HARNESSING LLMS WITH DIRECTION-SCOPE CONTROL FOR COMBINATORIAL SOLVER REFINEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Harnesses are increasingly unlocking the potential of large language model (LLM) agents by making their actions more controllable. In automatic heuristic design (AHD) for combinatorial optimization (CO), LLMs iteratively revise solvers using execution and verification feedback. However, these revisions often break previously validated solver behavior, making refinement unstable and costly. We characterize this pattern as *feasibility oscillation* and document it across CO benchmarks, AHD methods, and backbone LLMs. Tracing this oscillation, our analysis reveals that feasibility degradation often follows large solver revisions, while substantial feedback redundancy makes it harder to identify which failures deserve attention. Motivated by these findings, we introduce DiSH, a harness that jointly controls revision direction and scope. It selects representative failure evidence to guide each update and progressively narrows the revision scope as feasible behavior accumulates. Across nine CO benchmarks, DiSH consistently improves QYI scores and reduces feasibility oscillation, while roughly halving feedback token consumption relative to prior methods that use full feedback. These results establish DiSH as a strong baseline for solver refinement under harness control.

1 INTRODUCTION

Automatic heuristic design seeks to automate development for combinatorial optimization algorithms. Traditional methods search over heuristics predefined by human experts (Burke et al., 2013; Pillay & Qu, 2018), while large language models enable solvers to be generated and revised through iterative feedback (Romera-Paredes et al., 2024; Liu et al., 2024; Chen et al., 2026d). This execution-and-feedback loop naturally aligns with the harness, which has become a foundation for building LLM agents (Sumers et al., 2024; Yang et al., 2024; Wang et al., 2025a). Generally, a harness is the system surrounding an LLM that structures its observations and actions, executes its decisions, and returns results. In LLM-based AHD, it therefore determines how execution and verification evidence informs subsequent solver revisions.

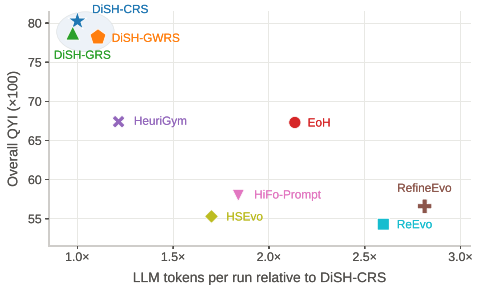


Figure 1: Overall QYI versus token cost.

Despite this central role, the harness-like components of existing LLM-based AHD methods remain largely limited to executing and verifying candidate solvers and returning evaluation feedback to the LLM (Romera-Paredes et al., 2024; Liu et al., 2024; Ye et al., 2024; Chen et al., 2026d). This is particularly problematic for CO solvers, whose constraints, decisions, and long-horizon logic are tightly coupled. A reported execution error or constraint violation identifies the symptom of a failure, but rarely specifies its root cause in the program or the extent of revision required to correct it. The LLM therefore decides which failures deserve attention and how extensively the solver should be modified. Existing methods expose what went wrong, but leave how to revise largely implicit in model generation. Recent methods regulate the search more finely, but they typically couple revision scope to search operators rather than constraining it independently before generation (Zheng et al., 2025; Chen et al., 2026a;b; Wu et al., 2026), while revision priorities largely depend on the model's interpretation of richer feedback (Yu et al., 2026; Cuong Van Duc et al., 2026; Gungordu et al., 2026; Lv et al., 2026). We discuss these lines of work in detail in Appendix A.

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.

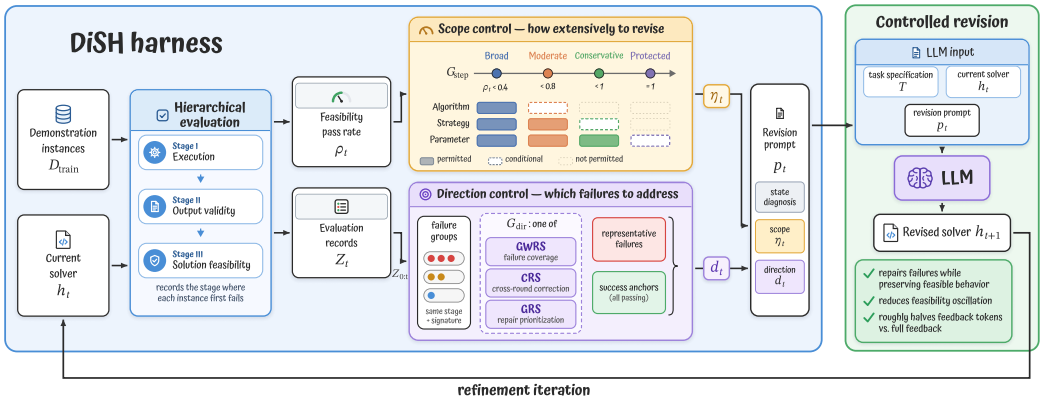


Figure 2: Overview of the DiSH framework. DiSH converts evaluation feedback into two complementary controls over what to revise and how extensively to revise it. Scope control adjusts the permitted revision magnitude, while direction control selects representative failures and successful instances to focus the revision. Together, they form the revision prompt that guides the LLM to repair remaining failures while preserving previously feasible behavior across iterations.

This leaves LLM-based AHD with a central challenge: *iterative revision does not reliably and efficiently accumulate improvement across rounds.* A revision may repair current failures while disrupting verified behavior, forcing later rounds to recover lost progress. We characterize this repeated loss and recovery as *feasibility oscillation*, and find it consistently across CO benchmarks, AHD methods, and backbone LLMs, indicating that it is a systemic rather than an isolated issue. Our analysis links feasibility loss to large changes between consecutive solvers and reveals substantial redundancy in the feedback guiding those changes, which obscures repair priorities. These findings motivate joint control of revision scope, which limits how extensively the solver may change, and revision direction, which determines which failures guide the update.

Guided by this analysis, we introduce DiSH, a harness framework designed to mitigate feasibility oscillation through joint control of revision direction and scope. As illustrated in Figure 2, DiSH first organizes execution and verification outcomes through hierarchical evaluation, preserving both instance-level failure evidence and the aggregate feasibility state. Direction control distills redundant records into representative repair evidence, making the target of each revision explicit. Scope control progressively restricts the permitted modification as feasible behavior accumulates, reducing the risk that a new revision disrupts verified behavior. Together, these components turn raw evaluation outcomes into revision instructions, guiding iterative revision toward cumulative refinement.

Figure 1 summarizes the performance and efficiency comparison. Across nine CO benchmarks, the three DiSH variants achieve higher average QYI while consuming fewer LLM tokens than existing AHD methods. Token-accounting details and the per-backbone search-cost breakdown are provided in Appendix E.3. DiSH also consistently reduces feasibility oscillation and roughly halves feedback-token usage relative to full feedback. Further analysis shows that narrower scope instructions lead to smaller realized revisions, while compact direction evidence preserves repair effectiveness. These results establish DiSH as a strong baseline for harness-controlled solver refinement.

Our main contributions are summarized as follows:

- We identify feasibility oscillation as a central obstacle to cumulative solver refinement and systematically characterize it across CO benchmarks, AHD methods, and backbone LLMs. Our analysis associates feasibility loss with large solver revisions and reveals substantial redundancy in the feedback used to guide these revisions.
- *We introduce DiSH, the first AHD harness to our knowledge to demonstrate stable and cumulative improvement across revisions*. DiSH jointly controls revision direction and scope by distilling repeated failures into representative repair evidence and progressively restricting solver modifications as feasible behavior accumulates.
- We evaluate DiSH across nine CO benchmarks and show consistent improvements in QYI and refinement stability. DiSH reduces feasibility oscillation and roughly halves feedback-token usage relative to full-feedback refinement, while preserving repair effectiveness and generalizing across direction strategies and backbone LLMs.

2 Problem Formulation

LLM-based automatic heuristic design (AHD) commonly improves solvers through iterative code revision (Liu et al., 2024; Ye et al., 2024; Chen et al., 2026d; Wu et al., 2026). Given a task specification $\mathcal{T}$ and demonstration instances $\mathcal{D}_{\text{train}}$, the current solver h_t is executed and evaluated on these instances. The resulting feedback F_t guides the next revision:

$$h_{t+1} \sim \mathcal{M}_{\text{LLM}}\left(\cdot \mid \mathcal{T}, h_t, \mathrm{F}_t\right), \tag{1}$$

where $\mathcal{M}_{\text{LLM}}$ denotes the conditional distribution over solver programs induced by the LLM. Applying (1) for K steps yields a trajectory $h_0, h_1, \ldots, h_K$. Define the feasibility pass rate of h_t as

$$\rho_t = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{x \in \mathcal{D}_{\text{train}}} \mathbb{I}[h_t \text{ produces a feasible solution for } x]\,. \tag{2}$$

Feasibility Oscillation. A revision may repair current failures while making previously feasible instances fail. Repeated losses and recoveries of feasibility lead to *feasibility oscillation*. A decrease in ρ_t indicates a net loss of feasibility, but individual regressions can also occur when the pass rate increases. Figure 3 illustrates this distinction. The first revision repairs I_2 and I_5 but makes I_1 infeasible, raising the pass rate from $2/5$ to $3/5$ despite this regression. The next revision restores I_1 and repairs I_3, reaching full feasibility. A further revision, however, makes I_4 infeasible and lowers the pass rate to $4/5$. Such regressions consume additional revision budget and hinder stable progress, motivating a closer analysis of the underlying solver revisions.

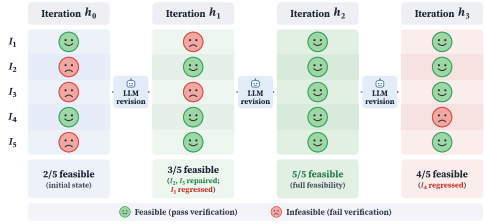


Figure 3: Instance-level feasibility oscillation.

Revision Distance as a Lens on Feasibility Oscillation. We measure the distance between consecutive solvers to examine how the revisions relate to feasibility oscillation. Let $T(h)$ denote the set of contiguous token sequences extracted from solver h. We use their Jaccard distance as an empirical measure of revision distance (Appendix D.1):

$$\Delta(h_{t+1}, h_t) = 1 - |T(h_t) \cap T(h_{t+1})| / |T(h_t) \cup T(h_{t+1})|. \tag{3}$$

Smaller distances indicate greater code retention, while larger distances indicate more extensive changes. Figure 4 compares trajectories from HeuriGym (Chen et al., 2026d), RefineEvo (Wu et al., 2026), and DiSH. The curves show feasible instance counts, and triangles show revision distances $\Delta(h_{t+1}, h_t)$. HeuriGym and RefineEvo drop from full feasibility to $0/5$ after substantial revisions with distances of 0.56 and 0.83, respectively. In contrast, DiSH maintains full feasibility once reached, with subsequent distances reported as 0.00. These examples suggest that extensive revisions can significantly disrupt established feasibility, leading to unexpected oscillation.

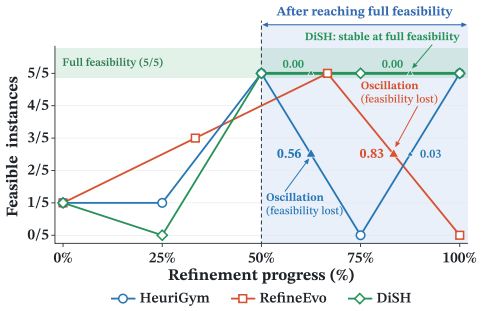


Figure 4: Revision distance and oscillation.

3 DiSH: Direction-and-Scope Harness for CO Solver Refinement

Combinatorial optimization (CO) solvers often involve sequences of interdependent decisions. As discussed in Section 2, revising their code can disrupt feasible behavior instead of improving it. We introduce DiSH, a harness that explicitly controls what to revise and how extensively to change the solver. It aims to stabilize feasibility gains while using feedback tokens efficiently. We first describe the controlled revision loop (Section 3.1). We then present hierarchical evaluation (Section 3.2), adaptive revision scope (Section 3.3), and revision-direction construction (Section 3.4).

3.1 From Iterative Revision to Harness-Controlled Solver Refinement

We turn evaluation feedback into explicit controls over the direction and scope of solver revisions.

Why Revisions Need Control. Feasibility oscillation highlights that two issues limit stable refinement. First, a single LLM call may not resolve failures involving closely linked constraints, decisions, and solver components. Second, repeating revisions can also undo earlier repairs. Thus, *revision* changes the solver, whereas *refinement* should accumulate feasible behavior across rounds.

Given h_t, DiSH produces per-instance evaluation records $\mathcal{Z}_t$ and the feasibility pass rate ρ_t. It then constructs a revision direction d_t, a scope instruction η_t, and a prompt p_t for the next LLM revision:

$$\begin{aligned} \text{Evaluation: } & (\mathcal{Z}_t, \rho_t) = \mathcal{E}(h_t, \mathcal{D}_{\text{train}}), \\ \text{Construction: } & d_t = \mathcal{G}_{\text{dir}}(\mathcal{Z}_{0:t}), \quad \eta_t = \mathcal{G}_{\text{step}}(\rho_t), \\ \text{Generation: } & p_t = \mathcal{H}(\mathcal{Z}_t, d_t, \eta_t), \quad h_{t+1} \sim \mathcal{M}_{\text{LLM}}(\cdot \mid \mathcal{T}, h_t, p_t). \end{aligned} \tag{4}$$

Here, $\mathcal{E}$ evaluates instances in successive stages. $\mathcal{G}_{\text{dir}}$ selects and organizes revision targets from current and historical evaluation evidence, while $\mathcal{G}_{\text{step}}$ determines the permitted scope from ρ_t. $\mathcal{H}$ combines the evaluation records and these controls into revision instructions p_t, which is then used to generate the next solver h_{t+1}. Appendix C provides the complete prompt structure and examples.

Rather than exposing all observed failure records directly to the LLM, direction control prioritizes them into representative repair evidence while retaining previously verified behavior. Scope control permits less extensive changes at higher pass rates, encouraging smaller revision distances $\Delta(h_{t+1}, h_t)$. Together, these controls aim to reduce regressions and support stable feasibility gains within the finite revision budget. Algorithm 1 in Appendix B gives the complete refinement loop.

