From Walls to Synergy: A Joint LLM-Evolution Framework for MILP Solvers

Ziao Guo[1], **Yuan Feng**[1], **Chenhao Ying**[1], **Junchi Yan**[1,2] *
[1]School of Computer Science & School of Artificial Intelligence, Shanghai Jiao Tong University
[2]Shanghai Innovation Institute
{ziao.guo, fy0123, yingchenhao, yanjunchi}@sjtu.edu.cn

Abstract

Machine learning (ML) methods have achieved notable success in enhancing individual components of mixed-integer linear programming (MILP) solvers. However, most existing approaches focus on single-component enhancement, neglecting critical interactions among components and often yielding diminishing returns when multiple approaches are integrated. The emergence of large language models (LLMs) provides a promising paradigm for jointly optimizing multiple solver components within a unified framework. In this paper, we propose **Jolly-MILP**, an LLM-guided framework to jointly optimize presolving, cut selection, and branching variable selection in exact MILP solvers. Our approach introduces a synergistic Coordinator to align component-level generators and a hierarchical Controller to produce adaptive, multi-stage strategies. Extensive experiments on nine MILP datasets demonstrate that Jolly-MILP consistently outperforms both ML-based approaches and existing LLM-based methods. Beyond raw speedups, our analysis shows that the gains do not stem from any single superior standalone component; rather, Jolly-MILP automatically discovers mutually adapted component strategies whose advantage emerges only under joint integration. These results indicate that explicitly modeling cross-component synergy is crucial for further gains in exact MILP solving, offering a new pathway toward solver-level co-design.

1 Introduction

Mixed-integer linear programming (MILP), as a fundamental combinatorial optimization (CO) problem, is significant across various areas, e.g., computer science and operations research, and also finds wide applications such as routing [36], planning [33], scheduling [15], etc. In practice, MILPs are typically solved with exact solvers like Gurobi [20] and SCIP [9], which operate within a sophisticated, multi-component computational framework. However, due to the NP-hard nature, MILPs are computationally intractable for large instances, necessitating the development of efficient solving methods.

Recently, there has been a growing trend in incorporating machine learning (ML) techniques to enhance and expedite traditional MILP solvers. Research efforts to enhance MILP solvers have focused on several key components. One line of work addresses the branching variable selection problem, employing ML models to predict and replace human-designed branching policies within solvers [17, 19]. In parallel, another stream of research targets the selection of cutting planes (cuts), utilizing ML models to adaptively choose which cuts to add [43, 38, 46]. Studies have also investigated the presolve component, leveraging ML models to tailor its parameters [30, 26].

*Correspondence author. This work was partly supported by National Cyber Security-National Science and Technology Major Project (No. 2026ZD1500700), NSFC (92370201, 12331014), and Shanghai Innovation Institute and the Shanghai Institute for Mathematics and Interdisciplinary Sciences (SIMIS) for the project (SIMIS-ID-26-AMS-001).

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

When practitioners seek to maximize solver efficiency, a natural inclination is to integrate ML methods that target different components of the MILP solver. However, a key issue arises: although the MILP solving process is executed in a unified manner across components, existing ML approaches are typically trained and optimized for individual components in isolation. This overlooks potential synergistic interactions between components. For example, an ML-based cut selection policy remains unaware of its impact on subsequent branch-and-bound process. Furthermore, integrating multiple pre-trained ML models into a single solver may shift the data distribution encountered during actual solving away from the distribution on which each model was separately trained, resulting in diminishing returns of the combined performance. A conceivable remedy would be to iteratively retrain the models while updating the training data after each round, yet this approach is clearly inefficient, imposes a substantial training burden, and is impractical for real-world deployment.

Meanwhile, the rise of large language models (LLMs) has introduced novel pathways for tackling CO problems by harnessing the broad knowledge inherent in these models. A growing research direction explores LLMs to automatically design algorithms for CO [32, 52], though much of this work has focused on generating heuristics for specific CO problem types, such as online bin packing and the traveling salesman problem (TSP). Recent research [27] has demonstrated that symbolic functions, which involve simple mathematical operations on input features, can yield highly effective branching policies. Such symbolic representations are also well-suited for generation by LLMs, offering a tractable and flexible space for algorithm synthesis. Crucially, the LLM-based algorithm generation framework offers high uniformity across different solver components. This uniformity naturally enables the *joint optimization* for multiple components within a single, cohesive pipeline, making it possible to systematically capitalize on the synergistic interactions between components that isolated ML approaches miss.

Building on this potential, we propose a joint LLM-based evolutionary framework for multiple components in MILP solvers, **Jolly-MILP**. Our approach considers three key components within the MILP solving pipeline—presolving, cut selection, and branching variable selection—and optimizes them under a unified algorithm generation framework that explicitly accounts for their synergistic interactions. This framework can produce algorithms for all three components simultaneously through a single generation process, eliminating the need for inefficient iterative retraining. Specifically, for each component, we instantiate a *Generator* responsible for generating candidate algorithms. Each Generator is paired with an *Analyzer* that analyzes performance variations and suggests component-specific evolutionary directions. Finally, a *Coordinator* synthesizes the suggestions from all three Analyzers, explicitly models the potential collaborative effects between components, and determines a coherent set of final evolutionary directions for the joint optimization of the components.

Moreover, we note that effective MILP solving often requires distinct algorithmic strategies at different stages of the solution process. For instance, the hybrid strong/pseudocost branching rule [2], which utilizes strong branching in the early stages and transitions to pseudocost branching in the later phases, has been shown to improve efficiency. To handle this multi-stage optimization within our framework, we devise a hierarchical evolutionary process for the branching variable selection component. Specifically, we introduce a dedicated *Controller*. The Controller monitors the effectiveness and timeliness of the algorithms generated by the Generator, ensuring that the most appropriate algorithm is applied at each stage of the solution process.

Extensive experiments are conducted to evaluate the effectiveness of Jolly-MILP. First, we evaluate our approach on nine popular MILP datasets, covering a broad range of instances from easy to hard. Jolly-MILP consistently outperforms integrated ML approaches and existing LLM-based approaches. Further analysis reveals that the performance of any *individually* LLM-generated component alone does not surpass that of ML models. **This finding underscores the critical role of joint optimization, and suggests a promising research direction that shifts from isolated component enhancement to solver-level co-design.** Then, we evaluate the generalization ability of our approach. We also perform ablation studies to examine the contribution of each element in our framework, and investigate the impact of different LLM backbones on overall performance. Additionally, We provide insight and visualization on our devised hierarchical evolutionary process.

The related works include ML for different MILP solver components and LLMs for solving CO, which we leave in Appendix A. **The main contributions of the paper are as follows.**

1) We propose Jolly-MILP, a unified LLM-guided framework that explicitly considers synergies and jointly optimizes multiple core components within exact MILP solvers. Unlike modular pipelines

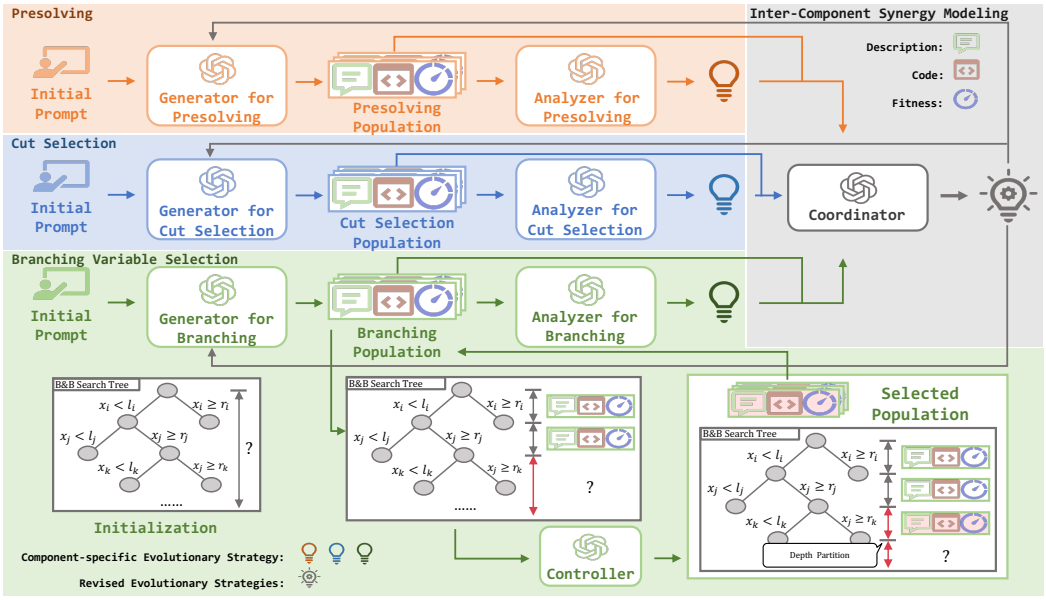


Figure 1: Overview of Jolly-MILP. The framework jointly optimizes three core components, i.e., presolving, cut selection, and branching variable selection, in a MILP solver. Each component is equipped with a dedicated Generator and Analyzer for algorithm generation and evolutionary strategy optimization, while a unified Coordinator aligns their evolution by modeling inter-component synergies. For the branching component, a Controller dynamically partitions the search tree, selects stage-specific strategies, and determines which branching strategies should be retained. The framework operates iteratively, continuously improving the overall performance of the solver.

that merely compose independently trained components at inference time, Jolly-MILP co-evolves the three components with a Coordinator that revises each component's search direction under integrated-solver fitness, so that the components are mutually re-optimized.

2) We design a hierarchical evolutionary process for branching variable selection, effectively enabling the generation of adaptive, multi-stage solving strategies. This mechanism addresses a key limitation of current LLM-based methods, which often lack the capacity to dynamically apply policies at different phases of the solution search.

3) Extensive empirical results show the effectiveness of Jolly-MILP. Crucially, our analysis reveals the decoupling between component-level and system-level performance, indicating that the performance gain stems not from superior standalone components, but from their coordination when integrated.

2 Methodology

In this section, we present the details of Jolly-MILP. Specifically, in Sec. 2.1, we first introduce the background regarding MILP, the branch-and-cut framework, and the presolving technique. Then, in Sec. 2.2, we elaborate on the basic structure of our multi-agent framework and outline a single evolutionary iteration. In Sec. 2.3, we concentrate on our proposed hierarchical evolutionary process based on different depth intervals of branch-and-bound search trees. Finally, in Sec. 2.4, we specify our devised fitness evaluation approach, which is critical for improving the efficiency of our framework. Fig. 1 shows the overall procedure of Jolly-MILP.

2.1 Preliminaries

Mixed Integer Linear Programming. Mixed Integer Linear Program (MILP) is an extension of linear programming (LP) where decision variables can be integers or continuous variables. A MILP $P = (\mathbf{A}, \mathbf{b}, \mathbf{c}, \mathbf{l}, \mathbf{u})$ is defined as:

$$\min_{\mathbf{x}} \{\mathbf{c}^\top \mathbf{x} | \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}\}, \tag{1}$$

where $\mathbf{c} \in \mathbb{R}^n$ is the coefficient of the objective function, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the coefficient matrix of the constraints, $\mathbf{b} \in \mathbb{R}^m$ is the right-hand side vector of the constraints, and $\mathbf{x} \in \mathbb{R}^n$ is the vector of decision variables. $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ are the lower and upper variable bound vectors, respectively. $p \leq n$ is the number of integer variables. If we drop the integrality constraints, the *LP relaxation* is obtained, which takes the form of:

$$\min_{\mathbf{x}} \{\mathbf{c}^\top \mathbf{x} | \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{x} \in \mathbb{R}^n\}. \tag{2}$$

Branch-and-Cut. Modern MILP solvers predominantly rely on the branch-and-cut framework, which interleaves branch-and-bound with the addition of cutting planes. Branch-and-bound constructs a search tree whose nodes are subproblems of Eq. 1; at each node, the LP relaxation is solved, and if its optimum $\mathbf{x}^*$ is fractional, a branching variable selection policy picks a fractional variable $\mathbf{x}_i$ and creates two children via $x_i \leq \lfloor \mathbf{x}_i^* \rfloor$ and $x_i \geq \lceil \mathbf{x}_i^* \rceil$. Integer-feasible relaxations yield upper bounds while LP objectives yield lower bounds, and the search proceeds until they converge (see Appendix B.1 for the standard details). Cutting planes are valid linear constraints added during solving that tighten the lower bound without excluding integer-feasible solutions, selected by a cut selection policy from a candidate pool. Crucially, cuts added at the root not only improve pruning but also reshape the LP solution values that serve as inputs to the branching policy, so the two components are tightly coupled rather than independent [47, 7]—an interaction our Coordinator explicitly exploits.

Presolving. Presolving is a standard preprocessing technique employed in modern MILP solvers. Its goal is to perform equivalence-preserving transformations on the original problem before the main solving process, typically to reduce the problem size through removing redundant constraints, tightening variable bounds, and strengthening the LP relaxation. Solvers usually maintain a pool of presolvers and execute multiple rounds of presolving. In each round, one presolver is selected and applied. Since the presolving phase directly determines the formulation passed to the branch-and-cut engine, it substantially influences the efficiency and behavior of the subsequent process.

2.2 Multi-Agent synergistic Framework

We propose a multi-agent framework to simultaneously generate and optimize presolving strategies, cut selection policies, and branching variable selection policies for MILP solvers. We first illustrate the roles of the three components.

1) The presolving strategy sets three key parameters of presolvers: priority, max-rounds, and timing, thereby deciding the applied presolver at each round during presolving.

2) The cut selection policy selects the subset of cuts in the pool of candidate cuts, and is applied at the root node of the branch-and-bound tree, aligned with ML counterparts.

3) The branching variable selection policy selects the variable for branching at every branching operation.

Our framework is designed to optimize all three components in a synergistic manner, leveraging LLMs to automatically generate and refine each algorithm while accounting for the performance and insight of other algorithms. Specifically, for each component, we devise two agents, including a Generator and an Analyzer, to capitalize on specific component evolution. Additionally, a joint Coordinator is utilized for all components to model and leverage potential synergistic interactions between components.

Generator. The Generator produces candidate algorithms based on given parent algorithms and an evolutionary strategy. Each output includes a code segment and a corresponding natural-language description (referred to as its "thought"). Inspired by symbolic-based approaches, Generator is instructed to select a subset of the extracted features from the solver and compose basic arithmetic expressions that reflect their semantic roles. The resulting algorithms are subsequently executed and evaluated, each obtaining a numerical fitness value.

