Process-Aware LNS for Large-Scale MILP via Context-Enhanced Fine-Tuned LLM-driven Selector

An Yan[1,2]* **Huigen Ye**[1]* **Hua Xu**[1]† **Jiahao Zhang**[1,3]

[1]Department of Computer Science and Technology, Tsinghua University
[2]Institute for Interdisciplinary Information Sciences, Tsinghua University
[3]The Hong Kong University of Science and Technology (Guangzhou)

Abstract

Solving large-scale Mixed-Integer Linear Programming (MILP) problems involving millions of variables is critical for industrial applications but notoriously intractable due to combinatorial explosion. While Large Neighborhood Search (LNS) has emerged as the premier heuristic strategy for tackling such scales, its efficacy hinges entirely on the underlying neighborhood selection mechanism. Recent Large Language Model (LLM)-driven LNS frameworks show remarkable promise but suffer from fundamental flaws: they lack generalizability across diverse problem classes and rely on static selection paradigms that remain entirely blind to the shifting dynamics of the iterative optimization process. To overcome this rigidity, we propose PALL, a Process-Aware Evolutionary LNS framework guided by a Context-Enhanced Fine-Tuned LLM-driven Selector. Instead of deploying a monolithic operator, PALL leverages an offline-evolved diverse operator ensemble and dynamically synchronizes the search strategy with evolving optimization states. Specifically, we fine-tune an 8B-parameter LLM on high-quality hindsight oracle trajectories to adaptively perform optimal operator selection. To bolster the LLM's sequential decision-making, we introduce a context-enhanced reasoning mechanism that leverages historical operator trajectories and optimization rewards as inferential feedback. Extensive evaluations on standard million-variable MILP benchmarks demonstrate that PALL achieves state-of-the-art performance. Supported by an asynchronous deployment strategy, it not only consistently outperforms commercial solvers like Gurobi and classical LNS algorithms but also delivers over a $6\times$ acceleration compared to state-of-the-art LLM-driven LNS frameworks.

1 Introduction

Mixed-Integer Linear Programming (MILP) stands as the cornerstone of modern combinatorial optimization, providing a profoundly versatile mathematical framework for modeling a myriad of complex real-world systems, ranging from supply chain logistics [Leon-Olivares et al., 2020] and production scheduling [Askari-Nasab et al., 2011] to energy grid management [Kassab et al., 2024]. Despite its unparalleled modeling expressiveness, solving MILP is intrinsically NP-hard [Kim et al., 2021]. As industrial applications increasingly demand the optimization of large-scale systems—often involving millions of decision variables and constraints—the underlying solution space expands exponentially, rendering the problem notoriously intractable. Consequently, developing scalable and computationally efficient MILP solving methodologies has remained a persistent and prominent challenge at the intersection of operations research (OR) and artificial intelligence (AI), driving continuous and intensive research efforts [Xu et al., 2025, Ye et al., 2026].

*Equal contribution.
†Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Historically, methodologies for solving MILPs can be broadly classified into two primary paradigms: exact methods and heuristic algorithms [Zhang et al., 2023]. Exact approaches, predominantly driven by Branch-and-Bound [Lawler and Wood, 1966] and Branch-and-Cut [Lucena and Beasley, 1996] algorithms, provide theoretical guarantees of global optimality and form the backbone of modern commercial solvers. However, when confronted with the aforementioned large-scale scenarios, these exact paradigms inevitably succumb to severe combinatorial explosion [Crainic et al., 2006]. The sheer volume of the search tree often prevents them from discovering even a trivial feasible solution within practical time limits. Consequently, the research community has increasingly pivoted towards heuristic algorithms [Fischetti et al., 2010]. Given the immense difficulty of obtaining high-quality solutions in a single computational pass, iterative heuristics have garnered significant attention. Within this domain, while techniques like the Feasibility Pump [Berthold et al., 2019] are instrumental for rapidly identifying initial feasible solutions, Large Neighborhood Search (LNS) [Hendel, 2022] has firmly established itself as the de facto standard for iterative improvement. By iteratively alternating between destroying a specific fragment of the current solution and repairing it to explore neighboring sub-spaces [Pisinger and Ropke, 2018], LNS effectively mitigates the curse of dimensionality, rendering it the most favored and promising strategy for tackling large-scale MILP.

The efficacy of an LNS framework hinges critically upon its neighborhood selection strategy—the core mechanism that determines which subsets of variables to destroy and reconstruct. Early research primarily relied on human-designed heuristics, such as RINS [Danna et al., 2005] and ACP [Ye et al., 2023a], which are rigidly constrained by hand-crafted rules and fundamentally lack the capability to adaptively learn from data. In recent years, data-driven approaches, particularly Imitation Learning (IL) [Sonnerat et al., 2021, Nair et al., 2020] and Reinforcement Learning (RL) [Wu et al., 2021, Song et al., 2020], have emerged as promising alternatives. However, in the realm of large-scale MILP, these traditional learning paradigms encounter severe bottlenecks: IL requires the prohibitive computational expense of collecting expert demonstration data on massive instances [Huang et al., 2023], while RL struggles to achieve convergence within intractably high-dimensional state-action spaces [Beggs, 2005]. Capitalizing on the advent of Large Language Models (LLMs), recent state-of-the-art (SOTA) methodologies have pioneered the integration of LLMs with Evolutionary Computation (EC) to automatically generate and evolve LNS operators, demonstrating remarkable potential [Ye et al., 2025]. Crucially, however, these LLM-driven frameworks suffer from two fundamental limitations. First, they necessitate an isolated and computationally expensive operator evolution process from scratch for every specific problem class. Second, and more profoundly, they operate under a rigid, static solving paradigm. By deploying a monolithic selection strategy throughout the entire search, these methods remain entirely blind to the shifting dynamics of the iterative optimization process. Given that different optimization phases—such as early exploration, mid-term stagnation, and late-stage convergence—inherently necessitate distinct neighborhood operators, relying on a static strategy severely limits search efficiency and exacerbates the vulnerability of being trapped in local optima.

To address these two limitations, we propose **PALL**, **Process-Aware Evolutionary LNS for Large-Scale MILP via a Context-Enhanced Fine-Tuned LLM-driven Selector**. Fundamentally departing from the reliance on a monolithic operator, PALL introduces a novel and synergistic paradigm: it first establishes a diverse ensemble of evolved neighborhood operators, and subsequently deploys a fine-tuned LLM as an intelligent dispatcher during the solving phase. We evolve the operator bank and fine-tune the selector on small-scale instances from all four benchmark families, then reuse the same bank and selector across these families at the million-variable scale. By leveraging the inferential prowess of the LLM to sequentially select the optimal operator based on evolving optimization states, we transition LNS from a rigid heuristic into a dynamic, context-aware, and self-adapting system. Our contributions are summarized as follows:

- **Tail-Aware Stepwise Operator Evolution:** To capture the shifting dynamics of the LNS lifecycle, we propose a novel offline ensemble-level evolutionary procedure. Rather than evolving a single best operator, we optimize a complementary bank of diverse operators evaluated by their stepwise coverage of LNS states. This paradigm guarantees that the resulting ensemble can collectively address diverse phases—from early exploration to tail-end stagnation.
- **Hindsight Oracle and Experience-Driven LLM Fine-Tuning:** To bridge the gap between static operators and dynamic selection, we design a hindsight selecting oracle that extracts optimal operator selection trajectories from small-scale MILP instances. For the first time, we fine-tune an 8B-parameter LLM on these high-quality oracle traces, intrinsically

equipping the model with the capability to autonomously map complex optimization states to the optimal operator and seamlessly transfer this selection policy to large-scale deployment.

- **Context-Enhanced Reasoning for Sequential Decision-Making:** To amplify the LLM's reasoning capacity, we formulate the operator selection as a context-conditioned sequential decision problem. During inference, the LLM observes not only the current problem state but also a compact historical context of recent actions and their normalized optimization rewards. This feedback mechanism empowers the model to dynamically evaluate the shifting effectiveness of the operator bank, significantly boosting its sequential reasoning.
- **State-of-the-Art Scalability via Asynchronous Deployment:** Extensive evaluations on million-variable MILP benchmarks demonstrate PALL's unparalleled superiority. By decoupling the offline operator evolution from the online search and implementing an asynchronous LLM deployment strategy to effectively mask inference latency, our framework consistently outperforms commercial solvers like Gurobi and achieves over a $6\times$ acceleration compared to state-of-the-art LLM-driven LNS frameworks.

2 Related Work

Large-Neighborhood Search for MILP. Mixed-Integer Linear Programming (MILP) optimizes a linear objective over discrete and continuous variables subject to linear constraints. To efficiently find high-quality primal solutions, Large Neighborhood Search (LNS) is widely employed. Mathematically, given an incumbent solution $\hat{x}$, LNS iteratively unfixes a subset of variables (defining a neighborhood $\mathcal{N}$) and re-optimizes the induced restricted sub-MILP. Traditionally, $\mathcal{N}$ relies on explicit distance constraints [Fischetti and Lodi, 2003] or linear relaxations [Danna et al., 2005]. Adaptive LNS uses online feedback to select among neighborhood heuristics [Hendel, 2022, Cai et al., 2025]. Learning-based approaches construct neighborhoods through imitation and reinforcement learning [Song et al., 2020, Wu et al., 2021, Sonnerat et al., 2021] or contrastive learning [Huang et al., 2023], with related developments in learned local branching [Liu et al., 2022] and BTBS-LNS [Yuan et al., 2025]. GNN&GBDT and Light-MILPopt further study learning-based optimization at the million-variable scale [Ye et al., 2023b, 2024].

LLM-Driven Heuristic Generation. Recent advances leverage large language models for automated heuristic design, successfully searching for novel scoring functions and executable code [Yang et al., 2024, Romera-Paredes et al., 2023, Liu et al., 2024]. In the context of MILP, LLM-LNS [Ye et al., 2025] pioneers this paradigm by evolving programmatic neighborhood-selection heuristics. However, a critical limitation persists: these approaches inherently treat the generated heuristic as a final, static artifact. They aim to find a single "silver bullet" rule to be applied universally across the entire optimization process. This monolithic deployment fails to adapt to the shifting dynamics of LNS. Our work breaks this bottleneck. Rather than relying on a solitary static heuristic, we shift the deployment paradigm: we evolve a complementary bank of diverse programs and leverage a fine-tuned LLM as a selector to dynamically orchestrate them based on real-time optimization states.

Positioning of PALL. Fundamentally, PALL bridges the gap between automated heuristic design and dynamic selection. While prior works view heuristic generation as the end goal, we treat it merely as the foundation to construct a diverse operator ensemble. We combine a complementary bank of evolved programs with a selector fine-tuned on small-scale hindsight trajectories. The same bank and selector are reused across problem families, with online decisions guided by the current optimization state and recent action–reward history.