3.2 Hierarchical Feasibility Evaluation and Feedback

We evaluate each instance in three stages to identify where it fails and provide feedback for revision.

From Outcome Evaluation to Failure Localization. Existing AHD pipelines mainly use execution and verification outcomes to score solver candidates (Romera-Paredes et al., 2024; Liu et al., 2024; Ye et al., 2024) or provide feedback (Chen et al., 2026d), without explicitly localizing failures for revision control. However, execution errors, invalid outputs, and constraint violations arise from various solving stages and require different repairs. DiSH therefore constructs a hierarchical evaluation pipeline that records both the failure stage and the associated diagnostic information.

Stage-Wise Instance Evaluation. To make evaluation feedback more diagnostic, DiSH applies three checks in order and records the first failed stage rather than reducing its outcome to a single score. For each instance $x \in \mathcal{D}_{\text{train}}$, Stage I checks execution within time and memory limits. Stage II checks output validity, and Stage III checks satisfaction of problem constraints. For example, a routing solver may time out on one instance but return a correctly formatted route that serves a customer twice on another. These failures call for different repairs: improving execution efficiency and correcting solution construction, respectively. Appendix B.2.1 provides the detailed criteria.

Hierarchical Evaluation Records. The resulting record is $z_t(x) = (s_t(x), e_t(x), q_t(x))$, where $s_t(x)$ denotes the terminal evaluation stage, $e_t(x)$ contains the corresponding error information, and $q_t(x)$ is the objective value of a feasible solution. Depending on the terminal stage, $e_t(x)$ may contain an exception, a parsing error, or a constraint violation. Failure records have no objective value, while successful records have no error information. The records from all instances are collected as $\mathcal{Z}_t = \{z_t(x)\}_{x \in \mathcal{D}_{\text{train}}}$. The hierarchical records support revision control at two levels. Aggregated across instances, they yield the feasibility pass rate ρ_t defined in Eq. (2), which informs the permitted revision scope. At the instance level, failure stages and diagnostics provide repair evidence, while successful records identify behavior to preserve.

3.3 Feasibility-Adaptive Revision Scope

We adapt the permitted revision scope to the current pass rate, allowing less extensive changes as more instances become feasible.

Conditioning Scope on Feasibility. Failure evidence indicates what requires repair, but not how broadly the solver should be revised. At low ρ_t, fundamental failures may require substantial revisions. At high ρ_t, such revisions risk disrupting verified behavior, as shown in Figure 4. We therefore use $\eta_t = \mathcal{G}_{\text{step}}(\rho_t)$ to progressively narrow the permitted revision scope as feasibility increases.

Semantic Revision Levels and Modes. To encourage smaller revision distances as feasibility improves, DiSH uses a prompt instruction η_t to specify the permitted semantic scope. We distinguish three revision levels. *Parameter* revisions adjust numerical constants without changing solver logic. *Strategy* revisions modify decision rules or local procedures. *Algorithm* revisions replace the core solving approach. The scope rule $\eta_t = \mathcal{G}_{\text{step}}(\rho_t)$ selects among four modes with progressively tighter restrictions. BROAD permits all three revision levels. MODERATE favors strategy revisions and permits algorithm replacement only when failures indicate a structural mismatch. CONSERVATIVE allows parameter changes and narrowly targeted strategy revisions but prohibits algorithm replacement. At full feasibility, PROTECTED instructs the LLM to preserve the solver by default, permitting only beneficial parameter changes that do not compromise currently feasible instances. Thus, the permitted scope narrows as ρ_t increases. Appendix B.2.2 details the mode thresholds and permitted revision levels, and Appendix C.4 provides the corresponding prompts.

Realized Revision Distance. To examine whether scope instructions lead to corresponding changes in realized solver modification, we provide *DiSH (w/o Dir)* as a controlled comparison, which retains full feedback and enables scope control. In Figure 5, lines show mean revision distances, shaded ribbons indicate 95% bootstrap confidence intervals, and background bands denote scope modes. As ρ_t increases, the realized revision distance of *DiSH (w/o Dir)* progressively contracts, whereas HeuriGym and RefineEvo do not exhibit the same trend. This pattern indicates that the revision scope specified by $\mathcal{G}_{\text{step}}$ before generation is reflected in progressively smaller realized revisions as feasibility increases.

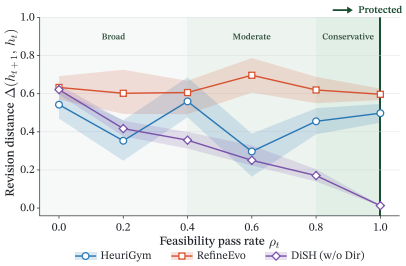


Figure 5: Revision distance vs. ρ_t.

3.4 REVISION-DIRECTION CONSTRUCTION

We build revision directions from representative failures and successful records, reducing repeated feedback while preserving useful repair guidance.

Distilling Evaluation Feedback into Revision Direction. Revision scope limits how extensively the solver may change, but a successful revision also depends on how the LLM uses evaluation feedback to decide what to change. The main obstacle is redundancy: many instance-level records describe the same failure, consuming tokens and making distinct repair needs harder to identify. Figure 6 illustrates this redundancy in a refinement example. Previously feasible instances I_1 and I_2 now fail with the same `IndexError` as I_3. The number of failure records increases from three to five, while the number of distinct failure groups remains three. To further quantify feedback redundancy, we analyze 69,890 failure records collected across revision rounds from five methods on nine benchmarks. Of these records, 95.6% share a failure pattern with another instance in the same evaluation. In 88.7% of solver evaluations, at least two instances exhibit the same failure pattern. These results motivate grouping related failures and distilling representative evidence instead of passing all records to the LLM.

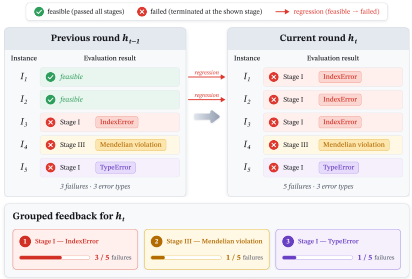


Figure 6: Failure redundancy.

Shared Failure-Group Representation. Let $\mathcal{P}_t$ contain the instances that pass all evaluation stages and let $\mathcal{F}_t = \mathcal{D}_{\text{train}} \setminus \mathcal{P}_t$ denote the set of failed instances. We group failed instances by their terminal stage and normalized error signature:

$$\mathcal{G}_t = \{G_t^{s,a} : G_t^{s,a} \neq \varnothing\}, \qquad \text{where } G_t^{s,a} = \{x \in \mathcal{F}_t : s_t(x) = s,\ \tau(e_t(x)) = a\}. \tag{5}$$

Here, τ maps the diagnostic information $e_t(x)$ to an error category a. Each group $G_t^{s,a}$ contains instances with the same failure stage s and error category a, and $\mathcal{G}_t$ collects all nonempty failure groups. Across the collected 69,890 failure records, normalization yields 53 distinct error signatures a. Figures 13 and 14 in Appendix F show their frequency distribution and representative normalized patterns, respectively. Instances within the same group therefore exhibit similar failure patterns and are expected to require similar revisions. A direction strategy selects a representative set $\mathcal{R}_t \subseteq \mathcal{F}_t$ from these groups. The records of $\mathcal{R}_t$, together with all successful records in $\mathcal{P}_t$, form the main

Table 1: QYI (%) across nine CO benchmark problems. The best, second, and third results in each column are shown in bold, underlined, and marked with ⋆, respectively.

Method	Operator	Intra	Protein	Pedigree	Crew	E-Graph	BPP	CVRP	TSP	Avg.
EoH	84.2	69.7	62.7	<u>62.0</u>	49.9	41.6	**99.7**	85.9	49.6	67.3
ReEvo	65.6	56.5	55.7	53.5	20.8	26.7	94.2	76.7	38.9	54.3
HSEvo	64.7	56.9	63.9	39.9	21.9	39.1	<u>99.5</u>	78.0	33.9	55.3
HiFo-Prompt	78.6	56.8	60.5	**64.3**	25.2	30.5	98.6⋆	72.4	35.3	58.0
RefineEvo	66.3	59.2	50.5	59.8⋆	35.1	41.1	87.2	74.5	35.7	56.6
HeuriGym	91.5⋆	68.4	73.7	44.4	40.7	51.1	**99.7**	85.5	52.0	67.4
DiSH-GWRS	**93.8**	69.9⋆	**81.7**	57.0	<u>67.3</u>	56.0⋆	**99.7**	<u>93.0</u>	<u>85.5</u>	78.2⋆
DiSH-CRS	<u>93.6</u>	<u>71.2</u>	<u>81.5</u>	57.7	65.5⋆	**73.2**	**99.7**	**93.6**	**86.9**	**80.3**
DiSH-GRS	90.5	**72.2**	80.7⋆	58.0	**69.4**	<u>63.0</u>	**99.7**	92.7⋆	83.6⋆	<u>78.9</u>

content of d_t. We instantiate revision direction from three alternative perspectives that emphasize failure coverage, cross-round correction, and repair prioritization, respectively.

Alternative Direction Constructions. The direction constructor is chosen from three alternatives: $\mathcal{G}_{\text{dir}} \in \{\mathcal{G}_{\text{GWRS}}, \mathcal{G}_{\text{CRS}}, \mathcal{G}_{\text{GRS}}\}$. Each independently constructs d_t from representative failures and successful records. As shown in Figure 7, they use 42.3–51.8% of the tokens required by full feedback while preserving the measured rate at which failure modes are repaired in the next revision. Thus, selecting one direction strategy is sufficient to substantially reduce feedback without diminishing the measured repair effectiveness.

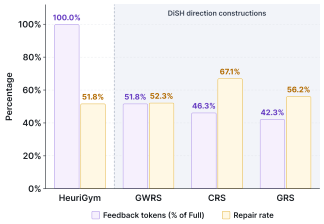


Figure 7: Feedback efficiency.

- **Group-Wise Representative Sampling (GWRS)** selects one representative from every current failure group. For each $G \in \mathcal{G}_t$, it chooses the largest instance, $r_G = \arg\max_{x \in G} n(x)$, where $n(x)$ is the input size, with ties broken by instance identifier. This rule preserves group coverage without repeating similar errors. The largest instance is chosen to expose the failure on a structurally richer input. Groups are reconstructed after each evaluation: repaired instances provide examples of behavior to preserve, while unresolved failures remain available for later revisions.
- **Correction Representative Sampling (CRS)** uses consecutive evaluations to relate current failures to the preceding revision. It selects the smallest instance $r_G = \arg\min_{x \in G} n(x)$ from each failure group, reducing incidental input complexity and making the remaining error easier to isolate. CRS additionally identifies the regression set $\mathcal{U}_t = \mathcal{P}_{t-1} \cap \mathcal{F}_t$, containing instances that were previously feasible but now fail. This comparison also annotates current failures as regressed, persistent, or changed, allowing d_t to emphasize behavior disrupted or left unresolved by the previous revision.
- **Scope-Aware Group Representative Sampling (GRS)** concentrates direction evidence on high-priority failure groups. It identifies a focal stage from the stage-level failure distribution and ranks the groups by stage priority and coverage $c_t(G) := |G|/|\mathcal{F}_t|$. It then retains the top $k(\rho_t)$ groups, where fewer groups are retained as feasibility increases, and selects the largest instance from each retained group. Consequently, low-feasibility states expose a broader range of repair needs, whereas high-feasibility states focus the revision on fewer, higher-priority targets.

4 Experiments

In this section, we conduct comprehensive comparison experiments and ablation studies to empirically evaluate the practical effectiveness of DiSH.

4.1 Experimental Setup

We evaluate DiSH-GWRS, DiSH-CRS, and DiSH-GRS on nine problems: six from HeuriGym (Chen et al., 2026d), plus bin packing (BPP), capacitated vehicle routing (CVRP), and traveling salesman (TSP) from BPPLIB (Delorme et al., 2018), CVRPLIB Set X (Uchoa et al., 2017), and TSPLIB (Reinelt, 1991), respectively. Demonstration instances guide refinement, while held-out instances are reserved for final evaluation. Baselines include EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), HSEvo (Dat et al., 2025), HiFo-Prompt (Chen et al., 2026b), RefineEvo (Wu et al., 2026), and the original HeuriGym refinement procedure. All methods share a complete-solver setting with Qwen3.5-122B-A10B (Qwen Team, 2026), DeepSeek-V3 (DeepSeek-AI, 2024), GPT-4.1-

Table 2: Overall QYI (%) across backbone models. The best, second, and third results in each column are shown in bold, underlined, and marked with ⋆, respectively.

Method	Qwen3.5-122B-A10B	DeepSeek-V3	GPT-4.1-mini	Gemini-2.5-Flash	Avg.
EoH	67.6 ± 1.5	57.3 ± 6.2	68.8 ± 4.7	75.2 ± 6.8	67.3
ReEvo	46.5 ± 15.3	43.4 ± 8.3	56.8 ± 8.9	70.3 ± 6.0	54.3
HSEvo	50.0 ± 11.9	44.4 ± 5.5	54.5 ± 7.1	72.4 ± 2.9	55.3
HiFo-Prompt	60.9 ± 1.8	44.3 ± 5.4	55.6 ± 3.2	71.2 ± 6.0	58.0
RefineEvo	42.1 ± 11.2	41.6 ± 4.1	67.8 ± 5.6	74.9 ± 5.7	56.6
HeuriGym	64.1 ± 5.3	62.8 ± 12.6	66.3 ± 11.2	76.6 ± 9.9	67.4
DiSH-GWRS	78.1 ± 9.0	75.9 ± 8.3⋆	77.9 ± 7.1⋆	**80.9 ± 5.7**	78.2⋆
DiSH-CRS	77.6 ± 7.5⋆	**80.6 ± 10.4**	**82.7 ± 4.9**	80.4 ± 7.7	**80.3**
DiSH-GRS	**80.1 ± 7.5**	76.6 ± 9.8	80.8 ± 8.6	77.9 ± 10.4⋆	78.9

mini (OpenAI, 2025), and Gemini-2.5-Flash (Comanici et al., 2025) as backbones. The five AHD baselines retain their native search operators and procedures. Following HeuriGym, we use the Quality-Yield Index (QYI) as the primary metric and report Quality and Yield separately. Quality uses verified subsets that may differ across methods. Section 4.3 additionally reports Convergence Rate, Survival Rate, and Tokens to Convergence on demonstration instances. Benchmark details, metric definitions, and baseline adaptations appear in Appendices E.1, D, and E.2, respectively.