Analyzer. The Analyzer is paired with the Generator and acts as a diagnostic expert. It takes pairs of the generated algorithms and their fitness values as input, and is instructed to perform comparative analysis on the potential reasons for differences in algorithm performance. It then produces causal insights and suggests possible evolutionary strategies for improved design. The Analyzer yields component-specific evolutionary directions to inform the Coordinator.

Coordinator. The Coordinator serves as the central module for joint optimization in our framework. While each Generator–Analyzer pair operates on a single solver component and thus cannot capture inter-component synergies, the Coordinator is specifically designed to bridge this gap. It takes as input the component-specific evolutionary directions suggested by the Analyzers and high-performing algorithms, and is asked to analyze the potential cooperative interactions among the components. The Coordinator then outputs a revised, globally consistent set of evolutionary strategies that collectively promote the overall performance of the solver. These integrated directions are subsequently used by all Generators in future rounds of algorithm generation.

Below, we summarize the key steps of a single evolutionary iteration in our framework:

1. **Algorithm Generation:** Each Generator produces new candidate algorithms (including code and natural-language descriptions) for its assigned solver component, given parent algorithms and the evolutionary strategy.
2. **Joint Evaluation:** The complete MILP solver, which integrates generated components, is executed on a set of instances to evaluate algorithm performances, yielding a fitness value for each algorithm to reflect its quality. Details are specified in Sec. 2.4.
3. **Causal Analysis:** Each Analyzer examines the evaluation outcomes, compares the performance of different algorithms for its component, and identifies the key factors contributing to performance differences. It then suggests promising evolutionary strategies for its component.
4. **Synergy-Aware Coordination:** The Coordinator receives all component-specific analyses from all Analyzers. It explicitly models interactions between components, resolves potential conflicts, and produces a revised evolutionary strategy for each component.
5. **Strategy Update:** The refined evolutionary strategies are stored, and the evolutionary strategy pool is managed and updated for future algorithm evolution.

2.3 Hierarchical Control for Multi-Phase Branching Strategy Evolution

The effectiveness of branch-and-bound can often be improved by dynamically applying different branching strategies at distinct phases of the solution process, as exemplified by hybrid strong/pseudocost branching rules [2]. To generate such adaptive, multi-phase strategies, we introduce a hierarchical evolutionary process enabled by a dedicated Controller agent, which serves as another core contribution of Jolly-MILP.

While the Generator for branching produces individual strategies for branching variable selection, the Controller, on the other hand, operates at a higher level to orchestrate their application across different stages of the search, defined by depth intervals in the branch-and-bound tree.

Specifically, the Controller receives the current stage of the solution process, i.e., the current depth range $[d_1, d_2)$ of the branch-and-bound tree, along with performance metrics for all branching strategies in the current algorithm population and a strong baseline algorithm. The performance metrics include solving time and node count at each depth level to provide comprehensive data. In practice, we select reliability pseudocost branching as the strong baseline.

Then, the Controller is required to analyze and compare the input data, conducting a depth partition through determining a depth partition point $d^\star$, where $d_1 < d^\star \leq d_2$, and selecting the promising branching strategies to be retained. After the depth partition, the currently best algorithm is assigned to the depth range $[d_1, d^\star)$, and is fixed in the following process. A new evolutionary process is subsequently initiated for the remaining depth range $[d^\star, d_2)$.

In this way, the capabilities of the Generator and the Controller are combined to dynamically manage the branching strategy population and optimize the solution process, ensuring that the most promising strategies are selected and assigned to appropriate stages in the solution process.

2.4 Fitness Evaluation

In this section, we detail the fitness evaluation scheme designed for our multi-component evolutionary framework. Designing an effective fitness function here is non-trivial, primarily due to the following interdependent challenges. First, to properly capture synergies among components, fitness must reflect the overall performance of the fully integrated solver. However, evaluating every possible combination of candidate algorithms from all components would incur prohibitive computational cost. Second, the component-specific Analyzers require fitness signals that are sensitive to the contribution

Table 1: Performance on easy, medium, and hard datasets. m and n refer to the number of constraints and variables, respectively. We report the 1-shifted geometric mean of the solving time/PD Integral, and improvement/effectiveness compared to the standalone solver (Default). The PD Integral is calculated within a 3600s time limit. We run the experiments 5 times with different random seeds.

	Easy: Multiple Knapsack (n = 720, m = 72)			Easy: Maximum Independent Set (n = 500, m = 1953)			Easy: Set Covering (n=1000, m=500)		
Method	Time(s) ↓	Improv. ↑	Effect. ↑	Time(s) ↓	Improv. ↑	Effect. ↑	Time(s) ↓	Improv. ↑	Effect. ↑
Default	2.57	-	-	13.30	-	-	16.47	-	-
SMAC3	2.02	21.40%	27%	12.19	8.35%	20%	12.78	22.40%	26%
EoH	2.58	-0.39%	6%	14.76	-10.98%	8%	15.94	3.22%	6%
ReEvo	2.06	19.84%	22%	10.28	22.71%	26%	13.66	17.06%	28%
Integrated-ML (config1)	1.86	27.63%	31%	9.48	28.72%	33%	11.07	32.79%	37%
Integrated-ML (config2)	1.94	24.51%	32%	9.13	31.35%	35%	10.64	35.40%	35%
Jolly-MILP (Ours)	**1.71**	**33.46%**	**45%**	**8.82**	**33.68%**	**48%**	**10.59**	**35.70%**	**46%**
	Medium: CORLAT (n = 466, m = 486)			Medium: MIK (n = 413, m = 346)			Hard: MIRP (n = 4120, m = 6857)		
Method	Time(s) ↓	Improv. ↑	Effect. ↑	Time(s) ↓	Improv. ↑	Effect. ↑	PD Integral ↓	Improv. ↑	Effect. ↑
Default	27.74	-	-	255.20	-	-	4804.68	-	-
SMAC3	22.16	20.12%	36%	245.29	3.88%	10%	4472.31	6.92%	30%
EoH	24.68	11.03%	12%	251.48	1.46%	5%	4727.19	1.61%	10%
ReEvo	23.72	14.49%	20%	248.45	2.64%	10%	4615.33	3.94%	10%
Integrated-ML (config1)	20.18	27.25%	52%	247.60	2.98%	10%	4326.11	9.96%	25%
Integrated-ML (config2)	19.22	30.71%	50%	241.15	5.51%	10%	4210.86	12.36%	30%
Jolly-MILP (Ours)	**18.31**	**33.99%**	**62%**	**236.81**	**7.21%**	**15%**	**3932.75**	**18.15%**	**40%**
	Hard: Item Placement (n = 1083, m = 195)			Hard: Load Balancing (n = 61000, m = 64304)			Hard: Anonymous (n = 37881, m = 49603)		
Method	PD Integral ↓	Improv. ↑	Effect. ↑	PD Integral ↓	Improv. ↑	Effect. ↑	PD Integral ↓	Improv. ↑	Effect. ↑
Default	304277.69	-	-	9462.41	-	-	146083.21	-	-
SMAC3	293248.72	3.62%	37%	8957.42	5.34%	34%	117962.47	19.25%	50%
EoH	310867.55	-2.17%	11%	9340.66	1.29%	16%	147231.44	-0.79%	5%
ReEvo	297246.38	2.31%	24%	9153.34	3.27%	27%	121686.85	16.70%	20%
Integrated-ML (config1)	291340.34	4.25%	46%	8774.50	7.27%	48%	110855.60	24.11%	60%
Integrated-ML (config2)	289463.98	4.87%	48%	8642.29	8.67%	46%	109492.37	25.05%	60%
Jolly-MILP (Ours)	**270763.40**	**11.01%**	**57%**	**8513.75**	**10.03%**	**59%**	**107324.21**	**26.53%**	**65%**

of their respective components; otherwise, the effect of a good or poor component-level algorithm could be obscured by the performance of other components.

To address these challenges, we design an iterative fitness evaluation scheme centered on a **test template** that holds one current algorithm for each of the three components (presolving, branching, and cut selection). When evaluating candidate algorithms for a particular component, we substitute only that component's algorithm in the template while keeping the other two fixed, and then measure the resulting overall solver performance. After evaluating candidates for all components, the template is updated to the best-performing combination observed in that iteration. Consequently, at most one component is replaced per evolutionary cycle.

We note that the single-component-per-cycle update pattern bears a surface similarity to coordinate descent, and thus could in principle inherit its susceptibility to local optima. Jolly-MILP mitigates this through the Coordinator, which refines the evolutionary strategies of each component using cross-component reasoning, so that every substitution is conditioned on the full joint context rather than greedily optimizing one axis in isolation. The iterative scheme additionally allows previously suboptimal components to re-enter the template in later rounds, preserving the ability to explore composite strategies over time. This represents a deliberate trade-off between evaluation efficiency and exhaustive exploration: exhaustive joint sampling of all components incurs prohibitive cost (>60× more time) while yielding only marginal gains (empirical results shown in Appendix J.5).

Ultimately, by holding other components fixed, the measured fitness cleanly isolates the contribution of the specific component under test, avoiding performance obscuration from unrelated algorithm interactions. Our design thus balances the goal of discovering synergistic combinations with the practical necessity of large-scale, computationally feasible evaluation.

Regarding the specific metrics used for fitness evaluation, we employ two complementary measures. The first and most direct metric is the solver's **solving time**, which offers an intuitive reflection of algorithm performance. However, for challenging problem instances where the solver cannot find the optimal solution within a short time frame, relying solely on solving time can significantly hinder evolutionary progress due to excessive evaluation cost. Therefore, for difficult instances, we adopt the **primal-dual gap integral (PD integral)** computed within a fixed, acceptable time limit as a surrogate performance metric. We provide more details on PD integral in Appendix B.4.

Table 2: Generalization ability to larger-scale instances. The reported metric is the solving time (lower is better).

Method	Maximum Independent Set		Set Covering	
	Medium (n=1000, m=3946)	Large (n=1500, m=5940)	Medium (n=1000, m=1000)	Large (n=1000, m=2000)
Default	258.71	2323.72	103.21	2046.75
ReEvo	236.43	2281.29	92.60	2087.84
Integrated-ML (config2)	202.68	2107.35	76.34	1719.06
Jolly-MILP (Ours)	**190.36**	**2058.63**	**67.87**	**1703.12**

We provide the formal mathematical description and general workflow in Appendix F, and a detailed algorithmic process in Appendix G.

3 Experiments

3.1 Experimental Settings

Datasets. We evaluate our approach on nine popular MILP benchmarks, including both classical synthetic MILP problems and challenging MILP problems from wide application scenarios. The benchmarks can be divided into three levels of difficulty: easy, medium, and hard. *1) Easy datasets* comprise three classical synthetic MILP problems: Multiple Knapsack [39], Maximum Independent Set [8], and Set Covering [6]. We artificially generate instances in line with prior research [17]. *2) Medium datasets* contain CORLAT [18] and MIK [5]. *3) Hard datasets* include real-world inspired datasets from the NeurIPS 2021 competition [16]: Item Placement, Load Balancing, and Anonymous; and a subset of MIRPLIB (Maritime Inventory Routing Problem, MIRP) [37], following [30]. Please refer to Appendix H for more details.

Metrics. We adopt SCIP 8.0.3 solver [9], a state-of-the-art open-source MILP solver, as the backend solver for all experiments. Following prior research [30], we utilize 1-shifted geometric mean of the solving time (**Time**, lower is better) and primal-dual gap integral (**PD integral**, lower is better) as evaluation metrics. We also calculate the performance improvement (**Improv.**, higher is better) over the standalone SCIP solver and the effectiveness (**Effect.**, higher is better), which is defined as the proportion of instances for which the evaluated method outperforms the standalone SCIP. We provide more details on PD integral in Appendix B.4.

Baselines. To ensure a fair comparison at the level of overall solver performance, Jolly-MILP is evaluated against baselines that integrate corresponding ML components. We construct two full-solver configurations by combining strong ML methods for each component. **1) Integrated-ML (config 1)**: L2P for presolving [30], L2C for cut selection [43], and GCNN for branching variable selection [17]. **2) Integrated-ML (config 2)**: L2P for presolving, HEM for cut selection [46], and Symb4CO for branching [27]. Note that HEM and Symb4CO are more recent and advanced approaches. We further compare with two LLM-based frameworks, **EoH** [32] and **ReEvo** [52], each used to independently generate and then integrate strategies for the same three components. In addition, we include **SMAC3** [29], a widely used automatic configuration tool, to perform holistic hyperparameter tuning of SCIP. The default setting of SCIP (**Default**) is also reported for reference.

Hyperparameters and detailed experimental setup for Jolly-MILP and baselines are provided in Appendix I. Moreover, although fully joint retraining of ML components is impractical in deployment (Sec. 1), we report a reduced-cost joint-alignment variant as an additional reference baseline in Appendix J.6. Notably, Jolly-MILP outperforms this variant on both easy tasks and the hard Item Placement task while consuming 10× less time on CPU.

3.2 Empirical Results and Further Analysis

Comparative Evaluation. The performance evaluation is summarized in Table 1. We also present the detailed standard deviation results in Table 13. Jolly-MILP achieves the best performance across the easy, medium, and hard datasets, surpassing both integrated ML solvers and existing

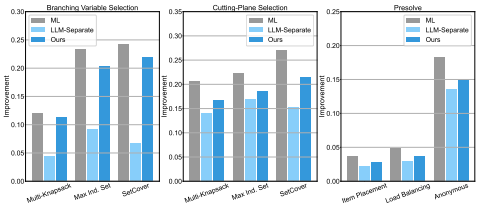


Figure 2: Comparison of the improvement when each component is individually applied to the SCIP solver.

Table 3: Ablation study on different elements of Jolly-MILP. The reported metric is the solving time (lower is better).

Dataset	Full	w/o Coordinator	w/o Controller	w/o Coordinator & Controller
Multiple Knapsack	1.71	2.13	2.07	2.50
Maximum Independent Set	8.82	10.15	9.54	11.26
Set Covering	10.59	13.24	11.98	14.29

Table 4: Improvement and Search Time (in brackets) starting from MK-evolved population.