3 Method

The proposed framework, **PALL**, fundamentally transforms Large Neighborhood Search (LNS) from a static neighborhood rule into a context-conditioned operator-selection problem. Instead of committing to a monolithic neighborhood operator for the entire run, we maintain an offline-evolved diverse operator ensemble and fine-tune an LLM-driven selector that chooses among them according to the current optimization state. Figure 1 provides a system-level overview, while Figures 2 and 3 zoom into the offline operator-bank evolution and the teacher-supervised LLM selector training,

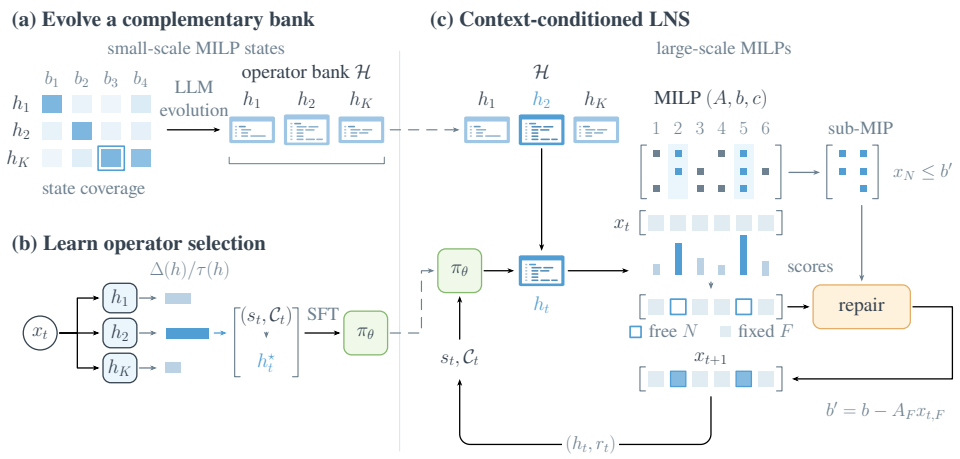

Figure 1: **Overview of PALL.** PALL casts LNS as a context-conditioned operator-selection problem $h_t = \pi_\theta(s_t, \mathcal{C}_t)$ over an offline-evolved operator bank $\mathcal{H}$, with the selected operator driving an identical sub-MIP repair routine. *(a) Tail-Aware Stepwise Operator Evolution* constructs $\mathcal{H}$ from a small-scale snapshot bank via tail-state feedback, LLM-generated candidates, and one-swap replacement (§3.2); *(b) Hindsight Oracle & Experience-Driven Fine-Tuning* extracts oracle labels h^* on small-scale instances and fine-tunes the selector π_θ via SFT/LoRA with context prompts (§3.3); *(c) Context-Enhanced Online Decision-Making* deploys π_θ on million-variable instances using state s_t and history $\mathcal{C}_t$, with feedback updating the state and action–reward context (§3.4).

respectively. Crucially, the selector is fine-tuned on hindsight oracle trajectories collected from small-scale instances, and is subsequently deployed online for large-scale MILP solving.

3.1 Problem Formulation and Framework Overview

We consider a general Mixed-Integer Linear Programming (MILP) problem defined as:

$$\sigma \quad f(x) = c^\top x \quad \text{s.t.} \quad x \in \mathcal{X}, \tag{1}$$

where $c \in \mathbb{R}^n$ is the objective vector, $\mathcal{X}$ denotes the feasible region, and $\sigma \in \{\min, \max\}$ specifies the optimization sense. Let $x_t \in \mathcal{X}$ denote the incumbent solution at LNS iteration t. A standard LNS step iteratively updates x_t by exploring a local neighborhood. Specifically, it selects a subset of variables to relax, solves the induced sub-MIP while fixing all remaining variables, and accepts the resulting candidate solution if it strictly improves the incumbent objective $f(x_t)$. Consequently, the efficacy of LNS hinges entirely on its **destroy operator** (i.e., the variable selection mechanism).

We formulate a destroy operator h as a scoring function that outputs a vector $r_t^h \in \mathbb{R}^n$. The solver forms the neighborhood N_t^h by selecting the top-scored variables under a dynamically adjusted size budget:

$$N_t^h = \{i \in [n] \mid r_{t,i}^h \text{ is among the selected scores at iteration } t\}. \tag{2}$$

The sub-MIP induced by N_t^h is then solved under a fixed time budget to yield a candidate solution $\tilde{x}_{t+1}^h$. PALL elevates this operator selection into an online sequential decision process. Given an offline-evolved operator bank $\mathcal{H} = \{h_1, \ldots, h_K\}$, a context-enhanced LLM selector π_θ dynamically dispatches the optimal operator at each iteration t:

$$h_t = \pi_\theta(s_t, \mathcal{C}_t), \qquad h_t \in \mathcal{H}, \tag{3}$$

where s_t captures the current optimization state (e.g., objective values, time progress, stagnation signals), and $\mathcal{C}_t$ records historical actions and their normalized rewards. This formulation elegantly decouples two roles: the **operator ensemble** defines *how* to search the solution space, while the **LLM selector** decides *when* to deploy each operator.

To maintain distribution consistency, identical sub-MIP repair rules are strictly enforced across all phases. As outlined below, PALL is driven by three core components: (1) constructing the operator

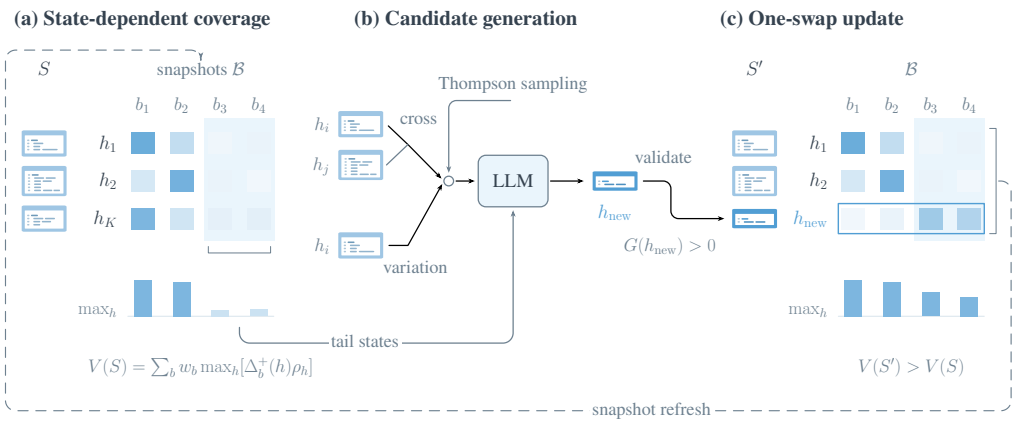

Figure 2: **Tail-Aware Stepwise Operator Evolution Pipeline.** Starting from a seed pool, the framework evaluates candidate operator ensembles on small-scale MILP snapshots. By specifically identifying weak instances and tail snapshots with low ensemble coverage, the LLM generates targeted candidate operators via crossover and mutation prompts. These candidates are then strictly vetted and integrated using a one-swap replacement mechanism, iteratively refining the offline operator bank for maximum complementary coverage. The matrices schematically show $\Delta_b^+(h)\rho_h$ across operators and snapshots; the bars show the maximum over operators, and S' denotes the ensemble after an accepted one-swap update.

ensemble $\mathcal{H}$ via offline evolution (Section 3.2); (2) extracting hindsight oracle trajectories for LLM fine-tuning (Section 3.3); and (3) deploying the context-enhanced LLM for asynchronous sequential decision-making (Section 3.4).

3.2 Tail-Aware Stepwise Operator Evolution

As depicted in Figure 2, the operator bank $\mathcal{H}$ is not a manually fixed list; rather, it is produced by an offline, step-level evolutionary procedure whose objective is to construct a complementary ensemble of neighborhood scoring rules. Each operator h is formulated as executable code with a common interface: it receives the sparse MILP representation, the initial solution, the current solution, and the objective coefficients, and returns a score for every variable. The top-scored variables are re-optimized in sub-MIP repair under their original types and bounds, while all remaining variables are fixed to the incumbent. Alongside the code, each operator carries a short natural-language *thought* that summarizes its intended search regime and scoring signals; this description is later utilized within the LLM selector's prompt.

The key departure from evolving a single globally optimal operator is that our inner objective evaluates an *ensemble* by its stepwise coverage of LNS states. We first build a snapshot bank $\mathcal{B}$ on small-scale training instances. Starting from an incumbent $x_{i,t}$ on instance i and iteration t, we evaluate every member of the current operator ensemble from the same incumbent, run the induced sub-MIP repair, and record the resulting signed improvement $\Delta_b(h)$ and wall-clock cost $\tau_b(h)$ for snapshot $b = (i, t)$. The snapshot trajectory is then advanced using the best improving candidate so that later snapshots correspond to realistic solver states rather than independent random perturbations. To avoid overfitting to the problem family that contributes the most snapshots, we apply class-balanced weights w_b.

Given a candidate ensemble S, its stepwise ensemble value is defined as:

$$V(S) = \sum_{b \in \mathcal{B}} w_b \max_{h \in S} \left[\Delta_b^+(h)\, \rho_h\right], \qquad \Delta_b^+(h) = \max\{0, \Delta_b(h)\}, \tag{4}$$

where $\rho_h = 1/\overline{\tau}(h)$ is the inverse average evaluation time of operator h on the snapshot bank. This objective rewards an operator only when it acts as the best available member for a specific solver state. A newly generated operator that merely duplicates an existing behavior obtains little marginal gain, while an operator that wins previously weak snapshots (i.e., tail states) can be incorporated even if it is not globally dominant as a standalone policy. The evolution loop alternates between

LLM-driven operator generation and ensemble-level replacement. We initialize from a small seed ensemble of valid operators and greedily re-select up to K members under $V(S)$. At each evolution step, the current ensemble is analyzed on the snapshot bank to identify weak training instances and tail snapshots where the ensemble's rate-weighted best improvement is critically low. The LLM then receives one of two generation prompts. A *cross* prompt combines multiple parent operators and asks for a novel standalone scoring rule, while a *variation* prompt mutates a single parent toward a different structural emphasis. A Thompson-sampling controller balances these two generation strategies according to previous replacement successes. The generated code is compiled, checked for the required function signature, and rejected if it contains obviously unscalable patterns (e.g., quadratic-size dense arrays). Valid candidates are evaluated on the snapshot bank and accepted only if a one-swap replacement strictly improves the overall ensemble value. For a generated candidate h_{new}, the replacement gain is:

$$G(h_{\text{new}}) = \max_{h \in S} \left[V\big((S \setminus \{h\}) \cup \{h_{\text{new}}\}\big) - V(S) \right]. \tag{5}$$

If $G(h_{\text{new}}) > 0$, the candidate replaces the maximizing member; otherwise, the current ensemble is retained. The snapshot bank is periodically refreshed using the latest ensemble, enabling later generations to target states induced by the evolved policy rather than merely the initial seed behavior.

