SM2-AHD: SEARCH EXPERIENCE MEMORY AND META-COGNITIVE REFLECTION FOR LLM-BASED AUTOMATIC HEURISTIC DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Automatic Heuristic Design (AHD) aims to construct high-quality heuristics with as little human intervention as possible; LLM-based AHD, which uses LLMs as code generators, has already produced heuristics that surpass hand-crafted ones through iterative search. Existing evolutionary methods, however, rely on a relatively fixed evolutionary framework and remain sensitive to the initial population and to the hyperparameters. We therefore propose SM2-AHD, a meta-level AHD framework in which the code population and the outer framework co-evolve throughout the search with two independent components. At the code level, **Search Experience Memory** (SEM) encodes each trajectory into a memory unit that carries its code lineage and failure attribution, and injects retrieved historical cases into the mutation prompt. At the level of the outer framework, **Meta-Reflection** periodically diagnoses verifiable search statistics and, under safety constraints, adjusts the exploration strength and the operator weights online. Experiments show that SM2-AHD outperforms several existing state-of-the-art methods across a range of heuristic design problems.

1 INTRODUCTION

Combinatorial optimization (CO) seeks the solution that optimizes an objective over a discrete feasible domain, and the vast majority of its problems are NP-hard; practice therefore relies on **heuristics**: experiential rules that trade an acceptable computational cost for high-quality solutions. Designing a good heuristic, however, depends on the domain insight and repeated trial and error of its designer, a cost that has long resisted automation. **Large language models** (LLMs) have recently opened a way out through their strength in code generation: by letting the model write *a program that solves the problem* rather than a solution to it, the LLM becomes a meta-solver that designs solvers. When this process is formalized and delegated to an algorithm it is called **Automatic Heuristic Design** (AHD) (Liu et al., 2023; van Stein & Bäck, 2025); when the search is driven by an LLM it is called **LLM-based AHD**, which has already produced solvers surpassing human-crafted heuristics on a range of NP-hard problems (Dat et al., 2025; Liu et al., 2024; Novikov et al., 2025; Romera-Paredes et al., 2024; Ye et al., 2024).

Work in this paradigm differs mainly along two axes: *how the search is organized*, whether population-based evolution, which keeps a pool of candidates and applies LLM-driven crossover, mutation, and selection (Dat et al., 2025; Liu et al., 2024; 2026a; Romera-Paredes et al., 2024; Yao et al., 2025; Ye et al., 2024), or tree search, which retains every candidate generated so far and balances exploration against exploitation (Gan et al., 2025; Gao et al., 2025; Kiet et al., 2026; Oren et al., 2025; Wang et al., 2025a;b; Zheng et al., 2025); and *what is searched over*, a single key function inside a prespecified solver framework (Liu et al., 2024; Romera-Paredes et al., 2024) or a complete algorithm (Liu et al., 2023; 2026d; Novikov et al., 2025; van Stein & Bäck, 2025; van Stein et al., 2025). Along both axes, however, attention concentrates on *generating the next candidate*.

Rich as the work in LLM-based AHD now is, the framework it operates in still leaves problems unresolved. Existing AHD systems hand the LLM only *part* of the modification authority, typically the key heuristic alone, so that an LLM edit cannot break the outer validation framework that already exists; but this also hard-codes the search's behavioral mode before evolution begins, fixing

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.

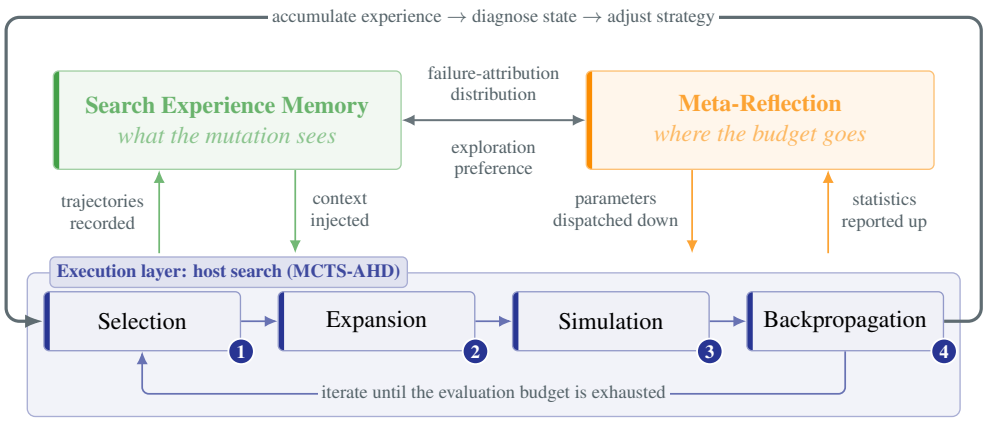


Figure 1: Overall architecture of SM2-AHD. Neither component touches the selection or update logic of the host search: the Search Experience Memory turns every finished trajectory into a retrievable unit and injects the retrieved cases into the next mutation, while Meta-Reflection reads the search statistics back and dispatches adjusted exploration and operator parameters. Together they form the cycle: accumulate experience, diagnose the state, adjust the strategy.

it rather than letting it adapt. Once the search under-explores and huddles in a few regions of the space, or entrenches a preference for low-yield operators and keeps spending budget on them, it can neither notice this nor recover, and LLM-based AHD is left sensitive to its initial population and hyperparameters. Worse, the evolutionary structure makes poor use of the search experience already accumulated. Every round of search leaves a complete record of under what circumstances, which kind of edit was made, and whether the outcome was a success or a failure, yet that record ends up in exactly one place: folded into a scalar fitness value. Once evaluation finishes the context is discarded, so change patterns validated repeatedly across paths cannot be reused and directions that have already failed may be attempted again; and whether or not a poorly performing candidate is kept, what survives is the candidate itself, never a genuinely effective **code-level reflection**.

We therefore propose **SM2-AHD** (Figure 1), a meta-level framework that adds, *without touching the selection or update logic of the host search*, two independent components. The first, **Search Experience Memory** (SEM), packages each trajectory as a structured unit (code lineage, edit differences, performance trace, and failure attribution) and retrieves the most similar historical cases into the mutation prompt, so that every generation builds on cross-path successful patterns and on failure lessons that have already been attributed. The second, **Meta-Reflection**, periodically reads verifiable search statistics, namely the convergence trend, the coverage of search directions, and each operator's success rate and average improvement, and adjusts the exploration strength and the operator weights online, so that the strategy evolves with the process rather than being hard-coded in advance. Realizing this is not free: failure experience may enter memory only after it has been *attributed*, and self-regulation must stay bounded, so neither component rewrites the logic of the search itself: what they change is *what the mutation sees* and *where the budget goes*. The two presuppose each other, memory supplying reflection with the distribution of failure attributions and reflection reshaping the trajectories memory will next receive, and together they form a closed loop, *accumulate experience* → *diagnose state* → *adjust strategy* → *accumulate again*.

Experiments on six classes of combinatorial optimization problems, across two solver frameworks and nineteen size configurations, compare SM2-AHD with four existing LLM-based AHD frameworks under a common protocol: SM2-AHD outperforms or matches the existing state-of-the-art methods on 19/19 configurations and outperforms them on 16/19, and on the remaining three the other evolutionary frameworks have all already come level with the optimum, so that SM2-AHD holds a general advantage over existing AHD frameworks.

Our contributions can be summarized in three points:

- We propose SM2-AHD, a meta-level AHD framework built from two core components, Search Experience Memory and Meta-Reflection. Acting together, they realize the cycle of

"accumulating experience, diagnosing the state and adjusting the strategy" in code evolution.

- Across the nineteen configurations of six combinatorial optimization problems, we obtain better performance than four existing state-of-the-art LLM-based AHD frameworks.
- Through ablation we analyze the contribution of the two components and show that each of them is indispensable for improving search performance.

2 Related Work

Automatic heuristic design and LLM-based AHD. AHD searches a heuristic space for the best-performing heuristic function; its traditional evolutionary route still needs hand-designed operators. EoH (Liu et al., 2024) and FunSearch (Romera-Paredes et al., 2024) first put an LLM inside a population-based loop: within a prespecified solver framework it designs only the key heuristic function, with ReEvo (Ye et al., 2024) steering it by "verbal gradients" and HSEvo (Dat et al., 2025) sustaining exploration by diversity. Later work advances in two directions: *how search is organized*, from MCTS-AHD (Zheng et al., 2025), the first to organize all historical heuristics with Monte Carlo tree search and so avoid discarding temporarily low-scoring individuals, through PoH (Wang et al., 2025b), MOTIF (Kiet et al., 2026), RF-Agent (Gao et al., 2025), and MASTER (Gan et al., 2025); and *the granularity of the object*, raised by AEL (Liu et al., 2023), LLaMEA (van Stein & Bäck, 2025), and AlphaEvolve (Novikov et al., 2025) from one function to a whole algorithm. The field's landscape is surveyed in (Liu et al., 2026b; Wu et al., 2025).

Agent memory. For a language agent, even a problem it has already solved is rederived from scratch (Banerjee et al., 2026). How to give a language agent memory is therefore a question worth attention. Existing work answers *what to store*, from validated programs (Wang et al., 2024) and experience streams (Park et al., 2023) through trajectory insights (Zhao et al., 2024), memory items (Ouyang et al., 2026) and dynamic buffers (Liu et al., 2025) to "memory skills" (Zhang et al., 2026a); and *how to retrieve*, by vector similarity (Zheng et al., 2024) or by similarity plus temporal contiguity (Fountas et al., 2025). Only whether *failure* experience should be admitted remains contested: LATS (Zhou et al., 2024) stores failed trajectories with their reflections, whereas AWM (Wang et al., 2025c) reports that incorrectly induced workflows degrade performance and CER's ablation (Liu et al., 2025) that such trajectories mislead the agent.

Metacognition and reflection. Metacognition is the ability to "monitor and regulate one's own cognitive processes" (Liu et al., 2026c). Reflexion (Shinn et al., 2023) stores verbal reflections in an episodic buffer, MARS (Hou et al., 2026) turns failure experiences into principled ones, and Retroformer (Yao et al., 2024) and PreFlect (Wang et al., 2026) improve the retrospective and prospective sides of reflection. In search, RoT (Hui & Tu, 2024) distills a weak LLM's tree search into task-level guidelines and MeLA (Qiu et al., 2026) traces the causal chain from a prompt's thinking to the heuristic it produces. Self-regulation must nevertheless stay conservative: prompting an LLM to generate feedback rarely yields reliable self-correction, the bottleneck being *feedback generation* (Kamoi et al., 2024), and agents that rewrite their own code carry their own risks (Yin et al., 2025; Zhang et al., 2026b).

3 Preliminaries and Motivation

3.1 Preliminaries

Automatic Heuristic Design (AHD). Given a task P (for example, a CO problem), AHD aims to search the heuristic space $\mathcal{H}$ for the best-performing heuristic h^*:

$$h^* = \arg\max_{h \in \mathcal{H}} \; g(h), \tag{1}$$

where a heuristic $h \in \mathcal{H}$ is an algorithm mapping a set of task instances $\mathcal{I}_P$ to a set of solutions $\mathcal{S}_P$, i.e., $h : \mathcal{I}_P \to \mathcal{S}_P$, and $g(\cdot)$ measures the heuristic's performance. For a CO problem with minimization objective $f : \mathcal{S}_P \to \mathbb{R}$, AHD estimates g by evaluating h on a task dataset $\mathcal{D}$:

$$g(h) = \mathbb{E}_{\mathbf{ins} \in \mathcal{D}}\big[- f\big(h(\mathbf{ins})\big)\big]. \tag{2}$$

In practice, AHD usually prespecifies a general solver framework (such as step-by-step construction or ant colony optimization, both defined in Appendix D) and lets the algorithm design only **one key heuristic function** within it, in order to reduce the difficulty of the search (Liu et al., 2023; van Stein & Bäck, 2025); we follow this setting and still denote the function to be designed by h.

