EVOLVING WITH FEASIBILITY: TURNING FAILURES INTO FEEDBACK FOR LLM-DRIVEN ALGORITHM EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly used to automate algorithm design by evolving executable solver programs. However, generated offspring solvers may fail to execute, exceed runtime limits, or violate problem constraints, preventing their algorithmic changes from being meaningfully evaluated. Such failures are particularly costly under limited candidate budgets: they waste generation opportunities, can terminate useful search trajectories, and may recur across generations. We propose **RevivEvo**, a feasibility-guided framework that treats feasibility not only as an evaluation outcome, but as feedback for evolution. RevivEvo uses parent-specific search and failure experience to prevent known risks during generation, type-aware bounded repair to recover infeasible offspring, and failure and repair memory to reuse experience across generations. Across 14 tasks from CO-Bench and HeuriGym with a budget of 20 candidates, RevivEvo increases the final candidate feasibility rate from 35.60% to 92.38% over the strongest feasibility baseline and improves the macro-averaged Test Avg. Score from 0.665 to 0.832 over the strongest performance baseline. Moreover, 52.4% of final selected solvers either result from repair or descend from a repaired candidate, showing that recovery can benefit subsequent evolution beyond immediate candidate repair. These results demonstrate that incorporating feasibility feedback throughout evolution improves both the reliability and effectiveness of LLM-driven algorithm design.

1 INTRODUCTION

Large language models (LLMs) are increasingly used to automate algorithm design through evolutionary search (Romera-Paredes et al., 2024; Liu et al., 2024; van Stein & Bäck, 2025). In each generation, an LLM modifies solver programs, evaluates the resulting offspring, and uses their performance to guide subsequent search. Recent methods improve this process through reflection, experience reuse, and structured exploration (Ye et al., 2024; Dat et al., 2025; Zheng et al., 2025; Chen et al., 2026b; Wu et al., 2026). As evolution moves toward complete solver programs (Chen et al., 2026a), however, offspring must not only improve solution quality but also remain executable and satisfy runtime and problem constraints. LLM-generated modifications can violate these requirements, making offspring infeasibility a bottleneck in algorithm evolution. Here, an offspring refers to a candidate solver generated from a parent during an evolutionary step.

Figure 1 illustrates three challenges caused by infeasible offspring. First, a modification can break the feasibility of a valid parent by overlooking its constraint-handling logic. Second, an implementation failure can prevent a potentially useful algorithmic change from being evaluated. Third, similar failures can recur across generations when previous failure experience is discarded. These failures are particularly costly under limited candidate budgets: they waste generation opportunities and can terminate useful search trajectories. This raises our central question: *how can LLM-driven algorithm evolution prevent, recover from, and learn from offspring infeasibility?*

We argue that feasibility should not be treated only as an evaluation outcome, but as feedback for evolution. Addressing the three challenges above requires using this feedback throughout the search process: generation should account for feasibility risks associated with the selected parent, recovery

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

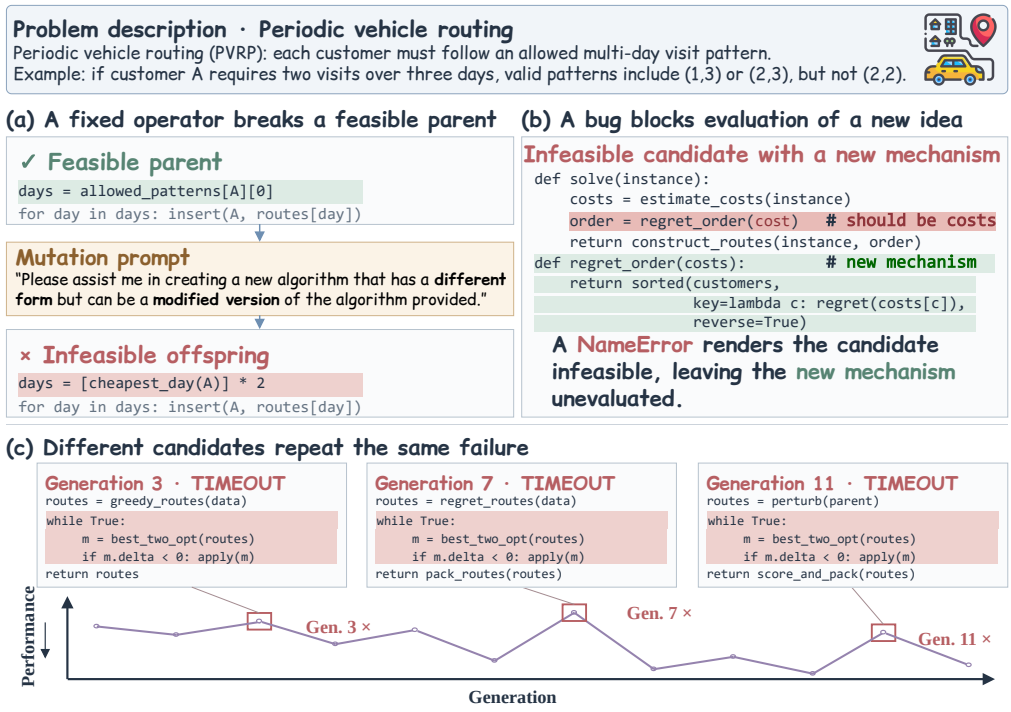

Figure 1: Three challenges caused by offspring infeasibility: a modification can break a feasible parent, a failure can prevent a potentially useful algorithmic change from being evaluated, and similar failures can recur across generations.

should preserve potentially useful modifications when failures occur, and failure and repair experience should guide subsequent generations.

To this end, we propose REVIVEVO, a feasibility-guided framework for LLM-driven algorithm evolution. **Parent-Specific Offspring Generation** uses the selected parent's search progress and failure history to guide revisions while avoiding known feasibility risks. **Type-Aware Bounded Repair** diagnoses code, timeout, and constraint failures and attempts to recover infeasible offspring under a fixed repair budget. **Failure and Repair Memory** retains failure lessons and repair experience for subsequent generation and recovery. Together, these components form a closed loop of failure prevention, candidate recovery, and experience reuse. We instantiate REVIVEVO on MCTS-AHD (Zheng et al., 2025), whose parent–offspring tree supports lineage-specific experience.

Across 14 tasks from CO-Bench (Sun et al., 2026) and HeuriGym (Chen et al., 2026c) under a 20-candidate budget, REVIVEVO improves both candidate feasibility and final solver quality. It achieves a 92.38% final candidate feasibility rate, compared with 35.60% for the strongest feasibility baseline, and improves the macro-averaged Test Avg. Score from 0.665 to 0.832 over the strongest performance baseline. Moreover, 52.4% of final selected solvers either result from repair or descend from a repaired candidate, showing that recovery can benefit subsequent evolution beyond immediate candidate repair. Our contributions are threefold:

- We identify offspring infeasibility as a bottleneck in LLM-driven algorithm evolution and formulate three challenges: preventing failures during generation, recovering infeasible offspring, and learning from failures across generations.
- We propose REVIVEVO, which addresses these challenges through parent-specific offspring generation, type-aware bounded repair, and failure and repair memory.
- Extensive experiments across CO-Bench and HeuriGym show that REVIVEVO improves both candidate feasibility and final solver quality, while lineage analysis demonstrates benefits of repair beyond immediate recovery.

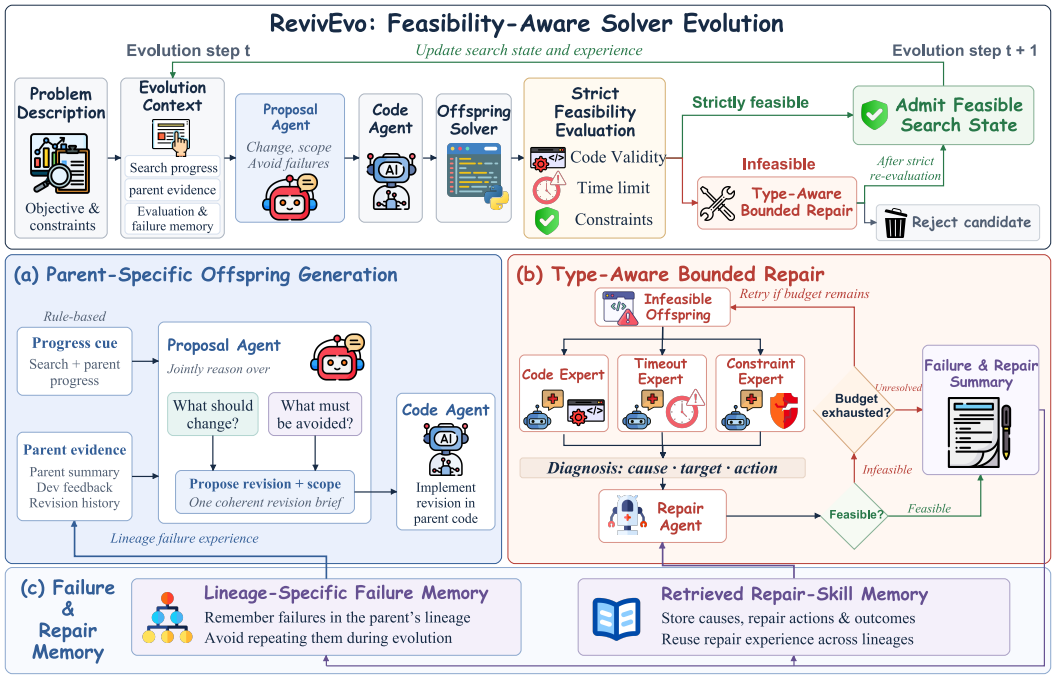

Figure 2: Overview of REVIVEVO. Feasibility feedback forms a closed loop across parent-specific generation, type-aware bounded repair, and failure and repair memory.

2 METHOD

Figure 2 presents REVIVEVO, a feasibility-aware evolutionary framework that turns offspring failures from terminal evaluation outcomes into feedback for subsequent search. An infeasible offspring may contain a useful algorithmic change whose evaluation is blocked by an implementation, runtime, or constraint-handling failure; discarding it also loses information that could prevent similar failures later. **Parent-Specific Offspring Generation** (§ 2.2) uses the selected parent's search and failure history to choose revisions while avoiding known risks. **Type-Aware Bounded Repair** (§ 2.3) diagnoses infeasible offspring and attempts to recover their underlying modifications. **Failure and Repair Memory** (§ 2.4) retains failure lessons and repair experience for later generations. Together, these components form a closed loop of failure prevention, candidate recovery, and experience reuse.

