MILP-EVO: CLOSED-LOOP FULLY AUTOMATIC DESIGN OF MILP SOLVERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Machine learning methods have shown that data-driven policies can accelerate mixed-integer linear programming (MILP) solvers, but many such approaches remain difficult to inspect, adapt, and deploy because the learned policy is represented as an external predictor or other opaque model. By contrast, explicit solver logic is easier to understand and integrate, but is usually hand-designed rather than learned from solver feedback. We study the automatic design of executable white-box components for MILP solvers, using their end-to-end behavior on MILP instances to guide iterative refinement. To this end, we introduce MILP-Evo, a framework implemented through PySCIPOpt, and instantiate it on the joint design of a cut selector and a branching rule. Candidate implementations are executed directly within SCIP and refined using their runtime outcomes and search-process diagnostics. The resulting solver components can be inspected, modified, and deployed within standard solver workflows. Across four families, MILP-Evo performs strongly on independent set, combinatorial auctions, and set cover, and remains competitive on facility location.

1 INTRODUCTION

Combinatorial optimization (CO) is a foundational and challenging area of mathematical optimization, with applications ranging from production planning and distribution scheduling to chip design (Liu et al., 2008; Chen, 2010; Ma et al., 2019). Many exact CO tasks are naturally modeled as mixed-integer linear programs (MILPs), a standard language for industrial and benchmark optimization instances (Bengio et al., 2021; Gleixner et al., 2021). Because broad classes of MILPs are NP-hard, the practical value of exact optimization depends heavily on the efficiency of modern solvers. Such solvers build on branch-and-bound (B&B) and branch-and-cut, combining tree search with cutting planes, presolve, primal heuristics, and other internal mechanisms to obtain certifiably optimal solutions (Achterberg, 2009).

Modern MILP solvers have historically relied on human-designed policies for these decisions. Such policies are widely deployed because they are compact, efficient, reliable, and naturally integrated into solver workflows. For example, branching and cut selection rules are often implemented as hard-coded scoring functions or carefully engineered procedures that reflect decades of optimization expertise. However, designing these policies requires substantial expert knowledge and manual tuning, and generic rules may miss distribution-specific structure in recurring problem families (Bengio et al., 2021; Lodi & Zarpellon, 2017; Kuang et al., 2024a). This tension has motivated a growing body of learning-based work that replaces or augments individual solver decisions with data-driven policies, especially for branching (Gasse et al., 2019; Gupta et al., 2020; Lin et al., 2024), cut management (Huang et al., 2022; Paulus et al., 2022; Turner et al., 2023; Wang et al., 2023; Li et al., 2023; Puigdemont et al., 2024), and other modules such as diving heuristics and presolving (Paulus & Krause, 2023; Liu et al., 2024a).

Although learned policies can improve solving efficiency on problem distributions with shared structure, many of them are implemented as external neural or graph-neural predictors. This introduces a different set of challenges: the learned decision logic can be difficult to interpret, may require nontrivial training and inference infrastructure, and is not always easy to deploy as a native part of a production solver. Recent work has begun to address this issue by learning more explicit and interpretable solver policies. In particular, symbolic-policy methods such as Symb4CO and GS4CO

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.

learn compact branching rules that are lightweight, interpretable, and closer in spirit to traditional solver heuristics (Kuang et al., 2024a;b). These works suggest that automatic algorithm discovery can produce white-box solver policies rather than only black-box predictors. However, existing interpretable approaches still primarily focus on a single solver module, most notably branching, and often optimize a policy through imitation or proxy objectives. Recent work has also begun to study coupled B&B decisions such as node and variable selection jointly (Du et al., 2025). These developments motivate the automatic construction of solver-integrated components whose interaction is assessed through complete branch-and-cut executions.

We introduce MILP-Evo, a framework for automatically constructing executable MILP solver components through PySCIPOpt (Maher et al., 2016). We study the joint design of a cut selector and a branching rule because the two components interact along the same branch-and-cut trajectory: cut selection changes the LP relaxations and candidate states seen by branching, while branching determines the future nodes and separation contexts. MILP-Evo represents the two callbacks as one deployable Python module, validates it against the PySCIPOpt interface, registers it in a fresh SCIP model (Achterberg, 2009), and evaluates it by end-to-end solving behavior. An LLM proposes and refines candidate logic, while SCIP supplies runtime, status, and failure diagnostics. We evaluate the resulting family-specific modules on four learn2branch problem families: set cover, combinatorial auctions, facility location, and independent set.

Our contributions are: (i) we introduce MILP-Evo, an automatic design framework whose output is an explicit, solver-integrated PySCIPOpt module evaluated through complete SCIP executions; (ii) we instantiate the framework for the joint design of cut selection and branching, and provide runtime validation and contract-preserving repair for candidate modules operating under SCIP callback constraints; (iii) we evaluate the resulting family-specific components on four learn2branch benchmark families; strong results occur on independent set, combinatorial auctions, and set cover, while facility location remains competitive and independent set supports detailed interaction and search analyses.

2 Related Work

2.1 MILP Solver Learning

Learning-augmented MILP solvers replace hand-designed components of branch-and-cut with policies trained from solver data. The most established line studies variable branching: GNN policies over the variable-constraint bipartite graph made it possible to imitate strong branching on recurring MILP distributions (Gasse et al., 2019), while later methods reduce deployment cost or improve sample efficiency and transfer through hybrid inference and contrastive augmentation (Gupta et al., 2020; Lin et al., 2024). A parallel line studies cutting-plane decisions. Early learned cut selectors rank local candidate cuts, often with imitation or lookahead signals (Huang et al., 2022; Paulus et al., 2022); more recent work broadens the control surface to separator configuration, cut removal, hierarchical cut selection, and global cut selection across the branch-and-cut tree (Turner et al., 2023; Wang et al., 2023; Li et al., 2023; Puigdemont et al., 2024; Zeng et al., 2025). These methods show that solver decisions expose learnable structure, but the learned object is typically a predictor or policy attached to one decision interface.

Recent work has also moved beyond isolated branching toward broader learned B&B control. Symbolic-policy methods such as Symb4CO and GS4CO learn compact branching rules that are closer to traditional solver heuristics than opaque neural predictors (Kuang et al., 2024a;b). Other approaches learn coupled decisions, such as node and variable selection (Du et al., 2025), or represent the entire B&B tree to select search nodes with reinforcement learning (Zhang et al., 2025). This progression is important for our setting because branching, node selection, and cut management interact through the solver trajectory rather than through independent one-step decisions. These works motivate studying interacting solver decisions along their shared trajectory. MILP-Evo represents cut selection and branching as a joint executable callback module.

LLMs have recently entered MILP research through complementary routes. MILP-Evolve uses an LLM-based evolutionary process to generate diverse MILP problem classes, enabling foundation-model training for integrality-gap prediction, learning to branch, and instance-text alignment across problem families (Li et al., 2024). LLM-LNS instead uses a dual-layer self-evolutionary LLM agent

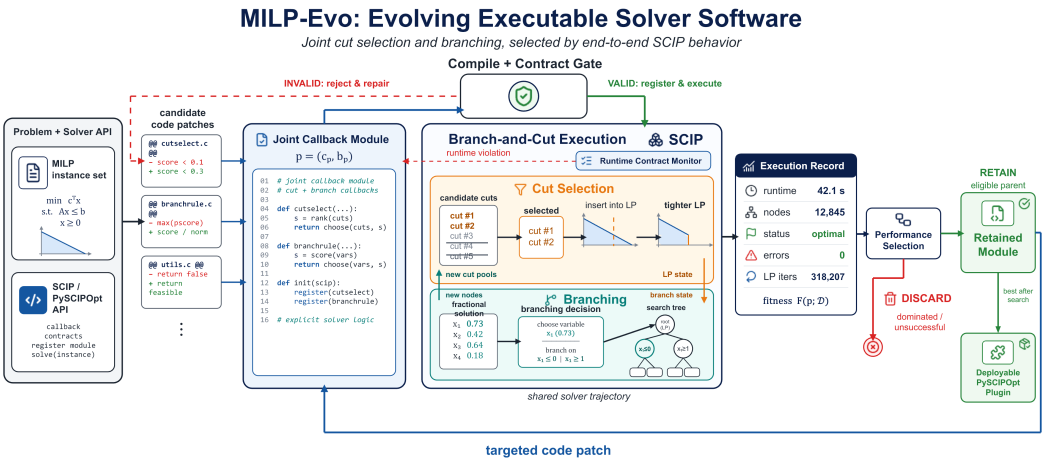


Figure 1: Software-level overview of MILP-Evo. Semantic patches modify one executable PySCIPOpt module that jointly implements cut selection and branching. Candidates pass compilation and callback-contract checks before registration, while runtime monitoring detects callback violations during complete branch-and-cut executions. The two callbacks interact through shared solver states; SCIP outcomes retain useful modules for later patches, and the best retained module is exported as a deployable plugin.

to design neighborhood-selection strategies for large-neighborhood search on large-scale MILPs (Ye et al., 2025). Together, these studies show that LLMs can support several stages of optimization-system design. MILP-Evo studies native PySCIPOpt modules containing cut-selection and branching callbacks that are executed and evaluated within SCIP.

