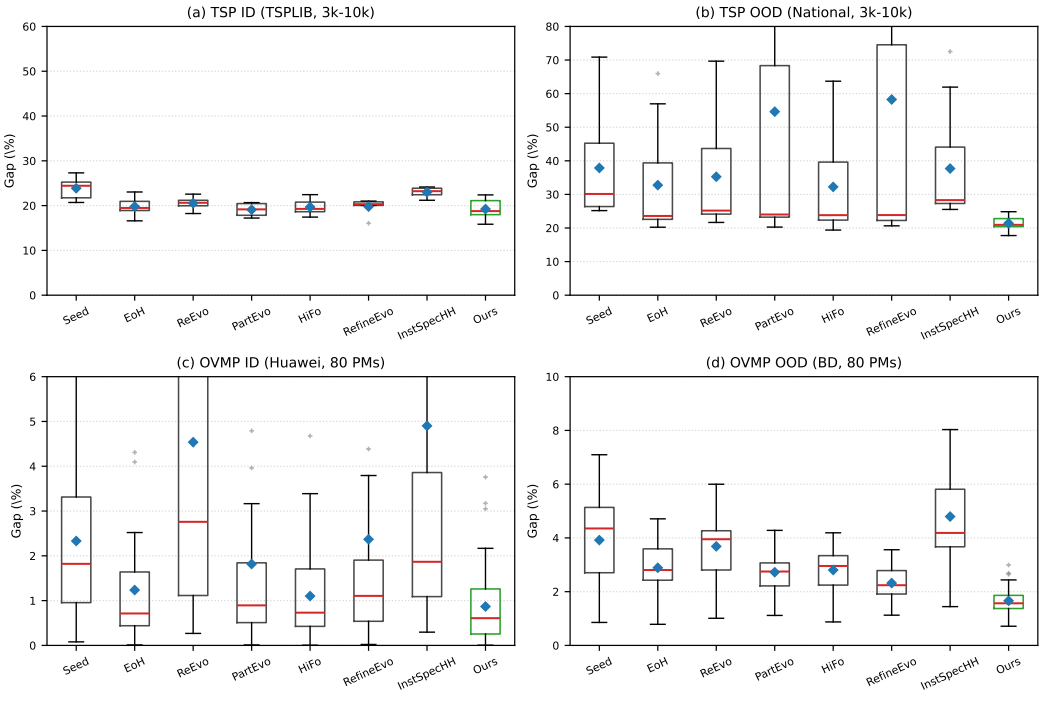

Figure 5: Per-instance five-run mean gaps for the main comparison on four ID/OOD partitions. Boxes show the interquartile range, red lines the median, blue diamonds the mean, and grey pluses outliers. TS-DCPPA is shown in green. Values above the y-axis limits are clipped.

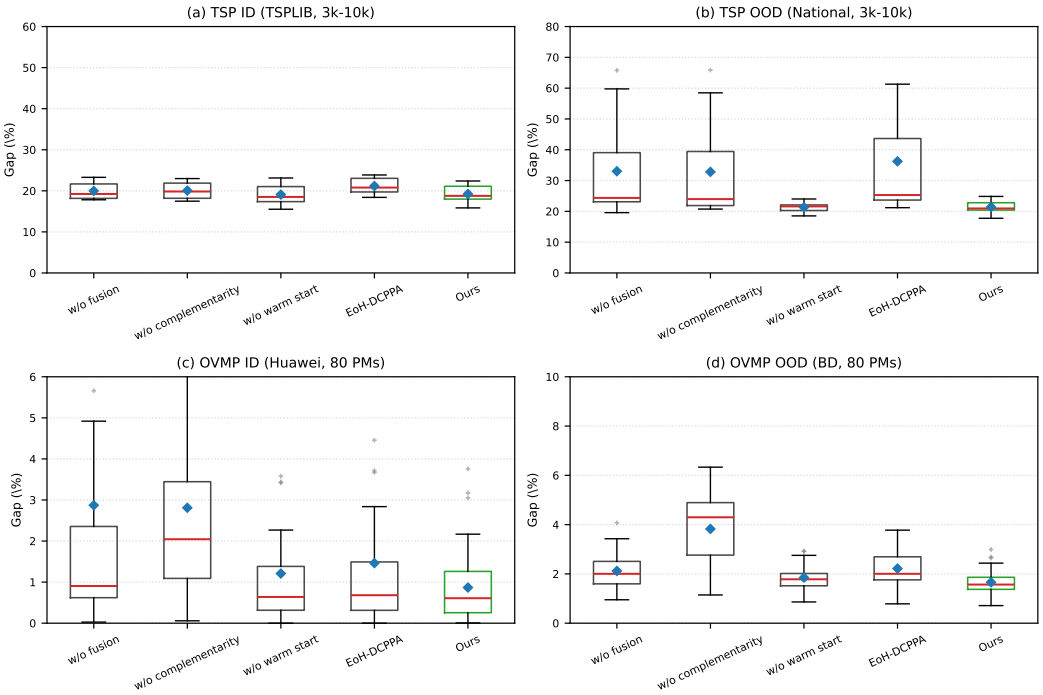

Figure 6: Per-instance five-run mean gaps for the ablation study on the same partitions as Figure 5, using the same visual conventions. Values above the y-axis limits are clipped.

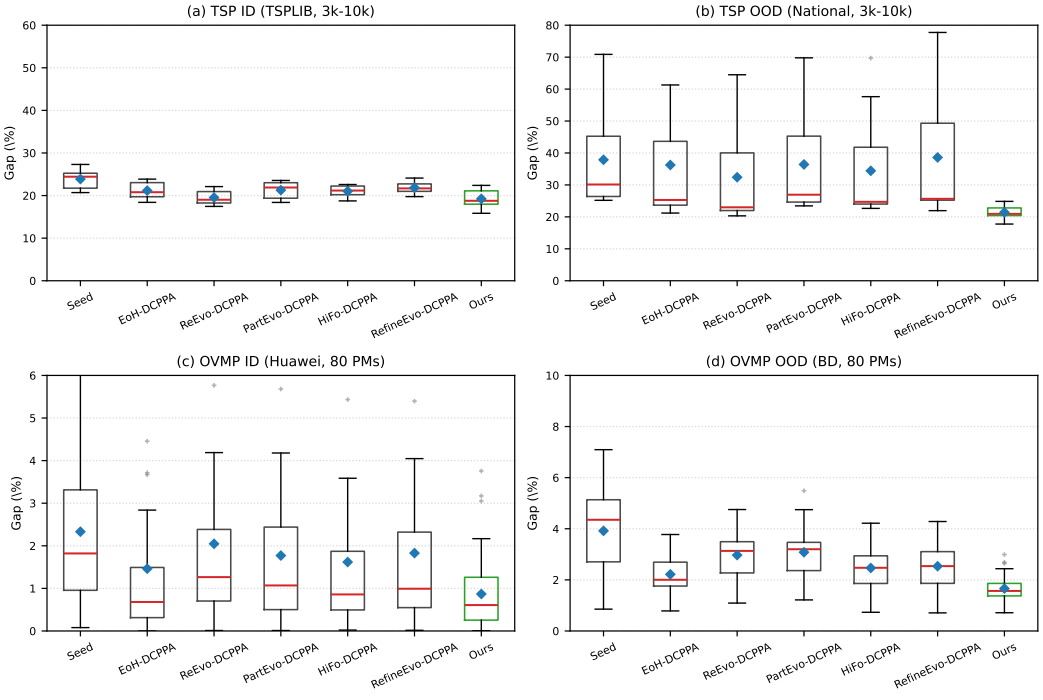

Figure 7: Per-instance five-run mean gaps for the DCPPA-constrained baselines on the same partitions as Figure 5, using the same visual conventions. Values above the y-axis limits are clipped.