MIS	SC	CORLAT	MIK	Item Placement	Load Balancing
33.75% (0.3h)	36.24% (0.4h)	29.08% (0.7h)	5.92% (1.4h)	8.41% (2.2h)	7.30% (1.8h)

Table 5: Sham-Coordinator and token-controlled ablations. The sham-Coordinator is a non-LLM rule without cross-component reasoning; its saved tokens are reallocated to extra rounds. The token-controlled variants grant the w/o-Coordinator model more round (token) budget. T_0 denotes Full's total LLM-token consumption.

Variant	**LLM tokens**	**MK (time ↓)**	**MIS (time ↓)**
Full (with Coordinator)	1.0 T_0	**1.71**	**8.82**
w/o Coordinator	~0.7 T_0	2.13	10.15
w/o Coordinator, 2× rounds	~1.4 T_0	2.02	9.94
w/o Coordinator, 4× rounds	~2.8 T_0	1.99	9.86
Sham Coordinator (mechanical merge, matched LLM tokens)	~1.0 T_0	2.08	10.06

LLM-based generation frameworks. Notably, Jolly-MILP not only attains the greatest improvement in Time or PD Integral but also exhibits a significantly higher effectiveness score. For example, on the Easy Multiple Knapsack dataset, Jolly-MILP reduces solving time by 33.46% (vs. 27.63% for the best ML baseline) while achieving a 45% effectiveness rate (vs. 31% for the same baseline). This pattern holds even on the challenging MIRP and Item Placement instances, where our method yields improvements of 18.15% and 11.01%, respectively—considerably higher than those of baselines. These results highlight two key findings: 1) joint LLM-driven evolution can outperform carefully integrated, individually trained ML components, effectively bridging the gap between LLM- and ML-based approaches on exact MILP solving tasks; and 2) the strategies generated by Jolly-MILP generalize more robustly across instances, as reflected in the consistently higher effectiveness metric.

Separated Components Evaluation. To further analyze the source of performance improvement, we evaluate each component in isolation by integrating only one component into the solver while keeping the other components fixed to the default configurations. The results are shown in Fig. 2, where ML denotes Symb4CO, HEM, and L2P, respectively; LLM-Separate refers to ReEvo. As the effect of presolving is not pronounced on easy problems, we report its performance on three hard datasets. An important and counter-intuitive finding is that LLM-generated single components underperform specialized ML counterparts, yet the fully integrated Jolly-MILP surpasses all baselines. This decoupling between component-level and system-level performance directly challenges the dominant paradigm of isolated component enhancement in the MILP-ML literature, and constitutes one of our core empirical contributions.

The significant improvement of our method on branching over the LLM baseline can be primarily attributed to the introduced hierarchical evolutionary process, which enables adaptive, multi-phase strategies. For the other components, the improvements mainly stem from the evolution of evolutionary directions. Furthermore, our method surpasses all baselines only when all components are jointly integrated, indicating that its overall advantage does not stem from the superior performance of any single component. Instead, it arises from Jolly-MILP's ability to explicitly exploit synergistic interactions through coordinated co-evolution.

Generalization. We evaluate the generalization ability by synthetically generating larger-scale Maximum Independent Set and Set Covering instances and testing on them using the models and strategies obtained from the easy datasets. Table 2 reports the solving time. Jolly-MILP consistently achieves the best performance on both medium and large instances. These results indicate that the

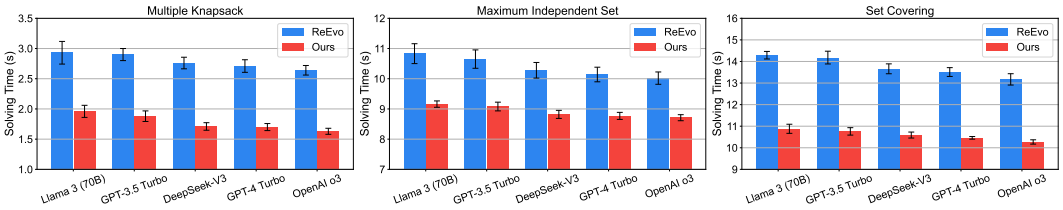


Figure 3: Performance with different LLM backbones. Three independent evolutionary runs are performed for each setting. For each run, the prompt is slightly modified and rephrased to evaluate robustness to prompt variations.

strategies generated by Jolly-MILP not only generalize well to broader instances of the same scale, but also transfer effectively across instance sizes while maintaining robust performance.

Ablation Study. Table 3 presents the effect of ablating core elements in Jolly-MILP: the Coordinator (the Analyzers' suggestions are used directly) and the Controller. The results show that removing either element leads to a clear performance drop. Notably, the degradation is more pronounced when the Coordinator is removed, underscoring that leveraging inter-component synergies plays a more critical role in generating high-quality strategies and improving overall solver performance.

Furthermore, we conduct two extra ablations:

- We grant the w/o-Coordinator variant 2× and 4× the round budget of the full model, resulting in 1.4× and 2.8× the token budget, and re-run the evolution.
- To separate genuine cross-component reasoning, we introduce a **sham-Coordinator**: a non-LLM rule that receives the same Analyzer suggestions as the full model but merges them without any cross-component analysis. The sham control exposes every Generator to a mechanical merge of all three suggestions, yet performs no reasoning about how the components interact.

Table 5 reports both results. The **sham-Coordinator** performs much worse, indicating that the benefit comes from the content of the cross-component guidance. As for the **token-controlled** ablation, even at the largest budget we test, the w/o-Coordinator variant stays well behind the full model, and the gap barely narrows as the budget grows.

Adopting Different LLM Backbones. To evaluate the robustness of Jolly-MILP to the adopted LLM backbones, we conduct experiments with several LLMs and compare the results with those of ReEvo. As shown in Fig. 3, our method demonstrates robustness across LLMs and consistently outperforms ReEvo. Moreover, even when employing a relatively weak LLM, its performance remains comparable with strong ML-based baselines. We also observe that more capable LLMs consistently yield better performance, indicating that future advances in LLMs can be directly leveraged to further improve the performance of Jolly-MILP.

Transferability. Beyond its performance on specific benchmarks, we evaluate Jolly-MILP's ability to generalize across different problem types. We conduct zero-shot transfer experiments to assess whether strategies discovered for one problem class maintain efficacy when applied to others without further training. As shown in Table 6, strategies evolved for one task maintain notable gains across diverse datasets, highlighting the zero-shot transferability.

Table 6: Improvement under zero-shot transfer setting. The generated strategies are evaluated on previously unseen datasets.

Train \ Evaluate	MK	MIS	SC
MK	–	6.17%	8.32%
MIS	10.25%	–	21.09%
SC	11.13%	19.62%	–

Furthermore, we evaluate whether Jolly-MILP achieves a rapid adaptation via warm-starting. We initialize the population with high-quality strategies derived from a different and easy dataset (e.g., MK), and run Jolly-MILP with a significantly reduced search time limit on harder problems. As demonstrated in Table 4, notable gains are achieved in short search times (0.3 to 2.2 hours), proving that pre-evolved strategies from simpler tasks can drastically accelerate deployment for entirely new and harder problem classes, and reflecting the significant practical value for real-world deployment.

Further Analysis of Hierarchical Control. To further elucidate the benefits of hierarchical control, we investigate the evolution of solving time throughout the evolutionary process. As shown in Fig. 4, once solving time approaches convergence, the Controller triggers a depth partition. This intervention effectively overcomes the convergence plateau and enables continued performance improvement,

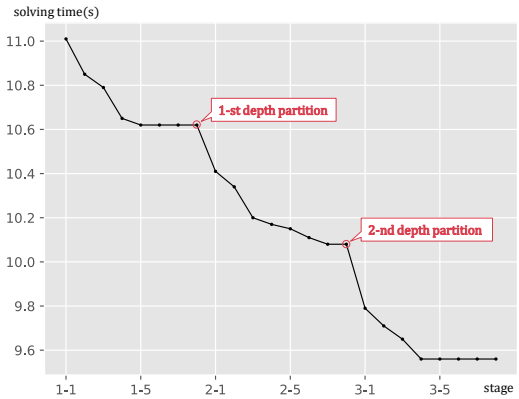


Figure 4: Solving time during the evolution process on a Setcover instance across different evolutionary stages. The x-axis tick 'a-b' represents the b-th evolution round within the a-th depth interval, while the y-axis tick denotes the solving time at the current stage.

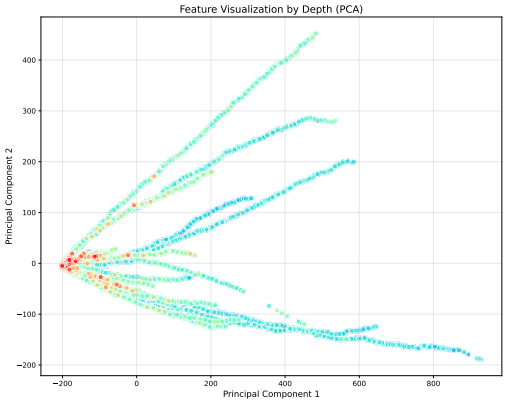


Figure 5: Visualization of variable features w.r.t depth. Each point on the two-dimensional plane represents the feature of a variable during the branching process in the branch-and-bound framework. The colors range from cold (teal) to warm (red), corresponding to the increasing depth of the current variable during branching.

Table 7: Impact of varying the number of depth intervals on the solving time (lower is better).

Dataset	1-Interval	2-Interval	3-Interval	4-Interval	5-Interval
Multiple Knapsack	2.07	1.86	1.71	1.70	1.70
Maximum Independent Set	9.54	9.08	8.82	8.82	8.81
Set Covering	11.98	11.02	10.59	10.61	10.58

thereby raising the achievable performance ceiling. Additionally, in Fig. 5, we visualize the variable features utilized for generating branching strategies across different tree depths. The visualization reveals a clear shift in feature distributions with respect to depth, further supporting our motivation for adopting multi-stage optimization of branching strategies.

Different Number of Depth Intervals. We further conduct an ablation study to analyze the impact of the number of depth intervals used for partitioning. Specifically, we evaluate configurations with the number of intervals ranging from 1 to 5, and report the corresponding solving times in Table 7. The results show that increasing the number of intervals initially leads to noticeable performance improvements, confirming our motivation for adopting multi-stage branching strategies. However, beyond three intervals, the marginal gains become limited, and the performance gradually converges for configurations with three to five intervals. Considering the additional computational overhead and longer evolutionary time associated with larger numbers of intervals, we set the number of depth intervals to three in our experiments, which provides a favorable trade-off between performance and evolutionary time.

4 Conclusion

In this paper, we propose Jolly-MILP, a unified LLM-guided evolutionary framework that jointly optimizes multiple core components of a MILP solver—presolving, cut selection, and branching variable selection—by accounting for their synergistic interactions. Additionally, a hierarchical control mechanism is introduced, enabling multi-stage branching policy generation. Extensive experiments show that Jolly-MILP outperforms both integrated ML approaches and existing LLM-based algorithm generation frameworks, effectively bridging the performance gap between ML and LLM-based approaches in enhancing MILP solvers and exact MILP solving. Ablation studies confirm the critical role of capturing inter-component cooperation and enabling adaptive multi-stage strategies. Our work demonstrates that joint optimization, rather than isolated component enhancement, is key to acquiring further gains in MILP solving, and offers a novel paradigm for leveraging LLMs to design more generalizable and synergistic strategies.

References

[1] Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. Constraint integer programming: A new approach to integrate cp and mip. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 6–20. Springer, 2008.

[2] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[3] Milad Taleby Ahvanooey, Qianmu Li, Ming Wu, and Shuo Wang. A survey of genetic programming and its applications. *KSII Transactions on Internet and Information Systems (TIIS)*, 13(4):1765–1794, 2019.

[4] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[5] Alper Atamtürk. On the facets of the mixed–integer knapsack polyhedron. *Mathematical Programming*, 98(1):145–175, 2003.

[6] Egon Balas and Andrew Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. *Combinatorial Optimization*, pages 37–60, 1980.

[7] 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.

[8] David Bergman, Andre A Cire, Willem-Jan Van Hoeve, and John Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[9] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021.

[10] Aymeric Blot, Holger H Hoos, Laetitia Jourdan, Marie-Éléonore Kessaci-Marmion, and Heike Trautmann. Mo-paramils: A multi-objective automatic algorithm configuration framework. In *Learning and Intelligent Optimization: 10th International Conference, LION 10, Ischia, Italy, May 29–June 1, 2016, Revised Selected Papers 10*, pages 32–47. Springer, 2016.

[11] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. *Handbook of metaheuristics*, pages 457–474, 2003.

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

[13] Edmund K Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R Woodward. A classification of hyper-heuristic approaches. *Handbook of metaheuristics*, pages 449–468, 2010.

[14] Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to optimize: A primer and a benchmark. *Journal of Machine Learning Research*, 23(189):1–59, 2022.

[15] Yves Crama. Combinatorial optimization models for production scheduling in automated manufacturing systems. *European Journal of Operational Research*, 99(1):136–153, 1997.

[16] Maxime Gasse, Simon Bowly, Quentin Cappart, Jonas Charfreitag, Laurent Charlin, Didier Chételat, Antonia Chmiela, Justin Dumouchelle, Ambros Gleixner, Aleksandr M Kazachkov, et al. The machine learning for combinatorial optimization competition (ml4co): Results and insights. In *NeurIPS 2021 competitions and demonstrations track*, pages 220–231. PMLR, 2022.

[17] 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.

[18] Carla P Gomes, Willem-Jan Van Hoeve, and Ashish Sabharwal. Connections in networks: A hybrid approach. In *International conference on integration of artificial intelligence (AI) and operations research (OR) techniques in constraint programming*, pages 303–307. Springer, 2008.

[19] 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.

[20] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[21] Abdelrahman Hosny and Sherief Reda. Automatic milp solver configuration by learning problem similarities. *Annals of Operations Research*, 339(1):909–936, 2024.

[22] Zhinan Hou, Xingchen Li, Yankai Zhang, Tianxun Li, and Keyou You. Llm4branch: Large language model for discovering efficient branching policies of integer programs. *arXiv preprint arXiv:2605.10401*, 2026.

[23] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of artificial intelligence research*, 36:267–306, 2009.

[24] Shubhra Kanti Karmaker, Md Mahadi Hassan, Micah J Smith, Lei Xu, Chengxiang Zhai, and Kalyan Veeramachaneni. Automl to date and beyond: Challenges and opportunities. *ACM Computing Surveys (CSUR)*, 54(8):1–36, 2021.