2.2 Automatic Design

Evaluator-in-the-loop program search has emerged as a general mechanism for discovering executable heuristics and algorithms. FunSearch evaluates LLM-proposed programs through external execution (Romera-Paredes et al., 2024); Evolution of Heuristics and ReEvo apply related ideas to heuristic design (Liu et al., 2024b; Ye et al., 2024); and AlphaEvolve studies code evolution across scientific and algorithmic tasks (Novikov et al., 2025). These systems separate candidate proposal from executable evaluation.

For MILP solver components, executable candidates must also satisfy runtime contracts imposed by the host solver. A PySCIPOpt module must compile, return legal SCIP objects and result codes, and remain stable under repeated callback invocation. Its cut-selection and branching components are evaluated together because their effects emerge along the complete branch-and-cut trajectory. These requirements motivate the validation, diagnostic, and repair mechanisms described next.

3 Method

3.1 MILP-Evo Overview

MILP-Evo treats a complete PySCIPOpt callback module as the unit of design. Each candidate module contains a cut selector and a branching rule, together with the code required to register both components in a SCIP model. Before evaluation, the module is compiled, checked against the callback interfaces, and loaded into a fresh SCIP model. SCIP then evaluates the candidate through complete branch-and-cut executions and returns runtime, status, node-count, and failure diagnostics.

The resulting execution record determines both candidate feasibility and candidate selection. Interface violations and runtime failures identify candidates that require repair, while mean process time ranks callback-valid candidates that solve the evaluation batch to optimality. Within these interface

constraints, candidate modules may vary how cuts are ranked, how many cuts are retained, which branching variable is selected, and how these decisions depend on solver state.

Figure 1 presents MILP-Evo as software-level evolution of a joint solver module. Starting from a MILP instance set, the SCIP/PySCIPOpt API, and callback contracts, the LLM proposes semantic code patches to the executable artifact $p = (c_p, b_p)$. Within SCIP, cut selection changes the LP states encountered by branching, and branching creates later nodes and separation contexts. The resulting execution record drives performance selection: retained modules seed targeted patches, and the best retained module is exported as a deployable PySCIPOpt plugin.

3.2 Joint Native Callback Search Space

Each candidate is a solver-integrated software artifact consisting of a Python module that defines a cut-selection callback and a branching callback. The evaluation harness registers both callbacks in a fresh SCIP model through PySCIPOpt. SCIP retains control of the branch-and-cut process and invokes the candidate module only at the corresponding cut-selection and branching decision points.

In this work, each program is a joint artifact

$$p = \big(c_p, b_p\big),$$

where c_p is a cut-selection callback and b_p is a branching callback:

- The cut-selection callback is invoked after candidate cuts have been generated. It scores, orders, and selects cuts from the candidate pool before they are added to the LP relaxation.
- The branching callback is invoked when the solver needs to branch at an LP node. It chooses a branching variable from the current fractional candidates.

The two callbacks are evolved as one source-code artifact and are loaded into the same SCIP model. This joint representation is essential because cut selection and branching are coupled through the solver trajectory. Cut selection changes the LP relaxation, bound progress, and candidate states later encountered by branching; branching changes the future nodes, LPs, and separation contexts in which cut selection is invoked. In Figure 1, the orange path denotes the LP state passed from cut selection to branching, while the teal return path denotes that newly created nodes lead to later separation rounds and hence new candidate cut pools. Searching over the pair (c_p, b_p) therefore targets their combined effect on branch-and-bound rather than the isolated quality of either component.

This representation also keeps the learned policy explicit. The output of evolution is not a neural predictor wrapped around the solver, but readable solver logic implementing concrete decision rules. The program may use signals exposed by SCIP, such as cut efficacy, cut sparsity, row parallelism, objective coefficients, lock counts, pseudo-cost information, branching candidate statistics, and node-depth-dependent state. Evolution searches over how these solver signals are composed into cut-ranking, cut-filtering, variable-scoring, and depth-dependent control logic, while the callback interface constrains the program to remain deployable as a native solver component.

The exported callback module remains fixed throughout each SCIP solve; no LLM call or program update occurs after optimization begins. Its decisions can still vary with solver state because the callbacks are repeatedly invoked on different cut pools, branching candidates, and node depths.

The search space is therefore not ordinary Python code. A generated source string first belongs to the unconstrained space $\widetilde{\mathcal{P}}$. It enters the effective search space only if it compiles, defines the required callback classes, satisfies the PySCIPOpt calling conventions, returns legal SCIP objects and result codes, selects cuts only from the candidate pool, and performs a valid branch action when the solver requires one. We define the feasible callback space operationally as

$$\mathcal{P} = \{p \in \widetilde{\mathcal{P}} : F(p; \mathcal{D}) < +\infty\},$$

where infeasible programs are assigned infinite fitness and are excluded from elite selection. This view is important for solver auto-design: a candidate can express a useful algorithmic idea while still being inadmissible as a SCIP program because it violates a low-level callback contract. Invalid artifacts may fail through syntax errors, Python exceptions, malformed return dictionaries, illegal cut objects, degenerate branch behavior, or C-layer solver assertions.

The resulting optimization problem is

$$p^\star \in \arg\min_{p \in \mathcal{P}} F(p; \mathcal{D}),$$

where F is induced by registering p inside SCIP and running the resulting solver on MILP instances $\mathcal{D}$. Thus the language model supplies search proposals, while SCIP execution supplies both the feasibility test and the selection pressure.

3.3 Contract-Preserving Evolution Loop

Given a candidate program, the evaluator first applies a compile-and-contract gate that compiles the module, loads the callback artifact, and checks that the required solver interfaces are present. Candidates that fail are marked invalid and returned with diagnostics for repair when recoverable, whereas candidates that pass are registered in fresh SCIP models. During optimization, runtime monitoring detects state-dependent callback violations that cannot be established before execution. The evaluator records process time, solve status, objective value when available, node count, solved-instance count, and structured error information.

We distinguish program-level callback validity from instance-level solve success. A program is callback-valid only if it satisfies the SCIP interface: the cut-selection callback must return cuts drawn from the original candidate pool, a legal selected-cut count, and a valid SCIP result code; the branching callback must perform a legal branch when LP branching candidates are available. Syntax errors, missing callback definitions, malformed return values, callback exceptions, invalid cut objects, and degenerate branching behavior are treated as program-level failures and receive $+\infty$ fitness. Among callback-valid programs, an instance run is successful only when SCIP terminates with the required target status, which is optimality in our experiments.

Let $\mathcal{D}$ denote the evaluation batch. For a callback-valid program p and instance x, let $T(p, x)$ be the measured process time of the SCIP run, and let $\overline{T}(p; \mathcal{D})$ be its mean over $x \in \mathcal{D}$. The selection fitness is

$$F(p; \mathcal{D}) = \begin{cases} \overline{T}(p; \mathcal{D}), & \text{if } p \text{ is valid and succeeds on } \mathcal{D}, \\ +\infty, & \text{otherwise.} \end{cases}$$

All other evaluation outcomes receive infinite fitness and cannot enter elite selection. This objective makes the scalar selection signal depend on complete end-to-end solver behavior while keeping structured failure information available for repair and reflection. The evaluator also records solved count, status, node count, objective values, and error traces as diagnostics, while the scalar score remains grounded in SCIP execution.

The evolution loop separates proposing solver logic from making it executable. In the semantic proposal stage, the LLM proposes a targeted modification to a seed or retained joint callback module. The modification may revise a parent's scoring formulas and control schedules or recombine useful motifs from two parents; although the evaluator receives a complete source module, the proposal is framed as a source-level patch to existing solver software. The targeted logic includes how to trade off cut efficacy and sparsity, how aggressively to remove nearly parallel cuts, how branching scores combine fractionality, locks, objective coefficients, and pseudo-costs, and how these decisions change across the branch-and-cut tree.