2.1 PROBLEM FORMULATION

Let $\mathcal{X}$ denote the instance space and $\mathcal{C}$ the space of complete candidate programs. A binary verifier $\mathcal{V}(x, y)$ determines whether an output y satisfies the constraints of instance x. For a program $p \in \mathcal{C}$ and an instance x, we define $v(p, x) = 1$ if p executes successfully within the prescribed time limit and produces an output y satisfying $\mathcal{V}(x, y) = 1$; execution failures, timeouts, and invalid outputs result in $v(p, x) = 0$. For an evaluation set $\mathcal{D}$, the strictly feasible program space is

$$\mathcal{C}_{\text{feas}}(\mathcal{D}) = \{p \in \mathcal{C} \mid v(p, x) = 1,\ \forall x \in \mathcal{D}\}. \tag{1}$$

A candidate is *feasible* if its program belongs to $\mathcal{C}_{\text{feas}}(\mathcal{D})$. Only feasible programs are admitted as valid states in the evolutionary search.

Within this feasible space, evolution aims to maximize solution quality on the development set $\mathcal{D}_{\text{dev}}$. Let $f(p, x) \in [0, 1]$ denote the task-specific normalized score assigned by the benchmark, with infeasible executions receiving a score of zero. We define

$$Q(p; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} f(p, x), \qquad p^\star \in \operatorname*{arg\,max}_{p \in \mathcal{C}_{\text{feas}}(\mathcal{D}_{\text{dev}})} Q(p; \mathcal{D}_{\text{dev}}). \tag{2}$$

The strict feasibility requirement creates a challenge for evolution: an infeasible offspring cannot enter the search even when its underlying modification may be useful. Simply discarding such candidates therefore loses both potentially valuable revisions and information about the failures that blocked their evaluation. REVIVEVO uses this information before generation to reduce recurring failures and after generation to recover potentially useful candidates.

2.2 PARENT-SPECIFIC OFFSPRING GENERATION

Predefined evolutionary operators specify generic transformation patterns, but the same revision strategy need not suit parents at different search states. A steadily improving lineage may benefit from refining its current mechanisms, whereas a stagnant lineage may require broader exploration; revisions that previously caused infeasibility should also be avoided. REVIVEVO therefore separates revision planning from code implementation, allowing parent-specific search and failure evidence to determine *what* to change before the Code Agent determines *how* to implement it.

For a selected parent p_t, evolutionary evidence $\mathcal{E}_t(p_t)$ summarizes its current program, evaluation feedback, and modification history. Quality and runtime feedback reveal improvement opportunities and computational bottlenecks, while previous modification outcomes identify promising directions and lineage-specific failure modes to avoid. A separate rule-based cue u_t controls the scope of the next revision based on recent search progress globally, along the parent's lineage, and among its direct offspring. Repeated improvements favor refinement of effective mechanisms, limited evidence motivates targeted exploration, and sustained stagnation encourages broader reconsideration of the current design. Given the task description $\mathcal{T}$, including its objective, constraints, and execution requirements, the Proposal Agent generates

$$b_t = \mathcal{A}_{\text{prop}}\big(\mathcal{E}_t(p_t), u_t, \mathcal{T}\big), \tag{3}$$

where b_t specifies the intended change, its scope, supporting evidence, and failure-avoidance requirements. The Proposal Agent thus converts heterogeneous evolutionary evidence into a concrete revision direction rather than directly editing program code. A separate Code Agent implements this proposal to produce the raw offspring

$$p_t^{(0)} = \mathcal{A}_{\text{code}}\big(p_t, b_t, \mathcal{T}\big). \tag{4}$$

2.3 TYPE-AWARE BOUNDED REPAIR

Even with failure-aware generation, an offspring may fail because of implementation errors, excessive runtime, or violated problem constraints. Immediately discarding it can lose a useful algorithmic modification because of a potentially repairable failure. Moreover, these failure types require different evidence and corrective reasoning: code failures expose execution errors, timeouts reflect computational bottlenecks, and constraint failures require relating verifier feedback to task requirements. REVIVEVO therefore separates failure diagnosis from code repair and uses type-specific experts to guide recovery.

Type-Aware Diagnosis. For the current program $p_t^{(r)}$ at repair round r, strict evaluation returns

$$e_t^{(r)} = \text{Eval}\Big(p_t^{(r)}\Big) \in \{\mathsf{feasible}, \mathsf{code}, \mathsf{timeout}, \mathsf{constraint}\}\,. \tag{5}$$

For an infeasible candidate, REVIVEVO collects failure evidence $\mathcal{F}_t^{(r)}$ according to the detected type and provides it to the corresponding Expert Agent:

$$\delta_t^{(r)} = \mathcal{A}_{\text{exp}}^{e_t^{(r)}}\left(\mathcal{F}_t^{(r)}\right), \qquad e_t^{(r)} \neq \mathsf{feasible}. \tag{6}$$

The evidence includes execution information for code failures, runtime characteristics for timeout failures, and verification feedback for constraint failures. The resulting diagnosis $\delta_t^{(r)}$ identifies the likely cause, repair targets, and recommended actions, separating reasoning about *why* the candidate failed from the subsequent code modification.

Bounded Repair. Given the diagnosis, the Repair Agent updates the infeasible candidate as

$$p_t^{(r+1)} = \mathcal{A}_{\text{rep}}\left(p_t^{(r)}, \delta_t^{(r)}\right). \tag{7}$$

The repaired candidate is re-evaluated on the same development instances under unchanged execution limits and constraint checks. If it remains infeasible, its latest failure type determines the next diagnosis and repair. Because repeatedly repairing one candidate can itself consume excessive inference and evaluation resources, a shared budget B_{rep} bounds the total repair rounds across all failure types. The candidate is admitted to the evolutionary search once it passes strict re-evaluation; otherwise, it is rejected when the budget is exhausted. Its repair trajectory, including diagnoses, repair actions, and outcomes, is retained for subsequent generation and repair.

2.4 Failure and Repair Memory

A failed offspring provides two distinct forms of experience: what should be avoided when generating future offspring, and how a similar failure may be repaired after it occurs. Because these forms of experience serve different decisions and have different scopes, RevivEvo maintains two complementary memories: lineage-specific failure memory for failure prevention and retrieved repair-skill memory for recovery reuse. Detailed memory update and reuse procedures are provided in Section B.3.

Lineage-Specific Failure Memory. Failure risks depend on the program being modified and its evolutionary history. After each generation-and-repair episode with a confirmed failure, we consolidate diagnoses, attempted repairs, and final outcomes into concise prevention lessons. A recovered child inherits these lessons along its lineage; if recovery fails, they are propagated back to the selected parent. The Proposal Agent incorporates the accumulated lessons into $\mathcal{E}_t(p_t)$ when constructing subsequent revision proposals, allowing known failure mechanisms to influence modification selection before implementation.

Retrieved Repair-Skill Memory. Repair experience can transfer beyond the lineage in which it was acquired. Episodes involving repair are summarized into reusable records linking failure conditions and diagnosed causes with corrective actions and observed outcomes. These records are shared across lineages within the same task and run. When a new failure occurs, related records are retrieved to guide the current repair process. Successful records provide corrective actions that previously restored feasibility, while unsuccessful records discourage repeating ineffective repairs under similar conditions. Together, the two memories feed failure-prevention experience back into offspring generation and recovery experience into future repair.

3 Experiments

3.1 Experimental Setup

Benchmarks. We evaluate RevivEvo on CO-Bench (Sun et al., 2026) and HeuriGym (Chen et al., 2026c). CO-Bench evaluates complete solvers for classical combinatorial optimization, whereas HeuriGym covers heuristic programs in scientific and engineering domains. We select seven tasks from each benchmark, covering diverse solver interfaces, constraint structures, and computational demands. Task descriptions and selection criteria are provided in Appendix A.1.

Baselines. We compare against direct LLM synthesis and agentic search methods. Direct synthesis uses Qwen 3.7 Flash, Doubao Seed 2.1 Turbo, DeepSeek V4 Flash, GLM 5.2, and DeepSeek V4 Pro, with each model generating one complete solver per run without execution feedback or iterative refinement. Agentic baselines include FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), HSEvo (Dat et al., 2025), MCTS-AHD (Zheng et al., 2025), HiFo-Prompt (Chen et al., 2026b), RefineEvo (Wu et al., 2026), and A2DEPT (Chen et al., 2026a). All agentic methods, including RevivEvo, use DeepSeek V4 Flash as the LLM backbone.

Implementation. Each method–task configuration is evaluated over three independent runs. Direct LLM synthesis generates one solver per run, whereas each agentic method receives $B = 20$

Table 1: **Overall performance on CO-Bench and HeuriGym measured by Test Avg. Score.** All agentic methods use DeepSeek V4 Flash with a budget of 20 candidates. Best and second-best values in each column are shown in **bold** and underlined, respectively.

Method	CO-Bench							HeuriGym							Macro avg.
	AL	BP1D	CWL	CDD	OSS	SC	PVRP	OS	CP	EG	IOP	PED	PSD	TM	
Direct LLM solvers															
Qwen 3.7 Flash	0.139	0.922	0.560	0.444	0.168	0.725	0.000	0.147	0.028	0.039	0.180	0.342	0.098	0.000	0.271
Doubao Seed 2.1 Turbo	0.204	0.565	0.705	0.440	0.470	0.719	0.000	0.333	0.303	0.333	0.398	0.165	0.108	0.000	0.339
DeepSeek V4 Flash	0.437	0.925	0.491	0.655	0.334	0.807	0.000	0.372	0.000	0.445	0.187	0.000	0.432	0.077	0.369
GLM 5.2	0.058	0.485	0.635	0.301	0.299	0.480	0.098	0.000	0.101	0.000	0.431	0.107	0.000	0.000	0.214
DeepSeek V4 Pro	0.194	0.620	0.490	0.534	0.232	0.781	0.186	0.504	0.368	0.099	0.328	0.104	0.349	0.000	0.342
Agentic baselines															
FunSearch	0.650	0.934	0.693	0.854	0.735	**0.886**	0.334	0.965	0.501	**0.876**	0.620	0.607	0.527	0.116	0.664
EoH	0.760	0.930	0.758	0.836	0.658	0.833	0.470	0.973	0.769	0.633	0.647	0.425	0.586	0.025	0.665
ReEvo	0.850	0.923	0.825	0.920	0.918	0.678	0.337	0.753	**0.903**	0.265	0.750	0.488	0.552	0.122	0.663
HSEvo	0.605	0.925	0.746	0.650	0.785	0.882	0.431	0.975	0.776	0.241	0.612	0.400	0.366	0.096	0.606
MCTS-AHD	0.757	0.932	0.706	0.838	0.529	0.748	0.156	0.323	0.261	0.092	0.588	0.708	0.457	0.000	0.507
HiFo-Prompt	0.830	0.940	0.719	0.741	0.717	0.870	0.179	0.982	0.607	0.687	0.508	0.828	0.523	0.068	0.657
RefineEvo	0.816	**0.950**	0.769	0.762	0.829	0.869	0.117	0.971	0.766	0.352	0.721	0.769	0.403	0.110	0.657
A2DEPT	0.428	0.941	0.665	0.478	0.905	0.805	0.188	0.919	0.648	0.436	0.665	0.527	0.541	**0.258**	0.600
Ours															
REVIVEVO	**0.857**	0.944	**0.849**	**0.935**	**0.951**	0.877	**0.872**	**0.986**	0.860	0.717	**0.770**	**0.842**	**0.961**	0.229	**0.832**

CO-Bench: AL: aircraft landing; BP1D: one-dimensional bin packing; CWL: capacitated warehouse location; CDD: common due-date scheduling; OSS: open-shop scheduling; SC: set covering; PVRP: periodic vehicle routing.
HeuriGym: OS: operator scheduling; CP: crew pairing; EG: e-graph extraction; IOP: intra-op parallelism; PED: pedigree; PSD: protein sequence design; TM: technology mapping.

candidate solvers per run. Each generated solver counts as one candidate regardless of feasibility. In REVIVEVO, repair updates the same candidate without consuming another candidate slot. All reported experiments start without externally supplied solver programs. The candidate budget therefore measures the ability of each method to obtain feasible, high-quality solvers under limited generation opportunities. All methods use identical development/test splits, evaluators, score normalization, and per-instance timeouts. Evolution uses only the development set. After search, the highest-scoring feasible development program is evaluated once on the test set. If no strictly feasible candidate is found, the candidate with the highest partial score is evaluated on the test set. Further details appear in Appendix A.3.