We also apply two-sided paired Wilcoxon signed-rank tests to the per-instance five-run mean gaps of TS-DCPPA and each competitor. Holm correction controls the family-wise error rate jointly across all row×column comparisons within each table; values below 10^{-3} are shown as $<10^{-3}$. In the main comparison (Table 12), all 42 adjusted p-values are below 0.05, including 36 below 10^{-3}.

Table 12: Holm-adjusted two-sided paired Wilcoxon signed-rank p-values between TS-DCPPA and each row in the main comparison (Table 1). Values are computed on per-instance mean gaps across the five runs; correction is applied jointly across all row×column pairs.

Method	TSP		OVMP		FJSP	
	ID	OOD	ID	OOD	ID	OOD
Seed heuristic	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
EoH	$<10^{-3}$	0.004	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
ReEvo	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
PartEvo	$<10^{-3}$	0.01	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
HiFo	$<10^{-3}$	0.02	$<10^{-3}$	$<10^{-3}$	0.001	$<10^{-3}$
RefineEvo	$<10^{-3}$	0.004	$<10^{-3}$	$<10^{-3}$	0.01	$<10^{-3}$
InstSpecHH	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$

In the ablation study (Table 13), w/o fusion, w/o complementarity, and EoH-DCPPA differ significantly from TS-DCPPA on all six partitions. For w/o warm start, the differences are significant on OVMP-ID, OVMP-OOD, and FJSP-ID, but not on TSP-ID, TSP-OOD, or FJSP-OOD (adjusted $p = 0.72$, 0.84, and 1.00).

Table 13: Holm-adjusted two-sided paired Wilcoxon signed-rank p-values between TS-DCPPA and each ablation variant (Table 3).

Method	TSP		OVMP		FJSP	
	ID	OOD	ID	OOD	ID	OOD
w/o fusion	$<10^{-3}$	0.02	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
w/o complementarity	$<10^{-3}$	0.008	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
w/o warm start	0.72	0.84	$<10^{-3}$	$<10^{-3}$	0.02	1.00
EoH-DCPPA	$<10^{-3}$	0.001	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$

For the DCPPA-constrained comparison (Table 14), all differences are significant except against ReEvo-DCPPA ($p = 0.16$) and RefineEvo-DCPPA ($p = 0.12$) on FJSP-ID. These two baselines also have the smallest FJSP-ID gap margins.

Table 14: Holm-adjusted two-sided paired Wilcoxon signed-rank p-values between TS-DCPPA and each DCPPA-constrained baseline (Table 6).

Method	TSP		OVMP		FJSP	
	ID	OOD	ID	OOD	ID	OOD
Seed heuristic	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
EoH-DCPPA	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
ReEvo-DCPPA	$<10^{-3}$	0.02	$<10^{-3}$	$<10^{-3}$	0.16	$<10^{-3}$
PartEvo-DCPPA	$<10^{-3}$	0.002	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
HiFo-DCPPA	$<10^{-3}$	0.002	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
RefineEvo-DCPPA	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	0.12	$<10^{-3}$

E CASE STUDIES

We examine one representative TS-DCPPA run on each of the TSP and OVMP tasks, showing the source algorithms, recovered presets, and learned configuration policy.

E.1 TSP CASE STUDY

The TSP source set contains five constructive node-selection algorithms. They combine three scoring terms with different weights: a regret bonus based on the difference between the two smallest current-to-unvisited distances, a destination-distance term favoring later visits to nodes near the destination, and a local-spread term. Sources 1 and 5 place greater weight on regret, while Sources 2–4 include nonzero spread weights (Table 15).

Fusion exposes seven parameters: `regret_weight`, `regret_sharpness`, `dest_weight`, `cur_norm_weight`, `spread_weight`, `gap_weight`, and `spread_fallback`. Each source preset matches its source algorithm on all 100 validation calls.

Table 15: TSP source presets. All five presets match their sources on $100/100$ validation calls. The remaining parameters, `regret_sharpness` and `spread_fallback`, are zero for every source.

Source	`regret_weight`	`dest_weight`	`cur_norm_weight`	`spread_weight`	`gap_weight`
Source 1	0.80	1.00	0.40	0.00	0.00
Source 2	0.30	0.70	0.00	0.10	0.00
Source 3	0.30	0.80	0.00	0.20	0.00
Source 4	0.25	0.70	0.00	0.15	0.15
Source 5	0.70	0.90	0.30	0.00	0.00

The deterministic configuration policy uses the current construction state. It computes a bottleneck score $(d_{(2)} - d_{(1)})/\Delta$, where $d_{(1)}$ and $d_{(2)}$ are the two smallest current-to-unvisited distances and Δ is the global distance range. It also uses the normalized nearest distance, distance-histogram entropy, the difference between mean distances from the current and destination nodes, and the fraction of unvisited nodes. These signals control the scoring parameters:

- Higher risk, derived from the bottleneck, nearest distance, and destination-distance difference, increases `regret_weight` and `spread_weight` and decreases `dest_weight`.
- Higher distance-histogram entropy increases `regret_sharpness` and `gap_weight`.
- When fewer than 15% of nodes remain, the policy increases `regret_weight`, `dest_weight`, and `cur_norm_weight`.

The policy thus updates the scoring rule at each construction step and can use parameter settings beyond the five source presets.

E.2 OVMP CASE STUDY

The five OVMP sources form two scoring families. Sources 1, 4, and 5 combine residual resources, CPU–memory imbalance, a minimum-resource bonus, and decaying load history. Sources 2 and 3 instead match each NUMA node's CPU fraction to recent requests and include a hotspot penalty. These families expose different ways to account for workload composition and cluster state.

Fusion unifies the two families through 23 parameters. On 10 validation sequences of 10 calls each, the five presets achieve similarities of 0.99, 0.90, 1.00, 1.00, and 1.00, respectively. The toggles `use_target_frac`, `use_hotspot`, and `normalize_by_avg` distinguish the target-fraction family (all 1) from the residual family (all 0). Table 16 shows selected varying parameters.

Table 16: Selected source presets recovered by fusion on the OVMP. Each row is a validated configuration of the parameterized NUMA scorer that reproduces the corresponding Stage 1 source. Only the parameters that vary across sources are shown.