The resulting operator bank contains diverse scoring rules occupying distinct search roles, such as exploring variables locked near the initial incumbent, repairing tight or violated rows, relieving critical slack through sign-aware flip estimates, avoiding high-degree hubs, and opening structural escape paths. The online LLM selector is therefore not tasked with synthesizing a new operator at test time; rather, it intelligently dispatches from a compact bank whose members have already been optimized for complementary behavior under a shared LNS repair procedure.

3.3 Hindsight Oracle and Experience-Driven LLM Fine-Tuning

The evolved operator bank provides complementary search behaviors across optimization states. We train the selector on small-scale hindsight trajectories to learn when to deploy each operator, conditioned on the current state and recent action–reward history (Figure 3).

Hindsight Teacher Trace Generation. To construct a strong supervision signal, we deploy a computationally expensive hindsight oracle on small-scale MILP instances exclusively for offline dataset generation. At each LNS iteration t, rather than committing to a single action, the oracle exhaustively evaluates every candidate operator $h \in \mathcal{H}$ from the identical incumbent solution x_t. For an operator h, let $\Delta_t(h)$ denote the signed objective improvement obtained by solving the corresponding sub-MIP:

$$\Delta_t(h) = \begin{cases} f(x_t) - f(\tilde{x}^h_{t+1}), & \sigma = \min, \\ f(\tilde{x}^h_{t+1}) - f(x_t), & \sigma = \max. \end{cases} \tag{6}$$

Let $\tau_t(h)$ denote the actual wall-clock time consumed by operator h, including both the scoring computation and the sub-MIP repair time. To explicitly optimize for efficiency, the oracle label $h^\star_t$ is defined as the operator yielding the maximum positive improvement per unit of charged time:

$$h^\star_t = \arg\max_{h \in \mathcal{H}} \frac{\Delta_t(h)}{\max(\tau_t(h), \epsilon)}. \tag{7}$$

If no operator improves the incumbent, the trajectory still advances using a fallback action, but such steps are strictly excluded from the positive training corpus to ensure high-quality supervision.

Context Prompt Construction. Each extracted high-reward transition is subsequently formatted into a supervised training pair. The input is a compact textual prompt encapsulating the optimization state s_t (e.g., problem family, progress, objective statistics, stagnation signals) alongside a sequential history context $\mathcal{C}_t$, and concise natural-language descriptions of the available operator bank. The target output is the exact name of the oracle-selected operator $h^\star_t$.

Supervised Fine-Tuning (SFT) for Scalable Transfer. Finally, we adopt the **Qwen3-8B** model as our foundational architecture and fine-tune it using these hindsight selection trajectories. This strategy leverages a powerful "easy-to-hard" transfer paradigm: extracting exhaustive oracle labels on large-scale MILPs is computationally prohibitive, whereas small-scale instances yield an abundance of diverse process states at minimal cost. By fine-tuning the LLM on these small-scale experiences, we intrinsically embed the capability for optimal operator selection into the model's weights, preparing it for zero-shot generalization to large-scale instances.

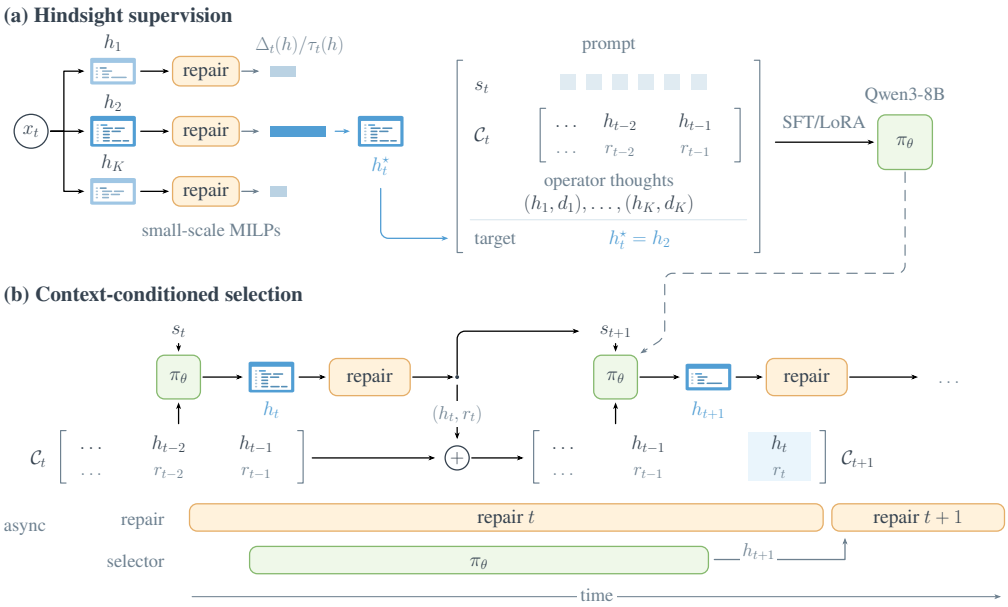

Figure 3: **Pipeline of the Context-Enhanced LLM Selector.** The diagram bifurcates into two complementary phases. **(a) Hindsight supervision:** High-quality teacher traces are generated via hindsight oracle evaluation of all operators in the evolved bank. These are formatted into context-rich prompts, where d_i is the short description of operator h_i, to perform supervised fine-tuning of an 8B LLM using LoRA adapters, resulting in the operational selector π_θ. **(b) Context-conditioned selection:** Embedded within the LNS loop, the operational selector dispatches the optimal operator h_t based on current state s_t and historical context $\mathcal{C}_t$. A feedback loop continuously updates the context $\mathcal{C}_{t+1}$ with observed normalized rewards r_t. The timing strip shows selector inference overlapped with repair, using the context available when the query is launched.

3.4 Context-Enhanced Reasoning for Sequential Decision-Making

During the online deployment phase on large-scale MILP instances, the fine-tuned LLM selector acts as an intelligent sequential decision-maker embedded directly within the LNS loop. As depicted in the bottom half of Figure 3, this phase relies on a continuous context feedback mechanism and an asynchronous execution strategy to ensure both adaptability and computational efficiency.

Context-Conditioned Action Formulation. Rather than treating LNS iterations as isolated steps, we formulate the operator dispatch as a context-aware sequential action $\pi_\theta(s_t, \mathcal{C}_t) \to h_t$, where $h_t \in \mathcal{H}$ is the selected operator for iteration t. The historical context $\mathcal{C}_t$ is profoundly essential to this formulation. The intrinsic value of an operator shifts drastically across different optimization phases: an operator designed to make aggressive, large-scale structural changes may be highly valuable during early exploration but strictly harmful once the search enters a delicate late-stage refinement phase. By explicitly exposing the model to recent action-reward traces, the selector conditions its decisions on the dynamically observed effectiveness of the operator bank, freeing it from a rigid reliance on static problem features.

Dynamic Execution and Context Feedback. Once the LLM selector predicts the one-step action h_t, the chosen operator scores the variables to define the current neighborhood. The solver then executes the sub-MIP repair and evaluates the resulting objective improvement. Crucially, this empirical gain is normalized into a reward signal r_t and appended to the history trajectory to form the updated context $\mathcal{C}_{t+1} = \mathcal{C}_t \cup \{(h_t, r_t)\}$. This continuous feedback loop empowers the LLM to meticulously track optimization progress and pivot its selection strategy in real-time when stagnation is detected.

Asynchronous Inference for State-of-the-Art Scalability. A fundamental challenge in deploying LLMs for combinatorial optimization is inference latency, which can severely bottleneck the underlying solver. To circumvent this, PALL employs a highly efficient asynchronous deployment

mechanism. While the LNS solver is actively executing the computationally intensive sub-MIP repair for the current iteration, the LLM selector concurrently prefetches the state information to compute the next operator decision. This elegant overlapping of LLM inference with the optimization workload effectively masks the model's computational overhead. Consequently, PALL successfully replaces the static neighborhood rules with process-aware intelligence while preventing the LLM from becoming the dominant runtime component.

In summary, the PALL framework seamlessly synergizes three distinct roles: the offline-evolved operator bank provides a complementary repository of search behaviors; the hindsight oracle distills small-scale search traces into high-quality sequential supervision; and the context-enhanced LLM selector deploys this intelligence as a highly scalable, dynamic policy for large-scale MILP solving.

4 Experiments

The primary objective of our experimental evaluation is to empirically validate whether the **PALL** framework—specifically its process-aware selection mechanism—substantially improves large-scale MILP search given a fixed offline-evolved operator bank and repair budget. We design comprehensive evaluations to answer the following core research questions: (1) Does the oracle-supervised LLM selector dynamically dispatch better LNS actions compared to both classical heuristic policies and modern non-LLM adaptive selection baselines? (2) Can the selector successfully generalize from easy-scale training instances to massive, million-variable MILP deployments? (3) Does the asynchronous deployment strategy effectively mask inference latency to deliver state-of-the-art acceleration? We report the final optimization objective values under a strict 600-second wall-clock budget across diverse benchmark families, systematically benchmarking PALL against commercial solvers, classical LNS variants, and recent state-of-the-art adaptive and LLM-driven LNS frameworks.

4.1 Experimental Setup

Benchmarks. We evaluate our framework on four highly challenging, large-scale MILP families: Set Cover (SC), Minimum Vertex Cover (MVC), Maximum Independent Set (MIS), and Multidimensional Knapsack (MIKS). SC and MVC are minimization problems, whereas MIS and MIKS are maximization problems. The hard instances reach the million-variable scale, with MIKS containing approximately $750{,}000$ binary variables and $250{,}000$ continuous variables. Detailed benchmark statistics, including exact dimensions and instance generation protocols, are deferred to Appendix B.

Baselines. We compare **PALL** with six selector baselines using the same offline-evolved bank of 10 operators: (1) a reward-updating *Thompson selector* that adjusts operator probabilities based on online multi-armed bandit feedback; (2) a *BALANS-Softmax selector* that applies the Softmax bandit mechanism of BALANS [Cai et al., 2025] with the same bank and repair routine; (3) a phase-agnostic *Round-robin selector* that naively cycles through the available operators; (4) a zero-shot *API-only LLM selector* that assesses the raw reasoning capability of a frozen foundation model without our supervised fine-tuning; (5) a pretrained *Qwen3-8B selector* using the same backbone, operator bank, prompt, and repair settings as PALL, without task-specific fine-tuning; and (6) a supervised *GNN selector* that directly replaces the LLM architecture with a Graph Neural Network trained on the identical oracle trajectories. External baselines include exact solvers (Gurobi [Gurobi Optimization, LLC, 2023], SCIP [Maher et al., 2016]), classical LNS heuristics (RINS [Danna et al., 2005], ACP [Ye et al., 2023a], Random LNS [Song et al., 2020]), scalable learning-based MILP frameworks (GNN&GBDT [Ye et al., 2023b], Light-MILPopt [Ye et al., 2024]), and state-of-the-art LLM-driven LNS frameworks (LLM-LNS [Ye et al., 2025], EoH-S [Liu et al., 2026]). For GNN&GBDT and Light-MILPopt, we use the official implementations and pretrained checkpoints under the same hardware and 600-second budget.