3.2 MAIN RESULTS

Table 1 reports final solver performance across all 14 tasks. REVIVEVO achieves the highest macro-averaged Test Avg. Score of 0.832, compared with 0.665 for the strongest agentic baseline, EoH, corresponding to a 25.1% relative improvement. It ranks first on nine tasks and among the top two on thirteen, indicating that the aggregate gain is not driven by a small subset of tasks.

These gains are particularly relevant under the fixed 20-candidate budget, where an infeasible offspring consumes a generation opportunity without contributing a valid search state. This issue is pronounced on tasks with coupled feasibility requirements; for example, PVRP jointly constrains multi-day visit patterns, routing, and vehicle capacities, so modifying one part of the solver can invalidate another. REVIVEVO addresses such losses through both failure-aware generation and recovery. We next examine these mechanisms separately and assess whether improved feasibility translates into more effective evolution.

3.3 ANALYSIS OF REPAIR

Evolutionary contribution. A useful repair mechanism should not merely make invalid programs executable; recovered candidates should remain competitive enough to contribute to evolution. Figure 3(a) shows that 72.2% of comparable repaired offspring match or outperform their feasible parents on the development objective, with 41.9% strictly improving. Repair therefore often preserves the algorithmic value of a failed modification rather than merely restoring feasibility.

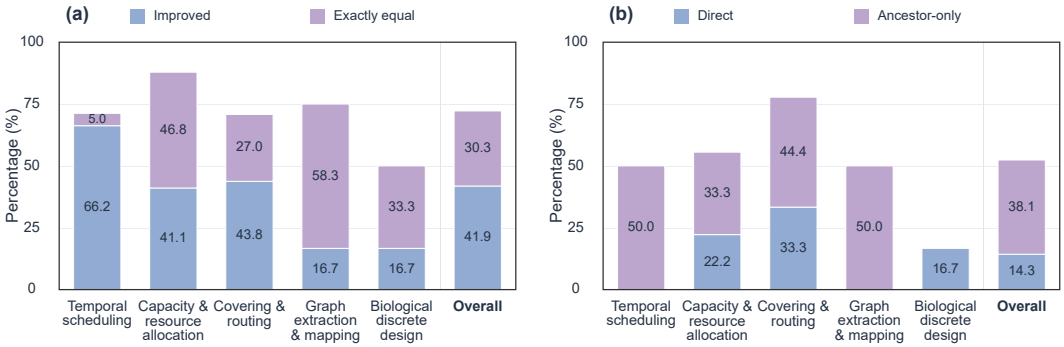

Figure 3: Evolutionary impact of repair: (a) quality comparison between repaired offspring and their feasible parents and (b) direct and lineage-based contributions of repaired candidates to final selected solvers.

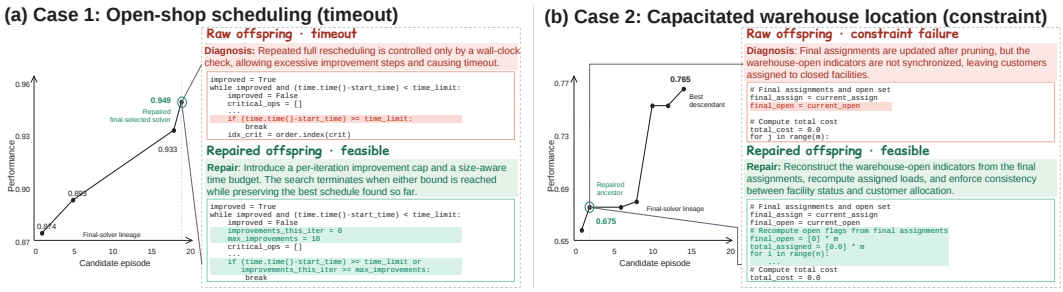

Figure 4: Repair case studies: (a) direct recovery with a repaired offspring selected as the final-best solver and (b) lineage recovery with a repaired ancestor producing a final-best descendant.

The benefit also extends beyond immediate recovery. Figure 3(b) shows that 52.4% of final selected solvers have repair in their lineage: 14.3% are directly repaired candidates, while 38.1% descend from repaired ancestors. The larger lineage contribution shows that recovery can preserve evolutionary branches whose later descendants become competitive, a benefit missed by considering only whether a repaired candidate is immediately selected.

Case study. Figure 4 illustrates these two contribution patterns. In open-shop scheduling, adding iteration and time-budget safeguards resolves a timeout while retaining the underlying search strategy, and the repaired program is selected as the final solver. In capacitated warehouse location, reconstructing facility-opening indicators from final customer assignments resolves a constraint inconsistency; the recovered candidate remains in the search and later produces a higher-quality descendant. These cases illustrate how repair can preserve an algorithmic modification while correcting the failure that blocks its evaluation.

Failure analysis. Table 2 shows that infeasibility remains substantial for all baselines: the highest baseline feasibility rate is 35.60%, compared with 92.38% for REVIVEVO. Timeout is the largest failure category for most baselines, while REVIVEVO reduces code errors, timeouts, and constraint violations to 0.60%, 4.29%, and 2.74%, respectively. The improvement therefore spans all three failure types rather than eliminating a single source of infeasibility. Together with the lineage results

Table 2: Candidate outcomes after each method's complete pipeline (%).

Method	Feasible↑	Code↓	Timeout↓	Constraint↓
FunSearch	17.38	20.95	41.31	20.36
EoH	23.10	19.76	40.71	16.43
ReEvo	<u>35.60</u>	18.10	<u>32.98</u>	13.33
HSEvo	17.74	33.33	34.76	14.17
MCTS-AHD	20.60	<u>16.07</u>	45.12	18.21
HiFo-Prompt	17.14	23.45	47.74	<u>11.67</u>
RefineEvo	20.83	19.40	46.31	13.45
A2DEPT	10.71	39.17	40.24	9.88
REVIVEVO	**92.38**	**0.60**	**4.29**	**2.74**

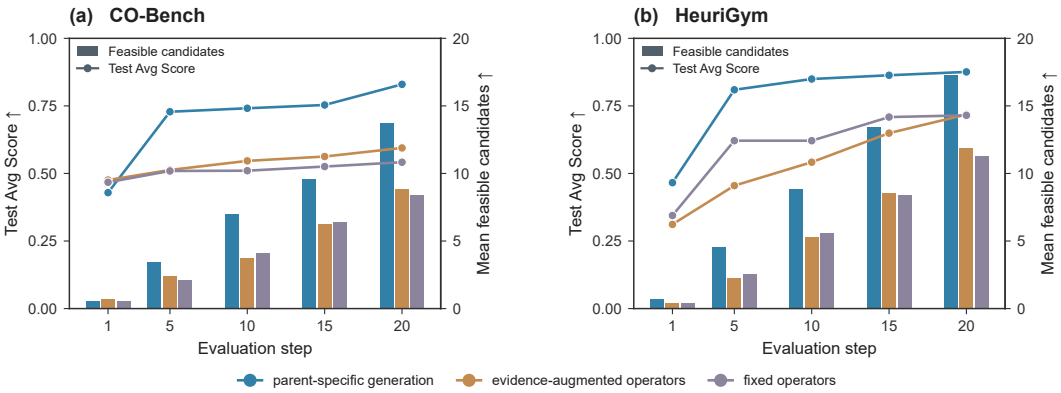

Figure 5: Impact of parent-specific generation on solution quality and feasibility without repair.

above, this indicates that improved feasibility preserves candidates that remain useful to subsequent evolution.

3.4 Analysis of Parent-Specific Generation

We next isolate whether parent-specific generation prevents failures before repair is invoked. Repair is disabled for all variants in this analysis. Figure 5 shows that parent-specific generation yields both higher final Test Avg. Scores and more feasible candidates than evidence-augmented and fixed operators, reaching an average of 13.71 feasible candidates on CO-Bench and 17.29 on HeuriGym within the 20-candidate budget. The advantage over evidence-augmented operators is particularly informative: providing parent evidence alone is less effective than first converting that evidence into a concrete revision proposal. This supports separating revision planning from code implementation rather than requiring the Code Agent to infer both simultaneously.

3.5 Ablation Studies

Table 3 isolates the components responsible for prevention and recovery. Initial CFR reflects pre-repair feasibility, while RepairSR reflects repair effectiveness. Removing failure memory or the Proposal Agent lowers initial CFR by 8.93 and 6.79 percentage points, respectively, with the latter causing the larger drop in final quality. For repair, removing diagnostic experts reduces RepairSR by 29.36 points, compared with 15.74 points without repair memory. These results support the intended division of roles: memory transfers prior failure and repair experience, while proposal and diagnosis explicitly reason about generation and recovery decisions.

Table 3: Component ablations of generation and repair.

Method	Rate↑	Test Avg.↑
Generation: Initial CFR (%)		
Full	80.00	0.824
w/o Failure Memory	71.07	0.760
w/o Proposal Agent	73.21	0.727
Repair: RepairSR (%)		
Full	79.84	0.824
w/o Expert LLM	50.48	0.726
w/o Repair Memory	64.10	0.762

3.6 Efficiency Analysis

Figure 6 examines whether the gains can be explained by additional search or LLM cost. In panel (a), ReviveEvo with 20 candidates achieves a higher macro Test Avg. Score than EoH, ReEvo, and RefineEvo with 80 candidates on the evaluated task subset, showing that increasing baseline generation opportunities does not close the performance gap within this budget range. Panel (b) accounts for the additional LLM calls introduced by proposal generation, diagnosis, and repair. Despite this overhead, ReviveEvo achieves the highest score with the lowest token cost per feasible candidate among the compared methods. The additional reasoning cost is therefore offset by a higher yield of feasible programs.