3.2 MOTIVATION

We observe two core problems in the LLM-based AHD framework: evolution itself focuses on producing the next generation of individuals, while comparatively little attention goes to *what can be learned from the evolution already performed, and what adjustment it should prompt*. Existing work is no exception. PathWise (Gungordu et al., 2026) builds a world model from an entailment graph and coordinates several agents, but its reflection and adjustment are confined to the evolution of a single generation, and the information it carries across generations is so limited that its cross-generation guidance is nearly meaningless; it remains, in essence, evolution over a relatively flat population. HiFo-Prompt (Chen et al., 2026) builds a pool of "improvement insights," using intelligent tools to generate and analyze the experience extractable from evolution and then injecting that pool as a prompt to steer it; but this route renders algorithmic insight as highly condensed natural language, extracted by way of semantic computation, so much of the meaning is lost in the process and the resulting insights need not apply. Learning from failure is missing in adjacent domains too: a system built for goal-oriented search admits it lacks a mechanism to learn from failure cases (Chopra & Shah, 2025). Inspired by the experience reviewed in Section 2, we adopt the idea of *meta-reflection* from agent evolution, that an agent should reflect not only on environmental feedback but also *on its own reflections*, so that the framework can adjust dynamically as evolution proceeds.

Following this, we build a **meta-level** AHD framework in which the LLM serves no longer merely as the "code generator" of population evolution but as the intelligent regulator of the evolutionary framework. At the code level we build **Search Experience Memory**: where existing work either discards the code outright or condenses its idea into natural language and thereby distorts it, we encode every trajectory as a *memory unit* carrying the code lineage, edit differences, performance trace, and failure attribution, and retrieve the most similar historical cases into the prompt at mutation time, so that generation consults both *successful patterns from other paths* and *failure lessons already attributed*. At the level of the outer framework, where previous frameworks focus only on balancing *exploration against exploitation*, **Meta-Reflection** periodically "observes–diagnoses–retunes," adjusting the exploration strength and the operator weights online from verifiable search statistics, that is, it **learns how to adjust the strategic framework**, so that the evolutionary strategy and the code population can *co-evolve*. In addition, we set a **safety threshold** that keeps each meta-reflection bounded and prevents the framework from collapsing (Kamoi et al., 2024; Yin et al., 2025; Zhang et al., 2026b).

4 METHOD: SM2-AHD

Overview. The two problems identified in the previous section, namely what can be learned from the evolution already performed and what adjustment it should prompt, motivate SM2-AHD, a new meta-level AHD framework. It consists of two independent components, one for each problem: *Search Experience Memory* (SEM), which changes "what the mutation sees" (Figure 2), and *Meta-Reflection*, which changes "where the budget goes" (Figure 3). Algorithm 1 states the loop they act on; the rest of this section takes the two components in turn.

4.1 COMPONENT ONE: SEARCH EXPERIENCE MEMORY (SEM)

We now turn to the core of the Search Experience Memory (SEM): how it stores search trajectories as memory, and how it uses that information as a prompt to guide code mutation. Figure 2 gives an overview of the two paths SEM maintains.

The memory unit. Once a search trajectory is complete, SM2-AHD packages it into a structured memory unit, a seven-tuple $M = (c, L, E, p, y, R, m)$, namely the problem context c, the evolutionary lineage L, the edit sequence E, the performance trace p, the success/failure label y, the reflection and failure attribution R, and metadata m. The label y compares the new heuristic with the one it was derived from: a trajectory counts as a *success* when that objective improves and a *failure* when

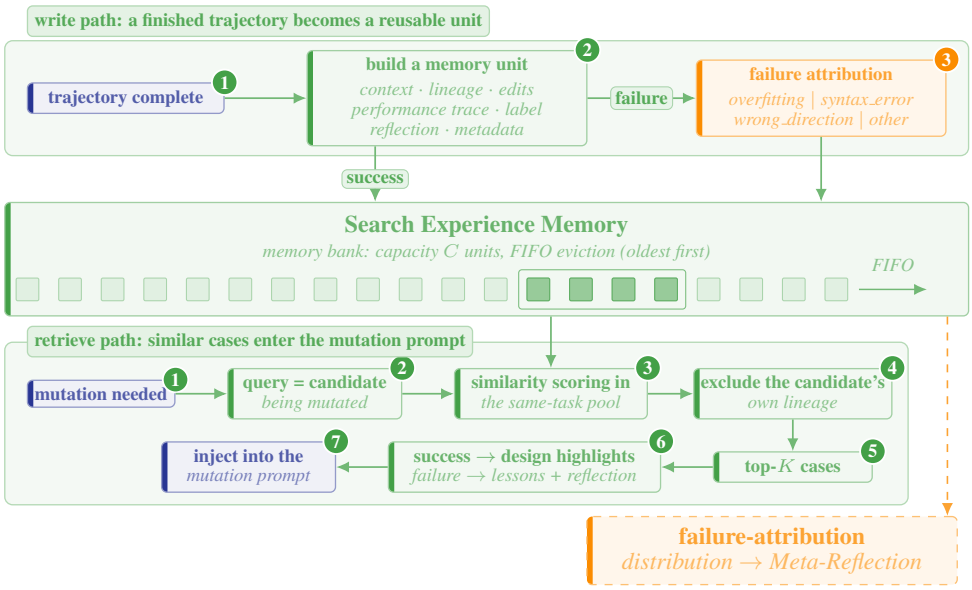


Figure 2: The two paths of the Search Experience Memory. *Write path*: a finished trajectory is packaged into a seven-tuple memory unit and stored, failed paths only after the failure has been attributed to a category; the bank is bounded and evicts oldest-first. *Retrieve path*: at mutation time the candidate being mutated serves as the query, cases are scored for similarity inside the same-task pool, the candidate's own lineage is excluded, and the top-K cases are formatted into the mutation prompt, successful ones as design highlights and failed ones as lessons and reflections. The failure-attribution distribution is handed on to Meta-Reflection.

it does not. The most critical entry is R: a failed path does not simply carry a "failure" label but is attributed to a category such as overfitting, syntax error, or wrong direction, so that failure itself becomes reusable knowledge. The store has a bounded capacity and evicts oldest-first to keep experience fresh, so the candidate pool always reflects "the most recent search experience."

Retrieval and injection. When a mutation is needed, SM2-AHD retrieves the historical cases most similar to the candidate being mutated from the *same-task* pool, excluding those on that candidate's own lineage so that "the path currently being traversed" is not mistaken for external reference; the retrieved entries are formatted and appended to the mutation prompt, successful cases contributing their design highlights and failed cases their lessons and reflections. The context a mutation depends on thus broadens from "the parent code alone" to "a cross-path, cross-generation library of historical cases," so the system can reuse successful change patterns and avoid failure patterns that have

Algorithm 1 SM2-AHD main loop (the full version is given in Appendix Algorithm 3)

Require: evaluator $\mathcal{E}$, evaluation budget T, mutation model $\mathcal{M}_{\mathrm{f}}$, reflection model $\mathcal{M}_{\mathrm{s}}$, operator set $\mathcal{O}$, memory capacity C, number of retrieved items K
Ensure: globally best heuristic h^*
1: initialize the evolution tree $\mathcal{T}$, the memory store, and the meta-controller parameters Θ
2: **while** $t < T$ **do**
3: $n \leftarrow \textsc{Select}(\mathcal{T})$; select an operator op according to the weights Θ and generate a child from n
4: if op is a mutation: retrieve similar historical cases and build the difference-aware context x from them; otherwise $x \leftarrow \textsc{DefaultCtx}(n)$
5: generate $h \leftarrow \mathcal{M}_{\mathrm{f}}(op, x)$, evaluate $o \leftarrow \mathcal{E}(h)$, backpropagate and record the operator event
6: write the memory unit M (failed paths are first attributed by $\mathcal{M}_{\mathrm{s}}$)
7: if a meta-analysis is triggered: collect statistics S, let $\mathcal{M}_{\mathrm{s}}$ output the adjustment $\Delta\Theta$, and update Θ after the safety filter
8: $t \leftarrow t + 1$
9: **end while**
10: **return** $\arg\max_{h \in \mathcal{T}} g(h)$

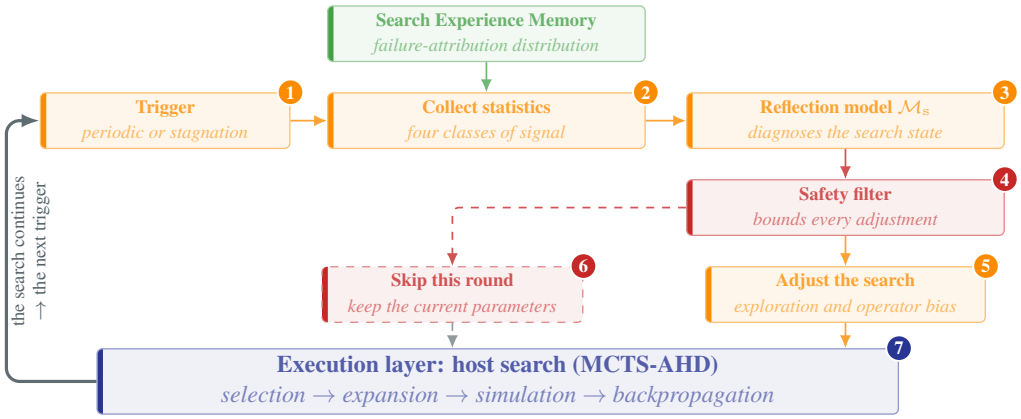


Figure 3: The life cycle of one meta-reflection round. Each box names one component of the round and, where the name alone is not enough, adds a single line of gloss; the mechanism behind each is given in the text. A trigger, whether periodic or stagnation, collects four classes of statistics, among them the failure-attribution distribution supplied by the Search Experience Memory. The reflection model diagnoses the state and outputs a structured adjustment, which reaches the search only after the safety filter has bounded it; the adjusted parameters change how the search explores, and the search runs on until the next trigger.

already recurred, and thereby solve the problem that search retains no failure experience (Chopra & Shah, 2025). The blocks that the two components inject into the prompt, and the prompt templates in full, are given in Appendix E.

4.2 Component Two: Meta-Reflection

We now turn to the core of Meta-Reflection: when it is triggered, and how it works to steer the framework and the code toward co-evolution. Figure 3 traces one round of the meta-reflection cycle.

Triggering and statistics. A meta-analysis is triggered when either a periodic timer or stagnation detection fires, the former counting a fixed number of evaluations and the latter firing once the search has gone a run of rounds without improvement, and a minimum interval separates two triggers, so that too-frequent adjustment does not itself cause oscillation. It then aggregates four classes of signal: convergence information (the current best value and the number of stagnant rounds), coverage of search directions (whether the search is over-concentrated in a few parts of the space), each operator's **true execution statistics** over the recent window (attempts, success rate, and average improvement), and the distribution of failure attributions supplied by SEM, the latter constituting a direct information channel between the two components.

Diagnosis, safety constraints, and synergy. The reflection model reads these statistics and outputs a structured decision, whose format Appendix E.5 gives in full: an absolute value for the novelty weight, an absolute value for the exploration constant, and a categorical bias direction for four of the operators. That decision does not take effect directly but must first pass a **safety filter**: a single round may move a parameter only within a neighbourhood of its previous step, a round is skipped altogether if a parameter has swung back and forth over several consecutive rounds, and every adjusted parameter stays inside the range allowed for it, since it is better to adjust slowly than to let the adjustment itself become a new source of oscillation. Further implementation details are given in Appendix B. Where a self-referential agent that directly rewrites its own code (Yin et al., 2025; Zhang et al., 2026b) risks collapsing its framework beyond repair, meta-cognitive adjustment confines itself to behavioural parameters and never modifies the code of the agent framework; even so, every adjustment acts on the whole of the remaining search, so its error cost far exceeds that of a single mutation and the magnitude of the change has to be kept within reason.