[25] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[26] Yufei Kuang, Xijun Li, Jie Wang, Fangzhou Zhu, Meng Lu, Zhihai Wang, Jia Zeng, Houqiang Li, Yongdong Zhang, and Feng Wu. Accelerate presolve in large-scale linear programming via reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2025.

[27] Yufei Kuang, Jie Wang, Haoyang Liu, Fangzhou Zhu, Xijun Li, Jia Zeng, HAO Jianye, 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*, 2024.

[28] Yufei Kuang, Jie Wang, Yuyan Zhou, Xijun Li, Fangzhou Zhu, HAO Jianye, 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*, 2024.

[29] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

[30] 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*, 2024.

[31] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. An example of evolutionary computation+ large language model beating human: Design of efficient guided local search. *arXiv preprint arXiv:2401.02051*, 2024.

[32] Fei Liu, Tong Xialiang, 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 *Forty-first International Conference on Machine Learning*, 2024.

[33] Thomas L Magnanti. Combinatorial optimization and vehicle fleet planning: Perspectives and prospects. *Networks*, 11(2):179–213, 1981.

[34] Muhammad Umair Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. Llmatic: neural architecture search via large language models and quality diversity optimization. In *proceedings of the Genetic and Evolutionary Computation Conference*, pages 1110–1118, 2024.

[35] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*, 2023.

[36] Carlos AS Oliveira and Panos M Pardalos. A survey of combinatorial optimization problems in multicast routing. *Computers & Operations Research*, 32(8):1953–1981, 2005.

[37] Dimitri J Papageorgiou, George L Nemhauser, Joel Sokol, Myun-Seok Cheon, and Ahmet B Keha. Mirplib–a library of maritime inventory routing problem instances: Survey, core model, and benchmark results. *European Journal of Operational Research*, 235(2):350–366, 2014.

[38] 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*, pages 17584–17600. PMLR, 2022.

[39] Lara Scavuzzo, Feng Chen, Didier Chételat, Maxime Gasse, Andrea Lodi, Neil Yorke-Smith, and Karen Aardal. Learning to branch with tree mdps. *Advances in neural information processing systems*, 35:18514–18526, 2022.

[40] Dhruv Shah, Michael Robert Equi, Błażej Osiński, Fei Xia, Brian Ichter, and Sergey Levine. Navigation with large language models: Semantic guesswork as a heuristic for planning. In *Conference on Robot Learning*, pages 2683–2699. PMLR, 2023.

[41] Thomas Stützle and Manuel López-Ibáñez. Automated design of metaheuristic algorithms. *Handbook of metaheuristics*, pages 541–579, 2019.

[42] Yiwen Sun, Xianyin Zhang, Shiyu Huang, Shaowei Cai, BingZhen Zhang, and Ke Wei. Autosat: Automatically optimize sat solvers via large language models. *arXiv preprint arXiv:2402.10705*, 2024.

[43] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *International conference on machine learning*, pages 9367–9376. PMLR, 2020.

[44] Mark Turner, Thorsten Koch, Felipe Serrano, and Michael Winkler. Adaptive cut selection in mixed-integer linear programming. *arXiv preprint arXiv:2202.10962*, 2022.

[45] Amala Mary Vincent and P Jidesh. An improved hyperparameter optimization framework for automl systems using evolutionary algorithms. *Scientific Reports*, 13(1):4737, 2023.

[46] 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.

[47] Franz Wesselmann and Uwe Stuhl. Implementing cutting plane management and selection techniques. In *Technical Report*. University of Paderborn, 2012.

[48] Xingyu Wu, Yan Zhong, Jibin Wu, and Kay Chen Tan. As-llm: When algorithm selection meets large language model. *arXiv preprint arXiv:2311.13184*, 2023.

[49] Hengjia Xiao and Peng Wang. Llm a*: Human in the loop large language models enabled a* search for robotics. *arXiv preprint arXiv:2312.01797*, 2023.

[50] 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.

[51] Milad Yazdani, Mahdi Mostajabdaveh, Samin Aref, and Zirui Zhou. Evocut: Strengthening integer programs via evolution-guided language models. *arXiv preprint arXiv:2508.11850*, 2025.

[52] 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. *arXiv preprint arXiv:2402.01145*, 2024.

[53] Huigen Ye, Hua Xu, An Yan, and Yaoyang Cheng. Large language model-driven large neighborhood search for large-scale milp problems. In *ICML*, 2025.

[54] Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies. In *Proceedings of the aaai conference on artificial intelligence*, volume 35, pages 3931–3939, 2021.

[55] Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. *Advances in Neural Information Processing Systems*, 36, 2024.

A Related Work

A.1 Neural Solver for MILP

Modern MILP solvers comprise key components such as branching, cut selection, and presolving, all of which play essential roles in improving solver efficiency [1]. Machine learning (ML) has been successfully applied to these components, leading to significant performance gains. Branching strategies, which determine the variable selection at each search tree node, have benefited from ML-driven approaches [17, 19, 54]. Cut selection, which refines the MILP formulation by selecting effective cutting planes, has also been optimized through ML, with adaptive strategies [43, 44, 38, 46]. Additionally, ML-based presolving methods, such as L2P [30], have demonstrated effectiveness in reducing problem size by identifying redundant constraints, fixing variables, and finding generalized upper bounds.

Despite these advancements, significant challenges persist. A central issue is that existing ML-enhanced components are typically optimized in isolation, without accounting for their performance when all of them are integrated into the same solver. First, improving one component individually may inadvertently impact others, leading to suboptimal global performance even when each component appears better in isolation. Second, because each model is trained on data generated separately, their integration often shifts the actual solving-time data distribution, resulting in out-of-distribution inputs and degraded real-world performance. Jolly-MILP jointly optimizes three core components—branching, cut selection, and presolving—within a single LLM-guided evolutionary process. Our approach explicitly guides the LLM to exploit synergies among components, aiming to achieve greater overall performance improvements.

A.2 LLM for Solving CO

With significant advancements in the capabilities of LLMs [35], emerging efforts have been made to leverage LLMs to solve CO problems. Some approaches utilize LLMs as optimizers [50], generating new solutions through in-context learning, struggling to handle large-scale problems. Others have employed LLMs to directly generate new algorithms [31, 42, 32, 52], e.g., by incorporating evolutionary computation into prompt engineering to facilitate automated algorithm improvement. Specifically, LLM-based design and enhancement of algorithms have been successfully applied to various classic CO problems, including online bin packing, traveling salesman problems (TSP), flow shop scheduling, and satisfiability (SAT).

Moreover, a growing line of work applies LLMs to individual stages of the MILP pipeline. LLM4Branch [22] prompts an LLM to synthesize or score branching rules, targeting the variable-selection stage in isolation. EvoCut [51] evolves cutting-plane generation strategies at the model level, focusing solely on the separation stage. LLM-LNS [53] instead operates outside the exact solving loop, using an LLM to steer a large-neighborhood-search metaheuristic over incumbent solutions. While each demonstrates that LLMs can improve a single slot of the pipeline, all three optimize their target stage independently of the others, leaving the cross-component interactions—precisely the effects our approach is designed to exploit—unaddressed.

B Additional Background

B.1 Branch-and-Bound Bound Convergence

Modern MILP solvers predominantly rely on the branch-and-cut framework, which integrates two key algorithmic pillars: the branch-and-bound procedure and the addition of cutting planes (cuts). The branch-and-bound search iteratively processes leaf nodes by solving their LP relaxations. If the LP optimum $\mathbf{x}^*$ satisfies all integrality constraints, it is feasible for the original MILP and provides an upper bound on the optimal objective value; meanwhile, the objective value of each LP relaxation naturally provides a lower bound. Throughout the search, these bounds are progressively tightened until they converge, at which point an optimal solution is certified. For cutting planes, solvers first generate a pool of candidate cuts, from which a cut selection policy selects a subset to add to the LP. The addition of cuts influences the subsequent branching process in two ways: (i) it improves the lower bound, enabling more effective pruning of the search tree; and (ii) it modifies features such as LP solution values that serve as critical inputs to the branching variable selection policy. Prior studies

have shown that cuts added at the root node can decisively influence the entire branch-and-bound tree [47, 7].

B.2 Automatic Algorithm Design

Hyper-heuristics, a paradigm for the automatic design of heuristic algorithms, have been extensively studied and developed to address complex optimization problems [11, 13, 12, 41]. These methods aim to generate or adapt heuristics for solving a wide range of problems without the need for manual tuning, thereby providing a general and flexible framework. Over the years, various approaches have been proposed to enhance the performance and generality of hyper-heuristics, including parameter configuration tools such as ParamILS [23], multi-objective optimization techniques like those in [10], and more recent advancements utilizing modern optimization frameworks such as Optuna [4]. Improvements in algorithmic efficiency and robustness have also been demonstrated, as highlighted by recent developments in hyper-heuristic strategies [45].

Machine learning has played a transformative role in the hyper-heuristic framework, enabling data-driven approaches to heuristic selection and generation [14, 24, 7]. These methods leverage the ability of machine learning models to capture complex patterns in problem-specific data, thus facilitating the design of heuristics that can dynamically adapt to diverse optimization scenarios. Among the various techniques, genetic programming has emerged as a particularly promising approach [3], offering a pathway to generating interpretable heuristic algorithms. This explainability stems from the tree-based representation of genetic programming solutions, which allows practitioners to analyze and understand the underlying heuristic logic. Despite its potential, the application of genetic programming to hyper-heuristics is not without challenges. It often relies on hand-crafted algorithmic components and significant domain-specific knowledge, which can limit its general applicability and scalability to new problems.

In recent years, the capabilities of large language models (LLMs) have advanced significantly, leading to their application in diverse domains [35]. A growing body of research explores the potential of LLMs as foundational components to enhance algorithmic performance. One prominent line of work leverages LLMs as optimizers, utilizing their in-context learning capabilities to directly generate new trial solutions [50]. While promising, this approach encounters significant challenges when applied to complex problems characterized by expansive search spaces, where the limitations of LLMs in handling such scale become evident [34, 55]. Beyond direct optimization, recent studies have investigated the integration of LLMs into the algorithm design process. In these applications, LLMs are employed to extract high-level algorithmic features for heuristic selection [48], offer guidance to improve heuristic strategies [40], and even design specific algorithmic components [49]. These approaches highlight the versatility of LLMs in algorithm development.

B.3 Machine Learning for Branching Variable Selection

Early research on ML for branching variable selection [17, 19] typically involves deep neural networks (DNNs), e.g., graph neural networks (GNNs), to approximate the strong branching policy, which optimizes the bounds to the greatest extent possible but may be computationally expensive. More recently, methods of symbolic learning for branching variable selection [27, 28] have emerged. These approaches are built upon symbolic regression (SR) to construct interpretable scoring functions based on instance features and have significantly improved inference efficiency. These SR-based approaches may involve searching through high-dimensional symbolic spaces from scratch. The symbolic-based approaches successfully prove that functions involving only simple mathematical operations can yield highly effective branching variable selection policies for MILP solvers, motivating us to leverage LLMs to generate symbolic functions for other solver components.

B.4 Details on the Primal-Dual Gap Integral

When we solve MILPs through modern MILP solvers, two important bounds are tracked, including the primal bound and the dual bound. The primal bound (a.k.a. global upper bound) refers to the objective value of the best feasible solution found so far. Assume at time t, the best feasible solution is $\mathbf{x_t^*}$, and the objective coefficient vector is $\mathbf{c}$. Then, the primal bound is calculated as $\mathbf{c}^\top \mathbf{x_t^*}$. The dual bound (a.k.a. global lower bound) corresponds to a solution of a valid relaxation of the MILP, and reflects the best lower bound of the original MILP problem. Finally, the **primal-dual gap**

integral is defined as the area between two curves, the solver's primal bound and the solver's dual bound. Therefore, this metric benefits both from improvements on the primal side (finding good feasible solutions), and the dual side (producing a tight optimality certificate). With a time limit T, the primal-dual gap integral can be calculated as:

$$\int_{t=0}^{T} \mathbf{c}^\top \mathbf{x}_\mathbf{t}^* - \mathbf{z}_\mathbf{t}^* \mathrm{d}t, \tag{3}$$

where $\mathbf{z}_\mathbf{t}^*$ denotes the dual bound at time t. The lower primal-dual gap integral refers to better performance. Note that an initial value is required at time $t = 0$. Therefore, MILP solvers such as SCIP always provide an initial trivial solution $\mathbf{x}_\mathbf{0}^*$ and a valid initial dual bound $\mathbf{z}_\mathbf{0}^*$. The primal-dual gap integral is recognized as a widely used and effective metric for evaluating solver performance. For instance, in the NeurIPS 2021 ML4CO competition [16], the primal-dual gap integral is adopted as a primary evaluation metric.

C Details on Prompts

C.1 Initialization Prompt for Algorithm Population

In this section, we illustrate the prompt used by the Generator for algorithm population initialization.

In the initialization process, the Generator produces the algorithm population based on the task context and the initialization strategy. The prompt structure consists of a problem description, initialization task instructions, a code format description, input-output specifications, and supplementary details. Notably, to enhance the quality of the initialized algorithm population, we also provide component-specific instructions for different components. These include descriptions of related high-performance heuristics and design insights, which assist the LLM in efficiently navigating the vast initial algorithm search space and locating truly effective algorithms.

1. Problem Description: A specific task description within the context of a particular type of MILP problems.

2. Component-Specific Instruction: Depending on different assigned components, different initialization instructions to the Generators are given. The instructions serve as the design guidance for each component, consisting of effective heuristics in high-performance solvers and inductive biases/design insights. We provide the specific instructions for the three components in Fig. 6, Fig. 7, and Fig. 8.

3. Code Format Description: This section standardizes the code format, specifying the function names, the number and names of parameters, parameter types, the number and names of return values, as well as return value types.

4. Input-Output Information: This section provides a more detailed description of the function's input and output, namely the information about parameters and return values. In Jolly-MILP, the input represents the instance and the current state of the solver. Based on prior research [17, 25], the state can be abstracted into a bipartite graph, and various extracted features can be used to characterize the bipartite graph. Therefore, the input consists of three feature maps: `constraint_features`, `edge_features`, and `variable_features`. For cut selection, the input also involves the candidate cuts (`cuts`), which is a list of scip Rows. The output varies for different components. For presolving, the output is a dictionary, `presolve_params`, which involves key parameters of 14 involved presolvers (specified in Table 11): priority, max-rounds, and timing. For cut selection, the output is a sorted list of candidate cuts, i.e., `sorted_cuts`, where the rank indicates the priority of selecting each cut. For branching, the output is a score array `scores` with a length equal to the number of variables. The values in this array are normalized within the range $[0, 1]$, and based on the scores, the branching variable b can be determined as $b = \text{argmax } scores_x$.