Source	`residual_w`	`imbalance_w`	`load_w`	`min_bonus_w`	`target_frac_w`	`corr_scale`	`hotspot_w`	`window`
Source 1	3.0	1.0	1.0	2.0	0.0	0.0	0.0	30
Source 2	0.3	0.0	0.0	0.0	12.0	6.0	1.5	30
Source 3	2.0	0.0	0.0	0.0	10.0	5.0	2.0	30
Source 4	3.0	1.0	1.5	2.0	0.0	0.0	0.0	100
Source 5	3.0	1.0	1.5	1.5	0.0	0.0	0.0	40

The OVMP policy uses recent request CPU fractions, the double-NUMA request ratio, cluster CPU–memory imbalance, and smoothed placement pressure p, computed from incoming request size relative to mean residual node capacity. It also tracks normalized CPU-fraction variance, named "entropy" in the generated code. The following branches are evaluated in order:

- $p > 0.9$ and cluster imbalance > 0.12: initialize the target-fraction weights from Source 2.
- $p > 0.75$, or a double-NUMA ratio > 0.6 with $p > 0.5$: use the residual-based settings of Source 5, with window size 40.
- Normalized CPU-fraction variance > 0.4 and mean CPU fraction < 0.5: use Source 4, with window size 100.
- Mean CPU fraction in $[0.35, 0.65]$ and cluster imbalance > 0.08: initialize the target-fraction weights from Source 3.
- Otherwise: use Source 1 with pressure-dependent residual and load weights.

After selecting a branch, the policy continuously modulates selected scoring weights using pressure and request CPU fraction, and adjusts the history window and smoothing rate. This combines discrete branch selection with continuous parameter adjustment within a shared placement algorithm.

F PROMPTS

This section reports the task descriptions, algorithm templates, fusion prompts, and configuration-policy prompts used in the reported TS-DCPPA experiments. Bracketed uppercase fields denote values substituted at runtime or verbatim fixed blocks whose full contents are inserted before an LLM call. Repeated text is factored into named blocks, but the instructions are otherwise reported without abbreviation. The standard reproduction prompts of EoH-S and EoH are not repeated because TS-DCPPA retains these templates and changes only the task context supplied to them. Constraints on determinism and function state in these prompts belong to the three evaluated task interfaces and are not general requirements of DCPPA.

F.1 TASK DESCRIPTIONS AND ALGORITHM TEMPLATES

For all three tasks, function attributes may retain information across consecutive decisions within an instance and are cleared before the next instance.

Task description for the TSP

```
The Traveling Salesman Problem (TSP) entails determining the shortest Hamiltonian cycle for
      a given set of nodes, such that each node is visited exactly once before returning to
      the origin. This problem can be addressed through an iterative constructive process,
     wherein the next node is sequentially selected from the set of unvisited nodes. Please
      design a novel algorithm to select the next node in each step.

Crucially, the algorithm should be simple and compact, using a few clear decision rules and
      avoiding complex search, simulation, and nested helper logic.

Constraints:
1. The algorithm must be strictly deterministic, excluding any stochastic components.
2. The generated Python code must be optimized for speed.
3. Function attributes can be used to store persistent data within one problem instance. In
      subsequent function calls, this persistent data can be updated and reused to
     accelerate execution. Avoid recomputing the same quantities across calls if they can
     be cached. Cached function attributes are automatically cleared before each instance
     evaluation. Initialize missing attributes; no cross-instance cache invalidation or
     instance-change detection is needed.
4. Use the pre-imported package as needed: np (as an alias for numpy). Do not use other
     packages or add any import statements in the function.
```

Algorithm template for the TSP

```
import numpy as np
```

```
def select_next_node(current_node: int, destination_node: int, unvisited_nodes: np.ndarray,
      distance_matrix: np.ndarray) -> int:
    """
    Select the next node in each step.

    Note: The logic is entirely contained within the function body, with no external
     classes or global functions defined.

    Args:
        current_node: ID of the current node.
        destination_node: ID of the destination node.
        unvisited_nodes: Array of IDs of unvisited nodes.
        distance_matrix: Distance matrix of the given set of nodes.

    Return:
        ID of the next node to visit.
    """
    # Below is a placeholder implementation that selects the first unvisited node.
    next_node = unvisited_nodes[0]
    return next_node
```

Task description for the OVMP

```
Given a set of NUMA nodes within a datacenter, each with specific available CPU cores and
     memory capacity, design a novel algorithm to select the optimal NUMA node for an
     incoming resource request. The algorithm must consider: (1) the request's CPU and
     memory requirements, (2) whether the request needs to span one or two NUMA nodes, and
     (3) the current resource availability across all NUMA nodes.

The objective is to maximize total request throughput: the number of successfully placed
     requests before the first request with no feasible placement under its single- or
     double-NUMA requirements.

The algorithm must be strictly deterministic, excluding any stochastic components. The code
      must be optimized for speed to ensure low latency. Use the pre-imported package as
     needed: np (as an alias for numpy). Do not use other packages or add any import
     statements in the function. Function attributes can be used to store persistent data
     within one problem instance. In subsequent function calls, this persistent data can be
      updated and reused to accelerate execution. Avoid recomputing the same quantities
     across calls if they can be cached. Cached function attributes are automatically
     cleared before each instance evaluation. Initialize missing attributes; no cross-
     instance cache invalidation or instance-change detection is needed.
```

Algorithm template for the OVMP

```
import numpy as np

def select_next_NUMA(req_hist: np.ndarray, observation: np.ndarray, availability: np.
     ndarray) -> int:
    """
    Selects a NUMA node for placing an incoming resource request.

    Note: The logic is entirely contained within the function body, with no external
     classes or global functions defined.

    System Architecture:
        - N physical machines in the cluster
        - Each physical machine has exactly 2 NUMA nodes
        - Total NUMA nodes in cluster: 2N (numbered 0 to 2N-1)
        - NUMA node ID mapping: for machine i and NUMA j, ID = 2*i + j

    Args:
        req_hist (np.ndarray): Sliding window of the most recent requests in chronological
     order, with the incoming request at req_hist[-1].
            Shape: (M, 3) where M is the current window length, bounded by the configured
     history window size
            This window does not necessarily contain the complete request history.
            Structure: Each row represents a single request [cpu_cores, memory_gb,
     is_double_NUMA]
                - Axis 0: Earlier requests at lower indices, most incoming request at index
      -1
                - Axis 1 (request attributes):
                    - [0] cpu_cores: CPU requirement in cores
                    - [1] memory_gb: Memory requirement in GB
```

```
                    - [2] is_double_NUMA: Request placement mode flag (0 or 1)
                        - 1: Request spans both NUMA nodes of a single physical machine
                             Resources are equally divided: each NUMA gets request/2
                        - 0: Request is placed on a single NUMA node

        observation (np.ndarray): Current available resources across all NUMA nodes.
            Shape: (N, 2, 2) where N is the number of physical machines
            Structure: observation[machine_id][numa_id] = [available_cpu, available_memory]
            Example for 2 machines:
                [[[cpu_m0_n0, mem_m0_n0], [cpu_m0_n1, mem_m0_n1]],  # Machine 0
                 [[cpu_m1_n0, mem_m1_n0], [cpu_m1_n1, mem_m1_n1]]]  # Machine 1

        availability (np.ndarray): NUMA node availability mask.
            Shape: (2N,) boolean array where True indicates the NUMA node can accept the
     request
            Index mapping: availability[2*i + j] corresponds to machine i, NUMA j

    Returns:
        int: Selected NUMA node ID (0 to 2N-1)

        For single-NUMA requests (is_double_NUMA=False):
            - Returns specific NUMA node ID where request will be placed

        For double-NUMA requests (is_double_NUMA=True):
            - Returns either NUMA ID of selected machine (2*i or 2*i+1 are equivalent)
            - Request will automatically span both NUMA nodes of the chosen machine
    """
    # Below is a placeholder implementation that selects the first available NUMA node.
    action = np.where(availability == 1)[0][0]
    return action
```