The two components described in this section, taken together, form a self-reinforcing closed loop that assists AHD: SEM supplies the meta-analysis with the distribution of failure modes, while Meta-Reflection's adjustment of exploration preferences changes the direction of future exploration

and hence the distribution of the memory SEM subsequently accumulates, forming the cycle "accumulate experience → diagnose state → adjust strategy → accumulate again."

5 EXPERIMENTS

Goal. To evaluate how SM2-AHD performs across the various kinds of LLM-based AHD, and to test the soundness and effectiveness of the method set out above, we compare SM2-AHD with the established heuristics corresponding to each problem and with four state-of-the-art LLM-based AHD frameworks, configuration by configuration, over six classes of combinatorial optimization problems, two solver frameworks and nineteen size configurations, and we ablate the two components under the same protocol so as to verify that each of them is indispensable.

5.1 MAIN EXPERIMENTS

Setup. We follow the protocol of the original MCTS-AHD paper (Zheng et al., 2025) throughout and instantiate SM2-AHD on MCTS-AHD, used purely as the base search that generates, evaluates and retains candidate heuristics; Appendix B.4 sets out why that base is a suitable one. Six problem classes are covered, in two solver frameworks defined in Appendix D, across nineteen test sets in all. Each test set is run independently three times, and every entry reports the mean objective of the resulting solver over those runs, to three decimals. Appendix F states the detailed configuration the five LLM-based frameworks are run under, the source of every column that is not an AHD method, and the remaining details behind the comparison.

Compared methods. For every problem class we report two families of methods. The first is *established heuristics*, which bound what can be reached without learning: for the constructive problems, the reference optima obtained by LKH for TSP (Lin & Kernighan, 1973) and by OR-Tools for KP (Perron & Furnon, 2024), the **Greedy Construct** heuristic, and the neural constructive policy **POMO** (Kwon et al., 2020); for the ant-colony problems, the default **ACO** of the solver framework (Desale et al., 2015; Dorigo et al., 2006) and the learned heuristic **DeepACO** (Ye et al., 2023). The second is *LLM-based AHD methods*: we compare against four existing frameworks, **ReEvo** (Ye et al., 2024), **HSEvo** (Dat et al., 2025), **PathWise** (Gungordu et al., 2026) and **MCTS-AHD** (Zheng et al., 2025), together with SM2-AHD, under a common protocol. Among the first-family algorithms, the model parameters published by some of the original frameworks are incomplete, so that no corresponding experimental data exists, and those entries are marked `N/A`; Appendix F gives the details. Because the first family contains exact solvers, its entries mark the level that learned methods have yet to reach, and the family is therefore reported as a reference line rather than as a competitor: the comparison this study draws is among the LLM-based methods.

Language models. All five LLM-based AHD methods generate code with the same model, `deepseek-flash`, so the comparison among them is drawn under a common code-generation model. SM2-AHD additionally calls a second, stronger model, `deepseek-v4-pro`, for its two reflective roles, the failure attribution of the Search Experience Memory and the meta-analysis of Meta-Reflection, on the ground that reflection gains more from a stronger model than generation does. Note that this part of the cost accounts for only a small fraction: we estimate that it adds about 9% in extra cost. Appendix F.2 discusses this.

Presenting the results. Following the comparison protocol used in prior work, we report the evaluation data of every heuristic evolved by an LLM-based AHD method under the same evaluation budget T. Appendix F gives the detailed configuration of each method. The "Gap" column of Tables 1–3 gives each method's relative gap to the best value reported for that test set over both families together, so a smaller value is always better and 0.00% marks the best value itself; that value is printed in bold. The best of the five LLM-based AHD methods is shaded grey. Gaps are given to three decimals, and maximization and minimization problems are marked with $\uparrow$ and $\downarrow$ respectively in the tables.

Results and analysis. Tables 1–3 give the comparison on all nineteen configurations. For the two constructive problem types, the optima obtained by fixed methods and by an operations-research solver are listed alongside, to show the gap between the heuristics and the exact optimum; they take no part in the comparison. Among the many LLM-based AHD methods, SM2-AHD designs a heuristic that outperforms or matches the existing state-of-the-art methods on sixteen configurations;

Table 1: Comparison on the seven constructive test sets (TSP-cons and KP-cons), between established heuristics (Optimal, Greedy Construct, POMO) and LLM-based AHD methods.

Task	TSP-cons								KP-cons					
Test set	N=20		N=50		N=100		N=200		N=50		N=100		N=200	
Methods	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap	Obj.↑	Gap	Obj.↑	Gap	Obj.↑	Gap
Optimal	**3.804**	**0.00%**	**5.754**	**0.00%**	**7.726**	**0.00%**	**10.697**	**0.00%**	**20.050**	**0.00%**	**40.694**	**0.00%**	**57.887**	**0.00%**
Greedy Construct	4.450	16.99%	7.066	22.80%	9.586	24.08%	13.399	25.25%	20.002	0.24%	40.656	0.10%	57.832	0.10%
POMO	3.807	0.07%	5.782	0.47%	7.790	0.82%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
LLM-based AHD: *deepseek-flash*														
MCTS-AHD	4.077	7.17%	6.339	10.16%	8.612	11.47%	12.034	12.50%	20.043	0.04%	40.688	0.015%	57.863	0.04%
ReEvo	4.012	5.46%	6.274	9.04%	8.537	10.49%	12.014	12.31%	20.014	0.18%	40.664	0.07%	57.835	0.09%
HSEvo	4.022	5.73%	6.315	9.74%	8.645	11.90%	12.195	14.00%	20.011	0.20%	40.660	0.08%	57.838	0.08%
PathWise	4.078	7.20%	6.324	9.90%	8.656	12.03%	12.135	13.44%	20.021	0.15%	40.669	0.06%	57.852	0.06%
SM2-AHD (ours)	3.920	3.05%	6.068	5.44%	8.278	7.14%	11.731	9.67%	20.041	0.05%	40.687	0.017%	57.871	0.03%

Table 2: Comparison on the six ant-colony test sets for TSP and CVRP (TSP-ACO and CVRP-ACO), between established heuristics (ACO, DeepACO) and LLM-based AHD methods.

Task	TSP-ACO						CVRP-ACO					
Test set	N=20		N=50		N=100		N=20		N=50		N=100	
Methods	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap
ACO	3.900	0.98%	6.554	15.05%	10.039	27.81%	9.046	95.10%	18.261	117.06%	30.014	104.46%
DeepACO	3.868	0.14%	N/A	N/A	8.383	6.74%	4.701	1.38%	N/A	N/A	14.881	1.37%
LLM-based AHD: *deepseek-flash*												
MCTS-AHD	3.866	0.10%	5.739	0.74%	8.057	2.59%	4.979	7.39%	9.150	8.76%	15.641	6.55%
ReEvo	3.871	0.23%	5.855	2.78%	8.399	6.93%	4.795	3.41%	9.125	8.47%	15.783	7.52%
HSEvo	3.869	0.18%	5.845	2.60%	8.385	6.75%	4.821	3.98%	9.293	10.46%	16.381	11.59%
PathWise	3.875	0.33%	5.884	3.28%	8.547	8.82%	4.728	1.98%	9.155	8.82%	15.989	8.93%
SM2-AHD (ours)	**3.862**	**0.00%**	**5.697**	**0.00%**	**7.854**	**0.00%**	**4.637**	**0.00%**	**8.413**	**0.00%**	**14.679**	**0.00%**

Table 3: Comparison on the six ant-colony test sets for MKP and BPP (MKP-ACO and BPP-ACO), between established heuristics (ACO, DeepACO) and LLM-based AHD methods.

Task	MKP-ACO						BPP-ACO					
Test set	N=100		N=300		N=500		N=120		N=500		N=1000	
Methods	Obj.↑	Gap	Obj.↑	Gap	Obj.↑	Gap	Obj.↓	Gap	Obj.↓	Gap	Obj.↓	Gap
ACO	16.589	3.12%	51.196	5.97%	122.229	8.87%	51.438	4.96%	218.016	7.56%	437.594	8.31%
DeepACO	N/A	N/A	53.105	2.46%	127.710	4.79%	49.109	0.21%	N/A	N/A	N/A	N/A
LLM-based AHD: *deepseek-flash*												
MCTS-AHD	17.101	0.13%	54.254	0.35%	133.759	0.28%	49.026	0.04%	202.698	0.001%	**404.016**	**0.00%**
ReEvo	17.077	0.27%	54.213	0.43%	132.544	1.18%	49.339	0.67%	204.224	0.75%	407.464	0.85%
HSEvo	17.023	0.59%	53.772	1.24%	130.659	2.59%	49.542	1.09%	205.271	1.27%	409.536	1.37%
PathWise	17.100	0.13%	54.330	0.21%	131.988	1.60%	49.318	0.63%	204.224	0.75%	407.401	0.84%
SM2-AHD (ours)	**17.123**	**0.00%**	**54.444**	**0.00%**	**134.132**	**0.00%**	**49.008**	**0.00%**	**202.695**	**0.00%**	404.055	0.010%

the winner on the remaining three is MCTS-AHD, yet SM2-AHD stays within a hair of it there; and against the other three LLM-based frameworks, ReEvo, HSEvo and PathWise, SM2-AHD is better on all nineteen. This bears out our conclusion: SM2-AHD holds a general advantage over existing AHD frameworks.

5.2 Ablation Study

To disentangle the roles of the two components we run two ablations of SM2-AHD against the reproduced baseline and the full model: SM2-AHD$^{-\text{Meta}}$, with Meta-Reflection removed and the memory kept, and SM2-AHD$^{-\text{SEM}}$, with the memory removed and Meta-Reflection kept. Further details are given in Appendix C. Tables 4 and 5 give the results, bold marking the best of the four variants in each test set.

Results. The full model is the only variant that is best on a majority of the nineteen configurations, winning $13/19$ of them. This ablation shows that the two independent components of the meta-level

Table 4: Ablation on the seven constructive test sets (TSP-cons and KP-cons): the reproduced baseline, SM2-AHD without Meta-Reflection (SM2-AHD$^{-\text{Meta}}$), SM2-AHD without the Search Experience Memory (SM2-AHD$^{-\text{SEM}}$), and the full SM2-AHD.

Task	TSP-cons ↓				KP-cons ↑		
Methods	20	50	100	200	50	100	200
MCTS-AHD	4.077	6.339	8.612	12.034	**20.043**	**40.688**	57.863
SM2-AHD$^{-\text{Meta}}$	4.047	6.275	8.563	11.989	20.021	40.671	57.849
SM2-AHD$^{-\text{SEM}}$	3.924	6.131	8.439	11.876	20.025	40.667	57.822
SM2-AHD (ours)	**3.920**	**6.068**	**8.278**	**11.731**	20.041	40.687	**57.871**

Table 5: Ablation on the twelve ant-colony test sets (TSP-ACO, CVRP-ACO, MKP-ACO, BPP-ACO): the reproduced baseline, SM2-AHD without Meta-Reflection (SM2-AHD$^{-\text{Meta}}$), SM2-AHD without the Search Experience Memory (SM2-AHD$^{-\text{SEM}}$), and the full SM2-AHD.

Task	TSP-ACO ↓			CVRP-ACO ↓			MKP-ACO ↑			BPP-ACO ↓		
Methods	20	50	100	20	50	100	100	300	500	120	500	1000
MCTS-AHD	3.866	5.739	8.057	4.979	9.150	15.641	17.101	54.254	133.759	49.026	202.698	**404.016**
SM2-AHD$^{-\text{Meta}}$	4.956	8.100	12.162	4.644	**8.225**	**14.055**	**17.150**	54.147	132.450	49.792	207.240	414.188
SM2-AHD$^{-\text{SEM}}$	3.862	5.701	7.862	5.410	9.565	15.416	17.134	54.126	130.036	49.068	203.276	405.339
SM2-AHD (ours)	**3.862**	**5.697**	**7.854**	**4.637**	8.413	14.679	17.123	**54.444**	**134.132**	**49.008**	**202.695**	404.055

TSP-ACO N=20: the two 3.862 entries are 3.8624 (SM2-AHD$^{-\text{SEM}}$) and 3.8622 (SM2-AHD).

framework of SM2-AHD are both indispensable, acting together to bring our method to its best performance.