Moreover, for all components, we have specifically provided the Generator with all feature names and their corresponding definitions, due to the excessive number and ambiguous meanings of variable features. For presolving, the descriptions of 14 presolvers are also involved. For cut selection, we also include the 15 features of the candidate cuts extracted through scip.Row API in the input information. We show the prompts in Fig. 9.

5. Additional Details: Additional supplementary information will be provided here, such as "do not output other explanatory information", "ensure that the array sizes match when performing numpy

[Presolving – Design Guidance]

You are now designing an algorithm for presolving.
Effective presolving strategies rely on simple, safe, and local reasoning that can be applied repeatedly at very low cost.

High-performance presolving heuristics commonly exhibit the following characteristics:
• Emphasis on bound tightening and variable fixing rather than complex transformations.
• Extensive use of single-constraint reasoning, where variable bounds are inferred from the minimum and maximum activity of a constraint.
• Preference for conservative reductions that provably preserve feasibility and optimality.
• Early elimination of trivial or dominated constraints to reduce model size.
• Exploitation of coefficient patterns, such as singleton variables or variables appearing with large coefficients, to enable safe eliminations.
• Iterative application of multiple lightweight rules, where each rule is independently useful but their combination yields stronger reductions.

Design insights:
• Presolving effectiveness comes from coverage, not aggressiveness.
• Local deductions applied many times are more valuable than rare deep inferences.
• Robustness and numerical stability are prioritized over maximal reduction.
• Presolving rules should generalize across diverse MILP instances, avoiding instance-specific logic.

Now, you should design your algorithm. First, You should describe your designed algorithm and main steps in two sentence. The description must be inside a brace.

Figure 6: Instruction for the presolving component design.

array operations", "avoid division by zero" and so on. This will assist the LLM in generating an algorithm that is both syntactically correct and effective.

We demonstrate a complete initialization prompt for the Generators of the three components in Fig. 10.

C.2 Initial Evolutionary Strategy Prompt

In this section, we present the initial Evolutionary Strategies used by the Generator for algorithm evolution. Note that the evolutionary strategies are also evolving in our framework through the Analyzer and Coordinator.

During evolution, the Generator generates an algorithm individual based on the task context, the current population, and the evolutionary strategy. Since MILP problems encompass a broad task scope and large data scales, the generalized strategies from [32] are not well-suited in this context. Considering the employed algorithm form, which can be divided into two steps: feature selection and arithmetic computation, we design initial evolutionary strategies for these two components accordingly.

We design five initial evolutionary strategies, categorized into two classes: **modification (m)** and **combination (c)**. Modification refers to randomly selecting a parent algorithm individual from the population and modifying it. A combination refers to randomly selecting several parent algorithm individuals from the population and performing recombination on them. The modification is further divided into three subcategories (m1, m2, m3), while the combination is divided into two subcategories (c1, c2).

The specific strategy prompts utilized in the population evolution are described as follows.

[Cut Selection – Design Guidance]

You are now designing an algorithm for cut selection.
Cut selection is a critical control mechanism that balances LP strengthening against numerical stability and computational overhead.

Successful cut selection heuristics are guided by the following principles:
- Preference for cuts that are strongly violated at the current relaxation while remaining numerically well-scaled.
- Penalization of dense cuts that significantly increase LP size and slow down reoptimization.
- Avoidance of highly parallel or redundant cuts, favoring diversity in geometric orientation.
- Explicit limitation of the number of selected cuts per round to prevent LP degradation.
- Consideration of historical effectiveness, where available, to downweight cuts with poor impact.

Design insights:
- The best cut is not necessarily the most violated one.
- A small, diverse, and stable cut set often outperforms aggressive cut addition.
- Multi-criteria reasoning consistently outperforms single-metric selection.
- Cut selection acts as a stabilizer of the branch-and-bound process, not merely an LP tightener.

Now, you should design your algorithm. First, You should describe your designed algorithm and main steps in two sentence. The description must be inside a brace.

Figure 7: Instruction for the cut selection component design.

m1: Attempt to select different features from the parent algorithm f with similar arithmetic operations.

m2: Attempt to select similar features from the parent algorithm f with different arithmetic operations.

m3: Attempt to modify the constant coefficients or add constant operations in the parent algorithm f.

c1: Combine all parent algorithms f_i, performing crossover and recombination of the extracted features, arithmetic operations, and coefficients.

c2: Combine all parent algorithms f_i while avoiding the use of identical features, arithmetic operations, and coefficients from them.

We select the strategy c1 as an example and demonstrate complete evolutionary prompt for the Generators of the three components in Fig. 11.

C.3 Prompt for Analyzer

The Analyzer is prompted to perform a comparative, causal analysis of algorithm performance. Its input consists of a pair of algorithms, one high-performing (randomly selected in the top-3 population) and one low-performing (randomly selected in the last-3 population), evaluated on the same benchmark instances. The prompt instructs the Analyzer to: 1) identify the key structural or functional differences between the two algorithms; 2) hypothesize how these differences lead to the observed performance gap; and 3) propose a concrete evolutionary strategy that can steer subsequent generations toward the higher-performing design.

We instruct the Analyzer to output a structured natural-language report that explicates the cause of the performance differential and supplies actionable evolutionary guidance, which is then forwarded to the Coordinator for synergy-aware integration. We take the branching variable selection component as an example and present the prompt for the Analyzer in Fig. 12.

[Branching Variable Selection – Design Guidance]

You are now designing an algorithm for branching variable selection.
High-quality branching decisions rely on carefully designed heuristics rather than purely greedy rules.

Common characteristics of effective branching heuristics include:
• Combination of fractional information with long-term impact estimates, such as pseudocosts.
• Gradual transition from exploration-oriented criteria at early nodes to exploitation-oriented criteria deeper in the search tree.
• Use of reliability mechanisms to mitigate noise in early pseudocost estimates.
• Penalization of variables with unstable or highly asymmetric branching behavior.
• Preference for variables that lead to balanced and informative subproblems.

Design insights:
• Pure fractionality-based branching is insufficient for difficult instances.
• Strong branching ideas are approximated through cheap historical statistics.
• Robust branching prioritizes consistency over occasional large improvements.
• Effective branching heuristics implicitly trade off search depth against bound improvement.

Now, you should design your algorithm. First, You should describe your designed algorithm and main steps in two sentence. The description must be inside a brace.

Figure 8: Instruction for the branching component design.

C.4 Prompt for Coordinator

The Coordinator is prompted to synthesize the component-specific analyses and produce a unified evolutionary direction that exploits inter-component synergies. Its input consists of: 1) the analysis reports from the Analyzer of each component; 2) the current high-performing algorithm code for each component. Then, the prompt instructs the Coordinator to: 1) integrate the separate analyses by summarizing the key strengths and weaknesses identified for each component; 2) evaluate potential interactions between components—e.g., how a more aggressive presolving strategy could affect the feature distribution for branching, or how cut selection influences the LP relaxations used in branching decisions; 3) formulate a coherent set of evolutionary strategies, one per component, that not only address each component's individual shortcomings but also promote cooperative improvements across the solver.

C.5 Prompt for Controller

The function of the Controller is to generate the depth partition point d and the population selection array $retained$ for the branching variable selection component. We provide it with the performance values of each algorithm individual in the population at each depth during the branch-and-bound process, as well as the performance values of a high-performing baseline algorithm (RPB) at each depth, allowing it to evaluate the performance of different algorithms at each depth. Based on this evaluation, the agent determines the most appropriate depth partition point and discards the algorithms that perform poorly within this depth range. For specific prompt designs, please refer to Fig. 13.

D Details on Involved Features

The adopted features comprise constraint features, edge features, and variable features. Constraint features include the `bias` of each constraint, while edge features represent the `constraint matrix` of the instance. Variable features are comparatively information-rich; following [27], for each variable, we employ 91 features, including both static features [17] describing the input problem and

Input-Output Information for Presolving

Input:
'constraint_features': a dictionary of type {'names': list, 'values': np.ndarray}, the features associated with the constraint nodes in the bipartite graph.

'edge_features': a dictionary of type {'names': list, 'indices': np.ndarray, 'values': np.ndarray}, the features associated with the edges in the bipartite graph (given as a sparse matrix in COO format).

'variable_features': a dictionary of type {'names': list, 'values': np.ndarray}, the features associated with the variable nodes in the bipartite graph.

Output:
'presolve_params': a list of dictionary {'priority': int, 'max-rounds': int, 'timing': int} involving hyperparameters for the presolvers; the length is 14, corresponding to 14 presolvers; the 'priority' value should be different with each other.

The meanings of the various features in 'variable_features' are as follows:
'type_0', 'type_1', 'type_2', 'type_3': Variable type;
'coef_normalized': Objective coefficient;
.....
'slack_distance', 'ceil_distance': Slack and ceil distances.

The corresponding 'names' for each index are as follows:
0 type_0
1 type_1
.....
90 ceil_distance

The description and indices of the presolvers are as follows:
1. Boundshift: Maps variables from the domain [a, b] to the domain [0, b-a].
.....

Input-Output Information for Presolving

Input:
'constraint_features': a dictionary of type {'names': list, 'values': np.ndarray}, the features associated with the constraint nodes in the bipartite graph.

'edge_features': a dictionary of type {'names': list, 'indices': np.ndarray, 'values': np.ndarray}, the features associated with the edges in the bipartite graph (given as a sparse matrix in COO format).

'variable_features': a dictionary of type {'names': list, 'values': np.ndarray}, the features associated with the variable nodes in the bipartite graph.

'cuts': a list of scip Rows; the cuts which we want to select from.

'cut_features': a np.ndarray, the features associated with each candidate cut.

Output:
'sorted_cuts': the sorted input 'cuts', the rank denotes priority.

The meanings of the various features in 'variable_features' are as follows:
'type_0', 'type_1', 'type_2', 'type_3': Variable type;
'coef_normalized': Objective coefficient;
.....
'slack_distance', 'ceil_distance': Slack and ceil distances.

The corresponding 'names' for each index are as follows:
0 type_0
1 type_1
.....
90 ceil_distance

The meanings of each index in 'cut_features' are as follows:
0: Constant shift of the cut.
.....
14: Whether the cut is removable from the LP.

Input-Output Information for Branching

'constraint_features': a dictionary of type {'names': list, 'values': np.ndarray}, the features associated with the constraint nodes in the bipartite graph.

'edge_features': a dictionary of type {'names': list, 'indices': np.ndarray, 'values': np.ndarray}, the features associated with the edges in the bipartite graph (given as a sparse matrix in COO format).

'variable_features': a dictionary of type {'names': list, 'values': np.ndarray}, the features associated with the variable nodes in the bipartite graph.

'scores': a list of type float, the scores for all nodes within the range [0, 1] to represent the probability of that node becoming the next branch node, whose length is equal to variable_features['values'].shape[0].

The meanings of the various features in 'variable_features' are as follows:
'type_0', 'type_1', 'type_2', 'type_3': Variable type;
'coef_normalized': Objective coefficient;
.....
'slack_distance', 'ceil_distance': Slack and ceil distances.

The corresponding 'names' for each index are as follows:
0 type_0
1 type_1
.....
90 ceil_distance

Figure 9: Input-output information for three components.

Initialization for Presolving
<Problem Description>

<Design Guidance>

<Code Format Description>
Then, Implement your designed algorithm as a python function named calculate_presolver_params. This function should accept 3 inputs: 'constraint_features', 'edge_features', 'variable_features'. The function should return 1 output: 'presolve_params'.

<Input-Output Information>

<Additional Details>
All are Numpy arrays. Please note that the shapes of the arrays need to match for the operation, and the indices need to be within the legal range. Please be careful to avoid division by zero. Please use more addition and multiplication. Do not give additional explanations.

Initialization for Cut Selection
<Problem Description>

<Design Guidance>

<Code Format Description>
Then, Implement your designed algorithm as a python function named calculate_cut_priority. This function should accept 5 inputs: 'constraint_features', 'edge_features', 'variable_features', 'cuts', 'cut_features'. The function should return 1 output: 'sorted_cuts'.

<Input-Output Information>

<Additional Details>
All are Numpy arrays. Please note that the shapes of the arrays need to match for the operation, and the indices need to be within the legal range. Please be careful to avoid division by zero. Please use more addition and multiplication. Do not give additional explanations.

Initialization for Branching Variable Selection
<Problem Description>

<Design Guidance>

<Code Format Description>
Then, Implement your designed algorithm as a python function named calculate_candidates_scores. This function should accept 3 inputs: 'constraint_features', 'edge_features', 'variable_features'. The function should return 1 output: 'scores'.

<Input-Output Information>

<Additional Details>
All are Numpy arrays. Please note that the shapes of the arrays need to match for the operation, and the indices need to be within the legal range. Please be careful to avoid division by zero. Please use more addition and multiplication. Do not give additional explanations.

Figure 10: Initialization prompt for the Generators of the three components.

dynamic features [25] reflecting the solving status at the current solving state. The descriptions of these 91 variable features are provided in Table 8.