4.2 Comparative Results

Per-problem performance. Across three variants, DiSH achieves the highest QYI on seven of nine problems and ties for best on BPP (Table 1). The largest advantages occur on TSP, E-Graph, and Crew Pairing, where the best DiSH variants achieve 86.9, 73.2, and 69.4, respectively. The strongest evaluated baselines on these problems reach 52.0, 51.1, and 49.9. DiSH also leads on Protein Design (81.7 vs. 73.7), CVRP (93.6 vs. 85.9), Operator Scheduling (93.8 vs. 91.5), and Intra (72.2 vs. 69.7). On BPP, all three DiSH variants tie with EoH and HeuriGym for the highest QYI of 99.7. Pedigree remains the exception, with HiFo-Prompt achieving 64.3 QYI compared with 58.0 for the best DiSH variant. Appendix G.1 further examines Quality and Yield on this benchmark through a complementary comparison with EoH. Overall, the performance advantage extends across most evaluated problems, although the leading DiSH variant varies by task.

Overall performance. Table 2 shows that all three DiSH variants outperform every evaluated baseline in mean QYI under each of the four backbones. DiSH-GRS leads on Qwen3.5-122B-A10B (80.1), while DiSH-CRS ranks first on DeepSeek-V3 (80.6) and GPT-4.1-mini (82.7). DiSH-GWRS leads on Gemini-2.5-Flash (80.9). DiSH-CRS also achieves the highest overall average QYI of 80.3, followed by DiSH-GRS (78.9) and DiSH-GWRS (78.2). Among the baselines, HeuriGym achieves the highest overall average of 67.4, closely followed by EoH (67.3). Overall, the performance advantage extends across all evaluated backbones, although the leading DiSH variant varies with the model. Further analysis in Appendix G.1 shows that these QYI gains are accompanied by aggregate improvements in both Quality and Yield for all three DiSH variants. Appendix G.2 examines how their gains over HeuriGym vary across problems and backbones. Results across shared initializations for all three variants are reported in Appendix G.3.

4.3 Effects of Direction and Scope

Table 3: Ablation of direction and scope control.

Method	QYI (%) ↑	Conv. (%) ↑	Survival (%) ↑	Tokens to Conv. ↓
HeuriGym	67.4	55.2	40.0	19,897
DiSH (w/o Sco)	69.0	55.6	46.6	16,335
DiSH (w/o Dir)	76.4	91.4	92.3⋆	14,804
DiSH-GWRS	78.2⋆	88.3	93.0	11,270⋆
DiSH-CRS	**80.3**	**92.9**	**94.2**	**10,271**
DiSH-GRS	78.9	91.4⋆	91.9	10,868

Table 3 shows larger gains in convergence and survival from scope control, while combining scope and direction further improves QYI and reduces Tokens to Convergence. We conduct a controlled 2×2 ablation with all other settings fixed. HeuriGym enables neither control; DiSH (w/o Sco) uses GWRS feedback alone; DiSH (w/o Dir) retains full feedback and enables scope control; and DiSH-GWRS combines both. DiSH-CRS and DiSH-GRS additionally test alternative direction strategies with both controls enabled. Evaluation includes held-out test QYI and three diagnostics on demonstration instances, defined in Appendix D.3. Compared with HeuriGym, DiSH (w/o Dir) raises Convergence Rate from 55.2% to 91.4% and Survival Rate from 40.0% to 92.3%. DiSH (w/o Sco) yields smaller

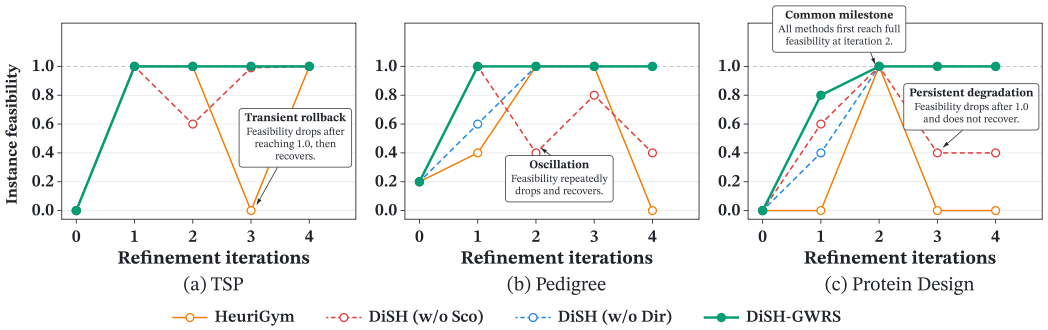


Figure 8: Representative feasibility trajectories on TSP, Pedigree, and Protein Design.

increases to 55.6% and 46.6%, respectively. Both configurations improve QYI and reduce Tokens to Convergence relative to HeuriGym. All three combined variants further improve these two metrics over either control alone, although convergence and survival do not improve uniformly. Relative to (DiSH (w/o Dir)), DiSH-GWRS has a lower Convergence Rate (88.3% versus 91.4%), and DiSH-GRS has a slightly lower Survival Rate (91.9% versus 92.3%). Overall, joint control improves final performance and token efficiency, with DiSH-CRS achieving the best results across all four metrics.

Feasibility stability across refinement. Representative trajectories in Figure 8 illustrate the stabilizing effect of adaptive scope control. On TSP, HeuriGym and DiSH (w/o Sco) temporarily lose feasibility before recovering. On Pedigree, DiSH (w/o Sco) oscillates, while HeuriGym drops from full feasibility to zero in the final iteration. On Protein Design, all methods first reach full feasibility at iteration 2, but HeuriGym and DiSH (w/o Sco) subsequently regress without recovery. In contrast, DiSH (w/o Dir) and DiSH-GWRS maintain full feasibility once reached in all three examples, with their curves overlapping on TSP. These patterns align with their higher Survival Rates in Table 3, suggesting that adaptive scope control based on feasibility helps reduce regressions.

4.4 Controlled Mechanism Analysis

Adaptive versus fixed revision scopes. We compare adaptive and fixed revision scopes to test whether the observed gains can be explained by a single fixed revision scope. The only difference is the scope policy: Adaptive selects the revision scope according to the current feasibility state, whereas the fixed baselines use Broad, Moderate, or Conservative scope throughout refinement. Figure 9 reports the QYI distribution and mean for each policy. Adaptive achieves the highest average QYI (72.8%), outperforming Broad (69.1%), Moderate (65.9%), and Conservative (65.4%), while also exhibiting lower variability than the fixed-scope baselines. These results suggest that the benefit comes not from any single fixed scope, but from adapting revision scope to the current feasibility state.

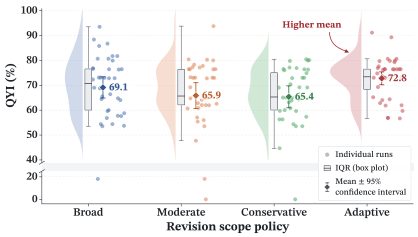


Figure 9: Adaptive vs. fixed scopes.

Structured revision direction. We compare structured direction construction with full feedback and random selection to distinguish evidence organization from feedback reduction alone. Random selection uses the same evidence budget as the corresponding structured strategy. Adaptive scope control is disabled, while revision scope and all other settings remain fixed. Figure 10 reports QYI changes relative to full feedback. Structured direction construction achieves gains of 5.1%, 6.5%, and 5.2% for GWRS, CRS, and GRS, respectively. Random selection yields changes of −1.8%, +0.9%, and −0.5%, showing no consistent improvement. All three strategies outperform random selection with the same evidence budget. This suggests that the gains come from organizing failure evidence, not just reducing feedback.

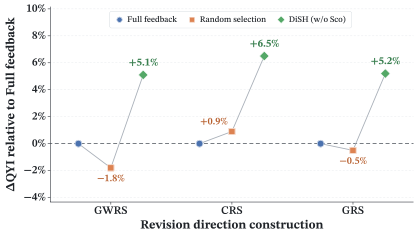


Figure 10: Evidence organization.

Joint Effects of Revision Direction and Scope. Across all three direction strategies, narrower semantic scopes are associated with smaller mean revision distances. Figure 11 shows a consistent decrease from Broad through Moderate and Conservative to Protected. Mean distances

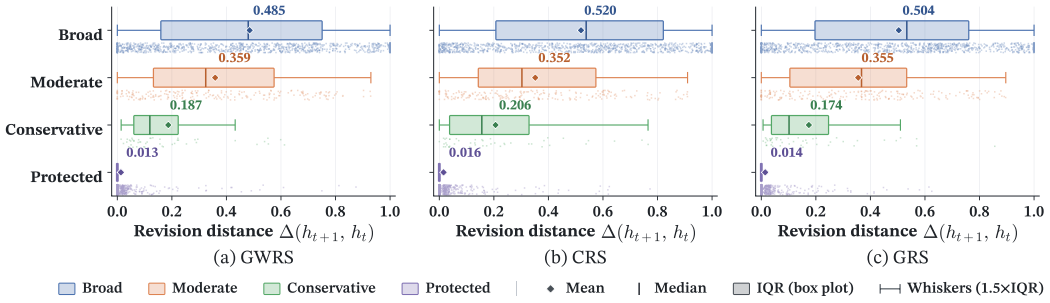


Figure 11: Revision distance by semantic scope.

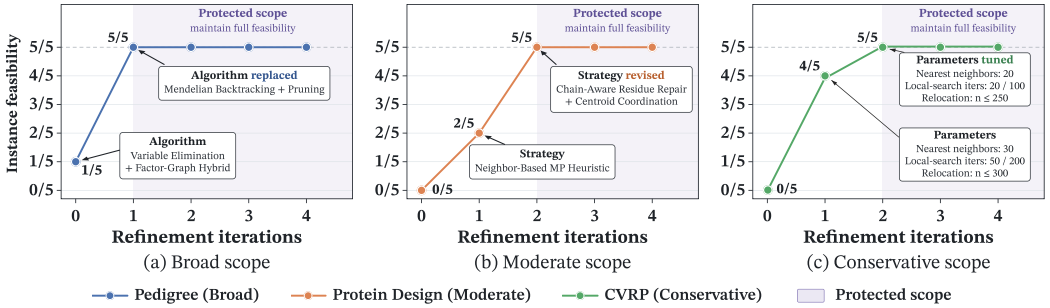


Figure 12: Representative solver revisions under different semantic revision scopes.

range from 0.485 to 0.520 under BROAD and approach zero under PROTECTED (0.013 to 0.016). These results indicate that scope guidance before generation is reflected in progressively smaller code changes across different direction constructions. This consistent pattern across direction strategies provides empirical support for the scope-control mechanism described in Section 3.3.

4.5 CASE STUDIES

Figure 12 presents three representative revisions from DiSH runs, illustrating how different scope modes guide solver modification. On Pedigree, only one of five demonstration instances passes, so BROAD mode permits aggressive changes to the core algorithm. The revision replaces the variable elimination and factor graph hybrid with Mendelian backtracking and pruning, raising feasibility from $1/5$ to $5/5$. On Protein Design, MODERATE mode applies at $2/5$ feasibility. The revision introduces chain-aware residue repair and centroid coordination, raising demonstration feasibility from $2/5$ to $5/5$. On CVRP, four of five instances already pass, and the CONSERVATIVE revision makes targeted parameter adjustments. It reduces the neighbor count, local search iterations, and relocation threshold without changing the routing or search logic, raising feasibility from $4/5$ to $5/5$. The purple regions mark PROTECTED mode, entered after full feasibility is reached. This mode preserves the solver by default and permits only parameter changes with a concrete benefit and no adverse effect on currently feasible instances. All three runs maintain full feasibility throughout the remaining iterations. Together, these cases illustrate how DiSH adapts revision scope to feasibility, allowing substantial repairs when needed and prioritizing preservation as feasibility improves.

5 CONCLUSION

This work shows that stable LLM-based refinement of combinatorial optimization solvers depends not only on the feedback itself, but also on how that feedback is translated into solver revisions. Existing approaches largely leave two key decisions to the LLM: *what to change* and *how much to change*. This can cause revisions to disrupt previously feasible behavior and lead to feasibility oscillation. DiSH makes these decisions explicit by controlling revision direction and scope: representative failure groups identify what should be addressed, while the current feasibility state determines how extensively the solver should be revised. Across nine combinatorial optimization problems and four backbone LLMs, DiSH achieves the highest overall average QYI, produces more stable refinement trajectories, and uses roughly half the feedback tokens required by full feedback. These results suggest that controlling the revision process itself provides another effective way to improve automatic heuristic design, complementing advances in search and solver generation.

AI USE STATEMENT

In this work, we used generative AI tools solely for language polishing, grammar checking, and discussions of experimental coverage. All AI-assisted content was reviewed by the authors, who take full responsibility for the final content of this work.

ETHICS STATEMENT

This work studies LLM-based solver refinement on combinatorial optimization benchmarks. The experiments focus on benchmark evaluation, and practical deployment would require further validation under application-specific requirements.

REPRODUCIBILITY STATEMENT

The method and refinement procedure are described in Section 3 and Appendix B. Appendix C provides the prompts and refinement examples. Metric definitions and experimental configurations are detailed in Appendices D and E, with additional results reported in Appendix G.

REFERENCES

Ali Ahmaditeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In *International Conference on Machine Learning*, 2024.

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

Edmund K. Burke, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward. A classification of hyper-heuristic approaches: Revisited. In Michel Gendreau and Jean-Yves Potvin (eds.), *Handbook of Metaheuristics*, pp. 453–477. Springer, 2019.

Bin Chen, Shouliang Zhu, Beidan Liu, Yong Zhao, Tianle Pu, Huichun Li, and Zhengqiu Zhu. A2DEPT: Large language model-driven automated algorithm design via evolutionary program trees. In *International Conference on Machine Learning*, 2026a.

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*, 2026b.