6 Conclusion

This paper proposed SM2-AHD, a meta-level LLM-based AHD framework composed of two independent components. **Search Experience Memory** encodes each complete trajectory into a structured memory unit carrying its code lineage, edit differences, performance trace, and failure attribution, and retrieves similar historical cases into the mutation prompt, so that mutation rests on cross-path successful patterns and attributed failure lessons alike; **Meta-Reflection** periodically diagnoses verifiable search statistics and, after safety filtering, adjusts the novelty weight, the exploration constant, and the operator biases, so that the strategy adapts online. Experiments show that SM2-AHD outperforms or matches the existing state-of-the-art methods on 19/19 configurations across six problem classes, and outperforms them on 16/19 with the remaining configurations nearly tied, which shows that SM2-AHD is a viable approach to AHD.

AI use statement

In this work we used generative AI tools. We used them mainly for:

- preliminary screening of the literature;
- writing code;
- translation and language polishing, and formatting and layout;
- drafting the initial manuscript.

We did not use generative AI tools for:

- reading the literature directly, or producing ideas and the core innovations;
- the concrete data processing;
- the direct writing and finalization of the paper's core argument.

In detail. We used an LLM to learn about recent progress on the problems we cared about, so as to grasp the current hotspots more quickly; we also had LLM assistance in writing code; in the early

stage of writing we asked AI to help draft some passages; and once the full text was complete, an LLM helped us with language polishing, layout, and presentation.

We did not use an LLM to generate the ideas directly: we consider that AI can produce erroneous, plausible-sounding, or non-essential research insight. Nor did we let an LLM take part in data processing directly (though in parts of the process we supplied de-identified material and had it generate scripts to assist the data processing), so as to prevent it from producing erroneous or fabricated data. Although we did ask an LLM to generate some draft paragraphs, the exposition of the paper's core argument, the checking of the references, and the second review of the finished article were all carried out independently by the authors, since an LLM's exposition can be imprecise or hallucinated; this way our thinking is conveyed completely and faithfully.

The ideas, code, data, and final polished text produced with LLM assistance were all verified a second time by the authors for correctness, so that no plagiarism, fabrication, or mismatch with reality remains. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

Reproducibility statement

All the work in this paper builds on existing open-source frameworks and adapts them. Details are given in the main text and in the corresponding appendices. We will release the source code used in this work once the paper is accepted.

References

Pratyay Banerjee, Masud Moshtaghi, and Ankit Chadha. APEX-EM: Non-parametric online learning for autonomous agents via structured procedural-episodic experience replay. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing*, 2026.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

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*, 2026.

Harshita Chopra and Chirag Shah. Feedback-aware Monte Carlo tree search for efficient information seeking in goal-oriented conversations. arXiv preprint arXiv:2501.15056, 2025.

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

Sachin Desale, Akhtar Rasool, Sushil Andhale, and Priti Rane. Heuristic and meta-heuristic algorithms and their relevance to the real world: A survey. *International Journal of Computer Engineering in Research Trends*, 2(5):296–304, 2015.

Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

Zafeirios Fountas, Martin A Benfeghoul, Adnan Oomerjee, Fenia Christopoulou, Gerasimos Lampouras, Haitham Bou-Ammar, and Jun Wang. Human-inspired episodic memory for infinite context LLMs. In *International Conference on Learning Representations*, 2025.

Bingzheng Gan, Yufan Zhao, Tianyi Zhang, Jing Huang, Yusu Li, Shu Xian Teo, Changwang Zhang, and Wei Shi. MASTER: A multi-agent system with LLM specialized MCTS. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 9409–9426, 2025.

Ning Gao, Xiuhui Zhang, Xingyu Jiang, Mukang You, Mohan Zhang, and Yue Deng. RF-agent: Automated reward function design via language agent tree search. In *Advances in Neural Information Processing Systems*, 2025.

Oguzhan Gungordu, Siheng Xiong, and Faramarz Fekri. PathWise: Planning through world model for automated heuristic design via self-evolving LLMs. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026.

Xinmeng Hou, Bohao Qu, Wuqi Wang, Peiliang Gong, Qing Guo, and Yang Liu. Learn like humans: Use meta-cognitive reflection for efficient self-improvement. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 28802–28824, 2026.

Wenyang Hui and Kewei Tu. RoT: Enhancing large language models with reflection on search trees. arXiv preprint arXiv:2404.05449, 2024.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. When can LLMs actually correct their own mistakes? A critical survey of self-correction of LLMs. *Transactions of the Association for Computational Linguistics*, 12:1417–1440, 2024.

Nguyen Viet Tuan Kiet, Dao Van Tung, Tran Cong Dao, and Huynh Thi Thanh Binh. MOTIF: Multi-strategy optimization via turn-based interactive framework. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 37000–37008, 2026.

Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. POMO: Policy optimization with multiple optima for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

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

Fei Liu, Xialiang Tong, Mingxuan Yuan, and Qingfu Zhang. Algorithm evolution using large language model. arXiv preprint arXiv:2311.15249, 2023.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, 2024.

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*, volume 40, pp. 37090–37098, 2026a.

Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Zhe Zhao, Xialiang Tong, Kun Mao, Zhichao Lu, Zhenkun Wang, Mingxuan Yuan, and Qingfu Zhang. A systematic survey on large language models for algorithm design. *ACM Computing Surveys*, 58(8):1–32, 2026b.

Gabrielle Kaili-May Liu, Areeb Gani, Jacqueline Lu, Jordan Thomas, Mark Steyvers, and Arman Cohan. Metacognition in LLMs: Foundations, progress, and opportunities. arXiv preprint arXiv:2607.11881, 2026c.

Yihong Liu, Junyi Li, Hongyu Lu, Xin Zhao, and Ji-Rong Wen. Experience-driven reflective coevolution of prompts and heuristics for autonomous algorithm design. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 4981–4995, 2026d.

Yitao Liu, Chenglei Si, Karthik R Narasimhan, and Shunyu Yao. Contextual experience replay for self-improvement of language agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14179–14198, 2025.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. Technical Report arXiv:2506.13131, Google DeepMind, 2025.

Yaniv Oren, Viliam Vadocz, Matthijs T. J. Spaan, and Wendelin Böhmer. Epistemic Monte Carlo tree search. In *International Conference on Learning Representations*, 2025.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, 2026.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–22, 2023.

Laurent Perron and Vincent Furnon. OR-Tools. Google, 2024. URL `https://developers.google.com/optimization/`.

Zishang Qiu, Xinan Chen, Long Chen, Wenjie Yi, and Ruibin Bai. MeLA: A metacognitive LLM-driven architecture for automatic heuristic design. *Expert Systems with Applications*, 330:133022, 2026.

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

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

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023.

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.

Niki van Stein, Diederick Vermetten, and Thomas Bäck. In-the-loop hyper-parameter optimization for LLM-based automated design of heuristics. *ACM Transactions on Evolutionary Learning and Optimization*, 2025.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in Neural Information Processing Systems*, 28, 2015.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024.

Hanyu Wang, Yuanpu Cao, Lu Lin, and Jinghui Chen. PreFlect: From retrospective to prospective reflection in large language model agents. arXiv preprint arXiv:2602.07187, 2026.

Hui Wang, Yang Liu, Xiaoyu Zhang, and Chaoxu Mu. CogMCTS: A novel cognitive-guided Monte Carlo tree search framework for iterative heuristic evolution with large language models. arXiv preprint arXiv:2512.08609, 2025a.

Hui Wang, Xufeng Zhang, and Chaoxu Mu. Planning of heuristics: Strategic planning on large language models with Monte Carlo tree search for automating heuristic optimization. arXiv preprint arXiv:2502.11422, 2025b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 63897–63911, 2025c.

Xingyu Wu, Sheng-Hao Wu, Jibin Wu, Liang Feng, and Kay Chen Tan. Evolutionary computation in the era of large language model: Survey and roadmap. *IEEE Transactions on Evolutionary Computation*, 29(2):534–554, 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*, volume 39, pp. 27144–27152, 2025.

Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, Ran Xu, Phil Mui, Huan Wang, Caiming Xiong, and Silvio Savarese. Retroformer: Retrospective large language agents with policy gradient optimization. In *International Conference on Learning Representations*, 2024.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Helan Liang, and Yong Li. DeepACO: Neural-enhanced ant systems for combinatorial optimization. *Advances in Neural Information Processing Systems*, 36, 2023.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27890–27913, 2025.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and evolving memory skills for self-evolving agents. arXiv preprint arXiv:2602.02474, 2026a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026b.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Learning Representations*, 2024.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte Carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, 2025.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, 2024.

A ALGORITHM FLOWS

Algorithm 2 gives the search procedure of the base MCTS-AHD, and Algorithm 3 gives the complete SM2-AHD main loop (Algorithm 1 in the main text is its condensed form). The base expands a selected node into a batch of children in one expansion, whereas the SM2-AHD loop advances one child per iteration: that is the same expansion unrolled, because each child needs its own operator, its own retrieved context and its own memory unit, so the two components act per child. The evaluation counter therefore counts evaluations, not iterations: the SM2-AHD loop advances it by one per iteration, while the base advances it by the number of children it has just evaluated in one expansion ($b = 2k + 2$ in Algorithm 2), which can carry it slightly past T on the last expansion. The budget T measures the same quantity in the two algorithms.

Algorithm 2 Base: the search procedure of MCTS-AHD (Zheng et al., 2025)

Require: evaluator $\mathcal{E}$, evaluation budget T, LLM $\mathcal{M}$, six operator classes $\mathcal{O} = \{\texttt{i1}, \texttt{m1}, \texttt{m2}, \texttt{e1}, \texttt{e2}, \texttt{s1}\}$
Ensure: globally best heuristic h^*

build the evolution tree $\mathcal{T}$ with a virtual root n_0 containing no heuristic; generate the N_I initial nodes with `i1` ▷ `e1` is a crossover and recombines existing heuristics, so it cannot initialize an empty tree
while $t < T$ **do**
  $n \leftarrow$ SELECT($\mathcal{T}$) ▷ selection: descend from the root taking the child with the largest UCT in Eq. (4)
  expand n into a batch of $2k + 2$ children, whenever n may still take children by progressive widening (Eq. (6)) ▷ expansion: k from `m1`, k from `m2`, one from `e2`, one from `s1`; at the root, progressive widening may instead draw on `e1`, which recombines whole subtrees; see Appendix B
  $o_j \leftarrow \mathcal{E}(h_j)$ for every child h_j of the batch ▷ simulation: evaluate the quality $g(h_j)$ of each child
  BACKPROPAGATE($\mathcal{T}, n, \{o_j\}$) ▷ backpropagation: Q takes the child maximum, N the sum over children
  $t \leftarrow t + b$; $\lambda \leftarrow \lambda(t)$ ▷ $b = 2k + 2$ is the number of newly evaluated children; the exploration coefficient decays linearly with the budget spent
end while
return $\arg\max_{h \in \mathcal{T}} g(h)$

B THE BASE SEARCH IN FULL: MCTS AND MCTS-AHD

B.1 MONTE CARLO TREE SEARCH

Monte Carlo tree search builds the search tree incrementally through repeated sampling. A node n_c represents a state and records a quality value $Q(n_c)$ together with a visit count $N(n_c)$; starting from the root n_r, each round *selects* a leaf n_l by descending repeatedly to the child with the largest UCT value, *expands* that leaf with sampled actions, *simulates* the newly generated nodes to obtain their quality, and *backpropagates* the result to the root. UCT balances exploitation against exploration through an explicit exploration term,

$$\mathrm{UCT}(c) = Q(c) + \lambda \cdot \sqrt{\frac{\ln\big(N(n_c) + 1\big)}{N(c)}}, \qquad c \in \mathrm{Children}(n_c), \tag{3}$$

where $N(n_c)$ is the visit count of the parent node and λ the exploration coefficient. The procedure has proven effective across a range of sequential decision-making tasks, including reward-function design (Gao et al., 2025) and heuristic planning (Wang et al., 2025b).