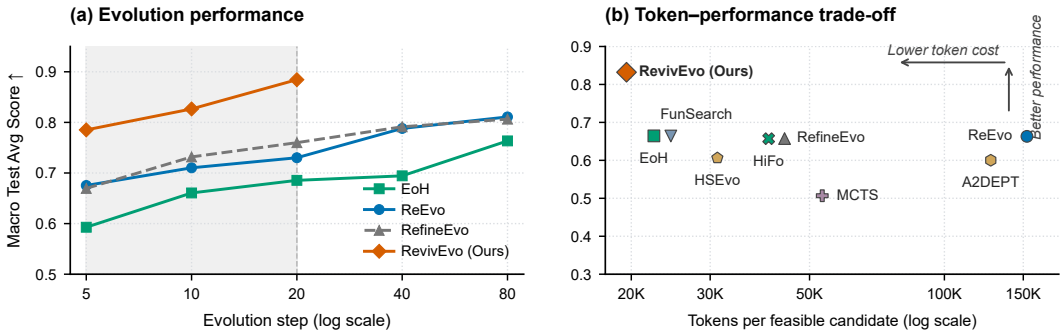

Figure 6: Search efficiency analysis: (a) performance scaling with candidate budget and (b) performance–cost trade-off measured by token cost per feasible candidate.

4 Related Work

LLM-Driven Automated Algorithm Design. Automated heuristic design searches algorithms rather than individual solutions (Burke et al., 2013). Language Model Crossover generates textual-parent variations (Meyerson et al., 2024), while FunSearch and EoH evolve heuristic code and ideas (Romera-Paredes et al., 2024; Liu et al., 2024). ReEvo, MCTS-AHD, and RoCo use reflection, tree-structured exploration, and multi-agent collaboration, respectively (Ye et al., 2024; Zheng et al., 2025; Xu et al., 2025). LLaMEA, A2DEPT, and AlphaEvolve explore metaheuristics, complete solvers, and evaluator-guided code optimization, respectively (van Stein & Bäck, 2025; Chen et al., 2026a; Novikov et al., 2025). We address offspring infeasibility through prevention and recovery during evolution.

Adaptive and Experience-Guided Evolution. Adaptive operator selection uses search feedback (Fialho et al., 2010); mutation prompting controls code modification scope (Yin et al., 2025). EvoPrompt evolves task prompts (Guo et al., 2024), while Promptbreeder also evolves their mutation prompts (Fernando et al., 2024). Self-Refine uses iterative self-feedback, while Reflexion, ExpeL, and Voyager reuse textual experience or executable skills (Madaan et al., 2023; Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024). Within AHD, HiFo-Prompt combines population-level guidance with reusable principles, and RefineEvo integrates operator planning with positive and negative parent–offspring experience (Chen et al., 2026b; Wu et al., 2026). We formulate revisions from parent-specific progress and lineage failures, considering improvement opportunities, revision scope, and failure risks before implementation.

Feasibility-Aware Algorithm Synthesis. Classical constraint handling and repair address invalid solutions (Deb, 2000; Salcedo-Sanz, 2009); generated solvers must satisfy execution and runtime requirements. CodeRL uses critic and unit-test feedback (Le et al., 2022), while CodeT uses generated tests to select candidates (Chen et al., 2023). Self-Edit and self-debugging use execution feedback for program revision (Zhang et al., 2023; Chen et al., 2024). Repair benefits depend on feedback quality and computational cost (Olausson et al., 2024). A2DEPT incorporates program maintenance into evolution (Chen et al., 2026a). We integrate failure prevention, type-aware bounded repair, and experience reuse across evolutionary search. Strict re-evaluation governs admission, while separate failure and repair memories inform subsequent generation and recovery.

5 Conclusion

We introduce RevivEvo, a feasibility-aware framework for LLM-driven algorithm evolution. It integrates parent-specific generation, type-aware bounded repair, and separate failure and repair memories to support both failure prevention and candidate recovery. Across 14 tasks from CO-Bench and HeuriGym, RevivEvo substantially improves candidate feasibility and final solver quality under limited search budgets. Furthermore, repaired candidates contribute not only through direct recovery but also by preserving productive descendant lineages. These results demonstrate that feasibility can serve as an effective search signal for guiding algorithm evolution.

AI USE STATEMENT

Generative AI tools were used to support literature search and discovery, research ideation, code implementation, figure and table preparation, manuscript drafting, and language refinement. The authors reviewed and revised all AI-assisted content, verified the reported results, and assume full responsibility for the accuracy and integrity of the final work.

REPRODUCIBILITY STATEMENT

The core workflow of REVIVEVO is described in the Method section, with additional methodological details and prompt templates provided in Appendices B and C, respectively. Implementation details and hyperparameter settings are provided in Appendix A.3, while baseline configurations are documented in Appendix A.2. The core implementation, benchmark data, results from the main experiments, and instructions for running REVIVEVO are available in the anonymous repository at `https://anonymous.4open.science/r/RevivEvo-5F91/`.

REFERENCES

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. doi: 10.1057/jors.2013.71.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=ktrw68Cmu9c`.

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. *arXiv preprint arXiv:2604.24043*, 2026a. URL `https://arxiv.org/abs/2604.24043`.

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. URL `https://openreview.net/forum?id=imSLzfZ6av`.

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*, 2026c. URL `https://openreview.net/forum?id=HWxHUO15Yy`.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=KuPixIqPiq`.

Gordon V. Cormack, Charles L. A. Clarke, and Stefan Buettcher. Reciprocal rank fusion outperforms Condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 758–759. ACM, 2009. doi: 10.1145/1571941.1572114. URL `https://doi.org/10.1145/1571941.1572114`.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. HSEvo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 26931–26938, 2025. doi: 10.1609/aaai.v39i25.34898. URL `https://ojs.aaai.org/index.php/AAAI/article/view/34898`.

Kalyanmoy Deb. An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186(2–4):311–338, 2000. doi: 10.1016/S0045-7825(99)00389-8.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 13481–13544, 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 60(1–2):25–64, 2010. doi: 10.1007/s10472-010-9213-y.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ZG3RaNIsO8.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pp. 282–293. Springer Berlin Heidelberg, 2006. doi: 10.1007/11871842_29. URL https://doi.org/10.1007/11871842_29.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223. PMLR, 2024. URL https://proceedings.mlr.press/v235/liu24bs.html.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Elliot Meyerson, Mark J. Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K. Hoover, and Joel Lehman. Language model crossover: Variation through few-shot prompting. *ACM Transactions on Evolutionary Learning and Optimization*, 4(4), 2024. doi: 10.1145/3694791.

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

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=y0GJXRungR.

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. doi: 10.1038/s41586-023-06924-6.

Sancho Salcedo-Sanz. A survey of repair methods used as constraint handling techniques in evolutionary algorithms. *Computer Science Review*, 3(3):175–192, 2009. doi: 10.1016/j.cosrev.2009.07.001.

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, pp. 8634–8652, 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Weiwei Sun, Shengyu Feng, Shanda Li, and Yiming Yang. CO-Bench: Benchmarking language model agents in algorithm search for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 33126–33134, 2026. doi: 10.1609/aaai.v40i39.40596. URL https://ojs.aaai.org/index.php/AAAI/article/view/40596.

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. doi: 10.1109/TEVC.2024.3497793. URL https://doi.org/10.1109/TEVC.2024.3497793.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

Yang Wu, Junran Pan, Yifan Zhang, Ning Xu, Fanshuo Zeng, and Jian Cheng. RefineEvo: Planning-guided heuristic evolution with bidirectional experience. *arXiv preprint arXiv:2607.11358*, 2026. URL https://arxiv.org/abs/2607.11358.

Jiawei Xu, Feng-Feng Wei, and Wei-Neng Chen. RoCo: Role-based LLMs collaboration for automatic heuristic design, 2025. URL https://arxiv.org/abs/2512.03762.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1381. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/4ced59d480e07d290b6f29fc8798f195-Abstract-Conference.html.

Haoran Yin, Anna V. Kononova, Thomas Bäck, and Niki van Stein. Controlling the mutation in large language models for the efficient evolution of algorithms. In *Applications of Evolutionary Computation*, volume 15613 of *Lecture Notes in Computer Science*, pp. 403–417, Cham, 2025. Springer. doi: 10.1007/978-3-031-90065-5_25. URL https://link.springer.com/chapter/10.1007/978-3-031-90065-5_25.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-Edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 769–787. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.acl-long.45. URL https://aclanthology.org/2023.acl-long.45/.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte Carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 78338–78373. PMLR, 2025. URL https://proceedings.mlr.press/v267/zheng25o.html.

A Experimental Details

A.1 Benchmark Tasks

We evaluate seven tasks from CO-Bench (Sun et al., 2026) and seven tasks from HeuriGym (Chen et al., 2026c), covering diverse solution representations, objectives, and feasibility requirements. Table 5 groups these tasks into five families for cross-task analysis. This grouping is descriptive, and our evaluation is limited to the selected 14 tasks.

Table 4 reports instance counts and per-instance solver time limits. For CO-Bench, we retain the development/test assignments specified in the task configurations. For HeuriGym, we construct a frozen, scale-stratified split from the original demonstration and evaluation instances: development instances are selected based on input-derived scale and structural features, while the remaining instances are reserved for testing.

A.1.1 CO-Bench Tasks

Aircraft landing. Each instance specifies aircraft landing windows, target landing times, earliness and lateness penalties, available runways, and pairwise separation requirements. The solver assigns a landing time and runway to each aircraft to minimize the total weighted deviation from target times. A feasible schedule satisfies all landing windows and separation requirements between aircraft assigned to the same runway. The output maps each aircraft to its assigned landing time and runway.

One-dimensional bin packing. Given item sizes and a common bin capacity, the solver partitions items into bins while minimizing the number of bins used. Each item must appear exactly once, and the total item size in each bin cannot exceed its capacity. The output contains the number of bins and the item indices assigned to each bin. This is an offline packing task where all items are available during solution construction.

Capacitated warehouse location. Each instance provides potential warehouses with opening costs and capacities, together with customer demands and assignment costs. The evaluated variant allows customer demand to be split among multiple open warehouses. The solver returns warehouse-opening decisions and a nonnegative allocation matrix. All demands must be fully satisfied, warehouse capacities cannot be exceeded, and closed warehouses cannot supply positive demand. For opening decisions z_i and allocated quantities q_{ji}, the evaluator minimizes

$$\sum_i f_i z_i + \sum_{j:d_j>0} \sum_i c_{ji} \frac{q_{ji}}{d_j},$$

where f_i denotes the opening cost, d_j denotes customer demand, and c_{ji} denotes the assignment cost for supplying the customer's entire demand from warehouse i.