When a candidate fails through a recoverable interface violation, we apply *contract-preserving repair*. The repair operator receives the candidate code together with compiler errors, Python tracebacks, callback-wrapper diagnostics, solver status, and the relevant PySCIPOpt interface constraints. Its role is to project the proposed solver logic back into the feasible callback space, not to replace it with a new heuristic. In practice this means repairing imports, class definitions, method signatures, return dictionaries, SCIP result codes, branch actions, and illegal API uses while preserving the candidate's cut-ranking and branching-scoring logic as much as possible. The repaired program is then re-evaluated by the same SCIP-grounded objective before it can enter the archive.

Finally, the archive maintains both performance and diversity. Each evaluated individual stores source code, fitness, parentage, solver diagnostics, error traces, reflection summaries, and two language-model-rated descriptors: *code quality* and *marginal utility*. These descriptors affect the exploration geometry by assigning candidates to cells in a two-dimensional MAP-Elites grid; they

do not decide which candidate is better inside a cell. Within each cell, replacement is determined solely by SCIP-grounded fitness F.

Within each island, each cell retains only a small number of finite-fitness elites. This prevents the search from collapsing immediately onto one visually similar family of programs, while still using solve time as the selection criterion inside each cell. Reflection converts solver feedback into strengths, weaknesses, and suggested evolving directions, which are fed back into later mutation and crossover prompts. Multiple islands evolve in parallel with periodic migration of high-quality individuals, providing a second source of diversity at the population level.

The final output is an explicit joint solver artifact that can be inspected, modified, and deployed through the same PySCIPOpt pathway used during evolution.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Baselines. We compare against a set of representative branch-and-cut baselines covering classical solver rules, learned branching policies, hybrid methods, and symbolic branching policies. FSB is full strong branching, RPB is reliability pseudo-cost branching, GNN is the supervised graph neural branching policy (Gasse et al., 2019), Hybrid combines learned and solver-side branching scores (Gupta et al., 2020), and GS4CO is a white-box symbolic branching baseline (Kuang et al., 2024a;b). We also include default SCIP as a solver-default baseline. MILP-Evo is evaluated as the exported PySCIPOpt artifact produced by execution-guided search, jointly implementing a cut selector and a branching rule. The primary comparison set consists of FSB, RPB, GNN, Hybrid, GS4CO, SCIP, and MILP-Evo; GNN-GPU and COPT (Ge et al., 2023) are included only as external references and are excluded from optimal-run win counts and boldface comparisons.

Benchmarks. We evaluate on four learn2branch MILP families: set cover, combinatorial auctions, facility location, and independent set (Gasse et al., 2019). We select these families because they cover distinct combinatorial structures and provide Easy, Medium, and Hard instance scales, allowing us to evaluate whether family-specific solver components transfer across instance sizes under a common experimental protocol.

Training and evaluation. For each problem family, MILP-Evo uses Gemini-3-flash-preview as the language-model backbone and performs 400 search iterations on Easy-scale instances to obtain a domain-specialized joint callback program. The exported PySCIPOpt policy is then evaluated without further adaptation across the Easy, Medium, and Hard scales of the same family. All methods are measured by mean process time with a 3600 second time limit, with unsolved runs charged the full time limit. We also report per-scale win counts over the primary comparison set and mean branch-and-bound node counts. MILP-Evo was run on machines equipped with Intel Xeon Platinum 8362 processors, using CPU execution only and no GPU acceleration. The GNN policy used by both variants was trained on a single NVIDIA A100 GPU with 40 GB of memory. At test time, GNN used CPU inference, whereas GNN-GPU used GPU inference on the same A100. The main MILP-Evo software environment used Python 3.11.9, PySCIPOpt 6.1.0, SCIP 10.0.2, and SoPlex 8.0.2. All baselines use the fixed defaults of their cited implementations without additional tuning.

4.2 COMPARATIVE EVALUATION AND SEARCH BEHAVIOR

Table 1 reports the main comparative results. Following prior branching-policy evaluations, each block gives mean process time, win count, and mean node count for one problem family across the three transfer scales.

Table 1: Comparative evaluation on four learn2branch families. Time is mean `proctime` in seconds; Wins is reported as W/S, where W counts runs solved to optimality fastest within the comparison set, and S counts runs solved to optimality within the time limit. GNN-GPU and COPT are external references.

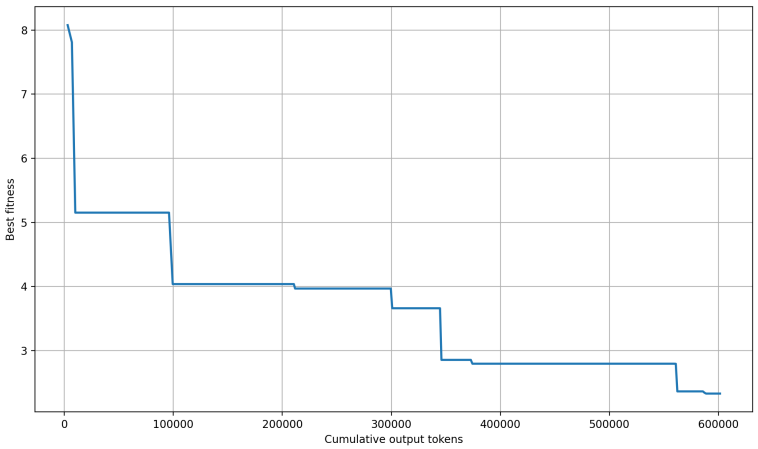
Indset:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	87.00	1/50	30.2	3318.97	0/7	150.5	3600.06	0/0	53.9
RPB	9.49	0/50	46.9	194.74	0/50	7442.9	2587.99	0/15	75329.3
GNN	5.80	0/50	62.2	1673.24	0/36	19872.9	3600.01	0/0	49010.6
Hybrid	6.47	0/50	53.1	1502.27	0/30	48887.3	3600.01	0/0	42854.8
GS4CO	7.29	0/50	54.1	147.26	0/50	2785.0	2057.08	0/30	67083.8
SCIP	14.80	0/50	164.6	377.99	0/50	4023.9	3155.52	0/18	22613.1
MILP-Evo	**2.57**	**49/50**	84.7	**59.20**	**50/50**	1051.9	**806.17**	**50/50**	17980.3
GNN-GPU	4.63	-/50	62.2	858.07	-/45	57818.8	3600.00	-/0	107167.4
COPT	2.62	-/50	9.6	62.18	-/50	707.7	1531.27	-/43	10279.2

Cauctions:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	10.80	0/50	25.5	286.99	0/50	253.1	3177.54	0/14	912.3
RPB	3.70	0/50	14.8	21.69	1/50	1169.0	**271.44**	**36/50**	17918.5
GNN	3.13	0/50	105.8	30.05	0/50	1047.9	662.39	0/50	17571.3
Hybrid	2.07	0/50	111.2	21.26	0/50	1210.7	718.47	0/50	30404.5
GS4CO	3.07	0/50	114.3	42.55	0/50	1353.1	1248.98	0/45	24867.6
SCIP	3.57	0/50	30.1	44.96	0/50	1169.9	307.55	3/50	15560.5
MILP-Evo	**0.81**	**50/50**	196.1	**11.79**	**49/50**	1536.0	381.90	11/50	34534.7
GNN-GPU	1.60	-/50	105.8	10.36	-/50	1047.9	191.11	-/50	17571.7
COPT	2.18	-/50	126.8	14.11	-/50	1542.2	161.34	-/50	19687.7

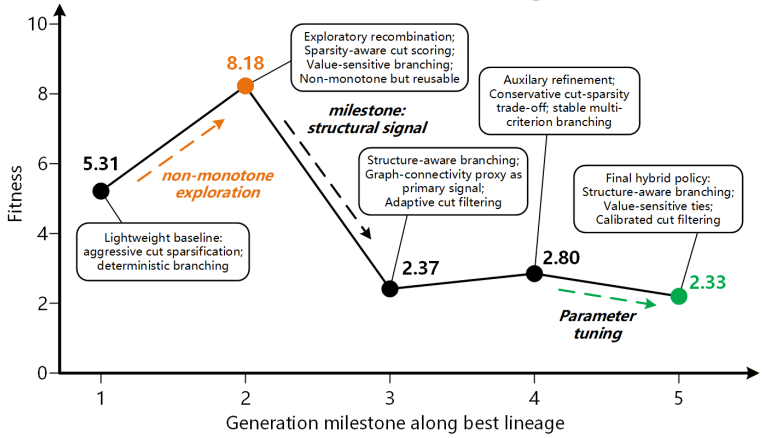
Setcover:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	29.07	0/50	45.7	1291.03	0/45	929.5	3600.03	0/0	658.3
RPB	11.20	0/50	129.6	162.70	0/50	10349.9	**2401.92**	6/20	118275.4
GNN	11.32	0/50	172.7	487.64	0/50	5990.5	3500.75	0/5	21036.2
Hybrid	8.26	0/50	201.9	171.41	0/50	7240.7	3079.13	0/10	63408.9
GS4CO	13.89	0/50	249.1	609.94	0/45	10303.9	2937.46	0/5	30500.2
SCIP	14.18	0/50	124.5	274.02	0/50	8361.4	2955.81	0/20	115143.9
MILP-Evo	**3.14**	**50/50**	269.0	**89.36**	**50/50**	11924.9	2641.34	**19/25**	188362.2
GNN-GPU	5.31	-/50	172.7	102.20	-/50	5990.5	2492.68	-/20	102635.8
COPT	6.88	-/50	297.4	99.64	-/50	12207.1	2718.71	-/25	175771.1