Task description for the FJSP

```
Given a set of jobs and machines, this is a Flexible Job Shop Scheduling Problem (FJSP).
     Each job consists of operations linked by precedence constraints, and each operation
     can be processed on any one of several eligible machines, each with its own processing
      time. The goal is to schedule all operations to minimize the makespan (the maximum
     completion time over all operations), subject to: (i) an operation can start only after
      all its predecessor operations have finished, and (ii) each machine processes at most
      one operation at a time. The schedule is built step by step: at each step you are
     given the candidate assignments of all currently ready operations (each candidate
     pairs a ready operation with one of its eligible machines and the corresponding
     processing time), current dynamic status, and read-only global instance data, and you
     must select one assignment to schedule next. Design a novel algorithm for this
     selection.

Crucially, the algorithm should be simple and compact, using a few clear decision rules and
      avoiding complex search, simulation, and nested helper logic.

Constraints:
1. The algorithm must be strictly deterministic, excluding any stochastic components.
2. The function must return exactly one element taken from feasible_operations (the
     selected (operation_id, machine_id, processing_time) tuple); returning an assignment
     that is not in feasible_operations is invalid.
3. The generated Python code must be optimized for speed.
4. Use the pre-imported package as needed: np (as an alias for numpy). Do not use other
     packages or add any import statements in the function.
5. Function attributes can be used to store persistent data within one problem instance. In
      subsequent function calls, this persistent data can be updated and reused to
     accelerate execution. Avoid recomputing the same quantities across calls if they can
     be cached. Cached function attributes are automatically cleared before each instance
     evaluation. Initialize missing attributes; no cross-instance cache invalidation or
     instance-change detection is needed.
6. Keep all algorithm logic directly inside determine_next_operation. Do not define
     external classes, global helper functions, or additional top-level code.
```

Algorithm template for the FJSP

```
import numpy as np

def determine_next_operation(current_status, feasible_operations):
    """
    Determine the next (operation, machine) assignment to schedule based on a dispatching
     rule.
```

```
    Args:
        current_status: A dict with dynamic scheduling status and read-only instance data:
            Dynamic current status:
            - 'machine_status': list[int], the time each machine becomes available.
            - 'operation_status': list[int], the ready time of each operation
              (max finish time of its predecessors; only meaningful for ready operations).
            - 'scheduled_operations': list[bool], whether each operation has already
              been scheduled.
            Read-only global instance data (not a current/remaining status):
            - 'options': tuple[tuple[tuple[int, int | float], ...], ...], where
              options[op] lists eligible (machine_id, duration) pairs for operation op.
            - 'predecessors': tuple[tuple[int, ...], ...], where predecessors[op]
              lists the direct predecessor operations of op.
            - 'successors': tuple[tuple[int, ...], ...], where successors[op] lists
              the direct successor operations of op.
            - 'operation_to_job': tuple[int, ...], mapping each operation to its job.
            - 'job_operations': tuple[tuple[int, ...], ...], where job_operations[j]
              lists the operations belonging to job j.
        feasible_operations: A list of candidate assignments. Each element is a tuple
            (operation_id, machine_id, processing_time) describing one eligible machine
            option of a currently ready (all predecessors finished, not yet scheduled)
            operation. A flexible operation contributes one candidate per machine option.

    Return:
        The selected assignment, a tuple (operation_id, machine_id, processing_time)
        taken from feasible_operations.
    """
    # Baseline: earliest-start dispatching rule, with SPT as a tie-breaker.
    machine_status = current_status["machine_status"]
    operation_status = current_status["operation_status"]
    next_operation = min(
        feasible_operations,
        key=lambda x: (
            max(operation_status[x[0]], machine_status[x[1]]),
            x[2],
            x[0],
            x[1],
        ),
    )
    return next_operation
```

F.2 Parameterized-Algorithm Fusion Prompts

The following templates are shared across the three tasks. Initial synthesis produces a complete parameterized algorithm and one parameter setting for each source algorithm. Each candidate is then checked for code validity and behavioral coverage. A candidate that passes both checks is assessed in a separate LLM call, which returns either a list of substantive issues or ==DONE==, without generating code. When issues are reported, a subsequent revision call generates an updated candidate and parameter settings. A candidate that fails either predefined check is revised directly from the corresponding diagnostic feedback. Two consecutive ==DONE== responses permit early stopping; after the soft round limit, acceptance requires the two predefined checks to pass.

To avoid repeating long fixed text, the stage-specific templates refer to the following blocks. Before an LLM call, each named block and bracketed field is replaced by its full content or current runtime value.

Fixed blocks for parameterized-algorithm fusion

```
[FUSION REQUIREMENTS]
1. Produce ONE Python function architecture that can approximate ANY provided algorithm by
     setting parameters.
2. Keep the original function signature as a prefix:
   [FUNCTION SIGNATURE]
3. Expose architecture parameters as additional function inputs without defaults.
4. The architecture must be a unified mechanism, not a concatenation of copied algorithms.
5. Architecture parameters must control shared computation, not select a copied whole-
     algorithm branch.
6. Keep the implementation compact and maintainable.

[SOURCE-PARAMETER REQUIREMENTS]
```

````
When producing a new or revised architecture, apply the following requirements to EACH
     source algorithm in [Algorithms], treating it as the target algorithm:
Derive one parameter setting for the parameterized architecture so that it approximates the
      target algorithm's behavior.
1. First, briefly analyze how the target algorithm's key decisions correspond to the
     architecture parameters.
2. Do not use fenced code blocks in the analysis.
3. Provide a dictionary of fixed values containing only the additional architecture
     parameters.
4. Do not omit, add, or rename any architecture parameter.
5. If the architecture has no additional parameters, use an empty dictionary {} for this
     target algorithm.
6. Place each dictionary under its existing 1-based Algorithm number in "source_parameters".
      Example names, values, and parameter count are illustrative; follow the actual
     architecture signature.
7. Use JSON values only (no Python expressions, NaN, or Infinity). Do not place the
     parameter table or a parameter generator inside the architecture code block.

[REVIEW CHECKLIST]
- Interface compatibility: keep the original function name and required input prefix
     unchanged; expose only additional architecture parameters without defaults.
- Return compatibility: preserve the output type and semantics of the source algorithms.
- State usage: function attributes can be used to store persistent data within one problem
     instance. In subsequent function calls, this persistent data can be updated and reused
      to accelerate execution. Avoid recomputing the same quantities across calls if they
     can be cached. Cached function attributes are automatically cleared before each
     instance evaluation. Initialize missing attributes; no cross-instance cache
     invalidation or instance-change detection is needed.
- Determinism: ensure repeated calls produce the same behavior when inputs, parameters, and
      any allowed function-attribute state are the same.
- No dispatcher structure: do not select among copied whole-algorithm branches.
- Unified mechanism: use shared intermediate features and let parameters modulate the
     computation smoothly.
- Continuous controls: prefer weights, thresholds, penalties, and other tunable continuous
     parameters over discrete template selectors.
- Configurator friendliness: keep architecture parameters interpretable and easy to tune.
- Runtime efficiency: keep the computation fast enough and avoid unnecessary heavy
     computation.
- Implementation simplicity: remove redundant parameters, unused variables, duplicated
     logic, unnecessary helper layers, and dead code.
- Comment discipline: include comments or docstrings only when they clarify non-obvious
     logic.

[ARCHITECTURE-AND-PRESET OUTPUT]
```python
# the complete fused function code only
```
```json
{
  "source_parameters": {
    "1": {"parameter_name_1": 0.5, "parameter_name_2": 2.0},
    "2": {"parameter_name_1": 1.0, "parameter_name_2": 0.3}
  }
}
```
````