B.2 THE SIX LLM ACTIONS OF MCTS-AHD

MCTS-AHD (Zheng et al., 2025) was the first to bring Monte Carlo tree search into LLM-based AHD, and in it every heuristic generated so far is retained in a single search tree: the root n_r is a virtual node representing no heuristic, and every other node corresponds to an executable heuristic function together with its natural-language description. The tree is initialized with $N_I = 4$ nodes and the search runs until the number of evaluations approaches the budget T, at which point the best-performing heuristic is returned. Expansion is driven by six classes of LLM actions (Zheng et al., 2025). `i1` initializes a heuristic from scratch. The mutation actions `m1` and `m2` take an existing heuristic as their input: `m1` introduces a new mechanism, while `m2` retunes parameters within the

Algorithm 3 SM2-AHD main loop (full version)

Require: evaluator $\mathcal{E}$, evaluation budget T, mutation model $\mathcal{M}_{\mathrm{f}}$ and reflection model $\mathcal{M}_{\mathrm{s}}$, operator set $\mathcal{O} = \{\texttt{i1}, \texttt{m1}, \texttt{m2}, \texttt{e1}, \texttt{e2}, \texttt{s1}\}$, memory capacity C, number of retrieved items K
Ensure: globally best heuristic h^*

build the evolution tree $\mathcal{T}$ and root n_0; initialize the memory store $\mathcal{M} \leftarrow \varnothing$ and the meta-controller parameters $\Theta \leftarrow \Theta_0$ (the operator weights, the novelty weight λ_{nov}, and the *base* exploration coefficient λ_0); set $t \leftarrow 0$
while $t < T$ **do**
  $n \leftarrow \textsc{Select}(\mathcal{T})$ ▷ selection: descend from the root taking the child with the largest UCT in Eq. (9)
  select an operator op according to the weights Θ, generate a child from n and add it to $\mathcal{T}$ ▷ expansion: progressive widening by visit count
  if $op \in \{\texttt{m1}, \texttt{m2}\}$ **then**
    $R \leftarrow \textsc{Retrieve}(\mathcal{M}, q_n, K)$ ▷ q_n is the code of the candidate n; retrieve similar historical cases by Eq. (8), excluding the current path
    $x \leftarrow \textsc{DiffAware}(n, R)$ ▷ build a difference-aware mutation context and inject the retrieved cases
  else
    $x \leftarrow \textsc{DefaultCtx}(n)$
  end if
  $h \leftarrow \mathcal{M}_{\mathrm{f}}(op, x)$; $o \leftarrow \mathcal{E}(h)$ ▷ simulation: generate a new heuristic and evaluate it
  $\textsc{Backpropagate}(\mathcal{T}, n, o)$; record the operator execution event (operator, success, improvement) ▷ backpropagation: Q takes the child maximum, N the sum over children
  $\textsc{Write}(\mathcal{M}, M)$ ▷ write the memory unit M; failed paths are attributed by $\mathcal{M}_{\mathrm{s}}$ and written as well; evict first-in-first-out once the store holds $C = 25$ units
  if $\textsc{Trigger}(t, \mathcal{T})$ **then** ▷ every tenth evaluation, or after twenty rounds without improvement, and never less than five evaluations after the previous trigger
    $S \leftarrow \textsc{CollectStats}(\mathcal{T}, \mathcal{M})$ ▷ convergence trend, branch visit distribution, operator success rates and average improvement, failure-attribution distribution
    $\Delta\Theta \leftarrow \mathcal{M}_{\mathrm{s}}(S)$; $\Theta \leftarrow \textsc{SafeFilter}(\Theta, \Delta\Theta)$ ▷ the novelty weight may move by at most 30% of its current value in one round, and the round is skipped outright if it has oscillated over the last four; both parameters are clipped to $[0.01, 0.50]$, and the operator weights are re-expressed as integers summing to the expansion batch; the exploration term of $\Delta\Theta$ moves the base λ_0, never the decayed λ
  end if
  $t \leftarrow t + 1$; $\lambda \leftarrow \lambda_0 \cdot \frac{T-t}{T}$ ▷ the exploration coefficient is re-derived from the current base λ_0 and decays with the budget spent, so a meta-adjustment of λ_0 survives this line
end while
return $\arg\max_{h \in \mathcal{T}} g(h)$

existing workflow. The crossover actions `e1` and `e2` recombine several existing heuristics: `e1` is invoked only at the root when progressive widening fires, and draws its inputs uniformly from two to five distinct subtrees, whereas `e2` takes its reference from a dynamically maintained elite set E of the top ten heuristics by $g(\cdot)$. Finally, `s1` performs **tree-path reasoning**: it analyses every distinct heuristic on the path from the leaf n_l under development back to the root n_r and synthesizes their advantageous designs into an enhanced heuristic. One expansion yields $2k + 2$ children: k from `m1`, k from `m2`, one from `e2`, and one from `s1`. Every generated heuristic is accompanied by a natural-language description; because the description is produced by a separate LLM call that reads the finished code, the two stay aligned (*thought alignment*).

Flat-population evolution such as FunSearch, EoH, ReEvo and HSEvo (Dat et al., 2025; Liu et al., 2024; Romera-Paredes et al., 2024; Ye et al., 2024) admits candidates into the population by score at every round, so code that is temporarily weak but still carries evolutionary potential is eliminated outright, which suppresses population diversity and drives the run toward local optima. Because the tree retains even a temporarily low-scoring heuristic, MCTS-AHD need not trade "retaining the best individual" against "maintaining diversity", and "worse first, better later" evolutionary paths stay available; `s1`, for which population-based methods have no counterpart, is a further capability that the tree structure itself provides.

B.3 Three adaptations for cross-task comparability and convergence

Three adaptations make this search comparable and convergent across tasks (Eqs. (4)–(7) of Zheng et al., 2025). The exploitation term is normalized so that UCT values share a scale across problems,

$$\mathrm{UCT}(c) = \frac{Q(c) - q_{\min}}{q_{\max} - q_{\min}} + \lambda \cdot \sqrt{\frac{\ln\big(N(n_c)+1\big)}{N(c)}}, \tag{4}$$

where $q_{\max}$ and $q_{\min}$ are the largest and smallest quality values met so far in the tree. Simulation sets a new leaf's quality directly to its heuristic's performance, and backpropagation updates each ancestor from its children,

$$Q(n_c) \leftarrow \max_{c \in \mathrm{Children}(n_c)} Q(c), \qquad N(n_c) \leftarrow \sum_{c \in \mathrm{Children}(n_c)} N(c), \tag{5}$$

with $Q(n_l) \leftarrow g(h)$ and $N(n_l) \leftarrow 1$ for a newly simulated leaf n_l. Progressive widening keeps non-leaf nodes, especially frequently visited ones, open to re-exploration by adding a child whenever

$$\big\lfloor N(n_c)^{\alpha} \big\rfloor \;\geq\; \big|\mathrm{Children}(n_c)\big|, \qquad \alpha = 0.5. \tag{6}$$

Finally, the exploration coefficient decays linearly with the evaluation budget spent,

$$\lambda = \lambda_0 \cdot \frac{T - t}{T}, \tag{7}$$

which encourages exploration early and guarantees convergence late ($\lambda_0 = 0.1$ for every task). When SM2-AHD is layered on this base, Meta-Reflection adjusts the base value λ_0 and leaves the schedule itself alone, so that the decay of Eq. (7) still governs how the adjusted coefficient is spent (see Appendix Algorithm 3).

B.4 Why MCTS-AHD is a suitable base

The two components of SM2-AHD are not tied to any particular way of organizing the search: as long as the search process leaves behind recordable trajectories, a population-based method can host them just as well, so **the framework can in principle be layered onto any LLM-based AHD framework**, such as FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024), or ReEvo (Ye et al., 2024). What is host-specific is only the *set of parameters* the meta-controller acts on: an exploration constant exists where the host has one, and a population-based host would expose its own knobs (selection pressure, population size, mutation rate) in its place. The mechanism does not depend on the host: it reads verifiable search statistics and adjusts them under a safety filter, whatever those statistics and parameters happen to be. We nevertheless select MCTS-AHD (Zheng et al., 2025) as the concrete base of SM2-AHD, used purely as the substrate that generates, evaluates and retains candidate heuristics, one that the two components only read statistics from and inject context into, so that either of them can be removed on its own for an independent ablation; Appendix B documents that substrate in full and Appendix Algorithm 3 gives the complete SM2-AHD main loop. We choose that base for three reasons. First, *its performance is well validated*: the authors report an advantage on nearly all test sets over population-based methods including FunSearch, EoH, and ReEvo, as well as over human-designed heuristics (Zheng et al., 2025); a number of tree-search methods have since performed more strongly, but they are not designed specifically for AHD, targeting instead multi-agent planning, sparse-reward reinforcement learning, and information seeking respectively (Chopra & Shah, 2025; Gan et al., 2025; Oren et al., 2025), and their gains come mainly from mechanisms unrelated to heuristic design. Second, *its structure fits naturally*: a single edit corresponds exactly to one edge of the tree and a single root-to-leaf path to one complete trajectory, so the tree doubles as the recording unit of memory without modification; by contrast, flat population-based evolution lacks such a hierarchical container, and once an individual is eliminated its trajectory is lost with it. Last, *it exposes the problem sharply*: MCTS-AHD deliberately retains individuals that perform poorly for now, which makes the loss from "experience being discarded" more conspicuous and the payoff from experience reuse more direct and easier to observe.

C Memory Retrieval and Execution-Layer Details

Beyond the two core components, the SM2-AHD execution layer contains two lightweight techniques that do not change the overall architecture and act only on the search's own internal parameters; they belong to the execution layer rather than to either component, and the ablation of

Section 5.2 does not vary them. The first is a **novelty-enhanced UCT**: in code search a mutation often changes only a few lines, so parent and child code are highly similar, while standard UCT allocates attempts solely by *visit count*, so the budget keeps flowing into nearly the same local direction; we therefore reward, at the selection stage, children that differ more from their parent (Eq. (9)). The second is **difference-aware mutation guidance**: if the prompt gives only the parent code, the model cannot know what was changed last time or with what effect and tends to fine-tune repeatedly in an already-converged direction, so we inject the previous edit and its effect into the prompt and explicitly require the model to **edit at a location different from last time**. Both are given in full below.

C.1 The Scoring Function for Memory Retrieval

When a node's code q is to be mutated, SM2-AHD retrieves the most similar entries from the *same-task* memory pool (we take $K = 2$):

$$\text{score}(M \mid q) = (1 - w) \cdot \underbrace{\text{Jacc}\big(\Phi(q), \Phi(M)\big)}_{\text{Jaccard similarity of identifier sets}} + w \cdot \underbrace{\text{rec}(M)}_{\text{recency within the pool}}, \tag{8}$$

The recency weight w in Eq. (8) takes values in $[0, 1]$ (we take 0, i.e., pure lexical-similarity retrieval). To control prompt length, the injected passage has a character budget (about 600 characters in this paper). This "similarity + recency" two-stage retrieval idea is in line with episodic-memory models that use similarity retrieval and temporal-contiguity retrieval together (Fountas et al., 2025).