Facilities:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	81.02	0/50	71.0	806.48	0/50	233.9	2108.21	0/41	109.6
RPB	36.03	16/50	90.4	305.37	6/50	588.6	**607.57**	15/50	260.1
GNN	50.48	0/50	287.6	478.05	7/50	881.4	669.15	8/50	462.3
Hybrid	**35.10**	**17/50**	253.2	325.92	14/50	895.3	631.09	**17/50**	456.6
GS4CO	68.69	0/50	273.6	558.56	1/50	933.8	731.28	4/50	449.9
SCIP	38.27	5/50	99.5	305.10	2/50	541.1	675.33	3/50	194.6
MILP-Evo	35.66	12/50	235.5	**266.21**	**20/50**	926.8	761.88	3/50	429.1
GNN-GPU	33.44	-/50	287.6	267.77	-/50	881.4	546.13	-/50	462.3
COPT	8.20	-/50	221.0	31.56	-/50	572.0	58.06	-/50	262.8

Overall, MILP-Evo performs strongly on independent set, combinatorial auctions, and set cover, attaining the best mean runtime in seven of their nine scale settings. Facility-location results are less



(a) Best fitness versus cumulative LLM output tokens.



(b) Milestones along the best lineage.

Figure 2: Search dynamics on independent set; lower fitness is better.

consistent but competitive: MILP-Evo nearly matches the best method on Easy, leads on Medium, and trails the fastest baseline on Hard.

These results show that joint executable search discovers effective domain-specific rules across diverse problem structures, although its gains remain problem-dependent and scale-dependent.

Search convergence. Figure 2 summarizes both the optimization trace and the ancestry of the best independent-set artifact. The convergence curve is staircase-shaped rather than smooth: search spends long periods on valid but non-improving programs, then improves when a structural edit changes how cut filtering and branching interact.

Lineage analysis. The lineage panel adds the mechanism behind this pattern. The final artifact does not descend through a monotone sequence of better programs. Exploratory recombination initially worsens immediate fitness while introducing reusable motifs such as sparsity-aware cut scoring and value-sensitive branching. Later descendants preserve these motifs and refine them into structure-aware branching, graph-connectivity signals, and calibrated cut filtering. The best program is therefore the result of non-monotone evolutionary reuse rather than a single isolated improvement.

4.3 Ablation Study

We use independent set as a representative case for the ablation study. The cut-only and branch-only variants isolate the two callback components. The separate-best variant combines the best indepen-

Table 2: Ablation on independent set. Time is mean `proctime` in seconds.

Variant	Easy			Medium			Hard		
	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
Cut only	6.33	4/50	45.1	179.96	0/50	2121.0	2522.53	0/28	22734.0
Branch only	4.85	1/50	137.1	166.36	0/50	4243.6	2452.77	0/26	46905.8
Separate best	3.95	7/50	132.9	124.48	4/50	4053.4	2299.23	0/29	52902.7
SCIP	14.80	0/50	164.6	377.99	0/50	4023.9	3155.52	0/18	22613.1
NoRepair	6.34	0/50	107.0	200.81	0/50	4016.6	2615.15	0/22	26773.5
MILP-Evo	**2.57**	**38/50**	84.7	**59.20**	**46/50**	1051.9	**806.17**	**50/50**	17980.3

Table 3: Repair success statistics. "Without repair" treats triggered repairs as pre-repair failures; "with repair" uses the final evaluation outcome.

Family	Without repair	With repair	Triggered repair	Repaired success
indset	88.07% (288/327)	98.47% (322/327)	38	34/38 (89.47%)
facilities	82.86% (290/350)	98.57% (345/350)	58	55/58 (94.83%)
cauctions	79.19% (312/394)	94.42% (372/394)	82	60/82 (73.17%)
setcover	68.95% (211/306)	90.85% (278/306)	94	67/94 (71.28%)
overall	79.96% (1101/1377)	95.64% (1317/1377)	272	216/272 (79.41%)

dently found single-component artifacts after search, without co-evolving them as one program. NoRepair disables the contract-preserving repair operator introduced in the Contract-Preserving Evolution Loop.

The ablation supports the importance of joint executable search. MILP-Evo outperforms both single-component variants and the post-hoc separate-best combination on all three transfer scales, reducing Hard-scale time from 2299.23 seconds for separate-best to 806.17 seconds. Disabling repair is also damaging: NoRepair is slower than MILP-Evo on every scale and wins no instances, even on Easy. Runtime gains are again not explained only by smaller trees; the evolved policy changes the cost and quality of the explored tree as well as its size.

Repair effectiveness. Table 3 reports repair-oriented success statistics over the stored candidate records. Repair substantially improves the reliability of executable callback generation, increasing the aggregate success rate from 79.96% to 95.64% and reducing final failures to 60 cases. The effect is especially visible on the families with lower pre-repair validity, where many otherwise discarded candidates can be projected back into the feasible callback space. Across all triggered repairs, roughly four out of five repaired candidates become final successful records. This suggests that repair is not only a convenience for handling syntax or interface mistakes, but also a stabilizing mechanism that lets the search retain useful solver logic proposed in imperfect executable form.

5 Conclusion

We introduced MILP-Evo, a framework for automatically designing MILP solver components as executable PySCIPOpt callbacks, which searches over native solver code and evaluates each candidate through end-to-end SCIP execution. This design keeps the discovered cut-selection and branching logic explicit, inspectable, and compatible with standard solver workflows. Our experiments show strong results on independent set, combinatorial auctions, and set cover, with competitive facility-location performance. The ablation and lineage analyses suggest that these gains come from joint callback evolution, non-monotone reuse of useful program motifs, and contract-preserving repair. More broadly, MILP-Evo points to a practical path for automatic solver design, where the learned object is explicit algorithmic logic shaped by end-to-end solver behavior rather than a black-box policy.

AI USE STATEMENT

MILP-Evo uses Gemini 3 Flash Preview as part of the proposed research method to generate, mutate, recombine, reflect on, and repair executable PySCIPOpt cut-selection and branching policies. The model also provides descriptors used for diversity-aware search. Candidate fitness is measured through SCIP executions rather than assigned by the language model. The exported policies are evaluated without further LLM calls.

The authors take full responsibility for the final manuscript, code, experimental results, and claims, including material produced with the assistance of generative AI.

REFERENCES

Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2): 405–421, 2021.

Zhi-Long Chen. Integrated production and outbound distribution scheduling: review and extensions. *Operations research*, 58(1):130–148, 2010.

Shuhan Du, Junbo Tong, and Wenhui Fan. Learning efficient branch-and-bound for solving mixed integer linear programs. *Applied Soft Computing*, 172:112863, 2025.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32, 2019.

Dongdong Ge, Qi Huangfu, Zizhuo Wang, Jian Wu, and Yinyu Ye. Cardinal Optimizer (COPT) userguide. https://guide.coap.online/copt/en-doc, 2023.

Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. Miplib 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3):443–490, 2021.

Prateek Gupta, Maxime Gasse, Elias Khalil, Pawan Mudigonda, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch. *Advances in neural information processing systems*, 33: 18087–18097, 2020.

Zeren Huang, Kerong Wang, Furui Liu, Hui-Ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming. *Pattern Recognition*, 123:108353, 2022.

Yufei Kuang, Jie Wang, Haoyang Liu, Fangzhou Zhu, Xijun Li, Jia Zeng, Jianye Hao, Bin Li, and Feng Wu. Rethinking branching on exact combinatorial optimization solver: The first deep symbolic discovery framework. In *The Twelfth International Conference on Learning Representations*, 2024a.