Task-specific interface blocks for fusion

```
[TSP TASK INTERFACE AND RETURN SEMANTICS]
Note:
    Args:
        current_node (int): ID of the current node.
        destination_node (int): ID of the destination node.
        unvisited_nodes (np.ndarray): Array of IDs of unvisited nodes.
        distance_matrix (np.ndarray): Distance matrix of the given set of nodes.

    Return:
        ID of the next node to visit.

[OVMP TASK INTERFACE AND RETURN SEMANTICS]
Note:
    System Architecture:
        - N physical machines in the cluster
        - Each physical machine has exactly 2 NUMA nodes
        - Total NUMA nodes in cluster: 2N (numbered 0 to 2N-1)
        - NUMA node ID mapping: for machine i and NUMA j, ID = 2*i + j
```

```
    Args:
        req_hist (np.ndarray): Sliding window of the most recent requests in chronological
     order, with the incoming request at req_hist[-1].
            Shape: (M, 3) where M is the current window length, bounded by the configured
     history window size
            This window does not necessarily contain the complete request history.
            Structure: Each row represents a single request [cpu_cores, memory_gb,
     is_double_NUMA]
                - Axis 0: Earlier requests at lower indices, most incoming request at index
      -1
                - Axis 1 (request attributes):
                    - [0] cpu_cores: CPU requirement in cores
                    - [1] memory_gb: Memory requirement in GB
                    - [2] is_double_NUMA: Request placement mode flag (0 or 1)
                        - 1: Request spans both NUMA nodes of a single physical machine
                             Resources are equally divided: each NUMA gets request/2
                        - 0: Request is placed on a single NUMA node

        observation (np.ndarray): Current available resources across all NUMA nodes.
            Shape: (N, 2, 2) where N is the number of physical machines
            Structure: observation[machine_id][numa_id] = [available_cpu, available_memory]
            Example for 2 machines:
                [[[cpu_m0_n0, mem_m0_n0], [cpu_m0_n1, mem_m0_n1]],  # Machine 0
                 [[cpu_m1_n0, mem_m1_n0], [cpu_m1_n1, mem_m1_n1]]]  # Machine 1

        availability (np.ndarray): NUMA node availability mask.
            Shape: (2N,) boolean array where True indicates the NUMA node can accept the
     request
            Index mapping: availability[2*i + j] corresponds to machine i, NUMA j

    Returns:
        int: Selected NUMA node ID (0 to 2N-1)

        For single-NUMA requests (is_double_NUMA=False):
            - Returns specific NUMA node ID where request will be placed

        For double-NUMA requests (is_double_NUMA=True):
            - Returns either NUMA ID of selected machine (2*i or 2*i+1 are equivalent)
            - Request will automatically span both NUMA nodes of the chosen machine

[FJSP TASK INTERFACE AND RETURN SEMANTICS]
Note:
    Args:
        current_status: A dict with dynamic scheduling status and read-only instance data:
            Dynamic current status:
            - 'machine_status': list[int], the time each machine becomes available.
            - 'operation_status': list[int], the ready time of each operation
              (max finish time of its predecessors; only meaningful for ready operations).
            - 'scheduled_operations': list[bool], whether each operation has already
              been scheduled.
            Read-only global instance data (not a current/remaining status):
            - 'options': tuple[tuple[tuple[int, int | float], ...], ...], where
              options[op] lists eligible (machine_id, duration) pairs for operation op.
            - 'predecessors': tuple[tuple[int, ...], ...], where predecessors[op]
              lists the direct predecessor operations of op.
            - 'successors': tuple[tuple[int, ...], ...], where successors[op] lists
              the direct successor operations of op.
            - 'operation_to_job': tuple[int, ...], mapping each operation to its job.
            - 'job_operations': tuple[tuple[int, ...], ...], where job_operations[j]
              lists the operations belonging to job j.
        feasible_operations: A list of candidate assignments. Each element is a tuple
            (operation_id, machine_id, processing_time) describing one eligible machine
            option of a currently ready (all predecessors finished, not yet scheduled)
            operation. A flexible operation contributes one candidate per machine option.
            Original zero processing times are exposed here as 1e-12.

    Return:
        The selected assignment, a tuple (operation_id, machine_id, processing_time)
        taken from feasible_operations.
```

Initial fusion prompt

```
# Task
You are given a set of algorithms with the same function signature. Please fuse them into
     ONE parameterized algorithm architecture. These algorithms exhibit complementary
     performance, each offering advantages under different instance distributions.
```

```
[Algorithms]
[SOURCE ALGORITHMS]

[TASK INTERFACE AND RETURN SEMANTICS]

# Requirements
[FUSION REQUIREMENTS]
[SOURCE-PARAMETER REQUIREMENTS]

# Instructions
1. Output a concise thought, including the source-to-parameter analysis specified above,
     then output the full fused function in one Python code block followed by its source
     parameter table in one JSON code block:

[ARCHITECTURE-AND-PRESET OUTPUT]
2. Do not rename the function. Do not remove original required inputs; only add new
     parameters without defaults.
3. The return type of the function should stay compatible with the provided algorithms.
4. Ensure that the code is deterministic and optimized for speed.
5. Function attributes can be used to store persistent data within one problem instance. In
      subsequent function calls, this persistent data can be updated and reused to
     accelerate execution. Avoid recomputing the same quantities across calls if they can
     be cached. Cached function attributes are automatically cleared before each instance
     evaluation. Initialize missing attributes; no cross-instance cache invalidation or
     instance-change detection is needed.
6. Do not use if/elif trees to switch among copied whole-algorithm branches.
7. Use a unified computation graph with shared intermediate features, where parameters
     smoothly modulate behavior.
8. Prefer continuous parameters (e.g., weights, thresholds, penalties) over discrete
     template selectors.
9. Keep architecture parameters easy for an outer configurator to tune.
10. Do not include explanatory comments/docstrings unless they clarify non-obvious logic.
```

Behavioral coverage is measured on LLM-generated input sequences containing N_{val} calls in total. Inputs on which a source algorithm cannot execute are excluded before comparison. The remaining inputs and source algorithms' outputs are reused across fusion rounds.