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

Hongzheng Chen, Yingheng Wang, Yaohui Cai, Hins Hu, Jiajie Li, Shirley Huang, Chenhui Deng, Rongjian Liang, Shufeng Kong, Haoxing Ren, Samitha Samaranayake, Carla P. Gomes, and Zhiru Zhang. HeuriGym: An agentic benchmark for LLM-crafted heuristics in combinatorial optimization. In *International Conference on Learning Representations*, 2026d.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed RL: Grounding large language models for authentic optimization modeling. In *Advances in Neural Information Processing Systems*, 2025.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games. In *Computer Games: 7th Workshop, CGW 2018, Held in Conjunction with IJCAI 2018, Revised Selected Papers*, pp. 41–75. Springer, 2019.

Cuong Van Duc, Minh Nguyen Dinh Tuan, Tam Vu Duc, Tung Vu Duy, Son Nguyen Van, Hanh Nguyen Thi, and Binh Huynh Thi Thanh. ReVEL: Multi-turn reflective LLM-guided heuristic evolution via structured performance feedback. *arXiv preprint arXiv:2604.04940*, 2026.

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 *AAAI Conference on Artificial Intelligence*, volume 39, pp. 26931–26938, 2025.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Maxence Delorme, Manuel Iori, and Silvano Martello. BPPLIB: A library for bin packing and cutting stock problems. *Optimization Letters*, 12(2):235–250, 2018.

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

Zhinan Hou, Xingchen Li, Yankai Zhang, Tianxun Li, and Keyou You. LLM4Branch: Large language model for discovering efficient branching policies of integer programs. In *International Conference on Machine Learning*, 2026.

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

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. ORLM: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 73(6):2986–3009, 2025.

Ziyao Huang, Weiwei Wu, Kui Wu, Wei-Bin Lee, and Jianping Wang. CALM: Co-evolution of algorithms and language model for automatic heuristic design. In *International Conference on Learning Representations*, 2026.

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

Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. ALE-Bench: A benchmark for long-horizon objective-driven algorithm engineering. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2025.

Xia Jiang, Yaoxin Wu, Minshuo Li, Zhiguang Cao, and Yingqian Zhang. Large language models as end-to-end combinatorial optimization solvers. In *Advances in Neural Information Processing Systems*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Yutao Lai, Kezhao Lai, Hai-Lin Liu, Yuping Wang, and Ping Guo. Es-ahd: An evolution strategy framework for automatic heuristic design. *arXiv preprint arXiv:2609.00023*, 2026.

Zhong Li, Hongliang Lu, Tao Wei, Yuxuan Chen, Wenyu Liu, Yuan Lan, Fan Zhang, and Zaiwen Wen. Constructing industrial-scale optimization modeling benchmark. In *International Conference on Machine Learning*, 2026.

Kuo Liang, Yuhang Lu, Jianming Mao, Shuyi Sun, Chunwei Yang, Congcong Zeng, Xiao Jin, Hanzhang Qin, Ruihao Zhu, and Chung-Piaw Teo. Large-scale optimization model auto-formulation: Harnessing LLM flexibility via structured workflow. *arXiv preprint arXiv:2601.09635*, 2026.

Fan Liu, Zherui Yang, Cancheng Liu, Tianrui Song, Xiaofeng Gao, and Hao Liu. MM-Agent: LLM as agents for real-world mathematical modeling problem. In *Advances in Neural Information Processing Systems*, volume 38, pp. 20881–20934, 2025a.

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 *International Conference on Machine Learning*, 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 *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), 2026b.

Haoyang Liu, Jie Wang, Yuyang Cai, Xiongwei Han, Yufei Kuang, and Jianye Hao. OptiTree: Hierarchical thoughts generation with tree search for LLM optimization modeling. In *Advances in Neural Information Processing Systems*, volume 38, pp. 120713–120781, 2025b.

Haoyang Liu, Yuyang Cai, Jie Wang, Xiongwei Han, Minyang Hu, Shuqi Liu, Mingxuan Yuan, Jianye Hao, and Feng Wu. Opt-Miner: Empowering information-seeking agent with tree-guided data synthesis for optimization modeling. In *International Conference on Machine Learning*, 2026c.

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

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. OptMATH: A scalable bidirectional data synthesis framework for optimization modeling. In *International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 40769–40802. PMLR, 2025.

Yidong Luo, Xuemin Chen, Chenguang Wang, Fangzhou Zhu, Tao Zhong, and Tianshu Yu. GRIMIP: A general framework for instance-specific configuration of MIP solvers using LLMs. *arXiv preprint arXiv:2606.23299*, 2026.

Haoze Lv, Ning Lu, Ziang Zhou, Yew-Soon Ong, and Shengcai Liu. AHD Agent: Agentic reinforcement learning for automatic heuristic design. *arXiv preprint arXiv:2605.08756*, 2026.

OpenAI. Introducing GPT-4.1 in the API. `https://openai.com/index/gpt-4-1/`, April 2025.

Nelishia Pillay and Rong Qu. *Hyper-Heuristics: Theory and Applications*. Springer, 2018.

Qwen Team. Qwen3.5: Towards native multimodal agents. `https://qwen.ai/blog?id=qwen3.5`, February 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.

Yifan Shi, Jiayi Wang, Minyi Wu, Ye Fan, Jialong Shi, and Jianyong Sun. MIRROR: A multi-agent framework with iterative adaptive revision and hierarchical retrieval for optimization modeling in operations research. *arXiv preprint arXiv:2602.03318*, 2026.

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

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024.

Weiwei Sun, Shengyu Feng, Shanda Li, and Yiming Yang. CO-Bench: Benchmarking language model agents in algorithm search for combinatorial optimization. In *AAAI Conference on Artificial Intelligence*, volume 40, pp. 33126–33134, 2026.

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

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.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025a.

Zhuohan Wang, Ziwei Zhu, Ziniu Li, Congliang Chen, Yizhou Han, Yufeng Lin, Zhihang Lin, Angyang Gu, Xinglin Hu, Ruoyu Sun, and Tian Ding. ORGEval: Graph-theoretic evaluation of LLMs in optimization modeling. *arXiv preprint arXiv:2510.27610*, 2025b.

Xuan Wu, Di Wang, Chunguo Wu, Lijie Wen, Chunyan Miao, Yubin Xiao, and You Zhou. Efficient heuristics generation for solving combinatorial optimization problems using large language models. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 3228–3239, 2025.

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

Tianshi Xu, Huifeng Wen, and Meng Li. Adapting the interface, not the model: Runtime harness adaptation for deterministic LLM agents. *arXiv preprint arXiv:2605.22166*, 2026.

Jiancong Yang, Xinan Chen, Rong Qu, Zishang Qiu, and Ruibin Bai. A multi-agent framework powered by large language models for automatic heuristic design. *Swarm and Evolutionary Computation*, 107:102487, 2026.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

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

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

Mingxin Yu, Ruixiao Yang, and Chuchu Fan. Heuristic search as language-guided program optimization. *arXiv preprint arXiv:2602.16038*, 2026.

Baoyun Zhao, He Wang, and Liang Zeng. G-LNS: Generative large neighborhood search for LLM-based automatic heuristic design. *arXiv preprint arXiv:2602.08253*, 2026a.

Chenyu Zhao, Shenglin Zhang, Wenwei Gu, Yongqian Sun, Dan Pei, Chetan Bansal, Saravan Rajmohan, and Minghua Ma. AgentTether: Graph-guided diagnosis and runtime intervention for reliable LLM agent operation. *arXiv preprint arXiv:2607.06273*, 2026b.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte Carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 78338–78373. PMLR, 2025.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024.

Appendix Contents

A Related Work **16**

A.1 Harness for Language-Model Agents 16

A.2 LLMs for Optimization Modeling and Solving 16

A.3 LLM-based Automatic Heuristic Design 16

A.4 Harnesses in LLM-based Solver Refinement 17

B Method Details **17**

B.1 Refinement Procedure 17

B.2 Evaluation and Scope Rules 18

C Prompt Design **18**

C.1 System Prompts 18

C.2 User Prompts 20

C.3 Harness State Diagnosis 21

C.4 Revision Scope Prompts 21

C.5 Revision Direction Construction 24

C.6 Complete Refinement Example 25

D Metric Definitions **26**

D.1 Revision Distance 26

D.2 Solution Performance Metrics 27

D.3 Refinement Trajectory Metrics 27

E Experimental Setup **28**

E.1 Benchmark Problems 28

E.2 Baseline Configurations 29

E.3 Search Cost and Efficiency 30

F Evidence of Failure Feedback Redundancy **30**

G Additional Performance Analysis **31**

G.1 Quality–Yield Analysis 31

G.2 Performance across Problems and Backbones 32

G.3 Performance across Initializations 33

H Limitations and Future Work **33**

A Related Work

A.1 Harness for Language-Model Agents

Before the agent harness emerged as an explicit systems concept, external environments already supported agent interaction and feedback. TextWorld (Côté et al., 2019), ALFWorld (Shridhar et al., 2021), and WebShop (Yao et al., 2022) provide executable environments for text, embodied, and web tasks. ReAct (Yao et al., 2023) integrates reasoning, action, and observation, while Reflexion (Shinn et al., 2023) carries failure feedback across trials through verbal reflection. Research has expanded to execution, verification, and interface design. SWE-bench (Jimenez et al., 2024) tests repository edits, while WebArena (Zhou et al., 2024) checks functional correctness against environment states. SWE-agent (Yang et al., 2024) shows that interface design can improve performance without changing the model. Together, these developments mark a transition from interaction environments to unified harnesses integrating execution, tooling, context, state maintenance, and verification.

Recent work further uses execution histories and failures to organize subsequent runs. LifeHarness (Xu et al., 2026) distills reusable interventions from recurring failures in training interaction histories while keeping the harness fixed during evaluation. AgentTether (Zhao et al., 2026b) localizes critical subtrajectories within failed executions and provides targeted guidance and runtime interventions for re-execution of the same agent. These works illustrate a shift from using execution outcomes for evaluation toward using them to organize subsequent execution.

A.2 LLMs for Optimization Modeling and Solving

LLMs are increasingly used to automate optimization modeling by translating natural-language problem descriptions into mathematical formulations and solver-executable code. Training-based approaches develop models specialized for this task through synthesized supervision. ORLM combines semi-automated data generation with open-source model training (Huang et al., 2025), while OptMATH constructs scalable bidirectional modeling data with forward verification and rejection sampling (Lu et al., 2025).SIRL further grounds training in solver-verifiable signals over generated models and executable code, providing rewards for syntactic validity, feasibility, and solution quality (Chen et al., 2025). Opt-Miner combines tree-guided data synthesis with reinforcement learning that rewards retrieval quality and modeling accuracy, training agents to acquire external knowledge for optimization problems with knowledge gaps (Liu et al., 2026c).

Complementary prompt-based and agentic approaches structure the modeling process itself. OptiMUS formulates models, generates solver code, and uses execution feedback to diagnose and repair modeling or implementation errors (Ahmaditeshnizi et al., 2024). OptiTree organizes formulation through a hierarchical modeling tree (Liu et al., 2025b), while MM-Agent studies broader mathematical modeling through staged workflows and hierarchical knowledge (Liu et al., 2025a). LEAN-LLM-OPT dynamically constructs agentic workflows for large-scale auto-formulation and delegates structured data operations to auxiliary tools (Liang et al., 2026). MIRROR combines hierarchical exemplar retrieval with execution-driven revision of mathematical models and solver code (Shi et al., 2026). Evaluation has also advanced through structural-equivalence measures (Wang et al., 2025b) and industrial-scale benchmarks derived from real mixed-integer programs (Li et al., 2026).

Beyond formulation, LLMs have been trained as end-to-end CO solvers (Jiang et al., 2025) and used to automate solver-level decisions, including instance-specific MIP configuration (Luo et al., 2026) and branching-policy discovery (Hou et al., 2026). These studies focus on problem formulation, direct solution generation, or solver-level decisions. A related line of work instead uses LLMs to automatically design and refine executable heuristics.

A.3 LLM-based Automatic Heuristic Design

Heuristic design for combinatorial optimization has long relied on expert knowledge. Hyper-heuristics and algorithm configuration methods search within predefined spaces of heuristics, components, or parameters (Burke et al., 2013; Pillay & Qu, 2018; Burke et al., 2019; Hutter et al., 2009; López-Ibáñez et al., 2016). LLMs extend this search to executable programs (Liu et al., 2026b). FunSearch (Romera-Paredes et al., 2024) evolves functions within fixed program skeletons, EoH (Liu et al., 2024) evolves heuristic ideas and implementations, and ReEvo (Ye et al., 2024) incorporates

reflection. Together, these methods establish *iterative revision* through repeated program evaluation and verification.

Subsequent research focuses on organizing and regulating this iterative search process. Approaches promote candidate diversity and branching (Dat et al., 2025; Zheng et al., 2025), introduce different modification granularities (Chen et al., 2026a;c), and adapt search behavior to current states (Chen et al., 2026b; Wu et al., 2026; Lai et al., 2026). Complementary extensions explore complete metaheuristics and coupled operators (van Stein & Bäck, 2025; Zhao et al., 2026a), collaborative search (Hu & Zhang, 2025), multiple objectives (Yao et al., 2025), and complementary heuristic sets (Liu et al., 2026a). Others improve guidance and evaluation efficiency (Wu et al., 2025) or adapt the generating model (Huang et al., 2026). These developments broaden both the algorithmic structures explored and the mechanisms governing their evolution.

Alongside these advances in search design, other work focuses on enriching and organizing the evidence used to guide revisions. HeuriGym (Chen et al., 2026d), LaGO (Yu et al., 2026), and ReVEL (Cuong Van Duc et al., 2026) provide execution, analytical, and structured feedback, respectively. PathWise (Gungordu et al., 2026) reuses search histories, AHD Agent (Lv et al., 2026) acquires targeted evidence, and ARES (Yang et al., 2026) combines structural ablation with accumulated design experience. These approaches extend revision guidance beyond aggregate performance toward richer, more structured, and more targeted evidence.

A.4 Harnesses in LLM-based Solver Refinement

For solver refinement, harness design must address not only evaluation and feedback, but also how each revision is controlled. HeuriGym (Chen et al., 2026d) integrates execution, output parsing, and verification to provide instance-level feedback. CO-Bench (Sun et al., 2026) and ALE-Bench (Imajuku et al., 2025) evaluate complete solvers through execution and performance measurement. However, mapping evaluation states to explicit revision controls remains underexplored.