Evaluation Metrics. The primary evaluation metric is the final optimization objective value achieved within the strict 600-second computational budget, averaged across the five instances per problem family to ensure statistical robustness. To facilitate standardized comparisons across diverse optimization senses, we define the relative advantage $\mathrm{Adv}(A, B)$ of a target method A over a

Table 1: Objective values on million-variable hard instances within 600 s. Lower is better for SC/MVC (↓); higher is better for MIS/MIKS (↑). Best values are bold. Entries marked with $*$ have no feasible solution within the time limit.

Method	SC ↓	MVC ↓	MIS ↑	MIKS ↑
Thompson selector	195,237	277,509	211,949	177,807
BALANS-Softmax	191,999	288,486	222,704	170,992
Round-robin selector	181,372	275,410	224,140	178,078
API-only selector	178,234	274,314	225,617	178,479
Qwen3-8B	198,586	275,244	223,709	**180,229**
GNN selector	187,455	275,538	224,648	178,258
Gurobi	320,026	330,924	168,984	129,336
SCIP	919,478	490,820	9,088	27,695
RINS	261,183	306,005	193,860	129,974
ACP	199,222	423,395	128,730	106,482
Random LNS	183,982	287,159	218,920	173,443
GNN&GBDT	$*$	$*$	$*$	$*$
Light-MILPopt	305,237	296,363	205,253	$*$
LLM-LNS	742,898	329,774	150,857	74,887
EoH-S	196,391	291,038	163,818	147,159
PALL (ours)	**169,842**	**274,100**	**225,728**	178,949

Table 2: Static single-operator results for the shared bank. Best static values are **blue**.

Operator	SC ↓	MVC ↓	MIS ↑	MIKS ↑
H_1	**171,000**	288,170	211,793	149,608
H_2	207,230	277,783	182,079	146,123
H_3	187,498	314,559	182,923	151,980
H_4	226,772	294,828	182,999	152,140
H_5	437,033	304,100	195,022	154,901
H_6	200,673	314,020	**225,284**	177,260
H_7	202,094	282,117	181,175	149,727
H_8	553,253	326,180	223,776	**178,217**
H_9	206,499	278,626	177,738	141,041
H_{10}	417,153	**275,779**	183,277	152,599
PALL (ours)	**169,842**	**274,100**	**225,728**	**178,949**

Table 3: Time to PALL's 600 s objective (s / speedup).

Problem	LLM-LNS	EoH-S
SC	$4200/7.00\times$	$2000/3.33\times$
MVC	$7900/13.17\times$	$1800/3.00\times$
MIS	$2900/4.83\times$	$6900/11.50\times$
MIKS	$5600/9.33\times$	$4900/8.17\times$
Overall	$\mathbf{8.58}\times$	$\mathbf{6.50}\times$

baseline method B such that a positive value strictly indicates the superiority of method A:

$$\mathrm{Adv}(A,B) = \begin{cases} (B-A)/|B|, & \text{for minimization problems,} \\ (A-B)/|B|, & \text{for maximization problems.} \end{cases} \tag{8}$$

4.2 Main Results

Table 1 summarizes the final objective values on million-variable hard instances. **PALL** outperforms all external baselines across the four problem families. Compared with Light-MILPopt, PALL improves the objective by 44.4% on SC, 7.5% on MVC, and 10.0% on MIS under the same budget.

Compared to internal ablations using the identical 10-operator bank, PALL yields a mean relative advantage of 5.35% over the reward-updating Thompson selector and 2.01% over the phase-agnostic round-robin policy, with the largest margin concentrated on Set Cover (13.01% and 6.36%, respectively), validating the necessity of context-aware LLM reasoning. PALL also outperforms the BALANS-Softmax selector across all four families. Against the pretrained Qwen3-8B selector, PALL achieves a mean relative advantage of 3.77% across the four family means, with the largest gain on SC (14.47%). Crucially, these results confirm our core hypothesis: an LLM fine-tuned on easy-scale oracle trajectories successfully transfers state-dependent scheduling intelligence to massive instances, robustly outperforming static or strictly online-adaptive rules.

To rigorously evaluate computational efficiency, Table 3 presents a time-to-target analysis against recent LLM-driven baselines (LLM-LNS and EoH-S). We measure the wall-clock time required for these baselines (given a 10,000-second extended horizon) to match PALL's 600-second objective value. PALL achieves a staggering overall speedup of $\mathbf{8.58}\times$ against LLM-LNS and $\mathbf{6.50}\times$ against EoH-S. Asynchronous prefetching hides 80.8–93.2% of selector inference time under the tested repair caps (Table 10).

A deeper family-wise analysis reveals that PALL excels most on SC and MIS, where the optimal operator distribution shifts drastically as the search progresses. Even on MVC and MIKS—which inherently favor stable, high-reward actions—PALL maintains a consistent edge over adaptive bandit policies. This demonstrates that the learned selector acts as a truly dynamic policy rather than merely recovering one globally dominant operator: it seamlessly pivots between aggressive exploration and delicate refinement based on the prevailing optimization context.

4.3 Ablations and Diagnostics

Table 2 compares PALL against every static operator within the identical 10-operator bank. This critical diagnostic isolates the intrinsic value of dynamic selection from the raw quality of the evolved bank: if a single operator dominated across all scenarios, the contextual LLM selector would be functionally redundant. As the results show, the best-performing static operator varies heavily depending on the problem family. Crucially, PALL strictly dominates the best static baseline across all four families (with relative gains of 0.68%, 0.61%, 0.20%, and 0.41% on SC, MVC, MIS, and MIKS, respectively). This definitively confirms that intelligently dispatching operators based on process context yields superior optimization trajectories compared to committing to any monolithic strategy.

Action–reward history complements the current optimization state: retaining this feedback improves the objective across all four families in the context ablation, with the largest improvement on SC ($169{,}880$ versus $416{,}137$ without history; Appendix G). These results support conditioning operator selection on recent search outcomes as the optimization progresses. Additional ablations on state-only prompting and operator-bank capacity are reported in Appendix G.

Summary. Overall, our experiments firmly establish that PALL's context-enhanced LLM selector dynamically orchestrates offline-evolved operators to yield vastly superior optimization trajectories. By significantly outperforming classical solvers and state-of-the-art adaptive baselines, while effectively masking LLM inference latency through asynchronous execution, PALL validates a highly scalable and effective paradigm for integrating LLMs into large-scale combinatorial optimization.

5 Conclusion

In this paper, we propose **PALL**, a context-enhanced Large Neighborhood Search framework for solving large-scale MILP problems, which dynamically orchestrates an offline-evolved operator bank using an oracle-supervised LLM selector. Extensive experiments on million-variable instances demonstrate that PALL consistently outperforms classical solvers, static heuristic rules, and state-of-the-art adaptive LNS frameworks in both objective quality and computational efficiency. Future work will integrate online reinforcement learning for the continuous adaptation of the LLM selector.

Acknowledgments and Disclosure of Funding

This work received no external funding. The authors declare no competing interests.

References

Hooman Askari-Nasab, Yashar Pourrahimian, Eugene Ben-Awuah, and Samira Kalantari. Mixed integer linear programming formulations for open pit production scheduling. *Journal of Mining Science*, 47(3):338–359, 2011.

Alan W Beggs. On the convergence of reinforcement learning. *Journal of economic theory*, 122(1): 1–36, 2005.

Timo Berthold, Andrea Lodi, and Domenico Salvagnin. Ten years of feasibility pump, and counting. *EURO Journal on Computational Optimization*, 7(1):1–14, 2019.

Junyang Cai, Serdar Kadıoğlu, and Bistra Dilkina. BALANS: Multi-armed bandits-based adaptive large neighborhood search for mixed-integer programming problems. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 2566–2574. International Joint Conferences on Artificial Intelligence Organization, 2025. doi: 10.24963/ijcai.2025/286. URL `https://www.ijcai.org/proceedings/2025/286`.

Teodor Gabriel Crainic, Bertrand Le Cun, and Catherine Roucairol. Parallel branch-and-bound algorithms. *Parallel combinatorial optimization*, pages 1–28, 2006.

Emilie Danna, Edward Rothberg, and Claude Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, 2005. doi: 10.1007/S10107-004-0518-7. URL `https://doi.org/10.1007/s10107-004-0518-7`.

Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical Programming*, 98(1–3):23–47, 2003. doi: 10.1007/S10107-003-0395-5. URL https://doi.org/10.1007/s10107-003-0395-5.

Matteo Fischetti, Andrea Lodi, et al. Heuristics in mixed integer programming. *Wiley Encyclopedia of Operations Research and Management Science. John Wiley & Sons, Inc*, pages 2–23, 2010.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

Gregor Hendel. Adaptive large neighborhood search for mixed integer programming. *Mathematical Programming Computation*, 14(2):185–221, 2022. doi: 10.1007/S12532-021-00209-7. URL https://doi.org/10.1007/s12532-021-00209-7.

Taoan Huang, Aaron M Ferber, Yuandong Tian, Bistra Dilkina, and Benoit Steiner. Searching large neighborhoods for integer linear programs with contrastive learning. In *International conference on machine learning*, pages 13869–13890. PMLR, 2023.

Fadi Agha Kassab, Berk Celik, Fabrice Locment, Manuela Sechilariu, Sheroze Liaquat, and Timothy M Hansen. Optimal sizing and energy management of a microgrid: A joint milp approach for minimization of energy cost and carbon emission. *Renewable Energy*, 224:120186, 2024.

Minsu Kim, Jinkyoo Park, et al. Learning collaborative policies to solve np-hard routing problems. *Advances in Neural Information Processing Systems*, 34:10418–10430, 2021.

Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

Edgar Leon-Olivares, Hertwin Minor-Popocatl, Omar Aguilar-Mejía, and Diana Sánchez-Partida. Optimization of the supply chain in the production of ethanol from agricultural biomass using mixed-integer linear programming (milp): a case study. *Mathematical Problems in Engineering*, 2020(1):6029507, 2020.

Defeng Liu, Matteo Fischetti, and Andrea Lodi. Learning to search in local branching. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3796–3803, 2022. doi: 10.1609/aaai.v36i4.20294. URL https://ojs.aaai.org/index.php/AAAI/article/view/20294.

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

Fei Liu, Yilu Liu, Qingfu Zhang, Tong Xialiang, and Mingxuan Yuan. Eoh-s: Evolution of heuristic set using llms for automated heuristic design. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 37090–37098, 2026.

Abilio Lucena and John E Beasley. Branch and cut algorithms. *Advances in linear and integer programming*, 4:187–221, 1996.

Stephen Maher, Matthias Miltenberger, João Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. pages 301–307, 2016.