Validation-sequence generation prompt

````
# Task
Provide a Python function that generates exactly [NUMBER OF SEQUENCES] independent input
     sequences for the target function. Each sequence must contain exactly [SEQUENCE LENGTH
     ] consecutive calls.

[Target function code]
```python
[FUNCTION SIGNATURE]
    pass
```

[TASK INTERFACE AND RETURN SEMANTICS]

# Requirements
1. Provide exactly one Python code block containing def generate_smoke_cases(): with no
     input parameters.
2. Return a list of sequences. Each sequence is a list of [SEQUENCE LENGTH] dicts, each
     containing "args" (a list) and "kwargs" (a dict).
3. Supply every required argument of the target function; do not invent or rename arguments.

4. Within each sequence, inputs represent consecutive steps of the same task instance. Keep
      dimensions, instance data, history, and evolving observations consistent with the task
      description. The function's internal state is retained between steps and reset before
      each new sequence.
5. Make different sequences small and diverse, including boundary cases. Repeated inputs
     within a sequence are allowed when meaningful.
6. Do not compile, import, or call the target function. Generate self-contained input
     sequences only; these are input replays, not simulations driven by the target's
     outputs.
7. Use only Python stdlib and numpy as np; do not rely on external state.
````

When the initial synthesis or a revision does not provide a valid parameter setting for a source algorithm, the following prompt derives the missing setting. The resulting setting is evaluated by the same predefined behavioral-coverage procedure.

Source-parameter derivation prompt (conditional)

```
# Task
Derive one parameter setting for the parameterized architecture so that it approximates the
      target algorithm's behavior.

[Parameterized architecture]
```python
[CURRENT ARCHITECTURE]
```

[Target algorithm]
```python
[SOURCE ALGORITHM]
```

[TASK INTERFACE AND RETURN SEMANTICS]

Architecture parameters to provide
[ARCHITECTURE PARAMETER NAMES]

Instructions
1. Briefly analyze how the target algorithm's key decisions correspond to the architecture
     parameters.
2. Do not use fenced code blocks in the analysis.
3. Then provide exactly one Python code block containing one function:
   def generate_arch_params():
4. The function must return a dict with this shape:
   ```python
   {
       "kwargs": {
           "<parameter_name>": <parameter_value>
       }
   }
   ```
5. Do not return an "args" key. Provide all architecture parameters through "kwargs" only.
6. The "kwargs" keys must exactly equal [ARCHITECTURE PARAMETER NAMES].
7. Do not omit, add, or rename any architecture parameter.
8. If [ARCHITECTURE PARAMETER NAMES] is empty, return exactly {"kwargs": {}}.
9. Do not import, compile, call, or reference either function.
```

Independent LLM review prompt

```
# Task
You are reviewing the current parameterized architecture against the requirements and
     checklist below. The provided algorithms exhibit complementary performance, each
     offering advantages under different instance distributions.

[Algorithms]
[SOURCE ALGORITHMS]

[TASK INTERFACE AND RETURN SEMANTICS]

[Current candidate architecture]
```python
[CURRENT ARCHITECTURE]
```

Requirements
[FUSION REQUIREMENTS]

Instructions
1. Check the candidate against the following criteria; report only as instructed in step 2:
[REVIEW CHECKLIST]

2. Output only concrete, substantive issues that still need improvement. For each issue,
     identify the relevant code or parameter, explain the problem, and suggest a direction
     for improvement. Do not invent issues or request unrelated stylistic changes. Do not
     output revised code or a parameter table. If there are no substantive issues under
     this round's checklist, output exactly the following marker alone:
==DONE==
```

The revision prompt receives either evaluator-generated program-validation feedback or the issue list returned by the independent reviewer. The review output is inserted verbatim. The optional

`Source:` line is included only when the evaluator can identify the failing line in the candidate, and an `==DONE==` review does not trigger a revision call. The runtime field follows the templates below.

Runtime feedback supplied to fusion revision

```
[PROGRAM-VALIDATION FEEDBACK]

For a code-validity or candidate-execution failure:
[EXCEPTION TYPE]: [ERROR MESSAGE]
Source: [FAILING SOURCE LINE]

For a behavioral-coverage failure:
The parameterized architecture's behavior is not similar enough to one or more original
     algorithms.
Low-similarity algorithms (1-based indices):
- Algorithm [SOURCE ID]: similarity=[SIMILARITY], parameters=[SOURCE PARAMETER DICTIONARY],
      threshold=0.900
[ADDITIONAL LOW-SIMILARITY ALGORITHMS]

[REVIEW ISSUE LIST]
The independent review output is inserted verbatim. A typical issue has the following
     structure:
- [RELEVANT CODE OR PARAMETER]: [SUBSTANTIVE PROBLEM]. Suggested direction: [REVISION
     DIRECTION].
[ADDITIONAL REVIEW ISSUES]
```

Fusion-revision prompt

```
# Task
You are aligning a set of algorithms into one parameterized architecture. These algorithms
     exhibit complementary performance, each offering advantages under different instance
     distributions.

[Algorithms]
[SOURCE ALGORITHMS]

[TASK INTERFACE AND RETURN SEMANTICS]

[Current candidate architecture]
```python
[CURRENT ARCHITECTURE]
```

[Issues of the Current Candidate]
[PROGRAM-VALIDATION FEEDBACK OR REVIEW ISSUE LIST]

Requirements
[FUSION REQUIREMENTS]
[SOURCE-PARAMETER REQUIREMENTS]

Instructions
1. Briefly analyze how to address the feedback in [Issues of the Current Candidate], then
     revise the candidate accordingly while preserving the requirements above. Keep the
     return type and semantics compatible with the provided algorithms, and keep behavior
     deterministic for identical inputs, parameters, and allowed state. Function attributes
      can be used to store persistent data within one problem instance. In subsequent
     function calls, this persistent data can be updated and reused to accelerate execution.
      Avoid recomputing the same quantities across calls if they can be cached. Cached
     function attributes are automatically cleared before each instance evaluation.
     Initialize missing attributes; no cross-instance cache invalidation or instance-change
      detection is needed.

2. After the analysis, output the complete improved architecture in one Python code block,
     followed by its source parameter table in one JSON code block:

[ARCHITECTURE-AND-PRESET OUTPUT]
```

F.3 Stage 2 Configuration-Policy Prompts

Stage 2 follows a two-phase curriculum. During the warm-start phase, which occupies the first 20% of the evaluation budget, the configuration policy must return one supplied source parameter setting without modifying or combining its values. During the main phase, the policy may generate new

parameter values for the same fixed parameterized algorithm. The warm-start prompt includes the source parameter settings and the fixed algorithm, whereas the main prompt retains only the fixed algorithm. The following templates are shared by the standard EoH reproduction operators. Repeated constraints are factored into a common block.