These features represent key information about the variables in an optimization problem, particularly within the context of the branch-and-bound algorithm. Below, we provide some detailed descriptions of critical features:

- `'type_0'`, `'type_1'`, `'type_2'`, `'type_3'`: These features represent the type of the variable, including binary, integer, continuous, and implied integer variables.
- `'coef_normalized'`: It provides a scaled value of the variable's objective function coefficient, which is used to compare the relative importance of variables after normalization.
- `'sol_val'` and `'sol_frac'`: The LP solution value `'sol_val'` represents the value of the variable in the solution obtained from the linear programming relaxation. `'sol_frac'`

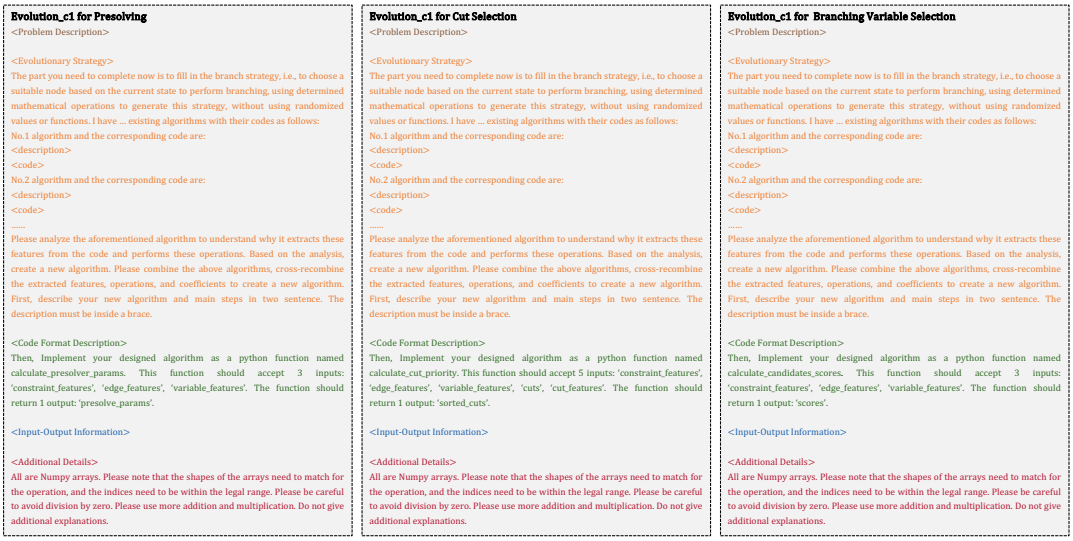


Figure 11: Evolutionary prompt for the Generators of the three components.

refers to the fractional part of the solution value, which is particularly relevant for integer variables in mixed-integer linear programming (MILP) models.

- `'basis_status_0'`, `'basis_status_1'`, `'basis_status_2'`, `'basis_status_3'`: These features refer to the basis status of the variable, which indicates whether the variable is currently part of the basis in the simplex method. The different indices represent different states within the simplex tableau, where `'basis_status_0'` represents that the variable is at its lower bound, `'basis_status_1'` represents that the variable is basic, `'basis_status_2'` represents that the variable is at its upper bound, `'basis_status_3'` represents that the free variable is non-basic and set to zero.
- `'reduced_cost'`: It indicates the cost of entering the basis in the simplex algorithm. If the reduced cost is negative for a maximization problem, the variable is eligible to enter the basis.
- `'inc_val'` and `'avg_inc_val'`: `'inc_val'` represents the best solution found so far. `'avg_inc_val'` indicates the average incumbent solution value, helping assess the quality of the current incumbent relative to past values.
- `'act_cons_coef_0'` to `'act_cons_coef_23'`: These features correspond to the coefficients of active constraints. The indices represent individual coefficients for each active constraint.
- `'dyn_cons_degree_0'` to `'dyn_cons_degree_6'`: A dynamic variant of statistics for constraint degrees, these features describe the degree (or complexity) of constraints that may change during the branch-and-bound process as nodes are explored.
- `'obj_func_coef_0'` to `'obj_func_coef_2'`: These represent the coefficients of the objective function for the variable, relevant for understanding how the variable contributes to the overall optimization goal.
- `'infeasibility_0'` to `'infeasibility_3'`: These features provide infeasibility statistics, which indicate how far the current variable's values are from satisfying the feasibility conditions of the optimization problem.
- `'coef_ratio_0'` to `'coef_ratio_7'`: These features represent the min/max ratios of coefficients in one-to-all comparisons. They help identify the relative magnitude of different coefficients in relation to each other, providing insights into the constraint's relative impact.
- `'pseudocost_0'` to `'pseudocost_4'`: These are the pseudocosts associated with the variable, often used in column generation and branch-and-bound methods to estimate the reduced costs of variables that are not currently in the basis.

Prompt for Analyzer (taking the branching variable selection component as an example)

<Task Description>
You are an expert algorithm analyst for MILP solvers. Your task is to diagnose the performance differences between two algorithms for the branching variable selection component and propose an evolutionary strategy for improvement.

<Input>
1. Algorithm A (High-Performing)
<Description and Code of Algorithm A>
2. Algorithm B (Low-Performing)
< Description and Code of Algorithm B>

<Instruction>
Perform a step-by-step comparative analysis:
1. Identify Structural Differences: Compare the formulas of Algorithm A and Algorithm B. List the key differences in their use of features and coefficients.
2. Hypothesize Causal Link: Explain how the specific differences identified in Step 1 likely lead to the observed gap in solving time and node count. Why does Algorithm A's formulation appear more effective?
3. Propose Evolutionary Strategy: Based on your analysis, suggest a concrete mutation-type evolutionary strategy in a paragraph. Your suggestion should aim to produce new algorithms that embody the strengths of Algorithm A. The evolutionary strategy should be concise, and the Generator can generate new algorithms given a parent algorithm and your evolutionary strategy.

<Output Format>
Provide a concise analysis report in the following structure:
- Differences:
- Causal Analysis:
- Proposed Evolutionary Strategy:
Do not give additional explanations.

Figure 12: Prompt for the Analyzer of the branching variable selection component.

- `'slack_distance'`, `'ceil_distance'`: `'slack_distance'` refers to the difference between the current value of the constraint and its slack (the remaining capacity). `'ceil_distance'` refers to the difference between the current value and the upper bound (ceiling) of a variable or constraint.

Note that during the presolving phase, since the problem has not yet begun to be solved, dynamic features that reflect the solving status at the current solving state cannot be obtained and are therefore set to zero.

E Details on Algorithm Individual and Evolutionary Strategy

Algorithm Individual. We define an algorithm individual as a combination of an algorithm description in the form of natural language, a segment of code, and a fitness value. The algorithm description summarizes the key thoughts and principles of the designed algorithm. The code specifies the detailed steps for executing the algorithm, and the fitness function value serves as a measure of performance.

For MILP problems, we optimize the components by designing a targeted algorithm form. Inspired by symbolic-based approaches, we extract the normalized features of variables, constraints, and edges from the solver. The LLM is required to select a subset of the features and apply basic arithmetic operations based on their meaning, thereby avoiding extensive and complex mathematical computations.

Evolutionary Strategy. Each evolutionary strategy consists of a natural-language instruction that guides the Generator in producing improved algorithm variants, along with an effectiveness metric that reflects the potential of the strategy. The effectiveness is defined by the performance of the best algorithm generated under that strategy. This metric is used to dynamically manage the evolutionary-strategy population: when a newly generated strategy exhibits higher effectiveness, it replaces the existing strategy with the lowest effectiveness in the population. This mechanism ensures that the search process continuously prioritizes the most promising evolutionary directions. Note that

You are an expert in Operations Research and Optimization, currently researching MILP and aiming to accelerate the solving process. Your approach is to optimize the Branching strategies used in MILP solvers by identifying better strategies for different search tree depths during the solving process.

You will receive information about the performance of different Branching strategies at various search tree depths.

Input format:
'''
{
  'start_depth': d1, # Starting search tree depth
  'end_depth': d2, # Ending search tree depth
  'algorithms': [
    {
      'node_count_dict': {depth1:<int>, depth2:<int>, ...}, # Node count explored at each depth from d1 to d2, the key is the depth and the value is the node count
      'sol_time_dict': {depth1:<float>, depth2:<float>, ...}, # Relative solving time used at each depth from d1 to d2, the key is the depth and the value is the solving time
      'total_sol_time': <float>, # Total time used to solve the problem
      'total_node_count': <float>, # Total number of nodes needed to solve the problem.
    },
    ...
  ],
  'baseline': { # Baseline algorithm for comparison and reference
    'node_count_dict': {depth1:<int>, depth2:<int>, ...},
    'sol_time_dict': {depth1:<float>, depth2:<float>, ...},
    'total_sol_time': <float>,
    'total_node_count': <float>,
  }
}
'''

Your task is to determine an optimal depth threshold `d` (d1 < d <= d2), where we will focus on optimizing the Branching strategy from depth `d1` to `d`. The value of d must satisfy d1 < d <= d2, where d1 and d2 correspond to the 'start_depth' and 'end_depth' in the input data. Additionally, decide which algorithms should be kept as seed algorithms for further optimization (excluding the baseline algorithm) from depth `d1` to `d`.

For each algorithm, you will decide whether to retain it or not:
- 1 means retain the algorithm.
- 0 means discard the algorithm.

Output format:
'''
d = <integer> # Optimal depth threshold you decide.
retained = [1, 0, 1, ...] # List of algorithms to retain (1 for retain, 0 for discard).
                # Each position corresponds to the input algorithms, excluding the baseline.
'''

No comments or explanations are needed in the output.

Query:

Input:
<A dictionary structured according to the Input Format>
Output:

Figure 13: Detailed prompts provided to the Controller.

we allow the Analyzer and the Coordinator to generate only modification-type evolutionary strategies. Initial combination strategies are preserved in the evolutionary strategy population throughout the process. We provide examples of the generated evolutionary strategies in Fig. 14.

F Formal Description and General Workflow of Jolly-MILP

F.1 Formal Description of Jolly-MILP

For completeness, this section provides a formal characterization of the Jolly-MILP framework, complementing the high-level description in Sec. 2.

F.1.1 Notation

Let $\mathcal{M}$ denote a MILP problem instance and $\mathcal{D}$ a distribution of instances of interest. We consider three solver components indexed by $k \in \mathcal{K} \triangleq \{p, c, b\}$, corresponding to **p**resolving, **c**ut selection, and **b**ranching variable selection. For each component k, let

- $\mathcal{A}_k$: the population of algorithms for component k;

Table 8: Brief feature names and their corresponding meanings, which are also part of the input-output information provided to the Generator.

Name	Meaning
type_0, type_1, type_2, type_3	Variable type
coef_normalized	Objective coefficient
has_lb	Whether it has lower bound
has_ub	Whether it has upper bound
sol_is_at_lb	Whether the solution is at lower bound
sol_is_at_ub	Whether the solution is at upper bound
sol_val, sol_frac	LP solution value
basis_status_0 to basis_status_3	Basis status
reduced_cost	Reduced cost
age	Age
inc_val, avg_inc_val	Incumbent solution value
act_cons_coef_0 to act_cons_coef_23	Statistics for active constraint coefficients
dyn_cons_degree_0 to dyn_cons_degree_6	A dynamic variant of statistics for constraint degrees
obj_func_coef_0 to obj_func_coef_2	Objective function coefficients
infeasibility_0 to infeasibility_3	Infeasibility statistics
num_of_cons	Number of constraints
neg_rhs_ratio_min, neg_rhs_ratio_max	Min/max for ratios of constraint coefficients to RHS
coef_ratio_0 to coef_ratio_7	Min/max for one-to-all coefficient ratios
pos_rhs_ratio_min, pos_rhs_ratio_max	Min/max for ratios of constraint coefficients to RHS
pseudocost_0 to pseudocost_4	Pseudocosts
cons_degree_0 to cons_degree_3	Statistics for constraint degrees
cons_coef_0 to cons_coef_9	Statistics for constraint coefficients
slack_distance, ceil_distance	Slack and ceil distances

- $a_k \in \mathcal{A}_k$: a single algorithm individual, represented as a tuple $a_k = (\texttt{desc}_k, \texttt{code}_k, f_k)$, where $\texttt{desc}_k$ is a natural-language description, $\texttt{code}_k$ is an executable Python function, and $f_k \in \mathbb{R}$ is its fitness value;

- $\mathcal{S}_k$: the space of evolutionary strategies for component k;

- $s_k \in \mathcal{S}_k$: a strategy, also represented as $(\texttt{desc}_k^s, e_k)$ with effectiveness $e_k \in \mathbb{R}$.

A **solver configuration** is a tuple $\boldsymbol{a} = (a_p, a_c, a_b) \in \mathcal{A}_p \times \mathcal{A}_c \times \mathcal{A}_b$. Given $\boldsymbol{a}$, the integrated SCIP solver induces a runtime measure

$$T : \mathcal{A}_p \times \mathcal{A}_c \times \mathcal{A}_b \times \mathcal{D} \to \mathbb{R}_{\geq 0}, \tag{4}$$

where $T(\boldsymbol{a}; \mathcal{M})$ denotes the solving time (or PD-integral, on hard instances) of $\boldsymbol{a}$ on $\mathcal{M}$. We write the expected performance over a problem distribution as

$$F(\boldsymbol{a}) \triangleq \mathbb{E}_{\mathcal{M}\sim\mathcal{D}}\big[T(\boldsymbol{a}; \mathcal{M})\big]. \tag{5}$$

The overall optimization objective of Jolly-MILP is

$$\boldsymbol{a}^\star = \arg\min_{\boldsymbol{a}\in\mathcal{A}_p\times\mathcal{A}_c\times\mathcal{A}_b} F(\boldsymbol{a}). \tag{6}$$

F.1.2 Agents as Stochastic Maps

Each agent in Jolly-MILP is modelled as a (stochastic) map between structured spaces, parameterized by an underlying LLM π_θ.

Generator. For component k, given a parent set $\mathcal{A}_k^{\text{par}} \subseteq \mathcal{A}_k$ and a strategy s_k, the Generator G_k samples a new candidate algorithm:

$$G_k\big(\mathcal{A}_k^{\text{par}}, s_k; \pi_\theta\big) \sim q_{G_k}(\cdot \mid \mathcal{A}_k^{\text{par}}, s_k). \tag{7}$$

Analyzer. Given a high-performing algorithm a_k^+ and a low-performing algorithm a_k^- for the same component, the Analyzer produces a component-local evolutionary direction $\delta_k \in \mathcal{S}_k$:

$$\delta_k = \mathcal{N}_k\big(a_k^+, a_k^-; \pi_\theta\big). \tag{8}$$

Analyzer's Evolutionary Strategy (Presolving):
Starting from a parent algorithm, mutate by (i) replacing linear averaged features with thresholded ratios or indicator functions derived from the same raw inputs, (ii) amplifying score separation by increasing weights on binary-like, tight-domain, and singleton indicators while compressing weights on generic averages, and (iii) enforcing early-presolver caps via hard-coded round/timing ceilings for cheap routines.

Analyzer's Evolutionary Strategy (Branching):
Starting from a parent policy, randomly select one or two high-impact scalar features (e.g., slack distance, infeasibility, pseudocost) and mutate the formula by (i) inserting multiplicative or quadratic interactions with an existing score term, (ii) reusing a single feature as an amplifier at multiple stages, and (iii) adding small corrective subtractions using complementary features. Reject mutations that only add linear weighted sums or feature averaging.