C.2 Novelty-Enhanced UCT

The role of the selection stage is to direct the limited evaluation budget toward the most promising regions as quickly as possible. SM2-AHD adds to Eq. (4) an extra novelty bonus measured by *code similarity*, making the selection stage favor children that differ more from their parent:

$$\text{UCT}(n, t) = \underbrace{\frac{Q(n) - q_{\min}}{q_{\max} - q_{\min}}}_{\text{exploitation}} + \underbrace{\lambda_0 \Big(1 - \frac{t}{T}\Big) \sqrt{\frac{\ln\big(N(p) + 1\big)}{N(n)}}}_{\text{exploration}} + \underbrace{\lambda_{\text{nov}} \Big(1 - \frac{t}{T}\Big) \eta(n)}_{\text{novelty}}, \tag{9}$$

where t is the evaluation budget already spent and T the total, p is the parent of n, novelty is the Jaccard distance between the identifier sets $\Phi(p)$ and $\Phi(n)$ of parent and child code, $\eta(n) = 1 - \frac{|\Phi(p) \cap \Phi(n)|}{|\Phi(p) \cup \Phi(n)|}$, and the two coefficients are separate quantities rather than two names for one: λ_0 is the initial value of the *exploration* coefficient, the same symbol and the same constant as in Eq. (7), so the middle term here is that equation's λ written out in full, while λ_{nov} is the initial weight of the *novelty* term. The two are set independently (both happen to be 0.1 in this paper) and the meta-controller adjusts them separately; the novelty term is there to suppress repeated visits to near-duplicate code regions and broaden coverage of the heuristic space. Attaching an extra bonus to the UCT selection term to guide the search is not original to this paper: feedback-aware tree search has used it for goal-oriented information seeking (Chopra & Shah, 2025), and cognitive tree search writes the uncertainty introduced by a learned model explicitly into a UCB-based deep-exploration term (Oren et al., 2025). We follow this form; the difference lies in the semantics of the bonus: it measures *novelty at the code level* rather than the information value of a solution or a state.

C.3 Difference-Aware Mutation Guidance

If the mutation prompt gives only the parent code, the LLM has no way of knowing "what was changed last time and with what effect," so it tends to fine-tune repeatedly in an already-converged local direction, spending the budget on edits nearly identical to the previous round. Against this, for the two mutation operators `m1`/`m2` SM2-AHD injects into the prompt the unified diff of the "grandparent $\rightarrow$ parent" step, the objective values before and after that edit, and the operator used, and explicitly requires the model to **edit at a location different from last time**, thereby steering the change toward regions not yet explored. This borrows from two existing lines of experience: Reflexion uses a retrospective model to summarize the root cause of failure from environment rewards and generate a follow-up action plan, and then rewrites the agent's prompt accordingly (Yao et al.,

2024), and we take only its form of "writing the failure root cause and the next plan into the context together," without training a rewriting model; and EvoPH shows that letting a prompt co-evolve with the heuristic under the drive of *fine-grained execution feedback* improves automatic algorithm design (Liu et al., 2026d), and we follow its idea of "driving prompt evolution with execution feedback," except that the feedback is compressed into a single diff and a single change in objective value.

D Problem Definitions and Solver Frameworks

This appendix states the five combinatorial optimization (CO) problems that Section 5 uses, and the two solver frameworks within which the single key heuristic function of Eq. (1) is designed. The problems are given in Appendix D.1, the frameworks in Appendices D.2 and D.3. Every constant quoted below is the one our implementation actually uses.

D.1 The five CO problems

An instance is a pair $(\mathbf{ins}, f)$ with $f : \mathcal{S} \to \mathbb{R}$ the objective and $\mathcal{S}$ the feasible set; n always denotes the problem size. TSP, CVRP and BPP are minimization problems, KP and MKP maximization problems. For all distance-based problems the nodes are points in the plane and $d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|_2$ is the Euclidean distance; the TSP family of benchmark instances of this form goes back to TSPLIB (Reinelt, 1991).

TSP. Definition. Given n nodes, find a closed tour visiting each exactly once with minimum total length. **Formulation.** With π a permutation of $\{1, \dots, n\}$ and $\pi(n{+}1) = \pi(1)$,

$$\min_{\pi} \sum_{i=1}^{n} d_{\pi(i),\pi(i+1)}. \tag{10}$$

Instance generation. Coordinates are drawn uniformly from the unit square, $\mathbf{x}_i \sim \mathcal{U}[0,1]^2$.

CVRP. Definition. A depot serves n customers, customer i demanding q_i; a fleet of identical vehicles of capacity Q must serve every customer, each route starting and ending at the depot, minimizing total travelled distance. **Formulation.** Let $x_{ij} \in \{0,1\}$ indicate that edge (i,j) is used and let node 0 be the depot:

$$\begin{aligned} \min \ & \sum_{(i,j)} d_{ij} x_{ij} \\ \text{s.t.} \ & \sum_{i} x_{ij} = 1 \qquad (j \neq 0), \\ & \sum_{j} x_{ij} = 1 \qquad (i \neq 0), \\ & \sum_{i \in S} \sum_{j \notin S} x_{ij} \ \geq \ \left\lceil \tfrac{1}{Q} \textstyle\sum_{i \in S} q_i \right\rceil \qquad \forall\, \varnothing \neq S \subseteq V \setminus \{0\}, \end{aligned} \tag{11}$$

the first two constraints being the in-degree and out-degree constraints, which make every customer the head and the tail of exactly one used edge, and the third the rounded capacity inequality that simultaneously forbids subtours and enforces the load limit. **Instance generation.** The depot sits at the centre of the unit square, $\mathbf{x}_0 = (0.5, 0.5)$; customer coordinates are uniform on the square and demands are uniform integers in $\{1, \dots, 9\}$; $Q = 50$.

KP. Definition. Choose a subset of n items, each with weight w_i and value v_i, to maximize total value without exceeding the capacity C; each item may be taken at most once. **Formulation.**

$$\max_{y \in \{0,1\}^n} \sum_{i=1}^{n} v_i y_i \quad \text{s.t.} \quad \sum_{i=1}^{n} w_i y_i \ \leq \ C. \tag{12}$$

Instance generation. Weights and values are both uniform on the unit interval, $w_i, v_i \sim \mathcal{U}[0,1]$; the capacity is $C = 25$ at $n = 100$ and $n = 200$, and $C = 12.5$ at $n = 50$.

MKP. Definition. As KP, but each item consumes one unit of resource in each of m dimensions and every dimension has its own budget. **Formulation.** With w_{ik} the consumption of item i in dimension k,

$$\max_{y\in\{0,1\}^n} \sum_{i=1}^{n} p_i y_i \quad \text{s.t.} \quad \sum_{i=1}^{n} w_{ik} y_i \;\le\; 1, \qquad k = 1, \ldots, m. \tag{13}$$

Instance generation. Prizes are uniform, $p_i \sim \mathcal{U}[0,1]$, as are the raw weights; with $m = 5$ dimensions the budgets are drawn as $C_k \sim \mathcal{U}[\max_i w_{ik}, \sum_i w_{ik}]$ and each column is then divided by its budget, which is what makes the right-hand side of Eq. (13) uniformly 1 and the instances "well stated" in the sense that no single dimension is trivially slack or trivially binding.

BPP. Definition. Pack n items of integer sizes q_i into as few bins of capacity C as possible; the items are all known in advance (the *offline* setting), so the packing is planned with full information. **Formulation.** With $\mathcal{B}$ a partition of $\{1, \ldots, n\}$ into bins,

$$\min \; |\mathcal{B}| \quad \text{s.t.} \quad \sum_{i\in b} q_i \;\le\; C \quad \forall\, b \in \mathcal{B}. \tag{14}$$

Instance generation. Sizes are uniform integers in $\{20, \ldots, 100\}$ and $C = 150$; the same generator also supplies the online variant, which we do not report.

D.2 Step-by-step construction

Framework description. The solution is built one element at a time. At step t let S_t be the partial solution and $\mathcal{C}(S_t)$ the set of elements that may still be appended without violating feasibility. A scalar scoring function h scores each candidate, and the framework commits to the best-scoring one,

$$c_{t+1} = \arg\max_{c\in\mathcal{C}(S_t)} h\big(S_t, c\big), \tag{15}$$

repeating until $\mathcal{C}(S_t) = \varnothing$, at which point S_t is the finished solution. The framework supplies the loop, the feasibility bookkeeping and the objective evaluation; **the only thing the LLM designs is the scoring function** h, which is why this framework makes a clean test bed for AHD: the search space is exactly the set of admissible scoring rules. Learning such a rule with a neural network is the setting of pointer networks (Vinyals et al., 2015) and of neural combinatorial optimization with reinforcement learning (Bello et al., 2016); here the rule is written in Python by an LLM instead.

Instantiation. Two constructive tasks appear in Section 5. For TSP the framework grows a tour from a start node and the designed function is

```
select_next_node(current_node: int, destination_node: int,
unvisited_nodes: set, distance_matrix: np.ndarray) -> int
```

which returns the index of the next node to visit; the objective is the tour length of Eq. (10). For KP the framework fills the knapsack and the designed function is

```
select_next_item(remaining_capacity: int, weights: np.ndarray,
                 values: np.ndarray) -> int
```

which returns the index of the next item to take; the objective is the total value of Eq. (12). The function name is read out of that signature by the framework rather than written into the loop.

D.3 Ant colony optimization

Framework description. Ant colony optimization (ACO) (Desale et al., 2015; Dorigo et al., 2006) solves a CO problem through a colony of constructive agents that deposit pheromone on the solution components they use, so that later agents are biased toward components that have proved useful. The framework keeps two matrices over the problem's components: a pheromone matrix $\boldsymbol{\tau}$, which is learned from the colony's own experience, and a heuristic matrix $\boldsymbol{\eta}$, which carries prior knowledge about the instance in front of it. An ant at component i chooses its next component j from the

feasible neighbourhood $\mathcal{N}_i$ by

$$p_{ij} = \frac{\left[\tau_{ij}\right]^{\alpha}\left[\eta_{ij}\right]^{\beta}}{\sum_{l\in\mathcal{N}_i}\left[\tau_{il}\right]^{\alpha}\left[\eta_{il}\right]^{\beta}}, \tag{16}$$

and after all ants have built their solutions the pheromone matrix is decayed and replenished,

$$\tau_{ij} \;\leftarrow\; \gamma\,\tau_{ij} + \Delta\tau_{ij}, \qquad \gamma = 1-\rho, \tag{17}$$

where ρ is the evaporation rate and the deposit $\Delta\tau_{ij}$ is task-specific. **The position we hand to the LLM is the heuristic matrix $\boldsymbol{\eta}$**, and this is the point at which our setting departs from textbook ACO: in the classical form $\boldsymbol{\eta}$ is a fixed prior such as $\eta_{ij} = 1/d_{ij}$, whereas here it *is* the designed object: the function the LLM writes is literally named `heuristics`, and its return value is used as $\boldsymbol{\eta}$ in Eq. (16). Neural-enhanced ant systems make the same substitution learned rather than written (Ye et al., 2023); we keep the substitution and change who supplies it.

Instantiation. Algorithm 4 gives the skeleton the four ACO tasks share, and Table 6 lists where they differ. The four are genuinely different implementations, not one implementation with four parameter settings: they disagree on what a pheromone entry means (an edge, an item, or an item *pair*), and consequently on what a deposit is. What they share, and what we verified across all four, is the selection rule of Eq. (16) with $\alpha = \beta = 1$ and a feasibility mask, and the fact that the only part written by the LLM is $\boldsymbol{\eta}$.

Algorithm 4 ACO as instantiated by the four ACO tasks: the shared skeleton

Require: instance $\mathbf{ins}$; the heuristic matrix $\boldsymbol{\eta}$ returned by the designed function h; ant count A; iteration count I; exponents α, β; decay γ
Ensure: the best solution found

$\boldsymbol{\tau} \leftarrow \mathbf{1}$ ▷ carrier is task-specific: an $n \times n$ matrix, or a length-$(n{+}1)$ vector (Table 6)
normalize $\boldsymbol{\eta}$ if the task requires it ▷ preprocessing, and it must precede every use of $\boldsymbol{\eta}$ as a sampling probability: see the note below
$s^* \leftarrow$ a solution built greedily from $\boldsymbol{\eta}$ ▷ the fallback returned if no ant ever beats the initial solution
for I iterations **do**
  for A ants **do**
    build a solution by repeatedly sampling the next component with probability proportional to $\tau^{\alpha} \odot \eta^{\beta} \odot \mathbf{mask}$ restricted to the feasible neighbourhood ▷ Eq. (16); the mask admits only components that keep the solution feasible
  end for
  evaluate every solution with the task's objective f
  $\boldsymbol{\tau} \leftarrow \gamma\,\boldsymbol{\tau} + \Delta\boldsymbol{\tau}$ ▷ $\Delta\boldsymbol{\tau}$ is task-specific; Table 6 gives all four rules
  $s^* \leftarrow$ the better of s^* and this iteration's best solution, in the direction of f
end for
return s^*

Preprocessing of $\boldsymbol{\eta}$. Because the LLM returns a raw array, the framework normalizes it before it reaches Eq. (16), and the four tasks differ here as well. TSP adds a floor of 10^{-9}. CVRP inspects the signature of the designed function and adapts its call: a function taking four arguments is passed the distance matrix, the coordinates, the demands and the capacity $Q = 50$, while one taking two arguments is passed the distance matrix and demands normalized by the capacity; a floor of 10^{-9} is then added. MKP appends one extra slot to the returned vector for a virtual "stop" node, with a fixed value of 10^{-8}. BPP clips the returned matrix into $[10^{-6}, 10^{6}]$, divides by its maximum, and clips again. This division of labour is deliberate: **the LLM writes one function and nothing else**, and each task's evaluator owns everything around it: instance generation, feasibility, evaluation and the normalization above.