Source-parameter context for the warm-start phase

```
Available source parameter configurations:

Source algorithm [SOURCE ID]:
Description: [SOURCE ALGORITHM DESCRIPTION]
Parameters: [SOURCE PARAMETER DICTIONARY]

[ADDITIONAL SOURCE CONFIGURATIONS]
```

Fixed parameterized algorithm for both phases

```
Fixed parameterized algorithm structure (read-only):
[ACCEPTED FUSED ARCHITECTURE]
```

Common configuration-policy constraints

```
1. The algorithm must be strictly deterministic, excluding any stochastic components.
2. The generated Python code must be optimized for speed.
3. Function attributes can be used to store persistent data within one problem instance. In
      subsequent function calls, this persistent data can be updated and reused to
     accelerate execution. Avoid recomputing the same quantities across calls if they can
     be cached. Cached function attributes are automatically cleared before each instance
     evaluation. Initialize missing attributes; no cross-instance cache invalidation or
     instance-change detection is needed.
4. Use the pre-imported package as needed: np (as an alias for numpy). Do not use other
     packages or add any import statements in the function.
```

TSP configuration-policy template

```
import numpy as np

def conf_select_next_node(current_node: int, destination_node: int, unvisited_nodes: np.
     ndarray, distance_matrix: np.ndarray):
    """
    Generates parameters for a fixed, parameterized TSP node-selection algorithm.

    Note:
        This function is a parameter configurator, NOT the final node selector.
        The returned parameters will be passed into a fixed algorithm structure:
            select_next_node(current_node, destination_node, unvisited_nodes,
     distance_matrix, *params, **params)

    Args:
        current_node: ID of the current node.
        destination_node: ID of the destination node.
        unvisited_nodes: Array of IDs of unvisited nodes.
        distance_matrix: Distance matrix of the given set of nodes.

    Returns:
        Algorithm parameters for the fixed algorithm structure.
        Supported return styles:
            - tuple/list/np.ndarray: treated as positional parameters
            - dict: treated as keyword parameters
            - scalar: treated as one positional parameter
            - None: no extra parameter
    """
    # Placeholder: two scalar parameters (example only).
    # Replace with deterministic logic based on current_node / destination_node /
     unvisited_nodes / distance_matrix.
    alpha = 1.0
    beta = 0.0
    return alpha, beta
```

Configuration-policy task descriptions for the TSP

```
[WARM START: SELECT PRESET PARAMETERS]
Design a deterministic parameter configurator for the Traveling Salesman Problem (TSP). The
      function does NOT directly output the next node. Instead, it must output parameters
     that will be fed into a fixed, parameterized node-selection algorithm.

The evaluated pipeline is:
1. conf_select_next_node(current_node, destination_node, unvisited_nodes, distance_matrix)
     -> params
2. select_next_node(current_node, destination_node, unvisited_nodes, distance_matrix, **
     params) -> next_node

Configurator requirements:
1. Analyze the current routing state to choose one complete supplied parameter
     configuration that adapts the fixed algorithm structure to the current route
     construction stage.
2. Return the selected configuration as a parameter dictionary whose key names match the
     given fixed-structure signature exactly. Return the parameters themselves, not a
     source algorithm index.
3. Return exactly one supplied configuration. Do not modify parameter values, combine
     configurations, or generate new values. The source configurations must be included in
     the generated function; they are not provided as runtime globals.

Objective: construct a high-quality Hamiltonian cycle with low total tour length.
[COMMON CONFIGURATOR CONSTRAINTS]

[MAIN: TUNE CONTINUOUS PARAMETERS]
Design a deterministic parameter configurator for the Traveling Salesman Problem (TSP). The
      function does NOT directly output the next node. Instead, it must output parameters
     that will be fed into a fixed, parameterized node-selection algorithm.

The evaluated pipeline is:
1. conf_select_next_node(current_node, destination_node, unvisited_nodes, distance_matrix)
     -> params
2. select_next_node(current_node, destination_node, unvisited_nodes, distance_matrix, *
     params, **params) -> next_node

Configurator requirements:
1. Analyze the current routing state to produce parameters that adapt the fixed algorithm
     structure to the current route construction stage.
2. Returned parameter form must match the given fixed-structure signature exactly (number/
     order for positional args; key names for keyword args).
3. Prefer continuous parameters over discrete algorithm/template selectors.

Objective: construct a high-quality Hamiltonian cycle with low total tour length.
[COMMON CONFIGURATOR CONSTRAINTS]
```

OVMP configuration-policy template

```
import numpy as np

def conf_select_next_NUMA(req_hist: np.ndarray, observation: np.ndarray, availability: np.
     ndarray):
    """
    Generates parameters for a fixed, parameterized NUMA-placement algorithm.

    Note:
        This function is a parameter configurator, NOT the final NUMA selector.
        The returned parameters will be passed into a fixed algorithm structure:
            select_next_NUMA(req_hist, observation, availability, *params, **params)

    System Architecture:
        - N physical machines in the cluster
        - Each physical machine has exactly 2 NUMA nodes
        - Total NUMA nodes in cluster: 2N (numbered 0 to 2N-1)
        - NUMA node ID mapping: for machine i and NUMA j, ID = 2*i + j

    Args:
        req_hist (np.ndarray): Sliding window of the most recent requests in chronological
     order, with the incoming request at req_hist[-1].
            Shape: (M, 3) where M is the current window length, bounded by the configured
     history window size
            This window does not necessarily contain the complete request history.
            Structure: Each row represents a single request [cpu_cores, memory_gb,
     is_double_NUMA]
```

```
                - Axis 0: Earlier requests at lower indices, most incoming request at index
     -1
                - Axis 1 (request attributes):
                    - [0] cpu_cores: CPU requirement in cores
                    - [1] memory_gb: Memory requirement in GB
                    - [2] is_double_NUMA: Request placement mode flag (0 or 1)
                        - 1: Request spans both NUMA nodes of a single physical machine
                             Resources are equally divided: each NUMA gets request/2
                        - 0: Request is placed on a single NUMA node

        observation (np.ndarray): Current available resources across all NUMA nodes.
            Shape: (N, 2, 2) where N is the number of physical machines
            Structure: observation[machine_id][numa_id] = [available_cpu, available_memory]
            Example for 2 machines:
                [[[cpu_m0_n0, mem_m0_n0], [cpu_m0_n1, mem_m0_n1]],  # Machine 0
                 [[cpu_m1_n0, mem_m1_n0], [cpu_m1_n1, mem_m1_n1]]]  # Machine 1

        availability (np.ndarray): NUMA node availability mask.
            Shape: (2N,) boolean array where True indicates the NUMA node can accept the
     request
            Index mapping: availability[2*i + j] corresponds to machine i, NUMA j

    Returns:
        Algorithm parameters for the fixed algorithm structure.
        Supported return styles:
            - tuple/list/np.ndarray: treated as positional parameters
            - dict: treated as keyword parameters
            - scalar: treated as one positional parameter
            - None: no extra parameter
    """
    # Placeholder: two scalar parameters (example only).
    # Replace with deterministic logic based on req_hist/observation/availability.
    alpha = 1.0
    beta = 0.0
    return alpha, beta
```