Common due-date scheduling. Each job has a processing time and separate earliness and tardiness penalty coefficients. All jobs share the due date $d = \lfloor h \sum_i p_i \rfloor$, where p_i denotes processing time and h is the due-date factor, set to 0.6 by default. The solver returns a job permutation, which the evaluator processes consecutively on a single machine from time zero. The objective minimizes total weighted earliness and tardiness relative to d. Feasibility requires each job to appear exactly once.

Open-shop scheduling. Each instance specifies jobs, machines, and operation processing times, without a fixed operation order within each job. The solver assigns a nonnegative start time to every operation and minimizes the makespan. Operations within the same job cannot overlap, and operations assigned to the same machine must also be mutually exclusive. The output is an operation start-time matrix.

Set covering. An instance consists of elements to cover, candidate subsets, and a cost for each subset. The solver selects subsets with minimum total cost such that every element is covered at least once. In the benchmark incidence representation, elements correspond to rows and candidate

subsets correspond to columns. The output lists selected column indices; leaving any row uncovered makes the solution infeasible.

Periodic vehicle routing. Each customer has a location, demand, and a set of admissible service-day patterns over a multi-day planning horizon. The solver selects one pattern for each customer and constructs daily vehicle routes to minimize total Euclidean travel distance. Daily visits must match the selected service patterns and satisfy vehicle capacities and fleet availability constraints. Each route starts and ends at the depot without intermediate depot visits. The output specifies both selected patterns and daily routes.

A.1.2 HEURIGYM TASKS

Operator scheduling. The input is a computation graph with operation dependencies, execution delays, and available functional units for each type. The solver assigns a start cycle to each operation to minimize the overall completion time. Each predecessor must finish before its successor starts, and the number of simultaneously active operations of each resource type cannot exceed the available units. The output maps operation identifiers to their assigned start cycles.

Airline crew pairing. Given dated flight legs and crew cost rates, the solver groups flights into pairings that cover each leg exactly once. The verifier checks chronological consistency and limits each duty to 14 hours, 10 block hours, and six flight legs. Consecutive duties must be separated by at least nine hours of rest. The simplified benchmark permits non-base starts and unrestricted ending airports, with an additional positioning cost for pairings starting away from the base. The objective minimizes duty, block-hour, and positioning costs. Each output line lists the flight-leg identifiers contained in one pairing.

E-graph extraction. An e-graph represents alternative equivalent expressions as e-nodes grouped into equivalence classes. Given node costs and designated root classes, the solver selects an extraction with minimum total node cost. A valid extraction selects exactly one node from each root class and each child class required by the selected nodes. The selected dependencies must also be acyclic. The output is the collection of selected e-node identifiers.

Intra-operator parallelism. Each computation node has alternative execution strategies with associated costs and memory requirements. Edges incur additional costs determined by the strategies selected at their endpoints. The solver chooses one strategy per node to minimize the sum of node and edge costs. Each node consumes memory over a fixed activity interval, and the combined usage at every time point must remain within the instance memory limit. The output lists the selected strategy index for each node.

Pedigree analysis. The input specifies family relationships and partially observed genotypes. The solver assigns an allele pair to each individual while satisfying Mendelian inheritance constraints. With two known parents, the child's assigned genotype must contain one allele inherited from each parent. Among consistent assignments, the objective minimizes the number of fully observed genotypes that require changes. Observation disagreement is therefore treated as a quality cost rather than a hard violation. The output provides genotype-domain indices in the individual ordering expected by the evaluator.

Protein sequence design. HeuriGym formulates this task using the Grand Canonical (GC) model of protein sequence design. Given a target protein structure in PDB format, the solver produces a binary hydrophobic/polar sequence over $\{H, P\}$. In this formulation, the objective is to minimize

$$\Phi(S) = \alpha \sum_{i,j \in S_H} g(d_{ij}) + \beta \sum_{i \in S_H} s_i,$$

where S_H denotes the hydrophobic positions, d_{ij} is the distance between residue side-chain centers, and s_i is the solvent-accessible area of residue i. The contact function g and coefficients $\alpha < 0$ and $\beta > 0$ favor hydrophobic contacts while penalizing solvent-exposed hydrophobic residues. The output contains one H/P label for each residue represented in the target structure.

Table 4: Instance counts and per-instance solver time limits.

Task	Dev	Test	Timeout (s)
CO-Bench			
Aircraft landing	20	29	10
One-dimensional bin packing	40	120	10
Capacitated warehouse location	12	37	10
Common due-date scheduling	21	49	10
Open-shop scheduling	30	30	10
Set covering	38	49	10
Periodic vehicle routing	5	5	10
CO-Bench subtotal	166	319	—
HeuriGym			
Operator scheduling	6	18	10
Airline crew pairing	5	9	10
E-graph extraction	6	17	10
Intra-operator parallelism	6	22	60
Pedigree analysis	5	15	10
Protein sequence design	6	18	10
Technology mapping	7	24	10
HeuriGym subtotal	41	123	—
Total	**207**	**442**	—

Table 5: Task families used in cross-task analyses. Tasks are grouped by their primary optimization structure.

Category	CO-Bench	HeuriGym
Temporal scheduling	Aircraft landing; Common due-date scheduling; Open-shop scheduling	Operator scheduling
Capacity & resource allocation	One-dimensional bin packing; Capacitated warehouse location	Intra-operator parallelism
Covering & routing	Set covering; Periodic vehicle routing	Airline crew pairing
Graph extraction & mapping	—	E-graph extraction; Technology mapping
Biological discrete design	—	Pedigree analysis; Protein sequence design
Total	**7 tasks**	**7 tasks**

Technology mapping. The input is a Boolean network in BLIF format. The solver produces an equivalent network implemented with lookup tables containing at most six inputs, while minimizing the number of LUTs. The generated network must preserve circuit functionality and satisfy the LUT input limit. The output is a mapped BLIF network. The benchmark uses ABC to verify functional equivalence and obtain the LUT count.

A.2 Baseline Configurations

We evaluate RevivEvo against two categories of competitors: direct LLM solvers with different LLM backbones and eight agentic search methods. The compared methods and their experimental configurations are described below. FunSearch, ReEvo, EoH, and MCTS-AHD use the implementations distributed with CO-Bench. HSEvo, HiFo-Prompt, RefineEvo, and A2DEPT use the authors' implementations through benchmark-specific adapters. All agentic baselines use DeepSeek

Table 6: Principal baseline settings and nominal allocation of the 20-candidate budget. Initialization and subsequent generation share the same budget; the final generation batch is truncated when necessary.

Method	Search size	Initial candidates	Subsequent generation	Total budget
FunSearch	10 islands	–	20 sequential generations from an initially empty program database	20
ReEvo	Population: 10	4	16 candidates through reflection-guided crossover and mutation	20
EoH	Population: 5	10	10 candidates through the configured evolutionary operators	20
MCTS-AHD	Elite population: 10	4	16 candidates through tree-guided expansion	20
HSEvo	Population: 2	6	14 candidates: $2 \times (2 + 1 + 4)$ from crossover, mutation, and harmony search	20
HiFo-Prompt	Population: 3	6	14 candidates in successive operator batches of $3, 3, 3, 3, 2$	20
RefineEvo	Population: 3	9	11 candidates allocated to planner-selected operator batches	20
A2DEPT	Parent set: 5	n_{init}	$20 - n_{\text{init}}$ candidates, with two offspring per selected parent	20

Notes. For FunSearch, "–" denotes the absence of a separately allocated initialization phase. For A2DEPT, n_{init} denotes the candidates consumed while initializing its five strategies. HSEvo's nominal schedule comprises two rounds, each with two crossover candidates, one mutation candidate, and four harmony-search evaluations.

V4 Flash to generate complete solvers. The shared LLM gateway uses a default temperature of 0.7 and an output-token limit of 8,192, with thinking mode disabled.

Table 6 summarizes the main search settings and the allocation of the 20-candidate budget. Population sizes are adjusted to accommodate both initialization and subsequent search under the fixed budget. HSEvo and RefineEvo maintain a 3:1 initialization-to-population ratio, whereas EoH and HiFo-Prompt initialize with twice their population size. Initialization consumes the same budget as subsequent generation, and the final batch is restricted to the remaining budget.

Direct LLM synthesis. We follow CO-Bench's DirectAnswer procedure, which generates one complete solver from the task description, required interface, and per-instance time limit. For HeuriGym, the prompt includes the prescribed program template and file-based input/output requirements. Each run uses one generation call, with a requested temperature of 0.7 and an output-token limit of 8,192. The same generated program is evaluated on development and test instances without feedback-driven revision, repair, format retries, or selection among multiple candidates.

FunSearch. FunSearch (Romera-Paredes et al., 2024) alternates LLM-based program generation with execution-based evaluation. It maintains an island-based database of valid, scored programs and preferentially samples high-performing examples to guide subsequent generations.

ReEvo. ReEvo (Ye et al., 2024) integrates evolutionary search with LLM-based reflection. Short-term reflection compares the relative performance of parent heuristics to guide crossover, while long-term reflection aggregates previous insights to guide elitist mutation.

EoH. EoH (Liu et al., 2024) represents heuristics as natural-language "thoughts" and executable code. It co-evolves these representations, allowing heuristic ideas and implementations to be revised jointly.

MCTS-AHD. MCTS-AHD (Zheng et al., 2025) organizes generated heuristics in a search tree and uses Monte Carlo tree search to guide expansion. The tree preserves evolutionary trajectories, enabling further exploration of temporarily underperforming heuristics.

HSEvo. HSEvo (Dat et al., 2025) combines LLM-driven genetic evolution with diversity-driven harmony search. It evaluates heuristic diversity and applies harmony-search updates to balance exploration and exploitation.

HiFo-Prompt. HiFo-Prompt (Chen et al., 2026b) combines hindsight and foresight in LLM-based heuristic evolution. Its Evolutionary Navigator adapts search guidance according to population dynamics, while its Insight Pool extracts reusable design principles from successful heuristics.

RefineEvo. RefineEvo (Wu et al., 2026) uses a Planner to schedule evolutionary operators and trigger refinement based on the search state. A Reflector distills positive and negative experience into a bidirectional pool for subsequent generation.

A2DEPT. A2DEPT (Chen et al., 2026a) evolves complete solver programs through tree-structured search. It combines simulated-annealing-based parent–child acceptance with supplementary Boltzmann sampling, applies adaptively scheduled hierarchical operators, and repairs missing dependencies through a program-maintenance loop.