Vinod Nair, Mohammad Alizadeh, et al. Neural large neighborhood search. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pages 99–127. Springer, 2018.

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, 2023. doi: 10.1038/s41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Jialin Song, Yisong Yue, Bistra Dilkina, et al. A general large neighborhood search framework for solving integer linear programs. *Advances in Neural Information Processing Systems*, 33: 20012–20023, 2020.

Nicolas Sonnerat, Pengming Wang, Ira Ktena, Sergey Bartunov, and Vinod Nair. Learning a large neighborhood search algorithm for mixed integer programs. *arXiv preprint arXiv:2107.10201*, 2021.

Yaoxin Wu, Wen Song, Zhiguang Cao, and Jie Zhang. Learning large neighborhood search policy for integer programming. *Advances in Neural Information Processing Systems*, 34:30075–30087, 2021.

Hua Xu, Xiaodong Li, Yuan Sun, and Huigen Ye. Intelligent evolution optimization: Guided from deep learning to large language model. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1724–1748, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Huigen Ye, Hongyan Wang, Hua Xu, Chengming Wang, and Yu Jiang. Adaptive constraint partition based optimization framework for large-scale integer linear programming (student abstract). In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 16376–16377, 2023a.

Huigen Ye, Hua Xu, Hongyan Wang, Chengming Wang, and Yu Jiang. GNN&GBDT-guided fast optimizing framework for large-scale integer programming. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 39864–39878. PMLR, 2023b. URL https://proceedings.mlr.press/v202/ye23e.html.

Huigen Ye, Hua Xu, and Hongyan Wang. Light-MILPopt: Solving large-scale mixed integer linear programs with lightweight optimizer and small-scale training dataset. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=2oWRumm67L.

Huigen Ye, Hua Xu, An Yan, and Yaoyang Cheng. Large language model-driven large neighborhood search for large-scale MILP problems. In *International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR, 2025. URL https://proceedings.mlr.press/v267/ye25j.html.

Huigen Ye, Hua Xu, and Carlos A. Coello Coello. Light-evoopt: A lightweight evolutionary optimization framework for ultralarge-scale mixed integer linear programs. *IEEE Transactions on Evolutionary Computation*, 30(1):333–347, 2026. doi: 10.1109/TEVC.2025.3550668.

Hao Yuan, Wenli Ouyang, Changwen Zhang, Yong Sun, Liming Gong, and Junchi Yan. BTBS-LNS: Binarized-tightening, branch and search on learning LNS policies for MIP. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=siHHqDDzvS.

Jiayi Zhang, Chang Liu, Xijun Li, Hui-Ling Zhen, Mingxuan Yuan, Yawen Li, and Junchi Yan. A survey for solving mixed integer programming via machine learning. *Neurocomputing*, 519: 205–217, 2023.

A Detailed Operator-Bank Evolution Procedure

This appendix gives the implementation-level procedure used to evolve the operator bank in Section 3.2. The goal of the procedure is not to find a single best LNS neighborhood rule, but to construct a compact ensemble of executable scoring operators whose strengths cover different solver states. Each operator takes the sparse MILP representation, the initial solution, the current incumbent, and objective coefficients as input, and returns a real-valued score for each variable. The LNS repair routine then frees the top-scored variables under the same neighborhood-size rule used in all offline and online evaluations.

Snapshot bank. Evolution is driven by a bank of step-level snapshots rather than by only full-run final objectives. A snapshot $b = (i, t, x_{i,t})$ stores the training instance index, the LNS step, the incumbent solution, the current objective value, progress statistics, and the class label of the instance family. From the same snapshot, every operator in the current candidate ensemble can be evaluated by running one identical sub-MIP repair and recording the signed improvement $\Delta_b(h)$ and charged time $\tau_b(h)$. This same-state evaluation makes the marginal value of an operator comparable across the bank.

Ensemble objective. For a candidate ensemble S, the value $V(S)$ in Eq. (4) is computed by taking, for each snapshot, the best rate-weighted positive improvement among members of the ensemble. An operator therefore helps the ensemble only when it wins some snapshots that are not already covered by other members. Class-balanced weights w_b prevent the largest problem family or the longest trajectory from dominating the objective. The time factor ρ_h penalizes operators that obtain occasional improvements only by making the scoring routine itself expensive.

Weak-state feedback. After scoring the current set, we identify weak states by sorting snapshots according to their covered value,

$$\max_{h \in S} \Delta_b^+(h)\rho_h.$$

The lowest-value snapshots and low-performing training instances are summarized into feedback for the LLM. This feedback contains the problem family, coarse progress stage, stagnation indicators, recent objective trend, and which existing operators failed to improve the incumbent. The prompt asks for a standalone scoring rule that targets these uncovered states rather than for a minor textual rewrite of a previous candidate.

Candidate generation and validation. The LLM proposes one executable operator at a time. We use two proposal modes. The *cross* mode combines ideas from multiple parent operators while requiring a new standalone scoring rule. The *variation* mode mutates one parent toward a different structural emphasis, such as locked variables, tight rows, critical slack, sign-aware flip estimates, or diversification away from high-degree hubs. A Thompson-sampling controller chooses between cross and variation according to their recent acceptance rates. Generated code is then compiled and checked for the required function signature, deterministic output shape, finite scores, and obviously unscalable operations such as dense quadratic-size arrays.

One-swap replacement and refresh. A valid candidate h_{new} is evaluated on the same snapshot bank. It is accepted only if replacing one existing member increases the set objective:

$$h_{\text{drop}} = \arg\max_{h \in S} V\big((S \setminus \{h\}) \cup \{h_{\text{new}}\}\big), \qquad G(h_{\text{new}}) = V\big((S \setminus \{h_{\text{drop}}\}) \cup \{h_{\text{new}}\}\big) - V(S).$$

If $G(h_{\text{new}}) > 0$, the candidate replaces h_{drop}; otherwise it is discarded. Periodically, the snapshot bank is rebuilt using the latest accepted ensemble. This refresh step is important because the distribution of later LNS states depends on the operator bank that generated earlier repairs.

Algorithms 1 and 2 separate two sources of adaptivity. The outer evolution loop changes the operator bank only when a candidate improves ensemble-level coverage, while the snapshot refresh loop changes the training states induced by the current bank. The selector described in Section 3.3 is trained only after this bank is fixed; it does not participate in the evolution objective.

Algorithm 1 Tail-Aware Stepwise Operator-Bank Evolution

Require: Training MILP families $\mathcal{D}$, seed operators H_0, bank size K, evolution budget M, refresh period R
Ensure: Evolved operator bank H

$H \leftarrow \textsc{GreedySelect}(H_0, K)$
$\mathcal{B} \leftarrow \textsc{BuildSnapshots}(\mathcal{D}, H)$
Evaluate every $h \in H$ on every snapshot $b \in \mathcal{B}$
for $m = 1, \ldots, M$ **do**
    Compute $V(H)$ using Eq. (4)
    $\mathcal{W} \leftarrow \textsc{FindWeakStates}(\mathcal{B}, H)$
    $o \leftarrow \textsc{SampleOperator}(\textsc{Cross}, \textsc{Variation})$
    $P \leftarrow \textsc{ChooseParents}(H, \mathcal{W}, o)$
    $h_{\text{new}} \leftarrow \textsc{LLMGenerateOperator}(P, \mathcal{W}, o)$
    if $\textsc{CompileAndSafetyCheck}(h_{\text{new}})$ fails **then**
        Update proposal statistics with rejection; **continue**
    end if
    Evaluate h_{new} on every snapshot $b \in \mathcal{B}$
    $h_{\text{drop}} \leftarrow \arg\max_{h \in H} V((H \setminus \{h\}) \cup \{h_{\text{new}}\})$
    $G \leftarrow V((H \setminus \{h_{\text{drop}}\}) \cup \{h_{\text{new}}\}) - V(H)$
    if $G > 0$ **then**
        $H \leftarrow (H \setminus \{h_{\text{drop}}\}) \cup \{h_{\text{new}}\}$
        Update proposal statistics with acceptance
    else
        Discard h_{new} and update proposal statistics with rejection
    end if
    if $m \bmod R = 0$ **then**
        $\mathcal{B} \leftarrow \textsc{BuildSnapshots}(\mathcal{D}, H)$
        Re-evaluate all $h \in H$ on all $b \in \mathcal{B}$
    end if
end for
return H

B Experimental Details and Reproducibility

This appendix records the experimental objects and implementation parameters used by the current draft. All reported results use the same MILP parser, initial-solution routine, sub-MIP repair routine, and operator-function interface.

B.1 MILP Form and Benchmark Families

We solve mixed-integer linear programs in the canonical form

$$\min_x \; c^\top x, \quad \text{s.t.} \quad Ax \le b, \quad l \le x \le u, \quad x_i \in \mathbb{Z},\ i \in \mathcal{I}. \tag{9}$$

Maximization benchmarks are converted to this minimization form by negating the objective. The parser stores the sparse constraint matrix A in CSR form, the objective vector c, right-hand side b, constraint senses, variable bounds, variable types, and the indices of binary/integer variables. Constraint senses are kept in the original LP sense; in the LNS repair step, constraints touched by at least one freed variable are rebuilt exactly, while all other variables are fixed to the incumbent value.

The benchmark instances are drawn from MILPBench,[3] using the SC, MVC, MIS, and MIKS families. Set cover (SC) is a binary minimization problem of the form

$$\min \sum_j c_j x_j \quad \text{s.t.} \quad Ax \ge \mathbf{1}, \quad x_j \in \{0,1\}. \tag{10}$$

Minimum vertex cover (MVC) is a binary minimization problem on graph edges E:

$$\min \sum_{v \in V} x_v \quad \text{s.t.} \quad x_u + x_v \ge 1,\ \forall (u,v) \in E, \quad x_v \in \{0,1\}. \tag{11}$$

[3] `https://github.com/thuiar/MILPBench`

Algorithm 2 Building and Refreshing the Snapshot Bank

Require: Training instances $\mathcal{D}$, current bank H, trajectory length T, snapshot stride q
Ensure: Snapshot bank $\mathcal{B}$

$\mathcal{B} \leftarrow \emptyset$
for each training instance $i \in \mathcal{D}$ **do**
  Construct an initial feasible incumbent $x_{i,0}$
  for $t = 0, \ldots, T-1$ **do**
    if $t \bmod q = 0$ **then**
      Add snapshot $b = (i, t, x_{i,t})$ to $\mathcal{B}$
    end if
    for each operator $h \in H$ **do**
      Score variables with h and solve the induced sub-MIP from $x_{i,t}$
      Record improvement $\Delta_{(i,t)}(h)$ and charged time $\tau_{(i,t)}(h)$
    end for
    $h_t^\star \leftarrow \arg\max_{h \in H} \Delta^+_{(i,t)}(h) / \max(\tau_{(i,t)}(h), \epsilon)$
    Advance the trajectory with the repaired incumbent produced by $h_t^\star$
  end for
end for
Assign class-balanced weights w_b to snapshots
return $\mathcal{B}$

Table 4: Dataset scales used in the current experiments. Hard results in the main tables are reported on instances 0–4.