Yufei Kuang, Jie Wang, Yuyan Zhou, Xijun Li, Fangzhou Zhu, Jianye Hao, and Feng Wu. Towards general algorithm discovery for combinatorial optimization: Learning symbolic branching policy from bipartite graph. In *Forty-first International Conference on Machine Learning*, 2024b.

Sirui Li, Wenbin Ouyang, Max Paulus, and Cathy Wu. Learning to configure separators in branch-and-cut. *Advances in Neural Information Processing Systems*, 36:60021–60034, 2023.

Sirui Li, Janardhan Kulkarni, Ishai Menache, Cathy Wu, and Beibin Li. Towards foundation models for mixed integer linear programming. *arXiv preprint arXiv:2410.08288*, 2024.

Jiacheng Lin, Meng Xu, Zhihua Xiong, and Huangang Wang. Cambranch: Contrastive learning with augmented milps for branching. *arXiv preprint arXiv:2402.03647*, 2024.

Chang Liu, Zhichen Dong, Haobo Ma, Weilin Luo, Xijun Li, Bowen Pang, Jia Zeng, and Junchi Yan. L2p-mip: Learning to presolve for mixed integer programming. In *The Twelfth International Conference on Learning Representations*, 2024a.

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. *arXiv preprint arXiv:2401.02051*, 2024b.

Songsong Liu, Jose M Pinto, and Lazaros G Papageorgiou. A tsp-based milp model for medium-term planning of single-stage continuous multiproduct plants. *Industrial & Engineering Chemistry Research*, 47(20):7733–7743, 2008.

Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, 2017.

Kefan Ma, Liquan Xiao, Jianmin Zhang, and Tiejun Li. Accelerating an fpga-based sat solver by software and hardware co-design. *Chinese Journal of Electronics*, 28(5):953–961, 2019.

Stephen Maher, Matthias Miltenberger, João Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. Pyscipopt: Mathematical programming in python with the scip optimization suite. In *International Congress on Mathematical Software*, pp. 301–307. Springer, 2016.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Max Paulus and Andreas Krause. Learning to dive in branch and bound. *Advances in Neural Information Processing Systems*, 36:34260–34277, 2023.

Max B Paulus, Giulia Zarpellon, Andreas Krause, Laurent Charlin, and Chris Maddison. Learning to cut by looking ahead: Cutting plane selection via imitation learning. In *International conference on machine learning*, pp. 17584–17600. PMLR, 2022.

Pol Puigdemont, Stratis Skoulakis, Grigorios Chrysos, and Volkan Cevher. Learning to remove cuts in integer linear programming. *arXiv preprint arXiv:2406.18781*, 2024.

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

Mark Turner, Thorsten Koch, Felipe Serrano, and Michael Winkler. Adaptive cut selection in mixed-integer linear programming. *Open Journal of Mathematical Optimization*, 4:1–28, 2023.

Zhihai Wang, Xijun Li, Jie Wang, Yufei Kuang, Mingxuan Yuan, Jia Zeng, Yongdong Zhang, and Feng Wu. Learning cut selection for mixed-integer linear programming via hierarchical sequence model. *arXiv preprint arXiv:2302.00244*, 2023.

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

Huigen Ye, Hua Xu, An Yan, and Yaoyang Cheng. Large language model-driven large neighborhood search for large-scale milp problems. In *Forty-second International Conference on Machine Learning*, 2025.

Shuli Zeng, Sijia Zhang, Shaoang Li, Feng Wu, and Xiang-Yang Li. Beyond local selection: Global cut selection for enhanced mixed-integer programming. *arXiv preprint arXiv:2503.15847*, 2025.

Sijia Zhang, Shuli Zeng, Shaoang Li, Feng Wu, and Xiangyang Li. Learning to select nodes in branch and bound with sufficient tree representation. In *The Thirteenth International Conference on Learning Representations*, 2025.

A Implementation Details

A.1 Benchmark Instance Generation

We generate the MILP instances using the same learn2branch-style generation protocol as Gasse et al. (2019). The released generation script writes CPLEX LP-format instances under problem-specific directories with `train`, `valid`, `transfer`, and `test` prefixes. For each benchmark family, MILP-Evo searches on Easy-scale training instances and evaluates the exported callback on held-out Easy, Medium, and Hard instance scales. The generator creates 1000 Easy-scale training instances and 200 Easy-scale validation instances per family; the evaluation scripts then select the held-out instances used for the reported transfer experiments.

The four benchmark families instantiate different MILP structures:

Set covering. Given a universe of elements and a collection of subsets whose union contains the universe, the set covering problem asks for a subcollection that still covers all elements. In our MILP instances, each binary variable indicates whether one subset is selected, and the objective is to obtain a cover with minimum total cost.

Combinatorial auction. In a combinatorial auction, bidders submit bids on packages of discrete, heterogeneous items rather than only on individual items. The associated winner determination problem selects a compatible set of package bids that maximizes the auctioneer's revenue or total accepted value. In our MILP formulation, each binary variable corresponds to accepting one bid, and item-conflict constraints ensure that no item is assigned more than once.

Capacitated facility location. Facility location problems ask where facilities should be placed or opened so that demand points can be served at low cost. In the capacitated variant used here, the model chooses which candidate facilities to open and how to route customer demand to open facilities. The objective combines fixed opening costs and transportation costs, while capacity constraints limit the total demand that each open facility may serve.

Maximum independent set. In graph theory, an independent set is a set of vertices with no edge between any selected pair. The maximum independent set problem seeks an independent set of largest possible cardinality in a given graph. Our MILP uses one binary variable per vertex, edge constraints to prevent adjacent vertices from being selected together, and a cardinality-maximization objective.

Table 4 reports the main scale parameters used for the Easy, Medium, and Hard instance families. Additional generator constants follow the learn2branch defaults used in our script: set-covering density is 0.05 with objective coefficients sampled up to 100, the combinatorial-auction bundle-extension probability is 0.7, the facility-location capacity-to-demand ratio is 5, and independent-set graphs are generated by the Barabasi–Albert model with affinity 4.

A.2 Search and Evaluation Protocol

We use the independent-set benchmark as a representative example of the training and evaluation protocol. The same protocol is applied to the other problem families with problem-specific instance generators and callback evaluation files.

Training. For a given problem family, MILP-Evo searches over a population of executable Python artifacts. Each artifact jointly defines a cut selector and a branching rule, and both callbacks are registered in the same SCIP model during evaluation. The search is initialized with diverse LLM-generated candidates and then alternates between mutation, crossover, reflection, and repair. Candidate quality is never judged by the LLM itself: each program is compiled, checked against the callback contracts, executed inside SCIP on a small batch of training instances, and scored by mean process time when all instances are solved to optimality. Programs that violate the interface, raise callback errors, or fail the required solve condition receive infinite fitness.

The population is maintained as a quality-diversity archive. Language-model feedback is used to propose code edits and to describe candidates, while selection within the archive is determined by

Table 4: Instance generation hyperparameters for the benchmark families.

Benchmark	Hyperparameters
Set covering	Easy: 500 rows, 1000 columns Medium: 1000 rows, 1000 columns Hard: 2000 rows, 1000 columns
Combinatorial auction	Easy: 100 items, 500 bids Medium: 200 items, 1000 bids Hard: 300 items, 1500 bids
Capacitated facility location	Easy: 100 facilities, 100 customers Medium: 100 facilities, 200 customers Hard: 100 facilities, 400 customers
Maximum independent set	Easy: 500 nodes, affinity 4 Medium: 1000 nodes, affinity 4 Hard: 1500 nodes, affinity 4

Table 5: Final hyperparameters shared across all four benchmark families. LLM sampling parameters not explicitly listed in the request use the provider defaults.

Hyperparameter	Value
Language-model configuration	
LLM backbone	Gemini 3 Flash Preview
Maximum output tokens	32,768
Temperature / top-p	Provider default
Evolutionary search	
Global search iterations	400
Number of islands	4
Population size per island	20
Cold-start individuals per island	4
Mutation probability	0.4
Crossover probability	0.6
MAP-Elites bins per descriptor	3
Elites retained per cell	2
Migration interval	25 iterations
Migrants per island and event	1
Training and evaluation	
Training instances per candidate	10
Training SCIP seed	0
Candidate-evaluation timeout	600 s
Evaluation instances per scale	10
Evaluation SCIP seeds	$\{0, 1, 2, 3, 4\}$
Per-instance evaluation time limit	3,600 s

SCIP execution. This separation is important: the LLM explores the program space, but the solver supplies the fitness signal that drives selection.