Analyzer's Evolutionary Strategy (Cut):
When evolving cut selection algorithms, replace any global or weakly aggregated variable signal with a structured cut-variable interaction term that combines fractionality, proximity to tight constraints, and objective/dual impact; normalize this signal independently to preserve variance, and ensure it acts as a differentiating criterion rather than a scalar bias.

Figure 14: Examples of evolutionary strategies generated by the Analyzers.

Coordinator. Given the per-component directions and the current best algorithms, the Coordinator produces a synergy-aware strategy update for all components jointly:

$$(\hat{s}_p, \hat{s}_c, \hat{s}_b) \;=\; \mathcal{C}\big(\{\delta_k\}_{k\in\mathcal{K}},\, \{\hat{a}_k\}_{k\in\mathcal{K}};\, \pi_\theta\big), \tag{9}$$

where $\hat{a}_k$ is the current incumbent algorithm of component k. Crucially, $\mathcal{C}$ is **not** decomposable as $\hat{s}_k = \mathcal{C}_k(\delta_k)$; its output for any single component depends on the directions of all components, which is the key mechanism that distinguishes Jolly-MILP from a coordinate-descent baseline.

Controller. Given a depth interval $[d_1, d_2)$, the current branching algorithm population $\mathcal{A}_b$, and a strong baseline (RPB) profile, the Controller outputs a depth partition point $d^\star$ and a binary retention vector $\boldsymbol{r} \in \{0,1\}^{|\mathcal{A}_b|}$:

$$\big(d^\star, \boldsymbol{r}\big) \;=\; \mathcal{H}\big([d_1, d_2),\, \mathcal{A}_b,\, a_b^{\text{RPB}};\, \pi_\theta\big), \quad d_1 < d^\star \le d_2. \tag{10}$$

F.1.3 Template-based Fitness Evaluation

Jolly-MILP maintains a **test template** $\mathcal{T} = (\hat{a}_p, \hat{a}_c, \hat{a}_b)$, holding the incumbent algorithm of each component. The fitness assigned to a candidate $a_k \in \mathcal{A}_k$ is defined by **partial substitution**:

$$\widetilde{f}_k(a_k \mid \mathcal{T}) \;\triangleq\; F\big(a_k,\, \hat{a}_{-k}\big), \tag{11}$$

where $\hat{a}_{-k}$ denotes the other two incumbents in $\mathcal{T}$. After all candidates of all components are evaluated in iteration t, the template is updated to the best observed combination:

$$\mathcal{T}^{(t+1)} \;=\; \arg \min_{\boldsymbol{a}\in\mathcal{A}_p^{(t)}\times\mathcal{A}_c^{(t)}\times\mathcal{A}_b^{(t)}} F(\boldsymbol{a}), \tag{12}$$

restricted (for tractability) to combinations that differ from $\mathcal{T}^{(t)}$ in at most one component, i.e. $\|\mathbf{1}[\boldsymbol{a} \neq \mathcal{T}^{(t)}]\|_0 \le 1$.

Remark (relation to coordinate descent). Eq. 12 coincides with one cycle of block-coordinate descent on F over the discrete blocks $(\mathcal{A}_p, \mathcal{A}_c, \mathcal{A}_b)$. However, the Coordinator (Eq. 9) injects coupling at the strategy level $\hat{s}_k$, which in turn shapes the proposal distribution q_{G_k} in Eq. 7. As a result, even though the evaluation step is single-component, the generation step is conditioned on the full joint context $\{\delta_j\}_{j\neq k}$. This circumvents the standard coordinate-descent failure mode of being trapped at non-stationary points caused by inter-block coupling, and is empirically validated against full joint sampling in Appendix J.5.

Synopsis of Component States:

Presolve.

Main strength: Presolve is highly effective at early, low-cost model reduction by using discretized, structure-aware ratios (binary-like, tight-domain, singleton) and hard score separation, which reliably triggers cheap, high-impact presolvers aligned with MIS structure.

Main limitation: The strategy is mostly static and structure-driven; it does not explicitly adapt its reductions to preserve signals that downstream branching and cut selection rely on.

Cut Selection.

Main strength: Cut selection uses structured cut–variable interaction terms (fractionality × slack proximity × coefficient and reduced-cost impact), normalized independently, which implicitly look ahead to downstream branching quality while maintaining sparsity and diversity.

Main limitation: Variable-aware signals are still globally aggregated and not explicitly conditioned on how presolve reshapes the model or which variables branching is likely to emphasize.

Branching.

Main strength: Branching achieves strong variable discrimination through guided nonlinear compositions, repeatedly amplifying a few high-impact features (slack distance, infeasibility, pseudocost) and creating large score separation that shapes the B&B tree efficiently.

Main limitation: The nonlinear pipeline can over-amplify noisy or poorly prepared LP states if upstream presolve and cut selection do not stabilize or enrich the right variables.

Critical Inter-Component Dependencies:

Structure-aware presolve simplifies the model and tightens domains, which stabilizes LP solutions and makes branching's nonlinear feature amplification more reliable. However, excessive elimination of moderately fractional or medium-degree variables can reduce the pool of informative variables for both cut generation and branching. Variable-aware cuts directly strengthen constraints around fractional, high-impact variables, sharpening slack- and infeasibility-based branching features, but if cuts become too dense or numerous they slow LP solves and weaken branching feedback loops. Branching's heavy reuse of slack distance and pseudocost magnifies whatever structure remains after presolve and cutting; thus, presolve must preserve "branch-relevant" variables, and cut selection must bias toward variables that branching can exploit.

Coordinated Evolutionary Directions:

For Presolve Generator:

Evolve presolve policies by systematically mutating linear or averaged structural features into thresholded ratios and indicator gates (e.g., binary-like ratios, tight-domain tests, singleton-degree indicators), then sharpening presolver ranking via discrete weight bands and hard caps on rounds and timing for cheap routines. Selection favors offspring that maximize early variable/constraint elimination while explicitly preserving a diverse set of fractional, nontrivial variables needed by cut selection and branching.

For Cut Selection Generator:

Evolve cut scoring by replacing any global or weak variable term with explicit cut–variable interaction signals combining fractionality, proximity to tightness, and objective/dual impact, independently normalized to preserve variance, and by mutating structural penalties into interacting sparsity–age–locality terms. Selection favors offspring that improve branching discrimination and LP stability simultaneously, avoiding dense or numerically harmful cuts.

For Branching Generator:

Evolve branching rules through guided nonlinear mutations that reuse 1–2 core state-dependent features (slack distance, infeasibility, pseudocost) as multiplicative or quadratic amplifiers across multiple stages, with small asymmetric corrective terms, while rejecting purely linear or averaging mutations. Selection favors offspring that increase score separation after presolve and cuts, producing shallower trees and faster bound tightening.

Figure 15: An example of the Coordinator's output.

F.1.4 Hierarchical Composition for Branching

The Controller induces a piecewise branching policy over the search tree. Let $0 = d_0 < d_1^\star < d_2^\star < \cdots < d_M^\star = d_{\max}$ be the partition produced by successive Controller invocations, and let $\hat{a}_b^{(m)}$ be the branching algorithm fixed for interval $[d_{m-1}^\star, d_m^\star)$. The composite branching policy applied at a node of depth d is then

$$\Pi_b(d) \;=\; \sum_{m=1}^{M} \hat{a}_b^{(m)} \cdot \mathbf{1}\big[d \in [d_{m-1}^\star, d_m^\star)\big]. \tag{13}$$

After fixing $\hat{a}_b^{(m)}$ for $[d_{m-1}^\star, d_m^\star)$, the framework re-initializes a fresh population for $[d_m^\star, d_{\max})$ and repeats Eqs. 7–12, restricted to the branching component. This construction yields the multi-stage, depth-aware behavior motivated in Sec. 2.3.

F.2 General Workflow of Jolly-MILP

Overall Algorithm. Jolly-MILP dynamically maintains two core data structures per component: an algorithm population $\mathcal{A}_i$, and an evolutionary-strategy population $\mathcal{E}_i$; and a test template $\mathcal{T} =< \hat{a_p}, \hat{a_b}, \hat{a_c} >$ that holds the current best algorithm combination. For the branching component, we also track the current depth interval $[d_1, d_2)$ under optimization and a set $\mathcal{B}$ of fixed branching policies for previously partitioned depth ranges.

Initialization. All algorithm populations are empty. Each evolutionary-strategy population $\mathcal{E}_i$ is seeded with general-purpose initial prompts (specified in Appendix C.2). The test template is

initialized with the solver's default configuration. The depth interval is set to $[0, d_{\max}]$, where $d_{\max}$ is a predefined maximum depth, and $\mathcal{B} = \emptyset$. In the first step, each Generator uses a component-specific initialization prompt (specified in Appendix C.1) to produce the initial algorithm individuals, populating $\mathcal{A}_p$, $\mathcal{A}_b$, and $\mathcal{A}_c$.

Main Evolution Loop. The core process then proceeds iteratively, following the key steps outlined in Section 2.2. In each iteration:

1. New algorithms and evolutionary strategies are generated through multi-agent collaboration.
2. Algorithms are evaluated using the scheme described in Section 2.4. The best-performing algorithms for each component are inserted into the corresponding population $\mathcal{A}_i$, replacing the worst-performing members.
3. New evolutionary strategies replace the least effective ones in $\mathcal{E}_i$, where effectiveness is measured by the performance of the best algorithm they produce.
4. The test template $\mathcal{T}$ is updated to reflect the current best combination of algorithms across all components. For example, if the combination $< \tilde{a_p}, \hat{a_b}, \hat{a_c} >$ achieves the best overall performance, then $\hat{a_p}$ in $\mathcal{T}$ is updated to $\tilde{a_p}$.

Hierarchical Control Trigger. The performance of the integrated solver is monitored continuously during the evolution. If no significant improvement in the solver performance is observed over several consecutive iterations, or the maximum number of iterations is reached, the hierarchical process (Section 2.3) is triggered. The Controller analyzes performance data and determines a partition depth $d^\star (d_1 < d^\star \leq d_2)$, selecting the most promising strategies to retain in $\mathcal{A}_b$. The best-performing branching strategy $\hat{a_b}$ for the interval $[d_1, d^\star)$ is fixed and added to the policy set: $\mathcal{B} = \mathcal{B} \cup \{[d_1, d) : \hat{a_b}\}$. In subsequent evaluations, the branching strategy in $\mathcal{T}$ is fixed for depths $[d_1, d^\star)$. The active depth interval is updated by $d_1 \leftarrow d^\star$, and evolution continues for $[d_1, d_2)$.

Termination. After a predefined number of depth partitions have been performed, final rounds of evolution are executed for the remaining depth interval up to d_2. The process stops when the overall solver performance converges or the maximum number of iterations is reached.

G Details on Algorithmic Process of Jolly-MILP

To provide a comprehensive and detailed exposition of the algorithmic workflow of Jolly-MILP, we present its core procedure in Algorithm 1. The algorithm outlines the end-to-end process, integrating joint component evolution with hierarchical control. Below, we provide an in-depth, step-by-step explanation of its implementation details, parameter settings, and underlying mechanisms.

- **Data Structures and Key Variables.** The algorithm maintains several core data structures and parameters throughout its execution:
  - Three **algorithm populations** $\mathcal{A}_p$, $\mathcal{A}_b$, and $\mathcal{A}_c$ for the presolving, branching, and cut selection components, each of a fixed size N^A.
  - Three **evolutionary strategy populations** $\mathcal{E}_p$, $\mathcal{E}_b$, and $\mathcal{E}_c$, which store textual prompts that guide the evolution for each component, each of size N^E.
  - The **test template** $\mathcal{T} = \langle \hat{a}_p, \hat{a}_b, \hat{a}_c \rangle$, a triple holding the current best-performing integrated algorithms involving each component. This template defines the baseline solver configuration used for efficient fitness evaluation.
  - The current **depth interval** $[d_1, d_2)$ within the branch-and-bound tree for which the branching strategy is actively being evolved.
  - The set of **fixed branching policies**, which is built incrementally as the hierarchical control partitions the depth of the tree. Once a policy is fixed for an interval, it is removed from the active evolutionary process.

  Key hyperparameters include the number of offspring per strategy N^O, the total number of depth partitions T, the maximum number of evolutionary rounds per interval M, a convergence tolerance m_{tol}, and the maximum tree depth $d_{\max}$.
- **Initialization (Lines 2-5).** The process begins by initializing the three algorithm populations. Each population is filled by querying the corresponding Generator with the designed initialization prompt, generating an initial set of candidate algorithms. The fitness of each algorithm is evaluated by integrating it into a solver while using default algorithms for the

other two components. The test template $\mathcal{T}$ is initially set to the solver's default configuration. The evolutionary strategy populations are initialized with the manually designed, general-purpose evolutionary prompts. Finally, the active depth interval is set to span the entire possible tree, $[0, d_{\max}]$.

- **Main Loop: Hierarchical Depth Partitioning (Lines 6-28).** The outer loop iterates T times, each representing a major phase of hierarchical control that partitions the branch-and-bound tree. The value of T is a trade-off between performance and evolutionary time.
- **Inner Loop: Co-evolution within a Depth Interval (Lines 8-23).** For the current depth interval $[d_1, d_2)$, an inner loop runs up to M rounds or until convergence. Convergence is tracked by $m_{\text{no_improve}}$, which increments if no improvement in the best-found fitness occurs in a round and triggers an early exit if it exceeds m_{tol}.

    1. **Offspring Generation (Line 9):** In each round, every component's evolutionary process is advanced. For each of the N^E strategies in $\mathcal{E}_i$, the corresponding Generator is prompted to produce N^O new "offspring" algorithms, resulting in a total of $N^E \times N^O$ new candidate algorithms per component. This design allows the parallel exploration of diverse evolutionary directions.
    2. **Joint Evaluation and Template Update (Lines 10-11):** The fitness of each new algorithm is evaluated following the efficient scheme described in Section 2.4. An algorithm for component i is evaluated by integrating it into the current test template $\mathcal{T}$, replacing only $\hat{a}_i$, while the algorithms for the other two components remain fixed. This measures its contribution within the fully integrated solver at a cost linear in the number of candidates. After all evaluations, $\mathcal{T}$ is updated to the combination (one algorithm per component) that achieved the best overall performance in this round.
    3. **Population and Strategy Refinement (Lines 12-15):** The algorithm populations for presolving ($\mathcal{A}_p$) and cut selection ($\mathcal{A}_c$) are updated by replacing their worst-performing members with the best offspring from the current round. The branching population $\mathcal{A}_b$ is updated separately within this inner loop (line 22). The Analyzer for each component then compares high and low-performing algorithms to diagnose performance differences and suggest new evolutionary strategies specific to that component. The Coordinator synthesizes these three separate suggestions, explicitly models potential interactions between components, and outputs a revised, unified set of evolutionary strategies. These new strategies replace the least effective ones in the evolutionary strategy populations $\mathcal{E}_i$.
    4. **Convergence Check and Control Trigger (Lines 16-21):** If the fitness of the best algorithm in the template $\mathcal{T}$ does not improve, the counter $m_{\text{no_improve}}$ is incremented. If this counter exceeds the tolerance m_{tol} or the loop reaches its maximum iteration M, the inner loop breaks, and control passes to the hierarchical partitioning step.