Family	Split	Instances	Variables	Constraints	Nonzeros
SC	easy	30	40,000	40,000	160,000
SC	hard	30	2,000,000	2,000,000	8,000,000
MVC	easy	30	20,000	60,000	120,000
MVC	hard	30	1,000,000	3,000,000	6,000,000
MIS	easy	30	20,000	60,000	120,000
MIS	hard	30	1,000,000	3,000,000	6,000,000
MIKS	easy	30	50,000	50,000	200,000
MIKS	hard	30	1,000,000	1,000,000	4,000,000

Maximum independent set (MIS) is the complementary binary maximization problem:

$$\max \sum_{v \in V} x_v \quad \text{s.t.} \quad x_u + x_v \leq 1,\ \forall (u,v) \in E, \quad x_v \in \{0,1\}. \tag{12}$$

Multidimensional knapsack (MIKS) is a mixed-integer maximization problem:

$$\max \sum_j p_j x_j \quad \text{s.t.} \quad Ax \leq b, \quad x_j \in \{0,1\}\ (j \in \mathcal{I}), \quad x_j \in [l_j, u_j]\ (j \notin \mathcal{I}). \tag{13}$$

Here, $\mathcal{I}$ indexes the binary variables; the remaining variables are continuous with the bounds specified by the original instance. In the local MILPBench pool, each reported family has 30 small-scale and 30 large-scale LP instances used by the current paper. The current paper trains/evolves from small-scale instances and reports the main large-scale results on instances 0–4. Table 4 gives the exact scales used by the cached LP parser.

B.2 LNS Repair and Solver Settings

All LNS variants start from a feasible incumbent produced by Gurobi after replacing the original objective with a zero objective. Parsed instance data and initial feasible solutions are cached and reused across runs. Each LNS iteration scores all variables with the selected operator, sorts variables in descending score order, frees the top n/p_t variables, fixes all other variables to the incumbent, and solves the induced sub-MIP with Gurobi. Freed variables retain their original types and bounds. The partition parameter is initialized at $p_0 = 10$. If the objective value two steps apart changes by at most $\epsilon |f(x_t)|$ with $\epsilon = 10^{-3}$ and $p_t \geq 3$, the next iteration uses $p_{t+1} = p_t - 1$, thereby increasing the free-variable fraction.

The total hard-evaluation wall-clock budget is 600 seconds per instance. Unless otherwise specified, the per-sub-MIP time limit is

$$\min\{T_{\text{remaining}}, T_{\text{total}}/5\}, \tag{14}$$

which is 120 seconds at the beginning of a 600-second run. Gurobi output is disabled and each sub-MIP model uses one solver thread. The reported same-bank methods use seed 42. For reproducibility under shared hardware, all CPU BLAS thread-count environment variables are set to one.

B.3 Operator Bank and Evolution Configuration

The operator bank used in the main experiments contains $K = 10$ executable scoring functions:

$$\{H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8, H_9, H_{10}\}.$$

The final bank was evolved from 12 easy training instances, three from each of SC, MVC, MIS, and MIKS. The evolution run used a bank size of 10, an LLM candidate-evaluation budget of 200 proposals, a trajectory horizon of 50 LNS steps, a 100-second easy-instance rollout budget, a 3-second sub-MIP budget during snapshot construction/evaluation, initial partition $p = 10$, 12 snapshot-bank workers, and refresh period 20. The final snapshot bank contained 497 snapshots, split into 249 fixed anchor snapshots and 248 live snapshots, with anchor weight 0.5. Candidate operators were required to expose the interface in Appendix B.7, return finite length-n scores, and avoid dense quadratic memory patterns.

B.4 Selector Training and Deployment

The LLM selector is a Qwen3-8B model fine-tuned with LoRA. The active SFT configuration uses LoRA rank 16, LoRA alpha 32, LoRA dropout 0.05, target modules `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`, 4-bit quantization, batch size 1, gradient accumulation 1, three epochs, learning rate 2×10^{-5}, zero warmup, maximum sequence length 20,000, and seed 42.

Training examples are one-step teacher labels in a thought-only compact prompt format. The main easy-oracle source contains 959 positive selection examples built from easy-instance oracle trajectories with selection window 1 and stride 1. The filters used for this source require a positive selected improvement larger than 10^{-9}, partition $p \geq 4$, stagnation count 0, and free-variable fraction at most 0.35. A state-matched easy subset used for live-environment alignment contains 530 examples, selected by matching easy states to hard-reference states with four nearest easy neighbors per hard state and at most one copy per easy example. Both datasets cover the same four reported families and use only the fixed 10-operator bank.

Oracle collection and selector fine-tuning use a single RTX PRO 6000 GPU with 96 GB memory. Collecting oracle trajectories for 40 easy-instance tasks at parallelism 15 takes 89 minutes, and fine-tuning takes 31–32 minutes per run. Operator-bank evolution with 200 candidate proposals, including LLM generation and local evaluation, takes approximately 5–20 hours depending on API latency.

During hard-instance deployment, the selector uses thought-only compact prompts, strict prompt sanitization, no warmup operator override, and one-step direct operator selection. The model outputs a single operator name; no external reranker, guard, or post-hoc correction changes the selected action. In the asynchronous implementation, the next selector query is prefetched while the current sub-MIP repair is running, so LLM inference latency can overlap with solver time. The main hard evaluation uses hard instances 0–4, a 600-second budget, seed 42, and the same LNS repair settings as Appendix B.2. Hard-instance evaluation uses an RTX PRO 6000 GPU with 96 GB memory, 22 CPUs, and 110 GiB of host memory, with one solver thread per task.

B.5 Baseline Settings

The same-bank Thompson selector and round-robin selector use the identical 10-operator bank and LNS repair procedure as the learned selector. Thompson selection updates operator preferences from observed online normalized improvements; round-robin cycles through the bank without conditioning on state. Static single-operator results run each operator alone with the same 600-second budget and hard instances 0–4.

Table 5: External assets used in the current paper and their licenses or terms of use.

Asset	Version/source	License/terms	Use in this work
MILPBench	Public GitHub repository	Apache-2.0 license	Source of the SC, MVC, MIS, and MIKS benchmark instances.
Gurobi Optimizer	12.0.2	Academic research license	Feasible-solution generation, sub-MIP repair, and direct-solver baseline runs.
SCIP	6.1.0	ZIB Academic License	Direct-solver baseline runs.
PySCIPOpt	6.1.0	MIT License	Python interface used for the SCIP baseline.
Qwen3-8B	Qwen/Qwen3-8B	Apache 2.0	Base model for the fine-tuned LLM selector.

Gurobi and SCIP are evaluated as direct solver baselines on the original hard MILP instances with a 600-second wall-clock budget, one solver thread, and solver output disabled. The Gurobi row uses Gurobi 12.0.2 through `gurobipy`. The SCIP row uses PySCIPOpt 6.1.0. Because the default SCIP run can report an invalidly large MIKS incumbent when continuous variables are left unbounded in the parsed model, the reported SCIP MIKS value uses the capped-variable variant of the same one-thread 600-second protocol.

The API-only selector uses the same 10-operator bank, prompt format, and online LNS repair settings as the fine-tuned selector, but queries the base API model directly without supervised fine-tuning on easy-instance oracle trajectories. Its averaged 600-second objectives are SC 178,234, MVC 274,314, MIS 225,617, and MIKS 178,479.

B.6 Existing Assets and Terms of Use

Table 5 summarizes the external data, software, and model assets used in the current experiments. The MILPBench repository is released under the Apache-2.0 license; we cite it as the source of the benchmark instances, use the instances under that license for academic evaluation, and do not redistribute them with the manuscript.

B.7 Operator Function Interface

Every generated operator is a Python function named `select_neighborhood` with the following signature:

```
def select_neighborhood(
    n, m, k, site, value, constraint,
    initial_solution, current_solution, objective_coefficient):
    ...
    return neighbor_score
```

Here n is the number of variables, m is the number of constraints, `k[i]` is the number of nonzeros in row i, `site[i]` stores the variable indices in row i, `value[i]` stores the corresponding coefficients, and `constraint[i]` is the right-hand side. The arrays `initial_solution` and `current_solution` are length-n incumbent vectors, and `objective_coefficient` is the objective vector c. The function must return a finite real-valued vector `neighbor_score` of length n; variables with larger scores are more likely to be freed in the next LNS repair. The operator does not receive the future repair outcome, the scores of other operators, or hard-instance oracle labels.

The interface is intentionally limited to sparse row access and incumbent/objective information so that generated operators can run on million-variable instances. The implementation rejects candidates that fail to compile, return malformed or non-finite scores, or use obviously unscalable dense all-pairs constructions.

C Per-Family Convergence Traces

Figure 4 complements the final-objective values in Table 1 with per-family relative-gap convergence trajectories on a separate set of hard test instances under the same 600-second wall-clock budget.