To make the search interface concrete, Config 1 and Prompt 2 provide shortened excerpts from the independent-set configuration and task prompt. The configuration excerpt shows how the benchmark, search budget, and population structure are specified. The prompt excerpt shows how the LLM is instructed to produce one joint callback artifact and to respect the PySCIPOpt callback contracts. The full configuration files and prompts are included in the supplementary material.

```
mutation_rate: 0.4
crossover_rate: 0.6
batch_size: 10
```

```
instances:
  source: learn2branch
  problem: indset
  limit: 10
  scip_seed: 0

population:
  iterations: 400
  population_size: 20
  island_num: 4
  bins_num: 3
```

Config 1: Shortened independent-set search configuration excerpt.

```
You are an expert in MILP solver design for maximum independent set
(stable set) instances. Your task is to co-evolve two SCIP plugins:
(1) a custom cut selector (CustomCutsel) that orders or filters
candidate cuts at separation, and
(2) a custom branching rule (CustomBranchingRule) that decides how to
branch on fractional LP candidates.

Both plugins run together on the same PySCIPOpt Model. One submission
must define both classes in the same module. CustomCutsel must return a
dictionary with keys cuts, nselectedcuts, and result, where cuts is only
a reordering or subset of the input cut pool. CustomBranchingRule must
branch when LP branching candidates exist and return SCIP_RESULT.BRANCHED
    .

Fitness is the mean process time to reach optimality over the evaluation
batch. Co-evolution means joint changes to cut selection and branching
must improve end-to-end SCIP behavior, not only one component in
    isolation.
```

Prompt 2: Shortened independent-set task prompt excerpt.

Evaluation. After search, the best finite-fitness program for each family is exported as the family-specific policy. No further LLM calls or adaptation are performed during evaluation. The exported callbacks are loaded into fresh SCIP models and tested on held-out learn2branch transfer instances at Easy, Medium, and Hard scales. We use the same joint registration path as in training and report solver status, process time, and branch-and-bound node counts under the time limit described in the main text.

B Best Discovered Cut-Branch Callback

This section presents the best joint cut-selection and branching callback artifact discovered by MILP-Evo. The artifact is an executable PySCIPOpt program registered inside SCIP during evaluation. It contains both the cut selector and the branching rule used for the corresponding benchmark family. For compactness, we show the independent-set artifact in full because it is the setting where MILP-Evo obtains its strongest gains and where we also report detailed search dynamics in the main paper. The complete exported programs for all four benchmark families are included in the supplementary material under `problem/`.

Program 3: Best discovered joint cut-selection and branching callback for independent set.

```
from pyscipopt import Model, Branchrule, SCIP_RESULT
from pyscipopt.scip import Cutsel

class CustomCutsel(Cutsel):
    def __init__(self):
        super().__init__()
```

```
    def cutselselect(self, cuts, forcedcuts, root, maxnselectedcuts):
        if not cuts:
            return {"cuts": [], "nselectedcuts": 0, "result": SCIP_RESULT
    .SUCCESS}

        scored_cuts = []
        for c in cuts:
            eff = self.model.getCutEfficacy(c)
            nnz = c.getNNonz()
            score = eff / (nnz + 1.0)**0.15
            scored_cuts.append((c, score))

        scored_cuts.sort(key=lambda x: x[1], reverse=True)

        best_score = scored_cuts[0][1]
        threshold = 0.85 * best_score

        n_pass = 0
        for _, score in scored_cuts:
            if score >= threshold:
                n_pass += 1
            else:
                break

        pool_limit = max(1, len(cuts) // 3)
        final_n = min(n_pass, maxnselectedcuts, pool_limit)

        return {
            "cuts": [x[0] for x in scored_cuts],
            "nselectedcuts": int(final_n),
            "result": SCIP_RESULT.SUCCESS
        }

class CustomBranchingRule(Branchrule):
    def __init__(self, model):
        super().__init__()
        self.model = model

    def branchexeclp(self, additional_cons):
        cands, sols, _, nlpcands, _, _ = self.model.getLPBranchCands()

        if nlpcands == 0:
            return {"result": SCIP_RESULT.DIDNOTRUN}

        best_cand = None
        best_score = (-1, -1.0, 1e12, -1.0) # (locks, obj, -index, frac)

        for i in range(nlpcands):
            v = cands[i]
            s = sols[i]

            locks = v.getNLocksUp()
            obj = v.getObj()
            idx = v.getIndex()
            frac = s * (1.0 - s) # maximized at 0.5

            current_score = (locks, obj, -idx, frac)

            if current_score > best_score:
                best_score = current_score
                best_cand = v

        if best_cand is not None:
            self.model.branchVar(best_cand)
            return {"result": SCIP_RESULT.BRANCHED}
```

```
        else:
            return {"result": SCIP_RESULT.DIDNOTRUN}
```

B.1 Code-Level Lineage Analysis

We trace the recorded ancestry of the best independent-set artifact to examine how solver logic emerges during evolution. The lineage is a fork–join directed acyclic graph rather than a monotone chain of improvements. The G0 seed uses efficacy-only cut ranking, a top-quarter cap, and smallest-index branching. A G2 crossover worsens fitness from 5.309 to 8.182, but introduces density-aware cut scoring that is later reused. This regression provides a concrete example of non-monotone exploration.

The lineage then splits into a lock-count-first branch and an auxiliary cut-calibration branch. Their final crossover attains fitness 2.331, a 56.1% reduction relative to the seed. Its cut selector uses $s_{\text{final}}(c) = \text{efficacy}(c)/(\text{nnz}(c)+1)^{0.15}$, retains cuts scoring at least 85% of the best candidate, and applies a one-third pool cap. The threshold makes the selected fraction adaptive, while the cap limits LP growth.

For a branching candidate v, let $n_{\text{up}}(v)$ denote its number of upward rounding locks, and let $w(v)$, $i(v)$, and $f(v) = x_v(1 - x_v)$ denote its objective coefficient, SCIP index, and fractionality score, respectively. The final branching rule uses the lexicographic score $b(v) = (n_{\text{up}}(v), w(v), -i(v), f(v))$. These quantities correspond to `getNLocksUp()`, `getObj()`, `getIndex()`, and the current LP solution value in the exported program.

Code-level inspection clarifies which terms are operational for this benchmark. The independent-set generator assigns the same objective coefficient to every vertex variable, so $w(v)$ does not distinguish branching candidates. Moreover, SCIP variable indices are unique and precede fractionality in the lexicographic tuple. The effective rule on this benchmark is therefore $b_{\text{eff}}(v) = (n_{\text{up}}(v), -i(v))$: it first maximizes the number of upward locks and then selects the candidate with the smallest SCIP index.

In the independent-set formulation, an upward lock indicates that increasing a binary vertex variable may violate a packing or clique constraint. We therefore interpret $n_{\text{up}}(v)$ as a formulation-aware proxy for constraint incidence or conflict pressure, rather than as the exact degree of the corresponding vertex. This qualification matters because the instance generator replaces some edge inequalities with aggregated clique inequalities before SCIP processes the model.

This lineage illustrates how the architecture encourages family-specific solver code: generation is conditioned on the target family, native SCIP quantities are exposed to the program, and cutting and branching are evaluated jointly by complete solver runs. End-to-end fitness can therefore preserve useful fragments even when an intermediate artifact is slower. Because the final program remains explicit, we can separate operational structural signals from syntactically present but inactive terms. The evidence supports the discovery and recombination of readable, family-conditioned solver motifs, not the stronger claim that every generated feature contributes independently.

C Additional Experimental result

C.1 Result Variability and Stability

Table 6 compares standard deviations across methods on the same held-out instances used in the main evaluation. Because the test set is fixed for all methods, the reported deviations allow us to compare how much each method's runtime varies across the same group of instances. These deviations should not be interpreted as solver randomness alone: the instances within a benchmark family are fixed, but they can still differ substantially in difficulty. Consistent with the main table, the discussion below focuses on the primary comparison methods; GNN-GPU and COPT are included as external references.

For process time, MILP-Evo has the smallest standard deviation among the primary methods in seven of the twelve family-scale settings and is among the three lowest in most settings. The pattern is clearest on independent set and on the easy and medium combinatorial-auction instances, where

low variance is paired with strong win counts and low mean runtime. This suggests that the improvement is not driven only by a few unusually easy instances: in these regimes, the discovered callback tends to reduce runtime consistently across the evaluation set. Similar runtime-stability behavior appears on the easy and medium set-covering instances and on medium-scale facility location.