This gap concerns both *revision scope* and *revision direction*. Scope determines how extensively a solver may change. Existing methods typically couple it to search operators rather than constrain it independently before generation (Zheng et al., 2025; Chen et al., 2026a;b; Wu et al., 2026). Direction specifies where changes should focus, yet priorities largely depend on the model's interpretation of richer feedback (Yu et al., 2026; Cuong Van Duc et al., 2026; Gungordu et al., 2026; Lv et al., 2026). The practical challenge is to repair current failures without disrupting previously verified behavior. This motivates exploring how a harness can use the execution and verification evidence it collects to jointly control revision direction and scope.

B Method Details

B.1 Refinement Procedure

DiSH follows a harness-controlled iterative refinement procedure, summarized in Algorithm 1. The process begins by evaluating the initial solver h_0 on the demonstration set to obtain the evaluation records $\mathcal{Z}_0$ and the corresponding feasibility pass rate ρ_0. At each refinement step t, the harness first determines the revision scope η_t from the current feasibility pass rate ρ_t and constructs the revision direction d_t from the accumulated evaluation records $\mathcal{Z}_{0:t}$. These controls, together with the evaluation feedback, are assembled into the refinement prompt p_t. The complete LLM context is then defined as $\mathcal{C}_t = [\mathcal{S}, \mathcal{U}, h_t, p_t]$, consisting of the fixed system prompt, task prompt, current solver, and round-specific refinement prompt. The LLM uses this context to generate the next solver h_{t+1}, which is evaluated in the same manner to obtain $\mathcal{Z}_{t+1}$ and ρ_{t+1}. All experiments use `history_rounds=1`, so each LLM call retains only this controlled context rather than the full interaction history. After K refinement steps, all evaluated solvers remain eligible for delivery, and DiSH selects as $h^\star$ the checkpoint with the highest demonstration feasibility pass rate ρ_t, breaking ties by lower mean cost over the feasible demonstration instances. The selected solver is then fixed and evaluated on the held-out test instances, which are not used during refinement or checkpoint selection.

Algorithm 1 Direction-and-Scope-Guided Solver Refinement

Input: system prompt $\mathcal{S}$; task prompt $\mathcal{U}$ (Appendix C); demonstration set $\mathcal{D}_{\text{train}}$; initial solver h_0; direction strategy $\mathcal{G}_{\text{dir}} \in \{\mathcal{G}_{\text{GWRS}}, \mathcal{G}_{\text{CRS}}, \mathcal{G}_{\text{GRS}}\}$; number of refinement steps K
Output: delivered solver $h^\star$

$(\mathcal{Z}_0, \rho_0) \leftarrow \mathcal{E}(h_0, \mathcal{D}_{\text{train}})$
for $t = 0, \ldots, K-1$ **do**
  $\eta_t \leftarrow \mathcal{G}_{\text{step}}(\rho_t)$
  $d_t \leftarrow \mathcal{G}_{\text{dir}}(\mathcal{Z}_{0:t})$
  $p_t \leftarrow \mathcal{H}(\mathcal{Z}_t, d_t, \eta_t)$
  $\mathcal{C}_t \leftarrow [\mathcal{S}, \mathcal{U}, h_t, p_t]$
  $h_{t+1} \sim \mathcal{M}_{\text{LLM}}(\cdot \mid \mathcal{C}_t)$
  $(\mathcal{Z}_{t+1}, \rho_{t+1}) \leftarrow \mathcal{E}(h_{t+1}, \mathcal{D}_{\text{train}})$
end for
$h^\star \leftarrow \text{Deliver}\left(\{(h_t, \mathcal{Z}_t)\}_{t=0}^{K}\right)$
return $h^\star$

B.2 Evaluation and Scope Rules

B.2.1 Instance Evaluation

The evaluation module $\mathcal{E}$ assesses each solver on the demonstration instances through three sequential stages. An instance proceeds to the next stage only after passing the current one. If a stage fails, the evaluation record identifies this first failure stage and the corresponding error. Instances passing all three stages are marked as `SUCCESS`. Table 4 summarizes the evaluation criteria and failure outcomes. These records distinguish execution failures, invalid outputs, and constraint violations. They are aggregated into the harness state diagnosis described in Appendix C.3, which summarizes the current solver state for subsequent revision.

Table 4: Three-stage feasibility evaluation for each instance.

Stage	Evaluation criterion
I Execution	Terminates normally within the time and memory limits; otherwise returns an exception, timeout, or memory error.
II Output validity	Produces a non-empty output and allows the evaluation pipeline to complete; otherwise reports an output or evaluation error.
III Solution feasibility	Receives a positive verdict from the problem-specific verifier; otherwise reports a verification failure.

B.2.2 Revision Scope Rules

The scope controller maps the current feasibility pass rate ρ_t to a revision instruction η_t. Table 5 specifies the four modes, their thresholds, and the permitted revision levels. The corresponding prompts are provided in Appendix C.4.

C Prompt Design

In this section, we detail the system and user prompts used by DiSH, together with the harness state diagnosis and the controls over revision scope and direction. We also provide a complete refinement example to illustrate how these components work together in practice. In the prompt text, "test case" is the literal label inherited from the HeuriGym feedback template and refers exclusively to demonstration instances in $\mathcal{D}_{\text{train}}$; held-out test instances are never exposed to the LLM.

C.1 System Prompts

Each refinement iteration begins with a task-agnostic system prompt that instructs the LLM to revise the current solver under a fixed set of execution, output, and refinement constraints. Beyond the basic solver interface and runtime requirements, the prompt specifies three revision levels, the three-stage

Table 5: Feasibility-adaptive revision-scope instructions.

Feasibility state	Instruction	Permitted revision scope
$\rho_t < 0.4$	BROAD	Parameter-, strategy-, and algorithm-level changes are permitted, including algorithm replacement.
$0.4 \leq \rho_t < 0.8$	MODERATE	Strategy-level changes are preferred; algorithm-level changes are permitted only when the observed failures indicate a structural mismatch.
$0.8 \leq \rho_t < 1$	CONSERVATIVE	Parameter-level changes and narrowly targeted strategy-level changes are permitted; algorithm-level changes are prohibited.
$\rho_t = 1$	PROTECTED	The current solver is preserved by default; only parameter-level changes with a concrete benefit and no adverse effect on currently feasible instances are permitted.

harness evaluation pipeline, and a feasibility-conditioned scope schedule that progressively restricts how much the solver may change as more demonstration cases succeed. Several placeholders are instantiated at runtime according to the problem configuration. For example, `{TIMEOUT}` specifies the wall-clock time limit, and `{N_DEMO}` denotes the number of demonstration instances used for evaluation feedback.

System Prompt

You are an expert solver engineer. Iteratively refine a correct, robust solution to the optimization problem below: keep what works and repair failures in a targeted way. How large a change is appropriate depends on how much of the solver already works—this is governed by the mode schedule in point 7, not by a blanket preference for small changes.

Follow these instructions strictly.

1. Implement only the `solve` function. Do NOT change its signature, argument names, or argument types. Helper functions, classes, and internal data structures are allowed.
2. The machine has 8 CPU cores. Time limit: `{TIMEOUT}` seconds. Use parallelism only when it is reliable and clearly beneficial within the budget.
3. Python 3.12. Permitted third-party libraries only:
   - `numpy==2.2.5`
   - `networkx==3.4.2`
   - `pandas==2.2.3`
4. Respond with code only—a complete `solve` implementation plus any helpers. No explanation, no Markdown, no extra text.
5. **Change scope**—three levels, distinguished by how much of the solver a change touches, not by which component is edited:
   - **Parameter**: numerical constants only; control flow and logic unchanged.
   - **Strategy**: logic changes within the current algorithmic paradigm; approach and complexity class unchanged.
   - **Algorithm**: replaces the core paradigm or changes the complexity class.

   Correctness fixes—a wrong loop bound, a wrong index, a mishandled constraint, an off-by-one—are NOT one of these three levels and are never restricted by mode. Fix them in any state, including when the defect lies in code that currently passing cases also execute.
6. Each round you receive concrete execution feedback computed from a fixed set of `{N_DEMO}` demonstration cases: per-case results, the dominant failure stage, and the demo SUCCESS proportion. The stages are:
   - **Stage I**: execution failure (crash, exception, import error, or timeout);
   - **Stage II**: the program runs but produces missing or malformed output;
   - **Stage III**: output is produced but rejected by the verifier (infeasible).

   This feedback is your primary signal—act on it; do not refine from intuition alone. The `{N_DEMO}` cases are representative but not exhaustive, and the final solver is judged on unseen instances, so prefer general robustness over fitting these `{N_DEMO}`. Breaking a currently passing case is a regression.

7. The permitted change scope narrows as demo SUCCESS rises—broad when little works, tight when most works. You will be told at the start of each round which mode is active and exactly what it permits; obey that instruction. This clause only states the overall shape:

- $< 40\%$ **(BROAD)**: all three levels open. Restore the dominant failure stage first and do not optimize cost yet. Even if a few cases already pass, treat this as full recovery of that stage—those cases may simply not exercise the faulty path.
- 40–80% **(MODERATE)**: Strategy by default; escalate to Algorithm only when a failing case points to a structural mismatch in the current paradigm; Parameter anytime.
- 80–99% **(CONSERVATIVE)**: Parameter; Strategy only when narrowly targeted at the specific failing cases; Algorithm not permitted.
- 100% **(PROTECTED)**: effectively frozen. Return the solver unchanged unless a Parameter change is both provably safe on every passing case and clearly beneficial.

Global rules (apply in every mode):

- Address the failure stage with the most failing cases first (ties go to the earlier stage). Overall priority: executable $\rightarrow$ valid output $\rightarrow$ feasible $\rightarrow$ lower cost. Never sacrifice an earlier stage to chase a later one.
- After any change, every previously passing case must still pass—judged by the end result, not by treating any block of code as untouchable.
- Correctness fixes (point 5) are permitted in every mode without restriction.

8. Respect the exact input/output format in the problem statement, handle edge cases, and keep input parsing and output writing stable.

9. Do not hardcode behavior for individual demonstration cases; the solver must generalize to unseen inputs.

Above all: keep each change within the scope the active mode permits—broad when little works, tight when most works. At 100%, usually change nothing. Correctness fixes are always allowed.

C.2 User Prompts

For each problem, the first refinement iteration begins with a task-specific user prompt whose structure is adapted from the HeuriGym setup (Chen et al., 2026d). The prompt provides the problem information, which specifies the optimization task together with its background, formalization, and input and output requirements, as well as a program template that defines the solver interface used throughout refinement. Accordingly, `{BACKGROUND}`, `{FORMALIZATION}`, `{INPUT_FORMAT}`, and `{OUTPUT_FORMAT}` are instantiated from the corresponding problem description, while `{SOLVER_TEMPLATE}` is filled with the problem-specific solver template.

User Prompt

```
# Problem Information

## Background
{BACKGROUND}

## Formalization
{FORMALIZATION}

## Input Format
{INPUT_FORMAT}

## Output Format
{OUTPUT_FORMAT}

# Program Template
{SOLVER_TEMPLATE}
```

C.3 HARNESS STATE DIAGNOSIS

After each Harness evaluation, the instance-level evaluation records Z_t of the current solver are organized into a round-level *state diagnosis*. For every evaluated solver state, the diagnosis reports the numbers and proportions of Stage I, Stage II, Stage III, and SUCCESS cases, together with the current `Focus stage`, `Current mode`, and the corresponding `Mode reason`. When sufficient refinement history is available, the refinement prompt additionally includes a compact `Progress Trend` summarizing the preceding evaluated solver states. These quantities help the LLM identify the remaining failure patterns, assess how much solver behavior has already been verified, and track changes across refinement rounds. The same structured state also provides the basis for the subsequent revision-scope control and revision-direction construction. The resulting diagnosis is placed at the beginning of each round-specific refinement prompt, providing the LLM with a concise view of the current solver state and its recent refinement progress.

Harness State Diagnosis

```
# Feedback from Previous Iteration (Iteration {ITERATION})

## Demo/Train Results Summary

- Total demo/train cases: {N_DEMO}
- Stage I failures (execution error): {N_I} ({R_I})
- Stage II failures (no/invalid output): {N_II} ({R_II})
- Stage III failures (verifier rejected): {N_III} ({R_III})
- Successful cases: {N_SUCCESS} ({R_SUCCESS})
- Focus stage: {FOCUS_STAGE}
- Current mode: {MODE}
- Mode reason: {MODE_REASON}

When sufficient refinement history is available:

## Progress Trend (demo/train)

{PROGRESS_TREND}
```

Specifically, SUCCESS proportion corresponds to the feasibility pass rate ρ_t defined in the main text. The revision scope then maps ρ_t to `Current mode`. The `Focus stage` identifies the evaluation stage with the largest number of failed cases $f_t = \arg\max_{s\in\{\mathrm{I,II,III}\}} |\{x \in D_{\text{train}} : s_t(x) = s\}|$. Ties are resolved by prioritizing earlier stages in the evaluation pipeline: Stage I $\rightarrow$ Stage II $\rightarrow$ Stage III. When all demonstration cases succeed, no failure stage is active and the `Focus stage` is reported as `none (all demonstration cases pass)`.

C.4 REVISION SCOPE PROMPTS

Revision scope control determines how extensively the LLM is allowed to modify the current solver at each refinement round. Early in refinement, substantial changes may be necessary to repair fundamental failures, whereas later revisions should increasingly preserve behavior that has already been verified. DiSH therefore adapts the permitted revision scope to the current solver state by mapping the SUCCESS proportion ρ_t to one of four modes, BROAD, MODERATE, CONSERVATIVE, or PROTECTED. As ρ_t increases, these modes progressively narrow the allowable changes from unrestricted Parameter, Strategy, and Algorithm level revisions to increasingly localized modifications and, ultimately, preservation of the current solver.

BROAD. When $\rho_t < 0.4$, the solver remains in an early recovery stage, where failures may arise from execution, output generation, or solution feasibility. DiSH therefore permits Parameter, Strategy, and Algorithm-level revisions without restricting the change scope. The guidance is further specialized by the current `Focus stage`, resulting in three corresponding prompts for Stage I execution failures, Stage II output failures, and Stage III feasibility failures.