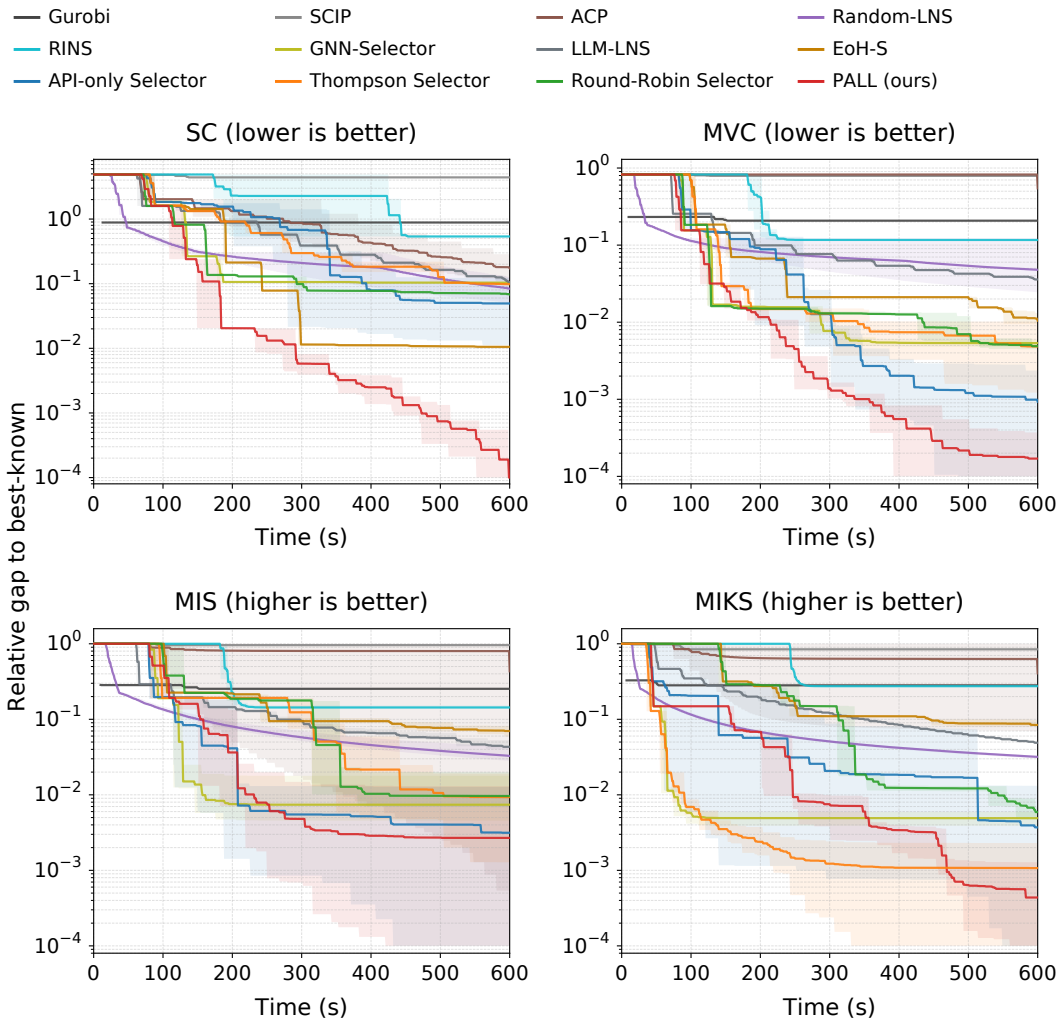

Figure 4: **Per-family relative-gap convergence** on a separate set of hard test instances under a 600-second budget. Each panel shows one benchmark family (SC and MVC are minimization, ↓; MIS and MIKS are maximization, ↑). The y-axis is the relative gap to the best-known incumbent on a logarithmic scale; the x-axis is wall-clock time in seconds. Solid curves are averaged across test instances, and shaded bands indicate the spread.

D Concrete Operator-Bank Evolution Trace

This appendix records a concrete saved evolution run behind the operator bank used in the main experiments. To keep the presentation independent of implementation-specific filenames and generated function names, we anonymize the final bank as $H_1, \ldots, H_{10}$ and generated candidates as C_t, where t denotes the candidate-evaluation index. The natural-language description stored with each generated candidate serves as the thought-style rationale used later to describe the bank to the selector.

Table 7 shows representative accepted replacements from the same run. We omit implementation-specific dropped and added function names because they do not affect the scientific claim; the important information is which mechanism was introduced and how much it improved the set objective. The largest gains occurred early, when broad structural failures in the seed set were

Table 6: Configuration and final metrics of the saved operator-bank evolution run. The ensemble objective is the stepwise snapshot-bank objective in Eq. (4).

Item	Value
Training pool	12 small-scale instances, 3 each from SC/MVC/MIS/MIKS
Proposal budget	200 generated candidates
Bank size and horizon	$K = 10$, 50 LNS steps
Snapshot bank	497 snapshots: 249 anchor and 248 live snapshots
Anchor/live weighting	Anchor weight 0.5 with class-balanced snapshot weights
Accepted one-swap replacements	64
Stepwise ensemble objective	$890.72 \rightarrow 1855.06$ (2.08×)
Diversity	$0.134 \rightarrow 0.266$ (1.99×)
Responsibility balance	$0.468 \rightarrow 0.810$ (1.73×)
Step coverage	1.0 at the final bank

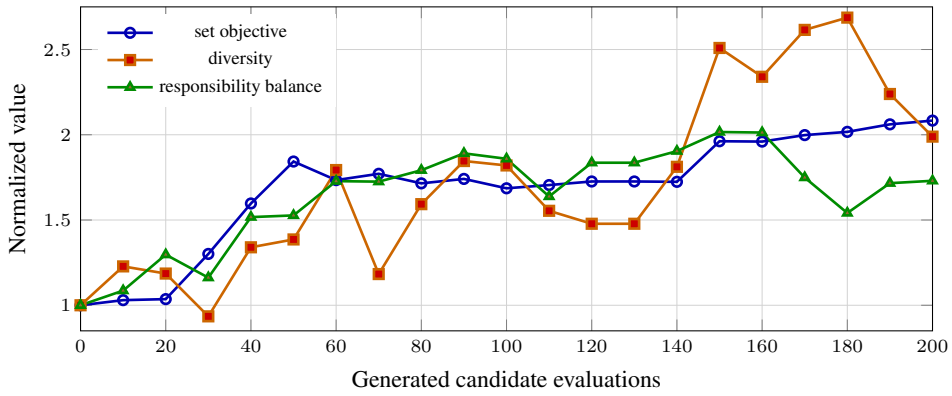

Figure 5: Evolution trace for the saved evolution run. Each curve is normalized by its value at the initial bank. The objective improves substantially while diversity and responsibility balance remain higher than at initialization, which is the desired behavior for a bank intended to support online selection rather than one static operator.

removed. Later accepted swaps have smaller immediate gains but shape the final bank toward complementary late-stage mechanisms.

The evolution prompts are deliberately short because the detailed state summary is built by the evolution driver from the current weak snapshots. Two stored prompt headers from the run illustrate the controller's intent:

```
Cross proposal:
Attack the worst 30% of snapshots; focus on all problem families, mid-stage states, and raw improvement.

Variation proposal:
Attack the worst 10% of snapshots; choose the weakest family automatically, late-stage states, and
     improvement rate.
```

The following excerpt shows how a late accepted candidate is represented internally. The full generated function is longer; the excerpt keeps the critical-set construction and sign-aware flip-impact calculation that motivated the one-swap acceptance of the candidate later anonymized as H_8.

```python
# H_8: brittle-coupling score
critical = (slack <= thr).astype(np.float64)
tight = 1.0 / (1.0 + slack / scale)
crit_w = critical * np.clip(tight, 0.0, 1.0)

edge_crit = crit_w[cons_idx]
pos_part = np.maximum(coef, 0.0)      # hurts if x_v increases
neg_part = np.maximum(-coef, 0.0)     # helps if x_v increases
can_edge = can_free[var_idx]

good_edge = edge_crit * np.tanh(0.9 * neg_part) * can_edge
bad_edge = edge_crit * np.tanh(0.9 * pos_part) * can_edge
```

Table 7: Representative accepted one-swap replacements from the saved evolution run. "Eval" is the candidate index in the generated-candidate stream.

Eval	Candidate	Gain	Candidate mechanism excerpt
33	C_{33}	768.94	Criticality-imbalance scoring contrasts violated and tight constraints through slack-sensitive structural signals.
142	C_{142}	185.38	Tight-core membership and variable coupling to the core, with anti-hub pressure for complementary coverage.
29	C_{29}	142.48	Directional criticality combines slack tightness, sign-aware contribution, and local degree structure.
45	C_{45}	98.36	Directional breakage blends slack-weighted coupling with a signed trajectory term.
22	C_{22}	93.66	Mixed constraint-coupling, violation-nearby, objective, and trajectory scoring.
169	C_{169}	30.27	Minimal-impact scoring estimates local slack improvement on critical constraints and applies anti-hub diversity.
182	C_{182}	38.44	Brittle-coupling scoring estimates sign-aware flip impact on the most critical constraints.
191	C_{191}	4.16	Structural-escape scoring mixes violated/tight coupling, degree risk, trajectory instability, and objective pressure.

Table 8: Final evolved operator bank used by the selector.

ID	Operator	Source	Thought-style role
1	H_1	seed	Free variables still locked near the initial incumbent; use objective magnitude as a tie-breaker.
2	H_2	eval 57	For repair phases dominated by tight or violated rows, focus on variables incident to those rows with a small trajectory-unstable boost.
3	H_3	eval 66	Estimate one-step slack relief on active constraints and damp high-degree hubs to avoid overly broad neighborhoods.
4	H_4	eval 148	Blend slack-rescue and violated-row propagation pressure while shifting with the current trajectory state.
5	H_5	eval 166	Pivot on conflict-critical variables that can move with limited damage to active tight rows.
6	H_6	eval 169	Prefer low-impact slack-improving variables with anti-hub weighting for conservative diversification.
7	H_7	eval 179	Combine tight/violated interaction pressure with coverage-gap pressure and a trajectory-aware stability gate.
8	H_8	eval 182	Score brittle critical-constraint phases by sign-aware flip impact, trajectory-freeing pressure, and anti-hub terms.
9	H_9	eval 184	Use signed residual pressure and participation diversity to find variables coupled to near-tight constraints.
10	H_{10}	eval 191	Combine degree risk, trajectory instability, and objective pressure to open a structural escape path.

```

sum_good = np.zeros(n, dtype=np.float64)
sum_bad = np.zeros(n, dtype=np.float64)
np.add.at(sum_good, var_idx, good_edge)
np.add.at(sum_bad, var_idx, bad_edge)
net = (sum_good - sum_bad) / (cnt + 1e-12)
core = 0.5 * (np.tanh(1.3 * net) + 1.0)
```

E Final Evolved Operator Bank

Table 8 lists the final 10 operators. The "thought" text is a compact version of the stored description used in selector training prompts. This bank intentionally contains overlapping sparse-MILP ingredients, but each member emphasizes a different state-action mechanism.