Configuration-policy task descriptions for the OVMP

```
[WARM START: SELECT PRESET PARAMETERS]
Design a deterministic parameter configurator for virtual machine placement on NUMA nodes.
     The function does NOT directly output a NUMA action. Instead, it must output
     parameters that will be fed into a fixed, parameterized algorithm structure.

The evaluated pipeline is:
1. conf_select_next_NUMA(req_hist, observation, availability) -> params
2. select_next_NUMA(req_hist, observation, availability, **params) -> action

Requirements:
1. Analyze the current decision context to choose one complete supplied parameter
     configuration that adapts the fixed algorithm structure to current workload and
     resource state.
2. Return the selected configuration as a parameter dictionary whose key names match the
     given fixed-structure signature exactly. Return the parameters themselves, not a
     source algorithm index.
3. Return exactly one supplied configuration. Do not modify parameter values, combine
     configurations, or generate new values. The source configurations must be included in
     the generated function; they are not provided as runtime globals.
4. The algorithm must be strictly deterministic, excluding any stochastic components.
5. The generated Python code must be optimized for speed.

Objective: maximize total request throughput under dynamic resource constraints.

Use the pre-imported package as needed: np (as an alias for numpy). Do not use other
     packages or add any import statements in the function. Function attributes can be used
      to store persistent data within one problem instance. In subsequent function calls,
     this persistent data can be updated and reused to accelerate execution. Avoid
     recomputing the same quantities across calls if they can be cached. Cached function
     attributes are automatically cleared before each instance evaluation. Initialize
     missing attributes; no cross-instance cache invalidation or instance-change detection
     is needed.

[MAIN: TUNE CONTINUOUS PARAMETERS]
Design a deterministic parameter configurator for virtual machine placement on NUMA nodes.
     The function does NOT directly output a NUMA action. Instead, it must output
     parameters that will be fed into a fixed, parameterized algorithm structure.

The evaluated pipeline is:
```

```
1. conf_select_next_NUMA(req_hist, observation, availability) -> params
2. select_next_NUMA(req_hist, observation, availability, *params, **params) -> action

Requirements:
1. Analyze the current decision context to produce parameters that adapt the fixed
     algorithm structure to current workload and resource state.
2. Returned parameter form must match the given fixed-structure signature exactly (number/
     order for positional args; key names for keyword args).
3. Prefer continuous parameters over discrete algorithm/template selectors.
4. The algorithm must be strictly deterministic, excluding any stochastic components.
5. The generated Python code must be optimized for speed.

Objective: maximize total request throughput under dynamic resource constraints.

Use the pre-imported package as needed: np (as an alias for numpy). Do not use other
     packages or add any import statements in the function. Function attributes can be used
      to store persistent data within one problem instance. In subsequent function calls,
     this persistent data can be updated and reused to accelerate execution. Avoid
     recomputing the same quantities across calls if they can be cached. Cached function
     attributes are automatically cleared before each instance evaluation. Initialize
     missing attributes; no cross-instance cache invalidation or instance-change detection
     is needed.
```

FJSP configuration-policy template

```
import numpy as np

def conf_determine_next_operation(current_status, feasible_operations):
    """
    Generates parameters for a fixed, parameterized FJSP operation-selection algorithm.

    Note:
        This function is a parameter configurator, NOT the final operation selector.
        The returned parameters will be passed into a fixed algorithm structure:
            determine_next_operation(current_status, feasible_operations, *params, **params
     )

    Args:
        current_status: A dict with dynamic scheduling status and read-only instance data:
            Dynamic current status:
            - 'machine_status': list[int], the time each machine becomes available.
            - 'operation_status': list[int], the ready time of each operation
              (max finish time of its predecessors; only meaningful for ready operations).
            - 'scheduled_operations': list[bool], whether each operation has already
              been scheduled.
            Read-only global instance data (not a current/remaining status):
            - 'options': tuple[tuple[tuple[int, int | float], ...], ...], where
              options[op] lists eligible (machine_id, duration) pairs for operation op.
            - 'predecessors': tuple[tuple[int, ...], ...], where predecessors[op]
              lists the direct predecessor operations of op.
            - 'successors': tuple[tuple[int, ...], ...], where successors[op] lists
              the direct successor operations of op.
            - 'operation_to_job': tuple[int, ...], mapping each operation to its job.
            - 'job_operations': tuple[tuple[int, ...], ...], where job_operations[j]
              lists the operations belonging to job j.
        feasible_operations: A list of candidate assignments. Each element is a tuple
            (operation_id, machine_id, processing_time) describing one eligible machine
            option of a currently ready operation. A flexible operation contributes one
            candidate per machine option.

    Returns:
        Algorithm parameters for the fixed algorithm structure.
        Supported return styles:
            - tuple/list/np.ndarray: treated as positional parameters
            - dict: treated as keyword parameters
            - scalar: treated as one positional parameter
            - None: no extra parameter
    """
    # Placeholder: two scalar parameters (example only).
    # Replace with deterministic logic based on current_status / feasible_operations.
    alpha = 1.0
    beta = 0.0
    return alpha, beta
```

Configuration-policy task descriptions for the FJSP

```
[WARM START: SELECT PRESET PARAMETERS]
Design a deterministic parameter configurator for the Flexible Job Shop Scheduling Problem
     (FJSP). The function does NOT directly output the next operation. Instead, it must
     output parameters that will be fed into a fixed, parameterized operation-selection
     algorithm.

The evaluated pipeline is:
1. conf_determine_next_operation(current_status, feasible_operations) -> params
2. determine_next_operation(current_status, feasible_operations, **params) ->
     next_operation

Configurator requirements:
1. Analyze the current scheduling state to choose one complete supplied parameter
     configuration that adapts the fixed algorithm structure to the current scheduling
     stage.
2. Return the selected configuration as a parameter dictionary whose key names match the
     given fixed-structure signature exactly. Return the parameters themselves, not a
     source algorithm index.
3. Return exactly one supplied configuration. Do not modify parameter values, combine
     configurations, or generate new values. The source configurations must be included in
     the generated function; they are not provided as runtime globals.

Objective: schedule all operations to minimize the makespan, subject to operation
     precedence and machine non-overlap constraints.
[COMMON CONFIGURATOR CONSTRAINTS]

[MAIN: TUNE CONTINUOUS PARAMETERS]
Design a deterministic parameter configurator for the Flexible Job Shop Scheduling Problem
     (FJSP). The function does NOT directly output the next operation. Instead, it must
     output parameters that will be fed into a fixed, parameterized operation-selection
     algorithm.

The evaluated pipeline is:
1. conf_determine_next_operation(current_status, feasible_operations) -> params
2. determine_next_operation(current_status, feasible_operations, *params, **params) ->
     next_operation

Configurator requirements:
1. Analyze the current scheduling state to produce parameters that adapt the fixed
     algorithm structure to the current scheduling stage.
2. Returned parameter form must match the given fixed-structure signature exactly (number/
     order for positional args; key names for keyword args).
3. Prefer continuous parameters over discrete algorithm/template selectors.

Objective: schedule all operations to minimize the makespan, subject to operation
     precedence and machine non-overlap constraints.
[COMMON CONFIGURATOR CONSTRAINTS]
```