BROAD Guidance: Stage I — Execution

Improvement Guidance — Mode: BROAD *Stage I focus: execution*

Many cases are failing at execution—the program crashes, raises exceptions, or times out before producing output.

Parameter-level, Strategy-level, and Algorithm-level changes are all permitted in this mode. The root cause of execution failures can sit at any level—if persistent timeouts suggest the current algorithmic paradigm is too expensive, switching to a paradigm with lower complexity is a reasonable option.

Please focus on:

1. Eliminating syntax, import, and runtime errors so the program runs end-to-end for every case.
2. Using only the allowed libraries.
3. Keeping the program within the time limit.

BROAD Guidance: Stage II — Valid Output

Improvement Guidance — Mode: BROAD *Stage II focus: valid output*

The program runs without crashing but does not produce valid output—the output file is empty, missing, or in the wrong format.

Parameter-level, Strategy-level, and Algorithm-level changes are all permitted in this mode. The root cause of output failures can sit at any level.

Please focus on:

1. Producing a non-empty, correctly formatted output file for every case, matching the format specified in the problem description.
2. Verifying the output file is created at the exact path the verifier expects.
3. Fixing edge cases where the program silently produces no output.

BROAD Guidance: Stage III — Feasibility

Improvement Guidance — Mode: BROAD *Stage III focus: feasibility*

Stage III is the dominant failure stage. In affected cases, the solver reaches verification, but the generated solutions are rejected as infeasible. The primary objective of this round is therefore to restore solution feasibility.

Parameter-level, Strategy-level, and Algorithm-level changes are all permitted in this mode.

Please focus on:

1. Repairing the recurring verifier failures using the provided failure evidence.
2. Making solutions feasible first; a valid but suboptimal solution is preferable to an invalid one at this stage.
3. Fixing the underlying defect at the appropriate revision level.
4. If the current algorithmic paradigm appears mismatched with the problem structure, replacing it is a reasonable option.
5. Do not optimise for cost in this round; feasibility comes first.

If a few cases already pass, that is not yet evidence that the algorithm is correct—those cases may simply not exercise the faulty path. Treat this round as feasibility recovery regardless.

MODERATE. When $0.4 \leq \rho_t < 0.8$, several demonstration cases already succeed, indicating that the current solver has established partially verified behavior. DiSH uses Strategy-level revision as the default in this mode, while allowing Parameter-level changes at any time. Algorithm-level revision is reserved for failures that suggest a structural mismatch in the current algorithmic paradigm.

MODERATE Guidance

Improvement Guidance — Mode: MODERATE *{SUCCESS_RATIO} Success*

Several demo/train cases already pass—the algorithmic skeleton has been validated on real instances. Some cases still fail; they may fail at any pipeline stage (execution, output, or verification). This round targets those remaining failures: make them pass first, then improve solution quality.

Parameter-level, Strategy-level, and Algorithm-level changes are all permitted in this mode. This mode is NOT a conservative mode and is distinct from the 80%+ CONSERVATIVE mode, which forbids Algorithm-level changes. Within the permitted set, prefer the smaller change scope first: make Strategy-level changes by default, and escalate to Algorithm-level changes when the failing cases' symptoms point to a structural mismatch in the current paradigm. Parameter-level changes are available at any time.

Fixing incorrect implementation is not restricted by the granularity above: if a failing case is caused by a defect—a wrong loop bound, a wrong index, a mishandled constraint—correct it directly, even if the defect lies in code that currently passing cases also run through.

Please focus on:

1. Use the provided failure evidence to repair the remaining failing cases.
2. After any change, the cases that passed before must still pass. Manage regression by checking this end result—not by treating any particular block of existing code as off-limits.
3. Securing feasibility on every case before attempting any cost improvement.

CONSERVATIVE. When $0.8 \leq \rho_t < 1$, most demonstration cases already succeed, and only a small number of failures remain. DiSH limits the revision scope accordingly: Parameter-level changes are allowed, Strategy-level changes must be narrowly targeted at the remaining failures, and Algorithm-level revisions are disabled to reduce the risk of regression.

CONSERVATIVE Guidance

Improvement Guidance — Mode: CONSERVATIVE *{SUCCESS_RATIO} Success*

Most demo/train cases are passing; only a small number remain failing. The case-by-case feedback above shows exactly which cases failed and how.

Parameter-level changes are permitted. Strategy-level changes are permitted only when narrowly targeted at the failing cases shown above. Algorithm-level changes are not permitted in this mode—they carry a high risk of regressing the currently passing cases.

Correctness fixes are exempt from this granularity limit (see point 5 of the system instructions).

Please focus on:

1. Diagnosing the specific failures shown in the case feedback above, and making the smallest change that addresses them.
2. Avoiding edits to logic that has no causal link to those failing cases.
3. When uncertain whether a change might affect passing cases, prefer not to make it.

PROTECTED. When $\rho_t = 1$, all demonstration cases succeed, and the current solver has established fully verified behavior on the demonstration set. DiSH therefore places the solver in a protected state, where preserving the current behavior becomes the default. Strategy and Algorithm level revisions are disabled, while Parameter level changes are allowed only when they provide a clear benefit without adversely affecting any currently successful case.

PROTECTED Guidance

Improvement Guidance — Mode: PROTECTED *100% Success*

All demo/train cases currently pass. The solver is in a frozen state. The default action in this mode is to return the existing solver code unchanged. This is the correct response, not a failure to follow instructions. In most rounds at 100% success, no change should be made.

Only Parameter-level changes are permitted in this mode. Strategy-level and Algorithm-level changes are not permitted.

Specifically:

- Do NOT add new functions, classes, or data structures.
- Do NOT add new control flow branches or new logic paths.
- Do NOT refactor existing code, even if a cleaner version would be equivalent.
- Do NOT replace existing logic with logic you consider "better".
- Do NOT make speculative efficiency edits without a concrete reason.

A Parameter-level edit is acceptable only if you can point to a concrete, specific reason it improves the solver, AND you can confirm it cannot adversely affect any currently passing case. If either condition is unclear, return the solver unchanged.

Correctness fixes are exempt from this restriction (see point 5 of the system prompt).

C.5 Revision Direction Construction

Revision direction construction determines which failures should receive attention and what evidence should guide the next solver revision. Rather than exposing all instance-level failures to the LLM, DiSH groups failures with shared signatures and selects representative evidence from these groups, while retaining successful cases as behavioral anchors to discourage regression. Based on this common representation, we consider three direction strategies, GWRS, CRS, and GRS, which differ in how failure evidence is prioritized, selected, and organized. The resulting guidance is dynamically constructed from the evaluation records available at each refinement round.

Group-Wise Representative Sampling (GWRS). GWRS groups failures by shared signatures and retains one representative case from each group, together with all successful cases. The feedback first summarizes the selected representative evidence and then presents the corresponding case-level examples, reducing redundancy while preserving distinct failure patterns.

GWRS Feedback

Summary. These are the test cases and results from the previous iteration. Feedback uses group-wise representative sampling: **{N_SHOWN_FAILING}** failing case(s) shown out of **{N_FAILING}** failing, with one representative per distinct failure group, plus all **{N_SUCCESS}** successful case(s). Cases with the same failure-group signature as a representative are omitted this round as redundant.

Representative Failure Case.

Test Case 1: `{test_name}`
Input file: `{content}`
Observed result: `{execution_message}`

Successful Anchor.

Test Case 2: `{test_name}`
Input file: `{content}`
Observed result: `{execution_message}`

Correction Representative Sampling (CRS). Unlike GWRS, which represents failures based only on their current signatures, CRS retains one smallest-input representative from each current failure group, together with all successful cases. Cross-round comparisons identify regressions, which are emphasized in the resulting revision guidance.

CRS Feedback

Regression Alert. The following cases were VALID in the previous iteration but FAIL now. These regressions must be repaired before optimizing new failures.

{case_id}: was valid in the previous round `{previous_cost}`
→ now Stage `{stage}`: `{execution_message}`

Failure Groups. Failing cases are grouped by a shared failure signature. The smallest-input case in each group is shown as its representative.

Representative Failure Case.

Test Case 1: `{test_name}`
Input file: `{content}`
Observed result: `{execution_message}`

Successful Anchor.

Test Case 2: `{test_name}`
Status: valid
Cost: `{cost}`

Scope-Aware Group Representative Sampling (GRS). GWRS retains one representative from every current failure group, while CRS further prioritizes cases that regress across refinement rounds. GRS instead performs group-level selection, retaining only the failure groups most relevant to the current round. Groups at the `Focus stage` are prioritized first, and the active revision mode determines how many high-priority groups are selected. One representative case is then retained from each selected group, together with all successful cases.

GRS Feedback

Failure Groups Selected for This Round.
Failing cases are grouped by a shared failure signature, and only the highest-priority groups are selected for repair this round. Groups at the focus stage are prioritized first. Mode **{MODE}**; **{N_SELECTED}** group(s) selected.

Representative Failure Case.

Selected failure group {G}; signature `{signature}`; covers {N_GROUP} cases.
Test case: `{test_name}`
Input: `{content}`
Observed result: `{execution_message}`

Success Anchors — already valid, must NOT regress.

```
- {case_id}:  cost {cost}
- {case_id}:  cost {cost}
```

C.6 Complete Refinement Example

To illustrate how the preceding components are instantiated in a complete refinement round, we present a real example from Protein Sequence Design. The example combines the current harness state, representative failure feedback, successful cases, and mode-specific revision guidance. The refinement prompt is combined with the fixed system and user prompts and the current solver h_t. These inputs are then provided to the LLM to generate the next solver h_{t+1}.

Instantiated Refinement Prompt

 # Feedback from Previous Iteration (Iteration 0)

Demo/Train Results Summary

- Total demo/train cases: 5
- Stage I failures (execution error): 0 (0.0%)
- Stage II failures (no/invalid output): 0 (0.0%)
- Stage III failures (verifier rejected): 3 (60.0%)
- Successful cases: 2 (40.0%)
- **Focus stage: Stage III**
- **Current mode: MODERATE**
- Mode reason: SUCCESS is 40%; the algorithmic skeleton has been validated, so strategy-level improvement is the default.

These are the test cases and results from the previous iteration. Feedback uses group-wise representative sampling: 1 failing case shown out of 3 failing, with one representative per distinct failure group, plus all 2 successful cases. Cases with the same failure-group signature as a representative are omitted this round as redundant.

Representative Failure Case

Test Case: 1a4y

Representative of 3 demonstration cases in the same failure group this round; the other 2 are grouped and not shown.

Result: Invalid solution
Error: Verification failed: Generated sequence length (460) does not match expected length (1166).

Successful Anchors

Test Case: 1aaj
Result: Valid solution with cost 45.714285714285715

Test Case: 1aba
Result: Valid solution with cost 40.229885057471265

 # Improvement Guidance — Mode: MODERATE *40% Success*

Several demo/train cases already pass—the algorithmic skeleton has been validated on real instances. Some cases still fail; they may fail at any pipeline stage (execution, output, or verification). This round targets those remaining failures: make them pass first, then improve solution quality.

Parameter-level, Strategy-level, and Algorithm-level changes are all permitted in this mode. Within the permitted set, prefer the smaller change scope first: make Strategy-level changes by default, and escalate to Algorithm-level changes when the failing cases' symptoms point to a structural mismatch in the current paradigm. Parameter-level changes are available at any time.

Please focus on:

1. Use the provided failure evidence to repair the remaining failing cases.

2. After any change, the cases that passed before must still pass.

3. Secure feasibility on every case before attempting any cost improvement.

D METRIC DEFINITIONS

We employ three groups of metrics to evaluate the magnitude of solver revisions, the performance of the resulting solutions, and the dynamics of the refinement process.

D.1 REVISION DISTANCE

Section 2 uses revision distance to quantify the magnitude of each solver revision. For a solver program h, we tokenize its Python source code and remove comments and formatting-related tokens, including indentation, line breaks, and end-of-file markers. Let $(z_1, \ldots, z_m)$ denote the resulting token sequence. We define $T(h) = \{(z_i, z_{i+1}, \ldots, z_{i+4}) \mid 1 \leq i \leq m - 4\}$ as the set of all contiguous five-token subsequences.

Jaccard Similarity. For two consecutive solvers h_t and h_{t+1}, we measure their code similarity using the Jaccard index:

$$J(h_t, h_{t+1}) = \frac{|T(h_t) \cap T(h_{t+1})|}{|T(h_t) \cup T(h_{t+1})|}.$$

Revision Distance. The revision distance is defined as the complement of their Jaccard similarity:

$$\Delta(h_{t+1}, h_t) = 1 - J(h_t, h_{t+1}).$$

A smaller $\Delta(h_{t+1}, h_t)$ indicates greater code similarity between consecutive solvers, whereas a larger value indicates a more extensive revision. We do not normalize variable names, canonicalize abstract syntax trees, or otherwise account for semantic equivalence between programs. Thus, $\Delta(h_{t+1}, h_t)$ serves as a transparent proxy for code-level revision magnitude rather than semantic distance.

D.2 Solution Performance Metrics

Following the HeuriGym evaluation protocol (Chen et al., 2026d), we evaluate each delivered solver on the held-out test instances using three metrics: Quality, Yield, and the Quality-Yield Index (QYI). Let $\mathcal{V}$ denote the set of test instances that pass verification, and let N be the total number of test instances. For each $x \in \mathcal{V}$, the normalized solution quality relative to the reference cost is defined as

$$q_x = \mathrm{Normalize}_p(c_x, c_x^\star).$$

Quality. Quality measures the average normalized solution quality over successfully verified test instances:

$$\mathrm{Quality} = \frac{1}{|\mathcal{V}|} \sum_{x \in \mathcal{V}} q_x.$$

Yield. Yield measures the proportion of test instances that pass verification:

$$\mathrm{Yield} = \frac{|\mathcal{V}|}{N}.$$

Quality-Yield Index (QYI). QYI jointly captures solution quality and verification coverage through their harmonic mean:

$$\mathrm{QYI} = \frac{2\,\mathrm{Quality}\,\mathrm{Yield}}{\mathrm{Quality} + \mathrm{Yield}}.$$

We set $\mathrm{QYI} = 0$ when no test instance passes verification. All three metrics are computed separately for each run before aggregating the run-level results.