Two short code excerpts illustrate the range of this bank. The seed operator is intentionally simple and exploratory:

```
def select_neighborhood(n, m, k, site, value, constraint,
                        initial_solution, current_solution,
                        objective_coefficient):
    import numpy as np
    not_changed = 1.0 - np.clip(np.abs(current_solution - initial_solution), 0, 1)
    obj_w = np.abs(objective_coefficient) / (np.max(np.abs(objective_coefficient)) + 1e-10)
    return not_changed * 1.5 + obj_w + 0.1 * np.random.rand(n)
```

By contrast, late evolved operators construct sparse row-variable edge lists and score variables through critical constraints. The following excerpt from H_{10} shows the structural-escape signal:

```
degree = np.zeros(n, dtype=np.float64)
np.add.at(degree, var_idx, 1.0)
lhs = np.zeros(m, dtype=np.float64)
np.add.at(lhs, cons_idx, coef * current_solution[var_idx])
slack = constraint.astype(np.float64) - lhs

w_tight = 1.0 / (1.0 + np.maximum(slack, 0.0) / scale)
w_viol = 1.0 + np.maximum(-slack, 0.0) / scale
tail_boost = np.ones(m, dtype=np.float64)
tail_boost[slack / scale <= 0.0] = 2.2
tail_boost[slack / scale <= -1.0] = 3.0
cons_weight = w_tight * w_viol * tail_boost

weighted_sum = np.zeros(n, dtype=np.float64)
np.add.at(weighted_sum, var_idx, np.abs(coef) * cons_weight[cons_idx])
```

F Selector Prompt and Deployment Examples

The selector is trained from hindsight oracle trajectories after the operator bank is fixed. Each SFT example contains a system prompt with the bank-level thought descriptions, a user prompt with the current online LNS state and recent action-reward context, and an assistant answer containing exactly one next operator name. The excerpt below is from a stored MVC small-scale training example.

```
System:
You are an expert MILP optimizer controlling an LNS solver; choose a short operator schedule from online
     search features and the operator Thoughts.

Available operators (excerpt from 10 total):
- H_1
  Thought: For exploration phases where many variables remain locked near the initial incumbent, free
     unchanged variables first and use objective magnitude as a tie-breaker.
- H_6
  Thought: For conservative diversification phases where neighborhoods should improve critical slack
     without collapsing onto hubs, prefer low-impact slack-improving variables with anti-hub weighting.
- H_7
  Thought: For coverage-gap phases where tight or violated interactions reveal missing repair structure,
     combine tight/violated interaction pressure with coverage-gap pressure and a trajectory-aware
     stability gate.
- H_8
  Thought: For brittle critical-constraint phases where flip direction matters, score variables by sign-
     aware flip impact, trajectory-freeing pressure, and anti-hub terms.
- H_10
  Thought: For structural-escape phases around violated or tight constraints, combine degree risk,
     trajectory instability, and objective pressure to prioritize variables likely to open a new repair
     path.

Use names exactly as written. Do not answer with a problem-family default cycle.

User:
Problem: MVC | sense=min | stage=develop | iter=11 | time=3.3%
Progress: total=43.91% recent=0.19% stagnation=0 parts=8 free=12.5%
Current solution structure: changed=34.0%, active=66.0%, violated=45.7%, tight=100.0%, near_tight=100.0%,
      critical_rows=45.7%, safe_flip=59.7%
Recent decisions:
  t-1: H_10 | gain=+0.04% | best-update
  t-2: H_10 | gain=+0.02% | best-update
  t-3: H_4 | gain=+0.01% | best-update
Tried operator leaders: H_2(avg=+8.71%, uses=5.0), H_6(avg=+0.24%, uses=1.0), H_10(avg=+0.03%, uses=4.0)

Assistant:
Step 1: H_7
```

Table 9 shows online deployment traces from the saved hard-instance policy logs. These traces are not used as training labels for the reported hard evaluation; they illustrate how the fine-tuned selector changes mechanisms as the incumbent evolves.

Finally, asynchronous prefetching overlaps the next selector query with the current sub-MIP repair. Hidden inference time is the portion overlapped with repair; exposed wait is the remaining time for which the solver waits. Table 10 reports the hidden fraction on hard instances under three repair

Table 9: Selected online selector decisions from hard instance 0 for each benchmark family. For minimization problems, positive improvement means objective decrease; for maximization problems, positive improvement means objective increase.

Family	Sense	Iter.	Selector choice	Hindsight best	Improvement	Current objective
SC	min	0	H_{10}	H_{10}	171,707.37	828,444.72
SC	min	2	H_1	H_1	180,420.47	644,866.57
SC	min	5	H_6	H_6	42,879.99	388,921.47
MVC	min	0	H_{10}	H_{10}	78,840.97	421,198.70
MVC	min	5	H_6	H_6	131.46	356,055.73
MIS	max	0	H_8	H_8	71,026.46	71,026.46
MIS	max	2	H_6	H_6	67,592.88	145,908.94
MIKS	max	0	H_8	H_8	68,502.00	68,502.00
MIKS	max	2	H_6	H_6	38,471.37	166,952.29

Table 10: Selector latency hidden by asynchronous prefetching on hard instances under different repair caps.

Repair cap (s)	Hidden inference time (%)
3	80.83
12	85.95
30	93.19

caps. Aggregate exposed wait decreases from 58.35 s with synchronous execution to 11.35 s with asynchronous execution, an 80.54% reduction.

G Context-Enhanced Ablation Studies

To deconstruct the architectural designs of PALL's selector, we report a unified ablation suite covering (i) the removal of the historical context $\mathcal{C}_t$ of recent action–reward pairs, (ii) a state-only prompt that further strips the operator-bank descriptions, and (iii) shrinking the offline-evolved operator bank from $K{=}10$ to $K{\in}\{4, 6, 8\}$. For each variant we re-fine-tune the LLM selector with the same LoRA recipe described in Appendix B.4 on the matching prompt or smaller bank, and re-deploy under the identical asynchronous LNS pipeline used for the main results. The target instance is the first million-variable hard instance per family under a 600-second budget at free-fraction 0.35 (single instance, matching index 0 of the main grid in Table 4).

Table 11 shows that the context feedback is the single most decisive design choice: removing the action–reward history collapses Set Cover by more than $2.4\times$ relative to the full PALL selector (SC: $416{,}137$ vs. $169{,}880$). Compared with the no-history variant, state-only prompting partially restores SC performance but degrades the remaining three families. On SC, increasing the operator-bank size steadily improves objective quality. $K{=}4$ already captures the dominant scheduling structure on MVC/MIS/MIKS but loses noticeable ground on SC, while $K{=}8$ closely tracks the full $K{=}10$ selector. PALL with the full bank is best on three of four families and is within 0.12% of the $K{=}6$ variant on MVC, indicating that both the context feedback and the diverse operator coverage are necessary to obtain the final headline performance.

Selector backbone. We fine-tune Qwen3-8B, Qwen3.5-9B, and Mistral-7B using the same training data, LoRA configuration, and evaluation protocol. Table 12 shows comparable objective values across the four problem families. All three selectors outperform the Thompson and round-robin baselines in Table 1.

Training stability. We independently train three LoRA selectors with training seeds 0, 1, and 42 and evaluate each on five hard instances from each of the four problem families under a 600-second budget. Using the same operator bank, prompt, repair protocol, and deployment seed 42, the three selectors achieve mean relative advantages of 3.15%, 3.89%, and 3.77%, respectively, over the pretrained Qwen3-8B control.

Table 11: Context-enhanced reasoning and operator-bank capacity ablations on a single million-variable hard instance (index 0) under a 600-second budget. SC and MVC are minimization (↓); MIS and MIKS are maximization (↑). The best entry per column is bold. "No history context" removes the recent action–reward trace $\mathcal{C}_t$ from the selector prompt; "State-only prompt" additionally strips the bank descriptions, leaving only the optimization state s_t; "Operator bank $K{=}\cdot$" re-fine-tunes the selector on a complementary subset of the evolved bank.

Variant	SC ↓	MVC ↓	MIS ↑	MIKS ↑
Context history				
No history context	416,137	275,784	224,377	178,170
State-only prompt	182,988	287,818	212,270	173,728
Operator bank capacity				
Operator bank $K{=}4$	191,167	274,295	225,811	178,650
Operator bank $K{=}6$	181,258	**273,776**	225,816	176,927
Operator bank $K{=}8$	170,762	277,514	225,816	178,624
PALL (full, $K{=}10$)	**169,880**	274,101	**225,843**	**178,904**

Table 12: Selector backbone comparison under the 600-second budget. Best values are bold.

Backbone	SC ↓	MVC ↓	MIS ↑	MIKS ↑
Qwen3-8B	**169,842**	274,100	**225,728**	178,949
Qwen3.5-9B	170,539	273,754	223,444	179,172
Mistral-7B	171,868	**273,299**	224,112	**179,184**

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the operator-bank plus LLM-selector framework, the hard-instance scope, and the main performance claims. These claims are consistent with the experiments and appendix details in Sections 4 and B.

   Guidelines:

   - The abstract and introduction should match the scope of the experiments and avoid overstating generality.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 3 describes the offline-evolved operator bank and offline-trained selector, and Section 4 specifies the evaluation on four benchmark families under fixed time budgets. Section 5 identifies online reinforcement learning for continued selector adaptation as future work.

   Guidelines:

   - The paper should state the main assumptions, missing ablations, and scope limits of the empirical evidence.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not present formal theoretical results, so there are no theorem statements or proofs to check.

   Guidelines:

   - Use [N/A] when the paper has no formal theory results.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Appendix B specifies the MILP form, benchmark scales, LNS repair procedure, operator-bank evolution settings, selector-training setup, baseline settings, and operator interface needed to reproduce the main results.

   Guidelines:

   - Reproducibility can be satisfied by detailed procedures, even if the code is not released.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: The manuscript provides reproduction details, but the code and local benchmark caches are not openly hosted in this draft. Readers therefore have a path to reproduction, but not an open-access release.

   Guidelines:

- If data or code are not released, explain the access path or why release is not possible.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The appendix lists the splits, hyperparameters, solver settings, seeds, budgets, and baseline configurations used for training, evolution, and evaluation.

   Guidelines:

   - Report enough hyperparameter and split detail for the reader to understand how results were obtained.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Appendix G reports results for three independently trained LoRA selectors with training seeds 0, 1, and 42. The main tables report point estimates; confidence intervals and significance tests are not reported.

   Guidelines:

   - If results are deterministic fixed-instance runs, it is acceptable to explain why error bars are not reported.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Appendix B reports the evaluation hardware and memory, solver threads, worker counts, runtime budgets, and offline data-collection, fine-tuning, and evolution times.

   Guidelines:

   - Report machine type, memory, and compute budget if those details matter for reproduction.

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work is algorithmic MILP benchmarking on standard or locally generated benchmark instances and does not involve human subjects, sensitive personal data, or other obvious ethics violations.

   Guidelines:

   - Answer [Yes] if the work conforms to the code of ethics as written.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: The manuscript does not include a broader-impact discussion covering both possible benefits and risks. The work is methodological, but the current draft does not spell out societal effects.

    Guidelines:

    - If broader impacts are relevant, discuss both possible benefits and harms.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The draft does not release a high-risk model or scraped dataset that would require special safeguard mechanisms.

    Guidelines:

    - Use [N/A] when the release does not pose the kind of misuse risk the question targets.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Appendix B.6 lists the external software, model, and benchmark assets used in the experiments together with their licenses. We cite MILPBench as the source of the benchmark instances and use them under its Apache-2.0 license without redistribution; Gurobi is used under its academic research license; SCIP and PySCIPOpt under the ZIB Academic License and the MIT License, respectively; and Qwen3-8B under the Apache-2.0 license.

    Guidelines:

    - Each asset's license is identified explicitly in Appendix B.6 and respected in the experiments.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: The paper documents the operator bank and selector setting for reproducibility, but it does not release a new standalone asset package in this draft.

    Guidelines:

    - Use [N/A] when no new asset package is being released with the paper.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or any human-subject study.

    Guidelines:

    - Use [N/A] when the paper has no human-subject component.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: No human-subject research was conducted, so IRB approval is not applicable.

    Guidelines:

    - Use [N/A] when no human-subject work was conducted.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLM usage is central to the core method: the paper fine-tunes Qwen3-8B as the selector and uses LLM-driven operator evolution and selection in the main pipeline.

Guidelines:

- Disclose LLM usage whenever it is part of the method itself, not just the writing process.