A.3 Implementation Details

RevivEvo configuration. All RevivEvo experiments use DeepSeek V4 Flash as the LLM backbone. The main experiments are repeated with three random seeds: 2024, 2025, and 2026. Each full task–seed run has a budget of 20 candidates, including initialization. The search maintains a population of at most 10 programs. RevivEvo adopts MCTS-AHD (Zheng et al., 2025) as the agent backbone and follows its tree-based search procedure. The underlying search uses normalized UCT selection and progressive widening. The initial exploration coefficient is $\lambda_0 = 0.1$, which decreases linearly as the candidate budget is consumed. The widening exponent is $\alpha = 0.5$, and tree selection descends at most 10 levels. After obtaining a feasible initial program, the search performs up to four bootstrap attempts and subsequently generates one candidate per tree expansion. The Proposal Agent considers at most eight recent trajectory outcomes and generates a revision brief of at most 150 words. Lineage failure memory retains the four most recent incidents, while repair retrieves at most two relevant repair records. The repair budget is set to $B_{\text{rep}} = 5$ for each infeasible candidate. Proposal, code generation, diagnosis, and repair use temperatures of 0.2, 0.7, 0, and 0.3, respectively; only proposal generation enables model reasoning. Their respective per-call `max_tokens` settings are 8192, 8192, 1536, and 8192.

Supplementary experiment settings. The ablation study and generation comparison use only seed 2024 and do not include repeated runs. For the efficiency analysis, the candidate-budget scaling experiment uses independent single-run evaluations on a subset of seven tasks: Periodic vehicle routing, Capacitated warehouse location, Pedigree, Operator scheduling, Crew pairing, Aircraft landing, and Open-shop scheduling. For both the generation-comparison and candidate-budget scaling curves, each reported test score is obtained by selecting the program with the best development-set performance among all candidates generated up to the corresponding checkpoint or budget, and then evaluating the selected program separately on the test set. Thus, both experiments use the same Test Avg. Score metric as the main experiments, while program selection depends exclusively on development-set performance. The performance–cost trade-off, measured by token cost per feasible candidate, is computed from the three-seed main experiments.

B Methodology Details

This appendix provides implementation details of RevivEvo beyond the component-level description in the main text. We first describe how candidate generation and recovery are integrated into the MCTS search process, then specify the construction of parent evidence and progress cues, and finally explain how experience is updated, inherited, and retrieved. Prompt templates and response schemas are presented separately in Appendix C.

B.1 MCTS-BASED EVOLUTIONARY SEARCH

We build on MCTS-AHD (Zheng et al., 2025) by retaining its tree-based selection, progressive widening, and value backpropagation. REVIVEVO modifies the initialization and expansion procedures: each ordinary expansion generates one proposal-conditioned candidate episode before returning to parent selection.

The normalized objective Q defined in Section 2.1 represents the desired solution quality. During implementation, however, the search process is guided by a task-specific development objective $J(p)$ formulated in minimization form. For CO-Bench, $J(p) = -Q(p; \mathcal{D}_{\text{dev}})$. For HeuriGym, J follows the configured development evaluation protocol by using either a native loss or a negative quality score. Tree selection, value updates, and progress comparisons are performed using J, while Q is used as the normalized reporting metric.

Initialization. Search begins with a strictly feasible solver, obtained from the first accepted external seed or, when no seed is provided, through budgeted parentless generation. External seed validation does not consume the generation budget, whereas generated initialization candidates do. Once a feasible seed is available, up to four bootstrap attempts use the seed and evaluated bootstrap portfolio as evidence. Accepted bootstrap programs are attached directly to the virtual root, although their generation reference remains the common seed. The virtual root itself does not represent an executable solver.

Before ordinary selection, up to two competitive bootstrap programs receive one priority expansion each. Candidates are considered in objective order, and repeated nonempty proposal signatures are skipped. Eligibility requires $J(p_v) \le J^b_{\min} + 0.10 \max(|J^b_{\min}|, \epsilon_{\text{mach}})$, where $J^b_{\min}$ denotes the best finite bootstrap objective. Bootstrap and priority expansions share the same candidate budget, and neither guarantees a feasible offspring.

Parent selection. Each program node v maintains a visit count N_v and a selection value H_v, which is distinct from the quality function Q. Selection descends from the virtual root. At a non-root node, progressive widening permits immediate expansion when $\lfloor N_v^{0.5} \rfloor > |\,\text{children}(v)|$. Otherwise, selection uses a normalized UCT-style score, based on UCT (Kocsis & Szepesvári, 2006) and adapted within the MCTS-AHD backbone (Zheng et al., 2025):

$$U_t(w) = \frac{H_w - h_{\min}}{h_{\max} - h_{\min}} + 0.1 \max\left(1 - \frac{n_t}{B}, 0\right) \sqrt{\frac{\log(N_v + 1)}{N_w}}, \tag{8}$$

where n_t denotes the charged candidate count, B denotes the run budget, and $h_{\min}, h_{\max}$ denote running reward-normalization bounds. If these bounds coincide, the least-visited child is selected instead. Descent terminates at a leaf or the selection-depth limit of ten, where one expansion is attempted.

Expansion and search updates. An expansion consists of proposal generation, code implementation, strict evaluation, bounded repair when needed, and experience consolidation. Repair attempts remain within the same candidate episode rather than creating separate tree nodes. During ordinary expansion, a feasible result is rejected if its source duplicates a program in the maintained population. An accepted program creates a child node with $N = 1$ and $H = -J$. For each ancestor v, the inherited update is

$$N_v \leftarrow N_v + 1, \qquad H_v \leftarrow \max_{w \in \text{children}(v)} H_w. \tag{9}$$

Thus, the backbone propagates the best child value rather than an average return. Unrecovered candidates consume their candidate opportunity but create no node and trigger no value or visit-count backpropagation. Their recorded outcomes and failure experience can still guide subsequent generation. A separate elite population retains up to ten programs with the lowest objectives; eviction from this population does not remove tree nodes. Search then returns to parent selection while the budget remains.

B.2 Parent Evidence and Progress Cues

Constructing parent-specific evidence. For a selected node v with program p_v, we construct

$$\mathcal{E}_t(p_v) = \big(a_v, \phi(p_v), \psi(\mathcal{R}_v), S_v, L_v, F_v\big), \tag{10}$$

where a_v denotes the algorithm description, $\phi(p_v)$ denotes a deterministic structural code profile, and $\psi(\mathcal{R}_v)$ summarizes the evaluation record of the parent. The profile captures source size and structural characteristics, including calls, loops, branches, and loop nesting, without exposing raw source code. Evaluation feedback contains available quality, runtime, and feasibility information. Short-term experience S_v links the latest parent–child revision to its observed outcome, while long-term experience L_v consolidates previous trajectory experience. The lineage-specific failure summary F_v records failures and their diagnosed causes. The task description $\mathcal{T}$ independently provides the objective, execution requirements, hard constraints, and permitted instance context.

Summarizing evolutionary progress. Progress cues are derived from recorded ordinary expansions along the selected parent's ancestry and attempts generated directly from that parent. Candidate and parent identifiers distinguish these outcomes from unrelated branches. Let $\mathcal{O}_t(v)$ denote the resulting ordered sequence reconstructed from the available bounded history. An outcome is considered informative only when its child is feasible, admitted to the search tree, and comparable with its parent:

$$\mathcal{I}_t(v) = \{e \in \mathcal{O}_t(v) : \mathrm{admitted}(e) = 1,\ \mathrm{feasible}(e) = 1,\ \Delta_e \in \mathbb{R}\}, \tag{11}$$

where $\Delta_e = J(p_{\mathrm{child}(e)}) - J(p_{\mathrm{parent}(e)})$. Here, J denotes the internal minimization objective on the development set, and the child refers to the final program after any permitted repair.

We define $I_t(v) = |\mathcal{I}_t(v)|$ and let $K_t(v)$ count informative outcomes satisfying $\Delta_e < -10^{-12}$. Local stagnation $s_t(v)$ counts consecutive non-improving informative outcomes rather than failed candidate attempts. Global stagnation g_t counts consecutive observed outcomes without an incumbent improvement. Once a finite incumbent exists, nonfinite outcomes also contribute to g_t.

Assigning progress cues. For ordinary parent expansion, the controller applies the following ordered rule:

$$u_t(v) = \begin{cases} \mathsf{hypothesis}, & I_t(v) = 0, \\ \mathsf{escape}, & I_t(v) \geq 2 \ \wedge\ \big[s_t(v) \geq 3 \ \vee\ (g_t \geq 5 \wedge s_t(v) \geq 2)\big], \\ \mathsf{specialize}, & I_t(v) \geq 2 \ \wedge\ K_t(v) \geq 2 \ \wedge\ s_t(v) = 0, \\ \mathsf{probe}, & \text{otherwise.} \end{cases} \tag{12}$$

A hypothesis cue requests a testable explanation of the parent's bottleneck, while a probe cue requests a controlled test under limited or mixed evidence. A specialization cue encourages refinement following repeated improvements, whereas an escape cue requests reconsideration of a soft design assumption without relaxing hard constraints. Initialization and bootstrap use separate cues to reflect their different evidence. The controller makes no LLM calls; only the selected label and its instruction are provided to the proposal prompt. These cues guide evidence interpretation rather than prescribing an editing operator or establishing a causal explanation.

Generating and implementing a revision. The Proposal Agent receives $(\mathcal{E}_t(p_v), u_t(v), \mathcal{T})$ and integrates improvement and failure evidence into a revision brief. The Code Agent then receives this brief, the parent description and complete source, and the task requirements. Relevant evidence is therefore conveyed through the brief rather than copied directly into the implementation prompt. The Code Agent checks the proposed revision against the actual source and returns a complete offspring solver. Feasibility is determined only by strict evaluation, not by the proposal or progress cue. The corresponding prompts and output schemas are provided in Appendix C.

B.3 Memory Update and Reuse

RevivEvo retains candidate experience to support subsequent generation and repair. Trajectory and failure memories provide lineage-specific context for the Proposal Agent, while repair skills enable cross-branch recovery within the same run. The repair-skill repository is not accessed during ordinary proposal generation.

Trajectory experience. For a feasible child c of parent v, an LLM compares their bounded code differences and development evaluation records:

$$\widetilde{S}_c = \mathrm{Compare}\big(\mathcal{T}, \mathrm{Diff}(p_v, p_c), \mathcal{R}_v, \mathcal{R}_c\big). \tag{13}$$

The comparison links the main revision to its observed outcome without claiming causality. When repair analysis is available, it is integrated with $\widetilde{S}_c$ to form short-term experience S_c; otherwise, $S_c = \widetilde{S}_c$. Long-term experience accumulates previous observations through

$$L_c = \mathrm{Update}(L_v, S_v; \mathcal{T}). \tag{14}$$

This update uses the parent's short-term experience S_v rather than the current child's S_c. When c is subsequently selected, the Proposal Agent receives the accumulated history in L_c and the latest revision experience in S_c separately.