D.3 Refinement Trajectory Metrics

We additionally report three trajectory-level diagnostics computed on the demonstration instances. Each run contains five evaluated solver states, consisting of the shared initial solver followed by four refinement steps. Let ρ_t denote the feasibility pass rate of state t, let N denote the total number of runs, and let $\mathcal{C}$ denote the set of converged runs. For a run whose final state is fully feasible, we define the stable convergence state as $t_{\mathrm{conv}} = \min\{t : \rho_t = \rho_{t+1} = \cdots = \rho_T = 1\}$. Thus, t_{conv} marks the beginning of the final uninterrupted fully feasible suffix rather than the first time full feasibility is reached.

Convergence Rate. Convergence Rate measures the fraction of runs that end in full feasibility:

$$\mathrm{ConvergenceRate} = \frac{N_{\mathrm{conv}}}{N},$$

where N_{conv} denotes the number of runs satisfying $\rho_T = 1$.

Survival Rate. Survival Rate measures the fraction of runs that preserve full feasibility after reaching it for the first time:

$$\mathrm{SurvivalRate} = \frac{N_{\mathrm{surv}}}{N_{\mathrm{hit}}},$$

Table 6: Instance statistics for the nine benchmark problems.

Problem	Source	# Demo	Demo size	# Test	Test size
Operator Scheduling	HeuriGym	5	3–34 operations	19	28–1,500 operations
Intra	HeuriGym	6	5–836 nodes	22	771–65,335 nodes
Protein Design	HeuriGym	5	87–1,166 residues	19	46–2,958 residues
Pedigree	HeuriGym	5	3–5 individuals	15	3–350 individuals
Crew Pairing	HeuriGym	5	37–44 flight legs	9	206–6,799 flight legs
E-Graph	HeuriGym	2	7–9 e-nodes	21	1,541–121,449 e-nodes
BPP	BPPLIB	5	100 items	20	300–1,000 items
CVRP	CVRPLIB Set X	5	119–227 customers	20	297–855 customers
TSP	TSPLIB	5	99–198 cities	20	264–1,655 cities

where N_{hit} denotes the number of runs that reach $\rho_t = 1$ at least once, and N_{surv} denotes those that remain fully feasible thereafter.

Tokens to Convergence. Among converged runs, Tokens to Convergence measures the average cumulative number of LLM input and output tokens required to reach stable convergence:

$$\mathrm{TokensToConv} = \frac{1}{|\mathcal{C}|} \sum_{r \in \mathcal{C}} \left(\sum_{k=1}^{t_{\mathrm{conv}}^{(r)}} c_k^{(r)} \right),$$

where $\mathcal{C}$ denotes the set of converged runs and $|\mathcal{C}|$ its size. For each run $r \in \mathcal{C}$, $t_{\mathrm{conv}}^{(r)}$ denotes the refinement step at which stable convergence begins, and $c_k^{(r)}$ denotes the token consumption of its k-th refinement call.

E Experimental Setup

This section details the experimental setup used to evaluate DiSH. We first describe the benchmark problems, then specify the configurations of the comparison methods. Finally, we evaluate the resulting trade-off between performance and efficiency.

E.1 Benchmark Problems

We evaluate DiSH on nine combinatorial optimization problems spanning a diverse range of problem structures and scales. Six benchmarks are inherited from HeuriGym (Chen et al., 2026d), for which we preserve the original demonstration and test splits as well as the corresponding evaluation setting to ensure direct comparability with prior work. To further assess generalization beyond the original benchmark suite, we include BPP, CVRP, and TSP from BPPLIB (Delorme et al., 2018), CVRPLIB Set X (Uchoa et al., 2017), and TSPLIB (Reinelt, 1991), respectively, with fixed and mutually disjoint demonstration and held-out test sets. This expanded benchmark suite provides a broader test of DiSH across both established and additional combinatorial optimization settings. Table 6 summarizes the benchmark statistics, with problem sizes reported in the standard units used by each benchmark.

Reference Costs and Verification. Solution quality is evaluated against a per-instance reference cost. For the six benchmarks inherited from HeuriGym, we use the objective values of the expert-provided solutions as the per-instance reference costs (Chen et al., 2026d). For BPP, CVRP, and TSP, we use the published reference objective values of the corresponding benchmark instances (Delorme et al., 2018; Uchoa et al., 2017; Reinelt, 1991), providing a strong and consistent basis for quality comparison. For these three additional benchmarks, the evaluator also independently verifies solution feasibility and recomputes the objective value from the submitted solution rather than relying on solver-reported values. Specifically, BPP requires every item to be assigned exactly once to non-empty bins without exceeding bin capacity, CVRP requires every customer to be served exactly once while satisfying vehicle capacity, and TSP requires a tour that visits every city exactly once. The evaluator then recomputes the number of used bins, route cost under the benchmark distance convention, or complete tour length including the closing edge, respectively.

Table 7: Initialization and search configurations for the five AHD baselines under the shared initial solver θ_0. The new-solver cap excludes θ_0.

Method	Shared-θ_0 initialization	New-solver cap	Key configuration
EoH	Singleton population $\{\theta_0\}$, bypassing the native initialization step	≤ 8	`use_seed=True`, `pop_size=2`, `n_pop=4`, `max_sample_nums=8`
ReEvo	θ_0 is evaluated first, followed by two seed-conditioned descendants	≤ 8	`init_pop_size=2`, `pop_size=2`, `mutation_rate=0.5`, `max_fe=9`
HSEvo	θ_0 is evaluated first, followed by two seed-conditioned descendants	≤ 8	`init_pop_size=2`, `pop_size=2`, `mutation_rate=0.5`, `hm_size=2`, `max_iter=1`, `max_fe=9`
HiFo-Prompt	Singleton population $\{\theta_0\}$ through the native seeded initialization path	≤ 6	`pop_size=2`, `n_pop=1`; operator E1 is applied twice, and E2, M1, M2, and M3 once each
RefineEvo	Singleton generation zero $\{\theta_0\}$, with no separately sampled initialization batch	≤ 6	`init_pop_size=0`, `pop_size=2`, `n_pop=12`, `offspring_per_operator=1`, $B_{\text{new}} = 6$

E.2 Baseline Configurations

Compared Methods. We compare DiSH against five representative LLM-based AHD baselines: EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), HSEvo (Dat et al., 2025), HiFo-Prompt (Chen et al., 2026b), and RefineEvo (Wu et al., 2026). The original HeuriGym (Chen et al., 2026d) refinement procedure serves as an additional reference. We evaluate three DiSH variants, DiSH-GWRS, DiSH-CRS, and DiSH-GRS, corresponding to the revision direction constructions introduced in Section 3.4. All methods operate in the same complete-solver refinement setting. For the five AHD baselines, adaptations are limited to incorporating the shared initialization, complete-solver evaluation and output adaptation, minimal compatibility adjustments, and token accounting and logging. Their native search operators and method-specific search procedures remain unchanged. All AHD baselines use the same demonstration-set scalar fitness, without method-specific fitness tuning or held-out feedback.

Shared Initialization. For each problem, we generate three independent initial complete solvers. Within each initialization, all methods start from the same solver, denoted by θ_0, so that differences in starting conditions do not confound the comparison. Each method is evaluated with three independent repeats per initialization under each backbone, yielding nine runs for every combination of problem and backbone. For the population-based AHD baselines, initialization is adapted to incorporate θ_0 while preserving the subsequent search procedure. EoH, HiFo-Prompt, and RefineEvo initialize generation zero with the singleton population $\{\theta_0\}$. ReEvo and HSEvo first evaluate θ_0 and then generate seed-conditioned descendants, retaining θ_0 as an elite or best reference. Subsequent parent selection, reflection, crossover, mutation, and population updates remain method-specific. HeuriGym also starts from θ_0 and retains its original refinement procedure. Table 7 summarizes the initialization and search configurations for the five AHD baselines.

Search and Refinement Settings. DiSH follows the $i = 5$ setting of HeuriGym, with the shared initial solver θ_0 replacing program generation in the first iteration. It then performs four refinement steps, yielding five evaluated solver states in total. For DiSH, HeuriGym, and the corresponding ablation configurations, we retain one round of refinement history in all runs. Each AHD baseline follows its own search procedure and is configured to allow at least one complete search cycle. EoH, ReEvo, and HSEvo evaluate at most eight additional solvers beyond θ_0, while HiFo-Prompt and RefineEvo evaluate at most six. For ReEvo and HSEvo, `max_fe=9` includes the evaluation of θ_0. We do not enforce identical candidate or iteration counts across methods to preserve their native search behavior. Each baseline retains its original resampling and termination rules, without additional fallback or rescue candidates. The number and structure of LLM calls also vary across methods. We therefore measure search cost by the total LLM input and output tokens consumed across all calls in a completed run.

Table 8: Cumulative LLM token consumption across backbone models.

Method	Qwen3.5-122B-A10B	DeepSeek-V3	GPT-4.1-mini	Gemini-2.5-Flash
EoH	102,504 (3.45×)	34,228 (1.45×)	45,519 (1.94×)	37,037 (1.70×)
ReEvo	97,541 (3.29×)	43,182 (1.83×)	71,916 (3.07×)	48,055 (2.20×)
HSEvo	51,594 (1.74×)	29,816 (1.27×)	46,582 (1.99×)	39,197 (1.80×)
HiFo-Prompt	79,241 (2.67×)	29,763 (1.26×)	48,110 (2.05×)	30,048 (1.38×)
RefineEvo	98,298 (3.31×)	60,809 (2.58×)	68,114 (2.90×)	53,570 (2.46×)
HeuriGym	37,485 (1.26×)	28,075 (1.19×)	28,378 (1.21×)	26,103 (1.20×)
DiSH-GWRS	30,133 (1.02×)	25,959 (1.10×)	26,657 (1.14×)	25,484 (1.17×)
DiSH-CRS	29,684 (1.00×)	23,536 (1.00×)	23,454 (1.00×)	21,808 (1.00×)
DiSH-GRS	27,065 (0.91×)	23,122 (0.98×)	23,478 (1.00×)	22,198 (1.02×)

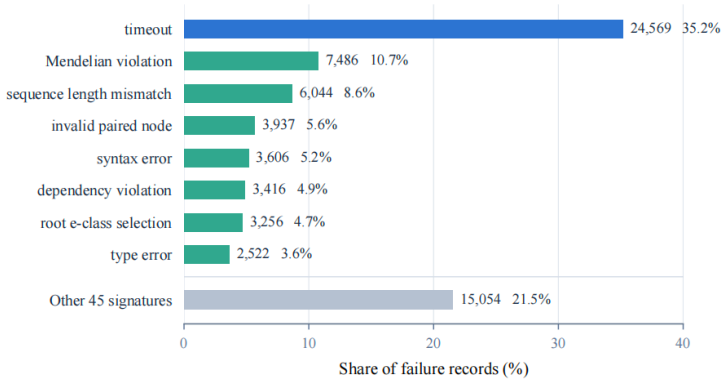


Figure 13: Distribution of normalized failure patterns across the 69,890 failure records collected from all nine benchmarks and refinement rounds. The eight most frequent patterns are shown individually, while the remaining 45 are grouped as `Other`.

E.3 Search Cost and Efficiency

Token Consumption. We measure search cost by the cumulative input and output token consumption across all LLM calls in a completed run. The reported token counts are obtained directly from the LLM provider. Token consumption is first averaged over repeated runs for each combination of problem, backbone, and initialization. We then average across the three initializations and compute a macro average over the nine benchmark problems. Because LLM providers may use different tokenization schemes and accounting rules, absolute token counts are compared only within the same backbone. For relative comparisons, token consumption is normalized to DiSH-CRS under each backbone. Table 8 reports the resulting token consumption for each backbone, together with values normalized to DiSH-CRS.

F Evidence of Failure Feedback Redundancy

To examine redundancy in the failure feedback encountered during solver refinement, we aggregate recurring failure patterns across all nine benchmarks and refinement rounds. The 69,890 recorded failures collapse into only 53 distinct normalized failure patterns, revealing substantial redundancy in instance-level feedback. This observation motivates the use of representative failure evidence during solver refinement. Figure 13 shows the frequency distribution of these patterns, while Figure 14 provides a complementary view of their recurrence and associated evaluation stages.

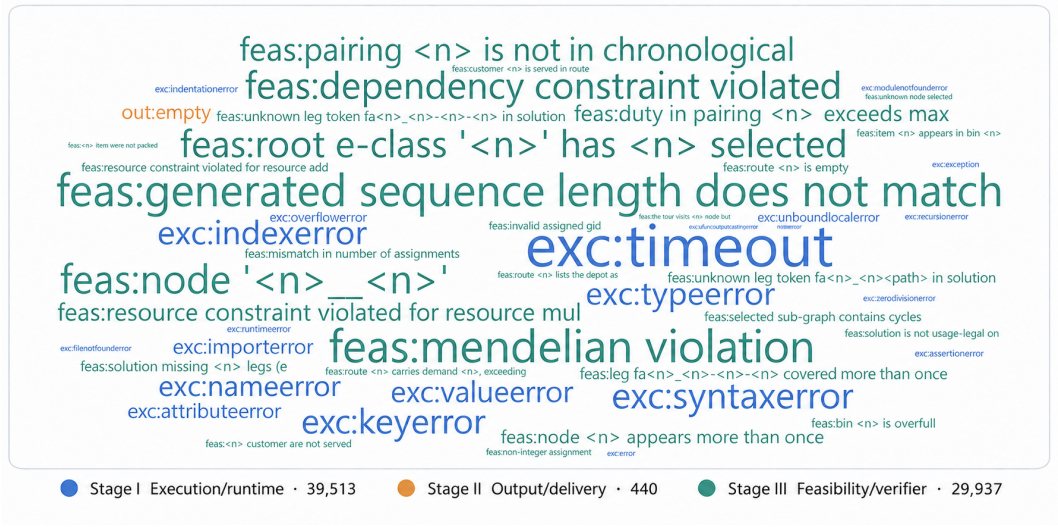


Figure 14: Visualization of the 53 normalized failure patterns. Word area scales with the square root of the observed frequency, while colors indicate the corresponding terminal evaluation stage: Stage I (execution/runtime), Stage II (output/delivery), and Stage III (feasibility/verifier).