The comparison also shows where this claim should be qualified. Very small runtime deviations for some baselines on hard instances can reflect consistent timeouts rather than robust solving, so standard deviation must be read together with mean time and win counts. Node counts require the same joint reading. Across several family-scale settings, methods that explore fewer branch-and-bound nodes are not necessarily faster, and MILP-Evo sometimes reaches a lower mean runtime while visiting more nodes. This pattern is consistent with a tradeoff between tree size and per-node overhead: a policy can reduce the number of nodes by spending more effort on branching or cut decisions, while a cheaper callback can traverse a larger tree but still finish sooner.

Thus, the larger node-count deviations of MILP-Evo in some settings should not be read as a direct contradiction of its runtime stability. They indicate that the evolved callbacks can produce more instance-dependent search trees, whereas the process-time columns show whether those trees are traversed efficiently enough to improve wall-clock performance. This distinction is most favorable on independent set and combinatorial auctions, where higher or more variable node counts often coincide with low mean runtime. On harder set-covering and facility-location instances, however, both runtime and node-count dispersion are larger, suggesting that transfer stability remains more problem-dependent outside the families where the learned solver behavior aligns most strongly with the instance structure.

C.2 Statistical Significance Tests

To avoid treating the five SCIP seeds for the same MILP instance as independent observations, we first average the five seed-level penalized runtimes for each instance. Non-optimal runs are assigned the 3600-second limit, yielding ten paired instance-level observations per method in each family–scale setting. We compare MILP-Evo with each of the six primary baselines using two-sided exact Wilcoxon signed-rank tests and apply Holm correction across the six comparisons within each setting. GNN-GPU and COPT are external references and are excluded from these tests.

Table 7 reports the comparison with the fastest primary baseline in each setting. W/T/L denotes the number of instances on which MILP-Evo is faster, tied, or slower after averaging over seeds.

After correction, MILP-Evo is significantly faster than the fastest primary baseline on all three independent-set scales and on the Easy and Medium scales of combinatorial auctions and set cover. The remaining five comparisons do not reach statistical significance. These nonsignificant results indicate insufficient evidence of a difference and should not be interpreted as evidence of equivalence.

C.3 Additional Search Curves

Figure 3 reports the search traces for all four benchmark families. Each curve tracks the best fitness found during the evolutionary search, providing a family-level view of how useful callback programs emerge beyond the independent-set example shown in the main text. Since the plotted value is the best archived fitness, the curves are monotone and should be read as discovery traces rather than as the performance of every sampled program.

The curves share a staircase pattern: long plateaus are interrupted by a small number of large drops. This behavior is consistent with program search in a constrained solver API. Many candidate edits are either invalid, behaviorally neutral, or only locally useful; occasionally, however, a structural change in cut filtering, branching priority, or their interaction produces a qualitatively better callback. After such a candidate is found, the archive retains it and subsequent search starts from a stronger incumbent.

The family-level dynamics differ in ways that match the quantitative results. Combinatorial auctions improve rapidly in the early part of search and then enter a long low-fitness plateau, suggesting that the search finds an effective solver strategy relatively quickly and later iterations mostly refine it. Independent set also improves early, but unlike combinatorial auctions it continues to ob-

Table 6: Main comparative results with standard deviations. Time and node counts are reported as mean±std over the same evaluation runs used for the main results. Wins and boldface follow the convention of the main-paper results table: GNN-GPU and COPT are external references and are excluded from primary win-count and boldface comparisons.

Indset:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	87.00±78.11	1/50	30.2±27.5	3318.97±789.79	0/7	150.5±46.0	3600.06±0.05	0/0	53.9±16.5
RPB	9.49±3.77	0/50	46.9±67.6	194.74±189.02	0/50	7442.9±8833.1	2587.99±792.07	0/15	75329.3±28439.3
GNN	5.80±2.55	0/50	62.2±79.1	1673.24±1473.07	0/36	19872.9±23832.4	3600.01±0.01	0/0	49010.6±9406.4
Hybrid	6.47±1.45	0/50	53.1±55.4	1502.27±1143.77	0/30	48887.3±37997.5	3600.01±0.01	0/0	42854.8±7286.9
GS4CO	7.29±2.27	0/50	54.1±50.7	147.26±93.78	0/50	2785.0±3313.0	2057.08±955.49	0/30	67083.8±42269.5
SCIP	14.80±5.91	0/50	164.6±141.2	377.99±254.89	0/50	4023.9±3369.8	3155.52±715.13	0/18	22613.1±6737.8
MILP-Evo	**2.57±1.04**	**49/50**	84.7±84.0	**59.20±47.27**	**50/50**	1051.9±1040.4	**806.17±679.78**	**50/50**	17980.3±19471.1
GNN-GPU	4.63±1.24	-/50	62.2±79.1	858.07±1214.30	-/45	57818.8±89060.7	3600.00±0.00	-/0	107167.4±20945.0
COPT	2.62±1.69	-/50	9.6±12.5	62.18±49.60	-/50	707.7±724.9	1531.27±1169.88	-/43	10279.2±7357.4

Cauctions:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	10.80±4.39	0/50	25.5±17.2	286.99±220.88	0/50	253.1±165.5	3177.54±784.12	0/14	912.3±231.9
RPB	3.70±1.33	0/50	14.8±12.7	21.69±8.98	1/50	1169.0±1072.4	**271.44±171.11**	**36/50**	17918.5±12358.2
GNN	3.13±1.33	0/50	105.8±65.0	30.05±22.76	0/50	1047.9±893.2	662.39±451.27	0/50	17571.3±12029.1
Hybrid	2.07±0.65	0/50	111.2±70.0	21.26±15.91	0/50	1210.7±1069.5	718.47±565.48	0/50	30404.5±24134.5
GS4CO	3.07±1.46	0/50	114.3±82.8	42.55±41.19	0/50	1353.1±1435.9	1248.98±866.53	0/45	24867.6±16419.1
SCIP	3.57±1.34	0/50	30.1±19.8	44.96±24.76	0/50	1169.9±984.0	307.55±172.59	3/50	15560.5±10578.3
MILP-Evo	**0.81±0.39**	**50/50**	196.1±96.7	**11.79±8.53**	**49/50**	1536.0±1087.2	381.90±396.98	11/50	34534.7±32040.0
GNN-GPU	1.60±0.43	-/50	105.8±65.0	10.36±5.62	-/50	1047.9±893.2	191.11±132.17	-/50	17571.7±12029.4
COPT	2.18±0.74	-/50	126.8±118.9	14.11±6.61	-/50	1542.2±1246.8	161.34±97.05	-/50	19687.7±13324.3

Setcover:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	29.07±32.68	0/50	45.7±59.1	1291.03±1153.67	0/45	929.5±640.3	3600.03±0.02	0/0	658.3±330.6
RPB	11.20±4.61	0/50	129.6±249.7	162.70±233.22	0/50	10349.9±17486.7	**2401.92±876.57**	6/20	118275.4±52191.2
GNN	11.32±8.39	0/50	172.7±182.5	487.64±709.82	0/50	5990.5±8593.8	3500.75±300.99	0/5	21036.2±3297.5
Hybrid	8.26±3.59	0/50	201.9±210.3	171.41±225.19	0/50	7240.7±9825.5	3079.13±1055.24	0/10	63408.9±28316.7
GS4CO	13.89±10.38	0/50	249.1±268.4	609.94±836.20	0/45	10303.9±13451.6	2937.46±189.53	0/5	30500.2±5994.1
SCIP	14.18±8.24	0/50	124.5±229.6	274.02±309.37	0/50	8361.4±12497.3	2955.81±982.01	0/20	115143.9±50744.5
MILP-Evo	**3.14±2.26**	**50/50**	269.0±242.4	**89.36±110.26**	**50/50**	11924.9±16808.0	2641.34±1158.47	**19/25**	188362.2±112933.8
GNN-GPU	5.31±2.14	-/50	172.7±182.5	102.20±138.75	-/50	5990.5±8593.8	2492.68±1269.69	-/20	102635.8±54080.0
COPT	6.88±3.32	-/50	297.4±412.9	99.64±115.85	-/50	12207.1±17679.6	2718.71±1135.03	-/25	175771.1±80573.4