Lineage-specific failure memory. After a candidate episode involving an algorithmic failure, its diagnoses, attempted repairs, and observed outcomes are consolidated into an episode summary ξ_c. This summary updates the selected parent's failure memory, yielding $F' = \mathrm{Update}_F(F_v, \xi_c; \mathcal{T})$. Each episode contributes at most one update rather than a separate warning for every repair attempt. The updated memory is assigned according to the pipeline outcome:

$$\mathrm{recipient}(F') = \begin{cases} c, & \text{if a feasible child is returned,} \\ v, & \text{otherwise.} \end{cases} \tag{15}$$

A recovered child carries the experience to its descendants. If recovery fails, writing the experience back to the parent allows subsequent offspring to benefit despite the absence of a new feasible child. Existing siblings and ancestors are not retroactively updated. Without a new eligible incident, the child inherits the parent's failure memory. Infrastructure-inconclusive episodes are excluded from failure and repair memory updates.

Cross-branch repair reuse. An episode with at least one applicable repair response can contribute a run-local repair-skill record. Each record summarizes the failure trigger, diagnosed cause, prevention guidance, attempted repair, and observed result, including both resolved and unresolved outcomes. During repair, the current failure and diagnosis form a query q. Records are filtered by task and execution-contract compatibility without requiring an identical failure type.

Eligible records are ranked by lexical matching and cosine similarity of character TF–IDF representations using three- to five-character n-grams. Let $\mathrm{rank}_{\mathrm{lex}}$ and $\mathrm{rank}_{\mathrm{char}}$ denote one-based ranks. For a record m, we combine the two rankings using a weighted variant of reciprocal-rank fusion (RRF) (Cormack et al., 2009):

$$s_{\mathrm{RRF}}(m \mid q) = \frac{0.4}{60 + \mathrm{rank}_{\mathrm{lex}}(m \mid q)} + \frac{0.6}{60 + \mathrm{rank}_{\mathrm{char}}(m \mid q)}. \tag{16}$$

If character ranking is unavailable, retrieval uses lexical ranking alone. At most two records are selected. When both outcomes are available, selection includes the highest-ranked resolved and unresolved records; otherwise, it selects the highest-ranked available records. Their summaries provide successful repair experience and warnings about unsuccessful attempts to the Repair Agent. Retrieved guidance informs repair but never replaces strict re-evaluation or guarantees admission.

C Prompt Engineering

This section presents the core prompt templates for parent-specific generation, type-aware diagnosis and repair, and memory consolidation. Each template specifies agent instructions, runtime inputs, and the required output format. Red brace-delimited fields denote placeholders filled with task-specific information during execution rather than actual model responses. Fixed instructions are reproduced from the implementation, except for the Repair Agent template, which provides a unified structure for failure-specific prompts.

C.1 GENERATION PROMPTS

Parent-specific proposal and implementation. The first template generates a revision brief based on the selected parent description, evaluated behavior, structural profile, and memories. The second implements this brief using the complete parent source code. The progress-cue slot is populated by the deterministic controller, and its selection rules are omitted here. We present the standard parent expansion case, excluding separate initialization and bootstrap variants. The Code Agent box shows the function-entry template. Class-entry tasks additionally append the execution-contract override from the implementation, requiring the complete class, methods, imports, and helpers within the same source block. The requested word limit and failure-avoidance priorities in the proposal are prompt-level instructions rather than independent guarantees of feasibility.

Parent-specific proposal generation prompt

You are the Evolution Proposal Agent in a two−agent solver−evolution pipeline. Based on the MCTS−selected parent and its memories, choose one parent−specific next evolutionary direction for a separate Code Agent.

[Evolution reasoning stage: {progress_cue_name}]
{progress_cue_text}

The stage controls how to read evidence; it is not a fixed mutation operator. Infer the concrete direction from the parent evidence.

Decide from the evidence whether the next move should be incremental, substantial, or exploratory. Do not default to either small or large changes. State what direction to pursue and why; leave source−level implementation to the Code Agent. You may use observed metrics and name algorithmic mechanisms, but do not write code or invent evidence.

Before choosing the direction, jointly reason over both experience families. Use short− and long−term trajectory experience to identify what may improve the evaluated parent, while using the selected lineage's failure experience to constrain the mechanism and evolutionary scope. Select the direction and its step size from this joint evidence rather than choosing a direction first and retrofitting warnings afterward. A direction that repeats a known failure mechanism is invalid unless the proposed direction first removes that mechanism. Failure evidence therefore has veto priority over novelty or expected quality gain. The resulting evolution_direction and failure_avoidance must be mutually consistent. The three fields must form one coherent brief; do not independently list or copy memory records.

Return JSON only using exactly these fields, with no more than 150 words across all field values:
{"evolution_direction": "the parent−specific next direction, its evolutionary scope (incremental, substantial, or exploratory), and why the joint trajectory and failure evidence supports that step size", "experience_guidance": "an integrated synthesis of how the relevant trajectory experience supports, corrects, or qualifies this direction", "failure_avoidance": "a direction−specific synthesis of the relevant past failures and what must not be repeated while pursuing this direction"}

[Task]
{task}

[Hard constraints and instance context]
{constraint_context}

[Parent algorithm description; no source code]
{parent_algorithm}

[Standardized parent feedback]
{parent_evaluation}

[Deterministic code profile; no source code]
{code_profile_json}

[Trajectory long−term memory]
{long_memory}

[Latest short−term experience]
{short_memory}

[Lineage failure experience]
{lineage_failure_memory}

Code generation prompt

{task}

You are the Code Agent in a solver−evolution pipeline. The Evolution Proposal Agent compressed the selected parent's feedback and trajectory memories into the brief below.

[Parent algorithm description]
{parent_algorithm}

[Parent code]
{parent_source}

[Evolution direction brief]
{revision_brief_json}

[Program interface, hard constraints, and instance context]
{hard_constraints_and_instance_context}

Inspect the complete parent source and decide the concrete code change that best follows the supplied direction. Use its direction, experience, and failure boundary together. Verify that the code−free brief fits the actual source. Match the scope of the change to the brief: it may be incremental, substantial, or exploratory. Do not default to either a minimal edit or a broad rewrite. Preserve the immutable contract and avoid every listed failure.

Use exactly two output parts: first, one sentence describing the implemented design change inside braces; second, one python code fence containing only the complete coherent Python source. The brace sentence or any JSON metadata must stay outside the code fence, whose first nonempty line must be an import, from−import, def, or class statement. The complete source must implement the child solver with function name '{entry_function}'. The declared inputs are {declared_inputs}; the declared output is {declared_outputs}. {input_output_description} {other_task_requirements}
Do not give additional explanations.

C.2 Diagnosis and Repair Prompts

Type-aware diagnostic prompts. The three Experts share a common output schema while analyzing different failure evidence: execution errors, excessive computation, and constraint violations. Each Expert receives a structured repair context containing failure evidence, candidate localization, and feedback from previous repair attempts.

Code Expert prompt

Diagnose this code−infeasible complete solver before repair. Identify the earliest executable or public−contract defect supported by the static issues or traceback. Do not redesign the algorithm and do not write or repeat code. repair_localization.parent_reference_at_candidate_targets shows how the strict−feasible parent implements these same candidate locations. If a candidate

location's mechanism there is essentially unchanged from the parent, that is evidence against it being the causal defect −− do not name it as a repair target on that basis alone; look instead at how the traceback's failing call actually reaches it. Only name a location as a repair target because of this reference when it reveals a genuinely different, restorable parent mechanism, and state that restoration explicitly in repair_action. Return JSON with exactly three fields: cause_with_evidence, repair_targets, repair_action. repair_targets must contain at most three exact qualified names from repair_localization.candidate_targets; use an empty list only when localization is impossible. Use one concise sentence per text field and no more than 180 words total.

[Repair context]
{repair_context_json}

Timeout Expert prompt

Diagnose why this complete optimization solver exceeded its evaluation time budget. Use dynamic stack frames as the strongest evidence and AST structure as supporting evidence. Loop depth or code length alone is not a cause: identify the concrete repeated work, its scale multiplier, and the smallest mechanism change that preserves feasibility and search intent. Do not write or repeat code. repair_localization.parent_reference_at_candidate_targets shows how the strict−feasible parent implements these same candidate locations. If a candidate location's mechanism there is essentially unchanged from the parent, that is evidence against it being the hotspot −− do not name it as a repair target on that basis alone; look instead at what scales with problem size around it. Only name a location as a repair target because of this reference when it reveals a genuinely different, restorable bounded parent mechanism, and state that restoration explicitly in repair_action. Return JSON with exactly three fields: cause_with_evidence, repair_targets, repair_action. cause_with_evidence must name the concrete repeated work, scale multiplier, and strongest stack/AST witness. repair_targets must contain at most three exact qualified names from candidate_targets and may include their enclosing caller. repair_action must state one minimal complexity change plus its measurable runtime guard. Use one concise sentence per text field and no more than 180 words total. If evidence is insufficient, say so rather than guessing.

[Repair context]
{repair_context_json}

Constraint Expert prompt

Diagnose this constraint−infeasible optimization solver before code repair. Using the validator evidence and relevant code changes, identify the earliest code mechanism that can produce the violation and distinguish it from downstream symptoms. Treat any deterministic schema/source audit as higher−confidence evidence than a free−form hypothesis; do not jump to resource thresholds while an upstream input/output representation defect remains. Read previous_attempts as counterfactual evidence: when progress is unchanged or worsened, do not repeat the same causal hypothesis, target, or patch mechanism unless new source evidence falsifies the prior test; when progress is improved, refine the same mechanism toward the exact validator semantics. Do not claim a deterministic audit that is absent.
repair_localization.parent_reference_at_candidate_targets shows how the strict−feasible parent implements these same candidate locations. If a candidate location's mechanism there is essentially unchanged from the parent, that is evidence against it being the causal defect −− do not name it as a repair target on that basis alone; look instead at how its result is consumed by the caller. Only name a location as a repair target because of this reference when it reveals a genuinely different, restorable parent mechanism, and state that restoration explicitly in repair_action. Do not write repaired code or repeat source. Return JSON with exactly three fields: cause_with_evidence, repair_targets, repair_action. cause_with_evidence must name the earliest causal mechanism and its strongest supplied witness. repair_targets must contain at most three exact qualified names from candidate_targets and may include their enclosing caller. repair_action must state one minimal patch action and its strict validation condition. Use one concise sentence

per text field and no more than 180 words total. Base claims on supplied evidence; say when the evidence is insufficient.