Three remarks on Table 6. First, ρ is the evaporation rate, so $\rho = 1 - \gamma$: the three tasks with $\gamma = 0.9$ evaporate at 0.10 and BPP, which decays at $\gamma = 0.95$, evaporates at 0.05. Second, the pheromone entry does not always denote an edge. MKP deposits on *items*, which is why its pheromone is a vector rather than a matrix, and it is a maximization problem, so its deposit grows with the objective rather than shrinking. BPP goes further and deposits on *pairs of items that share a bin*, so that what is reinforced is a co-occurrence rather than a transition; its deposit is the fitness $g = \frac{1}{|\mathcal{B}|}\sum_{b\in\mathcal{B}}(\text{occupied}_b/C)^2$, the mean squared bin utilization, which is larger for packings that fill their bins and is therefore maximized, while the reported objective is still the bin count. Third,

Table 6: The four ACO tasks. "τ carrier" is what one pheromone entry stands for; $\Delta\tau$ is the deposit of Eq. (17) with c the objective of the finished solution and A the number of ants. All four use the transition rule of Eq. (16) with $\alpha = \beta = 1$.

Task	τ carrier	Deposit $\Delta\tau$	ρ	#ants	#iters	Dir.
`tsp_aco`	$n \times n$ matrix	$\frac{1}{c}$ on *both* (i,j) and (j,i)	0.10	30	100	min
`cvrp_aco`	$n \times n$ matrix	$\frac{1}{c}$ on the travelled direction only, floored at 10^{-10}	0.10	30	100	min
`mkp_aco`	vector of length $n{+}1$	$Q\,c$ on each selected item, $Q = 1/\sum_i p_i$	0.10	10	50	max
`bpp_offline_aco`	$n \times n$ co-occurrence	g/A on every same-bin item pair	0.05	20	15	min

the iteration counts differ by more than sixfold across tasks (15 to 100), so the per-evaluation cost of a heuristic is far from uniform; we therefore report the evaluation budget T of Section 5 rather than wall-clock time as the common currency of the comparison.

E PROMPTS

This appendix records the prompts that the framework actually sends to the language model. They are reproduced verbatim rather than reconstructed, and the constants they contain (the number of retrieved memory items, the character budget, the diff truncation length) are the ones in force during the runs reported in Section 5. Two models are used, as Section 5 describes: the six operators go to `deepseek-flash`, the code-generating model shared by every method we compare against, while the failure attribution of SEM and the meta-analysis of Meta-Reflection go to the stronger `deepseek-v4-pro`. Beyond the model, what differs by role is the temperature: the operators run at temperature 1.0, which buys variety in the proposals, while the failure attribution and the meta-analysis run at 0.3, since both must produce a stable judgement rather than a varied proposal. Table 7 lists which prompt belongs to which component, which model receives it, and at what temperature.

Table 7: The prompts of this appendix, the component they belong to, the model that receives them, and the temperature they are driven at. The two models are those of Section 5; the temperatures are the ones the two clients are configured with: 1.0 for the flash client, 0.3 for the pro client.

Where	Prompt	Component	Model	Temp.
App. E.1	shared skeleton of the six operators	execution layer	`deepseek-flash`	1.0
App. E.2	operator-specific instruction (i1/e1/e2/m1/m2/s1)	execution layer	`deepseek-flash`	1.0
App. E.3	`m1` in full, with both injections	SEM + diff-aware	`deepseek-flash`	1.0
App. E.4	failure attribution	SEM (write path)	`deepseek-v4-pro`	0.3
App. E.4	historical-case block	SEM (retrieve path)	—	—
App. E.5	meta-analysis	Meta-Reflection	`deepseek-v4-pro`	0.3

E.1 HOW A PROMPT IS ASSEMBLED

Every prompt is built from four pieces, three of which are fixed per task and one of which depends on the operator. The task description is fixed per task and is the same text for every operator, as are the function signature and the input/output description; and the operator instruction is what distinguishes the six operators of Appendix B. The framework reads the function name out of the signature, which is why the name the model is told to use varies by task but not by operator: `select_next_node` for constructive TSP, `select_next_item` for constructive KP, and `heuristics_v2` for all four ACO tasks, where the literal {`version`} placeholder in the signature is filled with 2 before the name is parsed.

The skeleton below is the part common to all six operators, printed in the form that `m1` and `m2` give it. Colours follow the convention of HiFo-Prompt (Chen et al., 2026), whose prompts share this skeleton: green is the task description, blue the function name the model must define, red the declared counts, and orange the input and output names.

The common skeleton of the six operators, as `m1` and `m2` give it

{task description}
{the operator instruction, given separately for each of the six operators in Appendix E.2}
{for `m1` and `m2` only: the injected context of Appendix E.3}
First, describe the design idea based on the provided algorithm and main steps of the new algorithm in one sentence. The description must be inside a brace outside the code implementation. Next, implement it in Python as a function named {function name}.
This function should accept {N} input(s): {input names}. The function should return {M} output(s): {output names}. {input/output description}
Do not give additional explanations.

Two pieces of the printed skeleton need a word of explanation, because they are empty or absent in ways that are not obvious from the text. The slot after the output names, which in the base frameworks carries a task-specific hint, is present in our prompt builder but is hard-wired to the empty string, so no hint is ever injected there; the only task-specific knowledge that reaches the model is the task description and the input/output description. And the brace is load-bearing: the framework parses the model's reply by taking the text of the *first* brace-delimited group as the algorithm's natural-language description, and its code as the span running from the first `import` to the last `return`, so "the description must be inside a brace" is a parsing contract rather than a stylistic request. A reply from which either part cannot be extracted is not repaired but resampled, and the loop gives up after four further attempts.

E.2 The six operator instructions

The six operators differ chiefly in the instruction that follows the task description, and each then continues with the skeleton of Appendix E.1; the variations the other four introduce into that skeleton are collected at the end of this subsection. The instructions below are quoted verbatim; the framework substitutes the function name, the input and output lists, and the description of the inputs and outputs at the points marked in the skeleton, and the notation {`indivs`} denotes the one or more existing heuristics the operator is given, each rendered as its description, its code and its objective value.

Operator-specific instruction (inserted after the task description)

`i1` (initialization, from scratch). *[no preceding heuristic; the prompt opens directly with the skeleton]*

`e1` (crossover, divergent). I have k existing algorithms with their codes as follows: {indivs} Please create a new algorithm that has a totally different form from the given algorithms. Try generating codes with different structures, flows or algorithms. The new algorithm should have a relatively low objective value.

`e2` (crossover, assimilative). I have k existing algorithms with their codes and objective values as follows: {indivs} Please create a new algorithm that has a similar form to the No.k algorithm and is inspired by the No.1 algorithm. The new algorithm should have a objective value lower than both algorithms. Firstly, list the common ideas in the No.1 algorithm that may give good performances. Secondly, based on the common idea, describe the design idea based on the No.k algorithm and main steps of your algorithm in one sentence. The description must be inside a brace. Thirdly, implement it in Python as a function named {function name}.

`m1` (mutation, new mechanism). I have one algorithm with its code as follows. Algorithm's description: {description} Code: {code} Please create a new algorithm that has a different form but can be a modified version of the provided algorithm. Attempt to introduce more novel mechanisms and new equations or programme segments. *[the injected context of Appendix E.3 follows here]* First, describe the design idea based on the provided algorithm and main steps of the new algorithm in one sentence.

`m2` (mutation, parameter retuning). I have one algorithm with its code as follows. Algorithm's description: {description} Code: {code} Please identify the main algorithm parameters and help me in creating a new algorithm that has different parameter settings to equations compared to the provided algorithm. *[the injected context of Appendix E.3 follows here]* First, describe the design idea based on the provided algorithm and main steps of the new algorithm in one sentence.

`s1` (tree-path reasoning). I have k existing algorithms with their codes and objective values as follows: {indivs} Please help me create a new algorithm that is inspired by all the above algorithms with its objective value lower than any of them. Firstly, list some ideas in the provided algorithms that are clearly helpful to a better algorithm. Secondly, based on the listed ideas, describe the design idea and main steps of your

new algorithm in one sentence. The description must be inside a brace. Thirdly, implement it in Python as a function named {function name}.

Three differences among the six are visible only by comparing the transcript above with the skeleton of Appendix E.1, and are worth stating. The skeleton is printed in the form `m1` and `m2` give it; `e2` and `s1` fold the same sentences into their own numbered list, so there the brace sentence drops the clause "outside the code implementation" and "Next, implement it" becomes "Thirdly, implement it". `i1` and `e1` also open with the design idea of "*your* algorithm" where `m1` and `m2` say "based on the provided algorithm. . . of the *new* algorithm", and `m2` alone omits the "The function should return M output(s)" clause, so its output contract is weaker than the other five. And only three of the six, `e1`, `e2` and `s1`, carry a performance instruction, in all three phrased as "low": "a relatively low objective value", "a objective value lower than both algorithms", "lower than any of them". The other three carry none, and the direction of optimization is stated nowhere in the six; for the two maximization tasks (KP and MKP) a lower objective value is the worse outcome, and nothing in the prompt says so. We quote the instructions as they stand. That the two maximization tasks still reach the quality reported in Section 5 suggests the model was steered by the problem description rather than by this one word, but no control run removes it, so the size of its effect is not measured here.

E.3 `m1` IN FULL: THE TWO INJECTIONS

The mutation operators are where the two components of Section 4 enter the prompt. Both injections are inserted between the operator instruction and the format constraints, in that order: first the difference-aware context of Appendix C, then the retrieved cases of SEM. The transcript below is the assembled prompt for `m1`; the two injected blocks are marked, and all other text is the skeleton of Appendix E.1 with the operator instruction of Appendix E.2. Individual code and retrieved code are elided as placeholders: the framework inlines them in full, and we elide them here only because they run to hundreds of lines.

`m1` as sent: skeleton + diff-aware context + SEM retrieval

Solving Traveling Salesman Problem (TSP) via stochastic solution sampling following "heuristics". TSP requires finding the shortest path that visits all given nodes and returns to the starting node.
I have one algorithm with its code as follows.
Algorithm's description: {the parent's natural-language description}
Code:
{the parent's Python code, inlined in full}
Please create a new algorithm that has a different form but can be a modified version of the provided algorithm. Attempt to introduce more novel mechanisms and new equations or programme segments.

[injection 1 — difference-aware mutation guidance]
Note: The previous modification changed the objective from {parent objective} to {current objective} (using operator {'m1'}).
Code changes from the previous version are shown below:
{a unified diff of the grandparent → parent edit}
Focus your modification on DIFFERENT parts than the previous change. Avoid repeating the same type of modification.