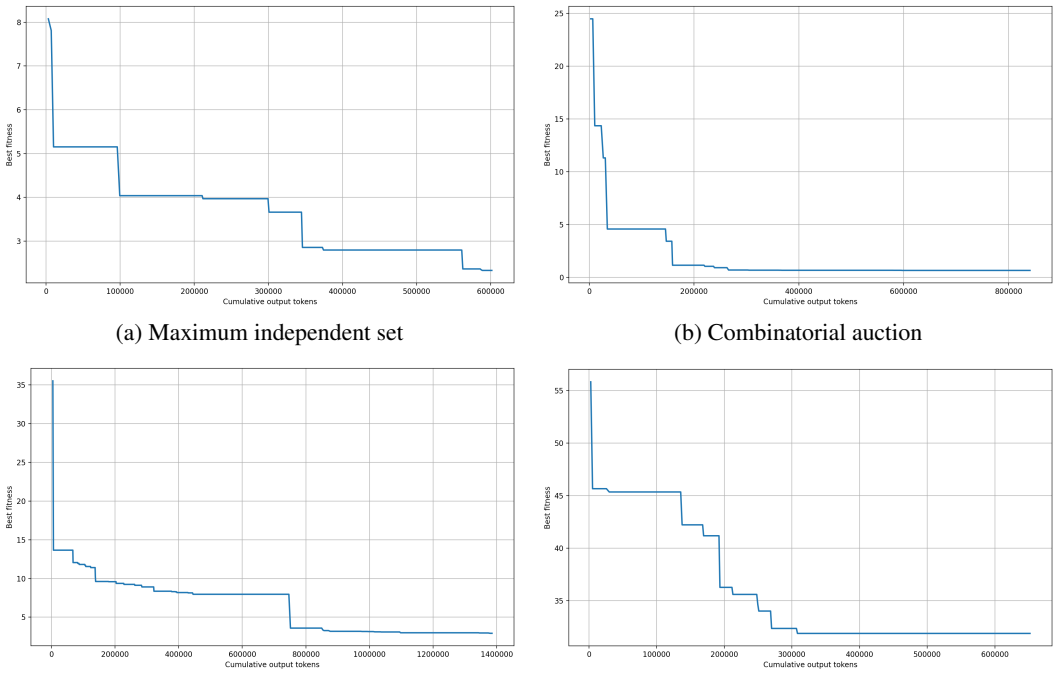
Facilities:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	81.02±59.56	0/50	71.0±61.1	806.48±669.60	0/50	233.9±223.3	2108.21±1143.00	0/41	109.6±77.2
RPB	36.03±25.70	16/50	90.4±110.2	305.37±232.06	6/50	588.6±645.0	**607.57±221.14**	15/50	260.1±171.7
GNN	50.48±41.74	0/50	287.6±271.4	478.05±510.82	7/50	881.4±827.2	669.15±321.01	8/50	462.3±300.8
Hybrid	**35.10±28.81**	**17/50**	253.2±244.9	325.92±284.35	14/50	895.3±896.8	631.09±296.42	**17/50**	456.6±297.0
GS4CO	68.69±64.17	0/50	273.6±267.3	558.56±475.23	1/50	933.8±861.2	731.28±387.37	4/50	449.9±294.0
SCIP	38.27±22.93	5/50	99.5±84.3	305.10±241.99	2/50	541.1±606.3	675.33±284.41	3/50	194.6±148.3
MILP-Evo	35.66±23.35	12/50	235.5±237.6	**266.21±218.47**	**20/50**	926.8±957.4	761.88±360.30	3/50	429.1±288.0
GNN-GPU	33.44±24.30	-/50	287.6±271.4	267.77±220.73	-/50	881.4±827.2	546.13±258.44	-/50	462.3±300.8
COPT	8.20±5.79	-/50	221.0±194.4	31.56±27.88	-/50	572.0±587.5	58.06±27.46	-/50	262.8±164.6

tain meaningful late-stage gains; this is consistent with the strong transfer performance of the final independent-set callback. Set covering and facility location show longer plateaus and smaller late improvements. Their curves indicate that the search still discovers better programs, but improvements arrive less frequently and the final incumbents are less dominant, matching the more mixed results in Table 6.

Overall, the search curves indicate that progress is uneven and problem-dependent. All four families exhibit incumbent improvements after initialization, but the timing and magnitude of these improvements vary substantially. Combinatorial auctions and independent set show clearer early or sustained gains, whereas set covering and facility location spend longer periods on plateaus and obtain smaller late-stage improvements. This pattern is consistent with the evaluation results, where transfer performance is strongest on the families with clearer search progress.

Table 7: Paired Wilcoxon comparisons between MILP-Evo and the fastest primary baseline in each family–scale setting. W/T/L counts paired instances on which MILP-Evo is faster, tied, or slower. Raw p values are two-sided; Holm-adjusted values account for comparisons with all six primary baselines within each setting.

Family	Scale	Baseline	W/T/L	Raw p	Holm p	Result
Indset	Easy	GNN	10/0/0	0.00195	**0.0117**	Faster
Indset	Medium	GS4CO	10/0/0	0.00195	**0.0117**	Faster
Indset	Hard	GS4CO	10/0/0	0.00195	**0.0117**	Faster
Cauctions	Easy	Hybrid	10/0/0	0.00195	**0.0117**	Faster
Cauctions	Medium	Hybrid	10/0/0	0.00195	**0.0117**	Faster
Cauctions	Hard	RPB	3/0/7	0.69531	1.0000	No significant difference
Setcover	Easy	Hybrid	10/0/0	0.00195	**0.0117**	Faster
Setcover	Medium	RPB	10/0/0	0.00195	**0.0117**	Faster
Setcover	Hard	RPB	3/5/2	1.00000	1.0000	No significant difference
Facilities	Easy	Hybrid	5/0/5	1.00000	1.0000	No significant difference
Facilities	Medium	SCIP	7/0/3	0.16016	0.5234	No significant difference
Facilities	Hard	RPB	3/0/7	0.37500	1.0000	No significant difference



(a) Maximum independent set

(b) Combinatorial auction

(c) Set covering

(d) Capacitated facility location

Figure 3: Additional search curves across the four benchmark families. Lower fitness values indicate better candidate callback programs under the search objective.

D Broader Impacts

MILP-Evo studies automatic design of explicit solver components for mixed-integer linear programming. The most direct positive impact is computational: better domain-specialized cut-selection and branching rules can reduce the time and energy required to solve recurring optimization problems. Because the discovered artifacts are executable PySCIPOpt callbacks rather than opaque neural predictors, they can be inspected, modified, and deployed within standard solver workflows. This transparency may make learned solver customization more usable for researchers and practitioners who need to audit optimization logic before using it in production systems.

The same capability also has potential risks. Faster MILP solving can improve decision systems in domains such as logistics, scheduling, manufacturing, network design, and resource allocation;

Table 8: Used assets and their licenses.

Asset (type)	License or terms; usage
learn2branch (Gasse et al., 2019) (dataset/code)	MIT License; instance generation and evaluation
Hybrid-learn2branch (Gupta et al., 2020) (code)	MIT License; baseline evaluation
GS4CO (Kuang et al., 2024a;b) (code)	No explicit root license; baseline evaluation
PySCIPOpt 6.1.0 (Maher et al., 2016) (code)	MIT License; callback implementation
SCIP 10.0.2 (Achterberg, 2009) (solver)	Apache-2.0 License; MILP solving
SoPlex 8.0.2 (solver)	Apache-2.0 License; LP solving
COPT (Ge et al., 2023) (solver)	Proprietary commercial solver used under a COPT academic license; external reference

some of these deployments may affect people if the underlying mathematical model encodes biased objectives, incomplete constraints, or sensitive business rules. Our method optimizes solver behavior, not the social desirability of the optimization model being solved. Thus, an automatically designed solver component should not be interpreted as a guarantee that the resulting decision process is fair, safe, or appropriate for a high-stakes application.

There are also technical risks specific to LLM-generated solver code. A candidate callback may be syntactically valid and fast on benchmark instances while still containing brittle assumptions, solver-version dependencies, or edge-case behavior that appears only on unseen instance distributions. We mitigate these risks in the research setting by using public synthetic benchmark families, running candidates inside SCIP with explicit callback-contract checks, assigning invalid or unsuccessful runs infinite fitness during search, and evaluating the final artifact by end-to-end solver execution. For any deployment beyond benchmark evaluation, we recommend manual inspection of the exported callback, stress testing on representative and adversarial instances, solver-version pinning, and domain-specific review of the MILP formulation and objective.

This work does not use private data, human-subject data, or scraped personal information. The experiments are conducted on standard synthetic MILP benchmark families, and the proposed method does not release a model that directly generates natural-language content for end users.

E Assets and Licenses

Table 8 lists the existing assets used in this work and their licenses or terms. We retain third-party notices and do not redistribute commercial solvers or code whose redistribution terms are not explicit.

All benchmark instances used in this work are synthetic MILP instances from standard learn2branch-style distributions. They do not contain personal data, human-subject data, copyrighted media, or scraped web content.