[Repair context]
{repair_context_json}

Unified Repair Agent prompt structure. The Repair Agent integrates failure evidence, expert diagnosis, and retrieved experience to revise the current solver. The unified template below summarizes failure-specific instructions, and each call activates only the corresponding branch. Runtime inputs remain placeholders, while the shared output protocol is reproduced verbatim.

Repair Agent: unified prompt structure

{failure_specific_repair_instruction}

[Required contract]
{program_contract}

[Failure evidence]
{failure_packet_json}
{static_issues_if_applicable}

[Diagnosis]
{expert_diagnosis_json}

[Required repair action]
{repair_action}

[Retrieved repair experience]
{retrieved_repair_experience}

{parent_reference_block_if_available}

[Output protocol]
Return exactly TWO required parts, with no diff or other explanation. FIRST return the complete corrected Python source for the entire program in one fenced code block (‘‘‘python ... ‘‘‘). The returned program must be syntactically complete and directly executable as−is −− never a partial excerpt, a patch, or the unmodified source. The diagnosis suspects the defect is in {diagnosed_target_names}, so prioritize that area, but you may change any other part of the program that is necessary for a correct, coherent fix. Preserve the public interface and all unrelated correct behavior. SECOND, after closing the Python fence, inspect the source you just returned and write <algorithm_summary>a concise description, preferably within 80 words, of that COMPLETE final solver</algorithm_summary>. Both parts are required. Describe only construction, search and fallback mechanisms actually present in the returned code. Do not describe the intended design, removed functions, or an earlier version; if repair reduced it to a greedy solver, describe only that greedy solver. Do not put metadata inside Python source.

[Source−localization instruction]
{branch_specific_source_localization_instruction}

[Complete current source]
‘‘‘python
{current_source}
‘‘‘

C.3 Memory Prompts

Failure experience and reusable repair records. The first template updates the selected lineage's failure summary after a completed episode. The second extracts a structured repair-experience

record from observed diagnoses, modifications, and evaluation outcomes. These templates consolidate experience rather than independently rediagnosing failures or certifying repairs. Record eligibility, inheritance, deduplication, and retrieval are determined by the code described in Appendix B.3.

Lineage failure-memory update prompt

Update one MCTS lineage's long−term failure experience after the complete bounded repair episode below. Keep the temporal order of distinct failures and repairs; do not merge different mechanisms merely because they share a broad error type. Do not rediagnose or invent evidence. State what failed, what was tried, the final outcome, and the concrete guard for future generation, in no more than 200 words. Return plain text only, not JSON.

[Task]
{task}

[Previous lineage failure experience]
{previous_lineage_failure_memory}

[Completed bounded repair episode]
{completed_episode_json}

Repair-experience record generation prompt

Convert this observed repair transition into a reusable prevention record. The main goal is to stop future generated programs from repeating the same error, not to narrate the whole event. Distinguish evidence from inference and do not invent a cause, code change, or validation result. For an unresolved repair, state what was attempted and what must not be repeated. Return concise JSON with these fields: trigger, root_cause, prevention_rule, repair_action, result. The prevention_rule must be an actionable pre−generation or pre−acceptance guard.

[Observed transition]
{observed_transition_json}

D Case Studies

D.1 Parent-Specific Generation

This example illustrates the generation of candidate C18 from parent C6 on the HeuriGym e-graph extraction task, using DeepSeek V4 Flash with seed 2025. The raw offspring passes strict development evaluation without requiring repair.

Example Problem and Parent Context

Task summary.

Read an e-graph from JSON and write the selected e-node IDs to a JSON file. Minimize extraction cost while covering the roots, selecting one representative per required e-class, preserving dependency closure, and avoiding cycles.

Recorded parent context.

Parent C6 was feasible on all six development instances. Its native development objective was 94.61084656 (lower is better). The progress cue recorded two informative lineage edges with no improvement.

Lineage failure experience: recorded excerpt.

Failed: SCC-based `strongconnect` closure in `solve` selected cyclic e-class dependencies, violating acyclicity (status `constraint_infeasible`).

Proposal Agent: Instruction and Response

Instruction excerpt from the frozen template.

Before choosing the direction, jointly reason over both experience families. Use short- and long-term trajectory experience to identify what may improve the evaluated parent, while using the selected lineage's failure experience to constrain the mechanism and evolutionary scope.

Response: `evolution_direction`.

Explore a substantial mechanism shift: replace fixed-level strict child ordering with a memoized recursive acyclic DP that allows any child e-class if a DFS visiting guard proves no cycle, using parent level-based DP as fallback. This challenges the soft assumption that strict level decrease is necessary for acyclicity, which may exclude better valid selections. Substantial because mature parent plateaued at J=94.611 despite safe lookup; failure evidence forbids SCC, so add explicit cycle guard/fallback.

Response: `experience_guidance`.

Long-term and short-term show preserving full solve and safer dict.get lookup kept J=94.611; parent level DP is feasible but likely overconstrained. Use its closure-cost DP and safe lookup, but relax level strictness only with dynamic cycle proof.

Response: `failure_avoidance`.

Do not repeat SCC/strongconnect closure or any cycle-blind selection; require per-selection DFS cycle detection and fallback to level-based acyclic witnesses. Reject directions that simplify solve or drop strict validity before scoring.

Code Agent Output

Algorithm.

Implemented a memoized recursive acyclic DP that selects each e-class's minimum-cost closed witness using DFS with an active-path cycle guard, falling back to the parent level-based DP when a cyclic child dependency is encountered.

Code excerpts. The excerpts below show recursive witness selection and the level-based fallback. Input processing, initialization, and output serialization are omitted.

```
def dp(ec):
    st = dp_state.get(ec, 0)
    if st == 2:
        return dp_cost[ec]
    if st == 3:
        return None
    if st == 1:
        # cycle on active path
        return None
    dp_state[ec] = 1
    active.add(ec)
    best = None
    best_total = float('inf')
    nlist = eclass_to_nodes.get(ec)
    if nlist:
        for i in nlist:
            ok = True
            total = node_cost[i]
            for cec in node_child_eclasses[i]:
                if cec == ec:
                    ok = False
                    break
                if cec in active:
```

```
                    ok = False
                    break
                sub = dp(cec)
                if sub is None:
                    ok = False
                    break
                total += sub
            if ok and total < best_total:
                best_total = total
                best = i
    active.discard(ec)
    if best is None:
        dp_state[ec] = 3
        return None
    dp_state[ec] = 2
    dp_cost[ec] = best_total
    dp_witness[ec] = best
    return best_total

# [...] Intermediate initialization and root processing omitted.

while stack:
    ec = stack.pop()
    if ec in visited_eclasses:
        continue
    visited_eclasses.add(ec)
    i = None
    if ec in dp_witness and dp_state.get(ec, 0) == 2:
        i = dp_witness[ec]
    else:
        # Fallback: level-based acyclic witness.
        lv = eclass_level.get(ec, -1)
        best = None
        best_total = float('inf')
        for j in eclass_to_nodes.get(ec, ()):
            ok = True
            total = node_cost[j]
            for cec in node_child_eclasses[j]:
                clv = eclass_level.get(cec, -1)
                if clv < 0 or clv >= lv:
                    ok = False
                    break
                cc = closure_cost.get(cec)
                if cc is None:
                    ok = False
                    break
                total += cc
            if ok and total < best_total:
                best_total = total
                best = j

        # [...] Remaining fallback and reconstruction omitted.
```

Evaluation Result

	Parent C6	Offspring C18
Feasible development instances	6/6	6/6
Dev Avg. Score ↑	0.5935	0.7554

The offspring passed development evaluation without repair, improving the Dev Avg. Score by 0.1619 over its parent.

D.2 TYPE-AWARE REPAIR

This example illustrates candidate C2 from the same e-graph extraction run. The generated solver executes successfully but violates the acyclicity constraint. One Constraint Expert–Repair Agent round restores feasibility.

Infeasible Candidate

Algorithm.

Replaced the level-based witness reconstruction with Tarjan SCC condensation over e-class dependencies, selecting witnesses only from acyclic SCCs or SCCs with a valid acyclic exit, and performing a topological DAG-safe reconstruction from roots.

Relevant code excerpt.

The fallback below checks whether child e-classes belong to the feasible set, but does not itself check whether selecting the node introduces a cycle.

```
if best is None:
    # last resort: pick any feasible node
    for i in eclass_to_nodes.get(ec, ()):
        ok = True
        for cec in node_child_eclasses[i]:
            if cec not in feasible_eclasses:
                ok = False
                break
        if ok:
            best = i
            break
```

Failure type. Constraint violation (`constraint_infeasible`).

Violated constraint. Acyclicity: the selected subgraph must not contain a directed dependency cycle. The candidate violated this requirement on one of the six development instances.

Verifier feedback. Verification failed: Selected sub-graph contains cycles (acyclicity violated).

Constraint Expert Output

Cause with evidence.

The earliest causal mechanism is the child's replacement of the parent's layered feasibility propagation with a Tarjan SCC-based `strongconnect` closure, which can select cyclic e-class dependencies; the strongest witness is the constraint-infeasible failure with dependency-cycle-risk instances and self-referential e-nodes.

Repair targets.

`solve.<locals>.strongconnect`, `solve`.

Repair action.

Restore the parent's level-based acyclic witness selection in `solve` instead of SCC closure, and validate that every selected node's child e-classes have strictly smaller assigned levels.

Repair Agent Output

Algorithm.

Level-based acyclic feasibility propagation: leaf e-classes get level 0; an e-class becomes feasible at level k+1 when a node's child e-classes are all feasible. Witness selection picks, per feasible e-class, the cheapest node whose child e-classes all have strictly smaller levels, guaranteeing acyclicity. Roots are expanded via DFS using these witnesses, with a same-level-checking fallback. No SCC closure is used.

Revised code excerpt.

The repaired witness-selection block requires every child e-class to have a strictly smaller level than the current e-class.

```
# Witness selection: for each feasible eclass, pick a node whose child
# eclasses all have strictly smaller assigned levels (guarantees
    acyclicity).
witness_node = {}
for ec in feasible_eclasses:
    best = None
    best_cost = float('inf')
    for i in eclass_to_nodes.get(ec, ()):
        ok = True
        for cec in node_child_eclasses[i]:
            if cec not in feasible_eclasses:
                ok = False
                break
            if eclass_level[cec] >= eclass_level[ec]:
                ok = False
                break
        if ok:
            if node_cost[i] < best_cost:
                best_cost = node_cost[i]
                best = i
    if best is not None:
        witness_node[ec] = best
```

Repair Result

	Before repair	After repair
Feasible	No	Yes
Dev Avg. Score ↑	0.5581	0.5935

After one Expert–Repair round, the candidate became feasible on all six development instances and its Dev Avg. Score increased from 0.5581 to 0.5935. For the infeasible candidate, we report the partial Dev Avg. Score, with failed instances contributing zero.