Table 9: Problem-level Quality and Yield decomposition. Each entry reports Quality/Yield (%).

Method	Operator	Intra	Protein	Pedigree	Crew	E-Graph	BPP	CVRP	TSP	Avg.
EoH	90.9 / 84.8	71.6 / 72.9	73.7 / 61.8	81.2 / 59.4	64.6 / 58.6	41.9 / 43.7	99.4 / 100.0	88.1 / 86.7	80.7 / 40.4	76.9 / 67.6
ReEvo	71.3 / 66.1	63.0 / 54.7	67.1 / 53.4	61.3 / 62.4	38.0 / 19.8	26.9 / 27.9	99.4 / 93.3	83.2 / 75.4	57.4 / 32.9	63.1 / 54.0
HSEvo	72.9 / 64.3	57.4 / 59.3	68.2 / 65.8	56.5 / 43.0	44.5 / 23.5	40.2 / 41.7	99.4 / 99.6	81.0 / 78.3	57.8 / 29.3	64.2 / 56.1
HiFo-Prompt	85.9 / 78.5	62.5 / 62.0	73.7 / 59.6	68.1 / 74.1	46.1 / 29.0	35.1 / 28.3	99.6 / 97.9	78.3 / 74.7	52.4 / 32.1	66.9 / 59.6
RefineEvo	74.4 / 64.8	74.1 / 55.1	63.6 / 48.8	73.7 / 57.6	52.6 / 35.2	40.4 / 45.2	91.1 / 85.0	77.4 / 76.0	63.0 / 29.2	67.8 / 55.2
HeuriGym	91.2 / 93.1	70.1 / 71.0	69.4 / 82.2	69.0 / 41.1	65.7 / 40.1	50.0 / 55.0	99.4 / 100.0	91.5 / 83.3	65.5 / 51.8	74.6 / 68.6
DiSH-GWRS	88.4 / 100.0	74.2 / 69.9	79.8 / 84.6	71.5 / 59.4	77.1 / 75.6	54.6 / 58.9	99.4 / 100.0	89.3 / 97.6	85.9 / 88.2	80.0 / 81.6
DiSH-CRS	88.1 / 100.0	71.0 / 73.5	80.4 / 84.1	74.4 / 58.5	66.6 / 79.0	69.3 / 78.2	99.4 / 100.0	90.2 / 97.6	87.2 / 89.2	80.7 / 84.5
DiSH-GRS	86.3 / 97.2	69.4 / 77.9	81.1 / 81.6	67.2 / 64.8	71.9 / 81.2	61.6 / 66.1	99.4 / 100.0	88.8 / 97.8	86.9 / 84.4	79.2 / 83.5

G Additional Performance Analysis

This section provides additional analyses of DiSH beyond the aggregate QYI results. We first examine the contributions of Quality and Yield, including a closer analysis of Pedigree. We then study performance across problems and backbone models, as well as across shared initializations.

G.1 Quality–Yield Analysis

To better understand the source of the QYI improvements, we examine Quality and Yield separately at both the problem and backbone levels. All three DiSH variants improve both Quality and Yield, indicating that their QYI gains do not come at the expense of either component. Quality measures average solution quality over successfully verified test instances, while Yield measures the proportion passing verification. Tables 9 and 10 report problem- and backbone-level results as Quality / Yield (%). DiSH-CRS achieves average Quality and Yield of 80.7% and 84.5%, corresponding to relative improvements of 4.9% and 23.2% over the respective strongest non-DiSH averages, EoH's 76.9% and HeuriGym's 68.6%. Even the lowest averages across the three variants, 79.2% for Quality and 81.6% for Yield, exceed these non-DiSH maxima. Across backbones, all three variants consistently outperform every non-DiSH method in Yield, although their Quality advantages are not universal. Problem-level patterns also vary: on Pedigree, all three variants have lower Quality than EoH but not consistently lower Yield, motivating further analysis of this benchmark.

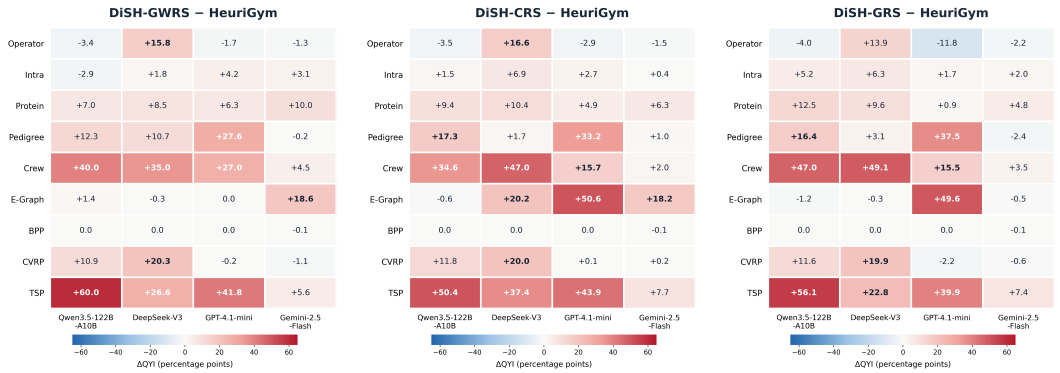
Analysis of the Pedigree Results. DiSH achieves the highest average QYI on all evaluated benchmarks except Pedigree. To better understand this exception, we compare the three DiSH variants with the baseline methods considered in this study. We use EoH as the main reference because it achieves the highest average QYI among the five AHD baselines (Table 1). Table 11 reports the corresponding Quality and Yield results. The results suggest that, relative to EoH, the lower QYI on

Table 10: Backbone-level Quality and Yield decomposition. Each entry reports Quality/Yield (%).

Method	Qwen3.5-122B-A10B	DeepSeek-V3	GPT-4.1-mini	Gemini-2.5-Flash	Avg.
EoH	77.2 / 71.4	69.1 / 56.8	75.6 / 69.7	85.8 / 72.4	76.9 / 67.6
ReEvo	56.3 / 45.9	53.5 / 43.7	64.2 / 56.9	78.4 / 69.3	63.1 / 54.0
HSEvo	58.2 / 52.4	54.7 / 44.6	64.3 / 55.3	79.6 / 72.0	64.2 / 56.1
HiFo-Prompt	69.4 / 64.0	55.5 / 45.0	60.1 / 60.9	82.5 / 68.4	66.9 / 59.6
RefineEvo	53.3 / 40.8	57.7 / 38.2	79.1 / 66.7	81.2 / 75.1	67.8 / 55.2
HeuriGym	71.2 / 68.1	73.5 / 62.5	70.4 / 67.9	83.4 / 76.0	74.6 / 68.6
DiSH-GWRS	82.5 / 80.0	78.2 / 80.0	76.5 / 82.2	82.8 / 84.3	80.0 / 81.6
DiSH-CRS	79.9 / 80.8	81.2 / 83.9	80.1 / 88.6	81.7 / 84.5	80.7 / 84.5
DiSH-GRS	80.4 / 84.1	76.0 / 81.5	80.5 / 85.8	79.8 / 82.3	79.2 / 83.5

Table 11: Quality and Yield on Pedigree for EoH and the three DiSH variants. Each entry reports Quality/Yield (%).

Method	Qwen3.5-122B-A10B	DeepSeek-V3	GPT-4.1-mini	Gemini-2.5-Flash	Avg.
EoH	84.0 / 67.4	83.3 / 47.4	66.1 / 67.4	91.5 / 55.6	81.2 / 59.4
DiSH-GWRS	75.1 / 48.9	68.4 / 67.4	62.5 / 61.5	79.9 / 60.0	71.5 / 59.4
DiSH-CRS	75.0 / 60.7	63.8 / 57.0	67.3 / 64.4	91.5 / 51.9	74.4 / 58.5
DiSH-GRS	62.1 / 65.2	60.6 / 63.7	76.1 / 65.2	69.9 / 65.2	67.2 / 64.8



(a) DiSH-GWRS (b) DiSH-CRS (c) DiSH-GRS

Figure 15: QYI differences between each DiSH variant and HeuriGym across problems and backbone models.

Pedigree is mainly attributable to lower Quality rather than consistently lower Yield. EoH achieves an average Quality of 81.2% and a Yield of 59.4%. DiSH-GWRS reaches the same Yield but with lower Quality, while DiSH-CRS achieves a similar Yield with a smaller reduction in Quality. Among the three DiSH variants, DiSH-GRS obtains the highest Yield at 64.8%, but its average Quality decreases to 67.2%. Overall, the three variants exhibit different balances between Quality and Yield on Pedigree, indicating that higher verification coverage does not necessarily lead to better solution Quality.

G.2 Performance across Problems and Backbones

To examine whether the aggregate gains are concentrated in particular problems or backbone models, we further compare each DiSH variant with HeuriGym across all combinations of problems and backbones. We quantify the improvement for each problem p and backbone b by its QYI gain over HeuriGym:

$$\Delta\mathrm{QYI}_{p,b} = \mathrm{QYI}_{\mathrm{DiSH},p,b} - \mathrm{QYI}_{\mathrm{HeuriGym},p,b}.$$

Figure 15 visualizes the resulting 36 differences for each DiSH variant. Each cell reports the QYI difference in percentage points, with a shared color scale centered at zero.

Table 12: Initialization-level paired comparisons between DiSH-CRS and the strongest non-DiSH method under each backbone. ΔQYI denotes DiSH-CRS minus the comparison method under the same initialization.

Backbone	Baseline	Init.	DiSH-CRS QYI (%)	Baseline QYI (%)	ΔQYI (%)
Qwen3.5-122B-A10B	EoH	1	84.5	69.1	+15.4
Qwen3.5-122B-A10B	EoH	2	78.7	66.1	+12.6
Qwen3.5-122B-A10B	EoH	3	69.6	67.5	+2.1
DeepSeek-V3	HeuriGym	1	87.3	68.9	+18.4
DeepSeek-V3	HeuriGym	2	85.8	71.1	+14.7
DeepSeek-V3	HeuriGym	3	68.6	48.3	+20.3
GPT-4.1-mini	EoH	1	88.0	64.5	+23.5
GPT-4.1-mini	EoH	2	78.3	68.1	+10.2
GPT-4.1-mini	EoH	3	81.8	73.8	+8.0
Gemini-2.5-Flash	HeuriGym	1	87.7	82.8	+4.9
Gemini-2.5-Flash	HeuriGym	2	72.4	65.2	+7.2
Gemini-2.5-Flash	HeuriGym	3	81.1	81.8	−0.7

Table 13: Summary of initialization-level paired QYI differences for the three DiSH variants. Each entry reports the mean ± standard deviation in percentage points over the three shared initializations, followed by the number of initializations on which the DiSH variant attains the higher QYI.

Variant	vs. EoH (Qwen)	vs. HeuriGym (DeepSeek)	vs. EoH (GPT)	vs. HeuriGym (Gemini)
DiSH-GWRS	$+10.5 \pm 8.5$, 3/3	$+13.1 \pm 4.3$, 3/3	$+9.1 \pm 6.3$, 3/3	$+4.3 \pm 7.1$, 2/3
DiSH-CRS	$+10.0 \pm 7.0$, 3/3	$+17.8 \pm 2.9$, 3/3	$+13.9 \pm 8.4$, 3/3	$+3.8 \pm 4.1$, 2/3
DiSH-GRS	$+12.5 \pm 6.6$, 3/3	$+13.8 \pm 2.9$, 3/3	$+12.0 \pm 12.7$, 3/3	$+1.3 \pm 8.3$, 2/3

The gains are distributed across multiple problems and backbones rather than being driven by a single setting. The largest improvements occur on TSP, Crew Pairing, and E-Graph, with gains exceeding 40 QYI points for several backbone models. Protein Design also exhibits consistently positive gains across all four backbones and all three DiSH variants. In contrast, BPP remains close to zero across nearly all settings, consistent with its near-saturated performance in the main results. Overall, these results indicate that the aggregate gains of DiSH arise from improvements across a broad range of problem–backbone settings rather than from a small number of favorable cases.

G.3 Performance across Initializations

Because all methods start from the same initial complete solver within each initialization, their performance can be compared directly at the initialization level. For each backbone, we compare DiSH against the strongest non-DiSH method according to its overall average QYI under that backbone. Since the comparison method is selected from the observed aggregate results and the twelve comparisons consist of four backbones with three shared initializations each, we report paired QYI differences and win counts descriptively rather than performing significance tests. Summary statistics are computed from the unrounded values. Table 12 reports the individual paired comparisons for DiSH-CRS, while Table 13 summarizes the corresponding results for all three DiSH variants.

The paired differences are positive on average for all three DiSH variants under every backbone, and each variant achieves a higher QYI in 11 of the 12 initialization-level comparisons. For DiSH-CRS, all three paired differences are positive under Qwen3.5-122B-A10B, DeepSeek-V3, and GPT-4.1-mini. Under Gemini-2.5-Flash, two of the three differences are positive, while the remaining difference is small (-0.7 percentage points). Overall, these results show that the aggregate gains persist across shared initializations rather than being driven by a single favorable initialization.

H Limitations and Future Work

Limitations. The main limitation of DiSH is that its control mechanism is designed primarily around feasibility, with less explicit control over solution quality. As the feasibility pass rate in-

creases, DiSH progressively narrows the revision scope to preserve verified behavior and reduce regressions. Once full feasibility is reached, revisions therefore become increasingly conservative. This improves refinement stability, but may also limit opportunities for further quality improvement. In addition, the current scope thresholds and direction strategies are manually specified. These rules are simple and interpretable, but may not adapt equally well to different problems, backbone LLMs, or refinement stages. Overall, DiSH provides effective control for stable feasibility, while quality-aware and more adaptive control remain open directions.

Future Work. In the future, we can extend DiSH toward more flexible control of both feasibility and solution quality. One direction is to continue improving solution quality after feasibility has been established, while explicitly protecting verified feasible behavior. Another direction is to learn the direction and scope policies from refinement trajectories. This could replace fixed thresholds and manually designed rules with controls that adapt to the current solver state, problem structure, and backbone LLM. A further extension is to combine such control with population-based automatic heuristic design, where multiple solver candidates are refined in parallel and can provide complementary search information. Together, these directions could make revision control more adaptive while supporting continued improvement beyond feasibility.