[injection 2 — retrieved cases from the Search Experience Memory]
— Reference history (SEM) —
[1] Similar past attempt (similarity=<2 decimals>, outcome=<outcome>, obj=<4 decimals>):
Last edit: <op>, obj <4 decimals> → <4 decimals>
Diff snippet: <at most 150 characters of the diff>, in a fenced block
Reflection: <the attributed one-sentence reflection>

First, describe the design idea based on the provided algorithm and main steps of the new algorithm in one sentence. The description must be inside a brace outside the code implementation. Next, implement it in Python as a function named heuristics_v2.
This function should accept 1 input(s): 'distance_matrix'. The function should return 1 output(s): 'heuristics_matrix'. The 'heuristics' function takes as input a distance matrix, and returns prior indicators of how promising it is to include each edge in a solution. The return is of the same shape as the input.
Do not give additional explanations.

E.4 The two prompts of the Search Experience Memory

SEM sends a prompt of its own on the write path as well. When a trajectory ends in a failure, the last step of that trajectory is handed to `deepseek-v4-pro` for attribution, and the resulting one-sentence diagnosis is stored alongside a category label that the framework parses back out of the reply. Which trajectories count as failures is fixed by the label y of Section 4.1: a trajectory whose final heuristic did not improve on the objective of the heuristic it was derived from.

SEM, write path: failure attribution

Analyze why the following heuristic code performed WORSE than its parent.
Parent objective: {
Child objective: {
Parent code:
{the parent's code}
Child code:
{the child's code}
Respond in exactly one sentence: what is the most likely cause of the degradation? Prefix with FAILURE_REASON: (overfitting / syntax_error / wrong_direction / other)

The reply is parsed by substring search for the three named categories, defaulting to `other`, and the diagnostic prefix is stripped before the sentence is stored as the reflection R of the memory unit. Two consequences follow, and they are properties of this prompt rather than of the framework's intent: the categories are matched anywhere in the sentence, not only in the prefix position the prompt asks for, so a reflection that merely mentions a word such as "overfitting" is filed under that category regardless of where it appears; and only the leaf and its parent are shown to the model, so a failure that is only visible against the wider trajectory is attributed from a single edit. We note both because the resulting category distribution is one of the four signals Meta-Reflection reads, as Appendix E.5 records, so a systematic misfiling would propagate. How often the substring rule actually misfiles is not measured here: both points are reported as properties of the parsing, not as effects we have quantified.

E.5 The Meta-Reflection prompt

Meta-Reflection is triggered on a timer or on detected stagnation, and its prompt is composed of five blocks of statistics that the host search collects and one output contract. It is the only prompt in the framework that reads the state of the search rather than a single solution, and it is the only one whose reply is parsed as JSON.

Meta-Reflection: state analysis and parameter adjustment

You are analyzing a running MCTS-AHD search process for combinatorial optimization. The goal is to discover high-quality heuristics automatically.
Current Search Status
- Evaluations used: {eval_times}/{fe_max}
- Current best objective: {current_best}
- Best objective 20 rounds ago: {best_20_ago}
- Stagnation: {stagnation_rounds} rounds without improvement
- Convergence rate (last 20 rounds): {convergence_rate:.4f}
Branch Distribution
{branch_summary}
Operator Performance (last 20 rounds)
{operator_table}
Current Strategy Parameters
- Novelty weight: {current_novelty_weight} (range: 0.01 $\sim$ 0.50)
- Exploration constant (UCT C): {current_exploration_constant}
- Operator weights: {operator_weights}
Memory Stats
- SEM bank size: {mem_total} (success={mem_success}, failure={mem_failure})
- Common failure reasons: {failure_reasons}

Your Task
Based on the above information, analyze the current search state.
Respond with EXACTLY a JSON object (no extra text, no markdown):
{ "analysis_1": "one-sentence health assessment", "analysis_2": "one-sentence main bottleneck diagnosis", "novelty_weight": <float or null>, "exploration_constant": <float or null>, "operator_bias": { "m1": "up" or "down" or "keep", "m2": ..., "e2": ..., "s1": ... }, "reasoning": "one sentence why this adjustment helps" }
Rules:
- novelty_weight ∈ [0.01, 0.50]; null = no change
- exploration_constant ∈ [0.01, 0.50]; null = no change
- Only change parameters that genuinely need adjustment.

Three features of this prompt constrain what Meta-Reflection can do, and all three are deliberate. A null field means "no change", so abstaining is expressible and is the default the model is told to prefer. Only four of the six operators appear in the contract, `m1`, `m2`, `e2` and `s1`, so only those four can be biased; the operator-statistics block above reports on five, adding the crossover `e1`, and `i1` is absent from both, since it seeds the tree at initialization and the expansion loop never draws it again. And the ranges are hard limits written into the prompt, which the safety filter of Section 4 then enforces independently: the prompt asks the model to stay inside them, and the filter clips the reply if it does not. The failure-attribution distribution of Appendix E.4 appears in the last block, which is the channel by which the two components are connected. One abbreviation is ours and not the code's: the operator-bias object of the JSON contract repeats the same three alternatives for each operator, and we elide the second to fourth with an ellipsis, whereas the prompt writes all four out. One name likewise comes from the code: the exploration constant is written λ_0 in the main text and $\lambda_0(1 - t/T)$ in Eq. (9), but the prompt calls it `UCT C`, the name it carries in the implementation, so the two refer to the same parameter.

E.6 Defaults, categories, and two prompts we do not use

For completeness, the framework contains two further prompts, both applied after a heuristic has been accepted. One describes a finished code in fewer than five sentences, and the other re-describes the algorithm, this time together with its code, in fewer than three. They matter to the method only indirectly, since the description they produce is what the operators of Appendix E.2 are shown in place of the parent's source, and its purpose is to keep the description and the code aligned.

F Experimental Setup

F.1 Setup

Five LLM-based AHD frameworks are run in this study: ReEvo, HSEvo, PathWise, MCTS-AHD and SM2-AHD. One setting is forced to a common value, and it is forced in the interest of fairness: the evaluation budget `max_fe` is 1000, the value the original MCTS-AHD paper gives LLM-based AHD methods on its test sets. It is the published default of MCTS-AHD and SM2-AHD; ReEvo and HSEvo use 100 and 450 and were brought up to it by an explicit override on the command line. Every other setting is left at the value its own framework specifies, so that each method runs under the configuration its authors chose. The one further override is the evolutionary scale of the same two frameworks, which default to 30: we set the initial population to 4 and the ceiling to 10, the values the base framework uses, so that the evolutionary scale is comparable across the family. The conditions of the calls are common as well: one and the same model through one and the same client layer, shared by all five frameworks, with a thread cap of 4 and timeouts of 60 s for the constructive classes and 120 s for the ant-colony ones.

PathWise is the exception on the budget, and the reason is fairness again. Its framework is multi-agent, and its original paper reports that $T = 500$ costs it about as many tokens as $T = 1000$ costs the frameworks it compares against; those comparisons were themselves run under that setting. We therefore leave PathWise at its own $T = 500$ and respect its authors' initial setting.

The four existing frameworks do not provide everything this comparison needs. Some problem classes have no implementation in the framework at all; in PathWise's case the instance-generation

routines and the validation data were missing for several classes, and its evaluation routine for `kp_constructive`, which reports a size the other frameworks do not, was replaced by the base one. What is missing we fill in by bit-level copy from the base framework, verified element by element, 25 of 25 arrays identical, so that all five frameworks are evaluated on the same instances.

The columns that are not AHD methods come from three different places. POMO, DeepACO and VSR-LKH are direct implementations of the algorithms of the same name, with the published weights and solvers used as they are and nothing retrained or refitted here. The solver behind the knapsack optima is Google OR-Tools itself, but the modelling recipe handed to it is not ours: it is copied verbatim from the knapsack recipe of MCTS-AHD, and extended from the single size that recipe covers to every size of both knapsack classes, for one of which MCTS-AHD gives no OR-Tools recipe at all. The last two rows are written by us from the description in the MCTS-AHD paper, because no AHD framework provides either of them: the Greedy Construct row, nearest neighbour for TSP and value over weight for KP, and the heuristic matrix of the default-ACO row, the distance matrix on TSP and CVRP, value over mean weight on MKP and an exponential of the deviation from capacity on BPP, with the ant-colony implementation underneath it being SM2-AHD's own.

Some entries of the three tables are marked `N/A`, because these algorithms have no implementation in the four AHD frameworks and, once we had implemented the methods of the original papers and wired them in, the published model parameters turned out to be incomplete: pretrained weights are missing for some problem sizes. Out of respect for their authors, we substitute and approximate nothing and mark the entries of such uncertain algorithms as missing values. The core comparison of this study is among AHD methods rather than against these other operations-research and deep-learning frameworks, so the missing entries do not affect the fairness of the comparison.

F.2 Language-model usage and token cost

Metrics. Table 8 reports, for each problem class, the token usage of one complete SM2-AHD training cycle (the runs of Section 5 that reached its evaluation budget of 1000 at the population of Appendix F, averaged over the three repeats of that class), split by the model that served each call (the six operators to `deepseek-flash`, the two reflective roles to `deepseek-v4-pro`) and, within each model, by the direction of the call, input or output. The calls were counted from the prompts and replies the runs logged.

Table 8: Token usage of one complete SM2-AHD training cycle per problem class, averaged over its three repeats, listed by the model that served each call and, within each model, by the direction of the call (input or output), together with the cost of that cycle, at the off-peak prices in force when the runs were made. The last column is the extra cost of running the two reflective roles on `deepseek-v4-pro` instead of `deepseek-flash`, that is $(A - B)/B$ with A and B as defined in the text below.

Problem class	`flash`		`pro`		**Cost**	**Extra cost**
	in	out	in	out		
TSP-cons	3,784,252	1,763,643	366,265	8,286	\$2.275	$+8.26\%$
KP-cons	2,627,968	978,300	192,767	3,473	\$1.370	$+7.05\%$
TSP-ACO	4,206,362	2,280,428	517,568	12,803	\$2.823	$+9.58\%$
CVRP-ACO	3,986,992	2,997,547	448,599	10,337	\$3.201	$+7.13\%$
MKP-ACO	4,035,277	2,362,455	642,458	11,304	\$2.920	$+11.46\%$
BPP-ACO	4,035,802	1,886,433	481,616	8,706	\$2.490	$+9.95\%$
Total	**22,676,653**	**12,268,806**	**2,649,274**	**54,910**	**\$15.079**	$\mathbf{+9.03\%}$

The extra cost of the second model. Let A be what the runs actually cost, computed from their token usage, and let B be what the same token usage would have cost had every stage run on `deepseek-flash`. The extra cost of the second model is then

$$\frac{A - B}{B} \times 100\%, \tag{18}$$

and over the six problem classes it comes to 9.03%: one complete cycle of each of the six classes costs \$15.079 in all, where running every stage on `deepseek-flash` would cost \$13.830. The

prices are those in force when the runs were made, at the off-peak rate: \$0.222 and \$0.666 per million input and output tokens for `deepseek-flash`, and three times that, \$0.666 and \$1.998, for `deepseek-v4-pro`, which accounts for \$1.874 of that bill. Pro supplies only 7.18% of the tokens, but because it is more expensive, it makes up 12.43% of the total bill.

Two remarks on how much that 9.03% can be trusted.

(i) It is an estimate, not a measurement. Language-model billing distinguishes cached from uncached input, and at the prices above the uncached rate is much the higher of the two. Whether and how often our calls hit the cache is not recorded in our logs, so we took the simplest consistent reading and priced every token at the *uncached* rate. In real billing a large share of tokens are cache hits, and those are billed at a small fraction of the uncached rate, a few per cent of it; the estimates above therefore overstate the expenditure actually incurred.

(ii) Analysing that error further, we believe the extra cost actually incurred is a smaller share. As the table above shows, `deepseek-flash` carries the core evolution of the code, and its cost comes mainly from *output* tokens; `deepseek-v4-pro` provides reflection, and its cost comes from *input* tokens. Under the prices above, flash's output price equals pro's input price, \$0.666 per million tokens, and the input side is exactly the side that can be served from cache; so the fall in what `deepseek-flash` spends will be far smaller than the fall in what `deepseek-v4-pro` spends, and the overhead actually incurred should be smaller than the 9.03% estimated above.