- **Hierarchical Control: Depth Partition and Policy Fixing (Lines 24-28).** Upon exiting the inner loop, the Controller is invoked. Its inputs are the current branching population $\mathcal{A}_b$, performance metrics (solving time/PD integral and node count) broken down by depth within $[d_1, d_2)$, and the strong baseline heuristic RPB. The Controller analyzes this data to determine an optimal partition depth $d^\star$ (where $d_1 < d^\star \leq d_2$) and selects a subset of the most promising branching strategies from $\mathcal{A}_b$ to retain for the next evolutionary phase. The best-performing strategy for the now-finalized interval $[d_1, d^\star)$ is fixed: it is set as the branching algorithm $\hat{a}_b$ in the test template $\mathcal{T}$ for that depth range and is added to the set of fixed policies. The active depth interval is then updated to $[d^\star, d_2)$ for the next partition cycle.
- **Finalization and Output.** After $T - 1$ partitions, the branch-and-bound tree is divided into T intervals, with fixed branching policies for the first $T - 1$ intervals. A final round of the inner evolutionary loop is executed for the last remaining interval $[d_1, d_2)$ to determine its optimal strategy. The entire process terminates when the solver's overall performance converges or a global computational budget is exhausted. The final output is the fully configured test template $\mathcal{T}$, which represents the set of three component algorithms (for presolving and cut selection) and the collection of depth-specific branching policies that together constitute the optimized solver.

Algorithm 1 Pseudocode of Jolly-MILP

Require: Algorithm population size N^A, evolutionary strategy population size N^E, number of offspring N^O, number of depth interval T, number of maximum rounds each depth interval M, maximum tolerance m_{tol}, maximum depth d_{max}.

Initialize three algorithm populations, each with size N^A, through three Generators using the initialization prompt.
Initialize the test template $\mathcal{T}$ to the default setting of the solver.
Evaluate the fitness value of each algorithm in the algorithm population.
Initialize three evolutionary strategy populations using our designed initial evolutionary strategies.
Set $d_1 = 0$, $d_2 = d_{\text{max}}$.
for $t = 1$ to T **do**
  Set $m_{\text{no_improve}} \leftarrow 0$
  for $m = 1$ to M **do**
    For the three populations, utilize N^E evolutionary strategies from the three evolutionary strategy populations to guide the Generators to generate three offspring pools with size $N^E \times N^O$.
    Evaluate the fitness value of each offspring using the test template.
    Update the test template $\mathcal{T}$ based on the fitness values.
    Update the presolving algorithm population and the cut selection algorithm population.
    Leverage the Analyzers to suggest evolutionary strategies.
    Leverage the Coordinator to revise evolutionary strategies for each component.
    Update the three evolutionary strategy populations.
    if no further improvement **then**
      $m_{\text{no_improve}} += 1$
    end if
    if $m_{\text{no_improve}} >= m_{\text{tol}}$ or $m == M$ **then**
      Trigger the hierarchical control and break to the outer layer.
    end if
    Update the branching algorithm population.
  end for
  if $t < T$ **then**
    Input the branching population, depth-level performance, d_1, d_2 to the Controller, acquiring $d^\star$ and the retained branching algorithm population.
    Fix the branching strategy for $[d_1, d^\star)$ through updating the test template $\mathcal{T}$, and set $d_1 \leftarrow d^\star$.
  end if
end for
return The test template $\mathcal{T}$

This detailed process ensures that Jolly-MILP efficiently explores the joint space of solver components, leverages their synergies through coordinated strategy updates, and dynamically constructs high-performing, multi-stage branching strategies via hierarchical control, all while maintaining computational tractability.

H Details on Benchmarks

We conduct experiments on nine different MILP benchmarks, which can be divided into three categories based on their difficulties: easy, medium, and hard. We provide details on the benchmarks below.

Easy datasets. Three synthetic datasets are included in the easy datasets: Multiple Knapsack [39], Maximum Independent Set [8], and Set Covering [6]. They are all widely used benchmarks and represent broad practical applications. We list the specific parameters used to generate the three benchmarks in Table 9. Please refer to `https://github.com/ds4dm/learn2branch` for instance generation. For each benchmark, 10000 instances are generated as the training set, 1000 for validation, and 100 for testing. Note that Maximum Independent Set and Set Covering are also utilized to investigate the generalization ability, and we generate 100 medium and large-scale instances for each benchmark. Moreover, synthetic datasets show high variance, indicating a high proportion of trivial

Table 9: The parameters utilized to generate MILP benchmarks.

Task	Easy	Medium	Large
Set Covering	500rows, 1000cols	1000rows, 1000cols	2000rows, 1000cols
Maximum Independent Set	500nodes	1000nodes	1500nodes
Multiple Knapsack	60items, 12knapsacks	/	/

instances. Since extremely simple instances yield poor signals for training ML models, we filter out instances with fewer than 10 tree nodes.

Medium datasets. The medium datasets are comprised of two benchmarks: CORLAT [18] and MIK [5]. MIK is a collection of mixed-integer linear programming problems featuring knapsack constraints. CORLAT is a real-world dataset originally developed to plan a wildlife corridor for grizzly bears in the Northern Rockies. We follow the provided train-test split, resulting in 80% instances for training, and 20% for testing. Please refer to https://atamturk.ieor.berkeley.edu/data/mixed.integer.knapsack/ for MIK and https://bitbucket.org/mlindauer/aclib2/src/master/ for CORLAT.

Hard datasets. We adopt three datasets from the NeurIPS 2021 ML4CO competition [16]: Item Placement, Load Balancing, and Anonymous; and a subset of MIRPLIB, denoted as MIRP. For the ML4CO benchmarks, we split the instances provided by their original benchmarks. Please refer to https://www.ecole.ai/2021/ml4co-competition/ for details. For MIRP, we select the instances and conduct the train-test split following [30]. These benchmarks are all challenging, representing complex real-world applications.

Table 10: Hyperparameters of Jolly-MILP.

Parameter	Value
LLM Backbone	DeepSeek-V3
Algorithm Population Size	10
Evolutionary Strategy Population Size	5
Number of Offspring	5
Number of Depth Interval	3
Max Rounds Each Depth Interval	15
Max Tolerance	3
Evaluation Time Limit (s)	300

I Details on Experimental Setup

Jolly-MILP. The adopted hyperparameters of Jolly-MILP are presented in Table 10. The fitness is evaluated on 10 distinct instances, ensuring no overlap with the test set. To mitigate excessively long runtimes from poorly performing algorithms, evaluation time limits are imposed on the solver, thereby bounding the total time required for fitness computation within a controllable budget. In our experiments, the complete evolution time for the three components ranges from 1 to 20 hours, depending on the hardness of the evaluated problems, a duration comparable to the GPU-based training time of the three ML components alone, despite excluding the overhead for label collection.

ML Baselines. ML baselines incorporate three categories, including L2P for presolving, L2C and HEM for cut selection, GCNN and Symb4CO for branching variable selection. For each method, we train the model in line with the settings in the original paper [30, 43, 46, 17, 27]. Note that for branching ML baselines, we generate a unified number of training samples on each dataset using the full strong branching (FSB) heuristic, i.e., 100000 samples for GCNN and 1000 samples for Symb4CO, in line with the original papers [17, 27]. To select the best policy for Symb4CO, the top ten symbolic policies based on imitation accuracy are executed on 20 validation instances. For the training of other ML baselines, since the datasets vary in size, there is no uniform amount of training data used. Their training settings remain consistent with those described in the original papers and will not be elaborated further here. We carefully train the ML baselines and meticulously integrate them into the backbone SCIP solver.

SMAC3. SMAC3 [29] is a versatile and advanced software tool for algorithm configuration, hyperparameter optimization, and the broader domain of automated machine learning (AutoML). It implements state-of-the-art Bayesian optimization techniques, primarily leveraging Sequential Model-

Table 11: Description of involved presolvers.

Presolver	Description
Boundshift	Maps variables from the domain [a, b] to the domain [0, b-a].
Convertinttobin	Transforms integer variables defined on the domain [a, a+1] into binary variables.
Domcol	Identifies dominance relationships among variables and deduces a lower bound.
Dualagg	Consolidates variables that share a specific structural pattern.
Dualcomp	Assigns a fixed bound when the collective value of continuous variables can offset a requirement.
Dualinfer	Tightens the bounds imposed on continuous variables.
Gateextraction	Isolates gate constraints and set-partitioning constraints.
Implics	A presolver that utilizes an implication graph to identify opportunities for variable aggregation.
Inttobinary	Reformulates integer variables with the domain [a, a+1] as binary variables.
Milp	Invokes an external presolve library and applies the returned postsolve data.
Qpkktref	Attempts to augment the problem by introducing KKT conditions as supplementary constraints.
Redvub	Eliminates variable upper bound constraints that are logically redundant.
Stuffing	Examines singleton continuous variables to determine if they can be fixed at a bound.
Trivial	Sets variables to a specific value if their lower and upper bounds are identical.

Table 12: Critical parameters involved in the hyperparameter tuning by SMAC3.

Parameter	Description
separating/maxroundsroot	Maximal number of separation rounds in the root node.
separating/maxcutsroot	Maximal number of separated cuts per separation round at the root node.
separating/maxcutsrootgenfactor	Maximal number of generated cuts per separation round at the root node.
separating/minefficacyroot	Minimal efficacy for a cut to enter the LP in the root node.
cutselection/hybrid/priority	Priority of cut selection rule <hybrid>.
cutselection/ensemble/priority	Priority of cut selection rule <ensemble>.
cutselection/dynamic/priority	Priority of cut selection rule <dynamic>.
cutselection/hybrid/efficacyweight	Weight of efficacy in cut score calculation.
cutselection/hybrid/dircutoffdistweight	Weight of directed cutoff distance in cut score calculation.
cutselection/hybrid/minorthoroot	Minimal orthogonality for a cut to enter the LP in the root node.
branching/relpscost/conflictweight	Weight in score calculations for conflict score.
branching/relpscost/inferenceweight	Weight in score calculations for inference score.
branching/relpscost/minreliable	Minimal value for minimum pseudo cost size to regard pseudo cost value as reliable.
branching/relpscost/maxreliable	Maximal value for minimum pseudo cost size to regard pseudo cost value as reliable.
branching/relpscost/sbiterquot	Maximal fraction of strong branching LP iterations compared to node relaxation LP iterations.
branching/relpscost/maxbdchgs	Maximal number of bound tightenings before the node is reevaluated.
branching/relpscost/maxproprounds	Maximum number of propagation rounds to be performed during strong branching before solving the LP.
branching/relpscost/lowerrortol	Low relative error tolerance for reliability.
branching/relpscost/higherrortol	High relative error tolerance for reliability.
branching/relpscost/confidencelevel	The confidence level for statistical methods.

Table 13: Improvement on nine benchmarks. We run the experiments 5 times with different random seeds and report the average results with the standard deviations.

Method	MK	MIS	SC	Corlat	MIK	MIRP	IP	LB	AN
SMAC3	21.40%±1.78%	8.35%±0.76%	22.4%±0.54%	20.12%±1.22%	3.88%±0.27%	6.92%±0.65%	3.62%±0.31%	5.34%±0.17%	19.25%±0.94%
EoH	<0	<0	3.22%±1.22%	11.03%±1.98%	1.46%±0.34%	1.61%±0.93%	<0	1.29%±0.15%	<0
ReEvo	4.28%±3.60%	22.71%±1.42%	17.06%±1.05%	14.49%±1.62%	2.64%±0.56%	3.94%±0.82%	2.31%±0.44%	3.27%±0.09%	16.7%±1.25%
Integrated-ML (config1)	27.63%±1.43%	28.72%±1.20%	32.79%±0.49%	27.25%±2.14%	2.98%±0.35%	12.36%±1.08%	4.25%±0.28%	7.27%±0.12%	25.05%±0.86%
Integrated-ML (config2)	24.51%±1.61%	31.35%±0.97%	35.40%±0.52%	30.71%±1.06%	5.51%±0.43%	9.96%±0.98%	4.87%±0.39%	8.67%±0.21%	24.11%±1.16%
Jolly-MILP (Ours)	**33.46%±1.80%**	**33.68%±0.84%**	**35.70%±0.63%**	**33.99%±1.45%**	**7.21%±0.59%**	**18.15%±0.73%**	**11.01%±0.52%**	**10.03%±0.26%**	**26.53%±0.87%**

Based Optimization (SMBO). We leverage SMAC3 to tune the critical hyperparameters of the three components in SCIP. There are a total of 82 hyperparameters involved: 42 for presolving, 20 for cut selection, and 20 for branching variable selection. Table 11 demonstrates the descriptions of 14 different presolvers, where each presolver includes three parameters: priority, max-rounds, and timing. We also present the most critical hyperparameters involved for cut selection and branching in the hyperparameter-tuning process in Table 12. The hyperparameter search is allocated a time budget of 100 hours, a duration deemed sufficient for securing a well-tuned configuration.

LLM-based Baselines. For the EoH [32] and ReEvo [52] baselines, we carefully adapt the prompts to align with our specific tasks, aiming to ensure a fair comparative analysis. To this end, we maintain consistent experimental conditions: the population size is identical to our method, and the procedure for fitness evaluation remains the same. Also, the same LLM backbone is adopted. The evolution process for each component is capped at 40 iterations. In practice, however, we observe that performance typically converges with no distinct improvement beyond 20 iterations. Therefore, we ensure that LLM baselines **converge** under our experimental setting, and increasing computational budget does **not** result in significantly better performance.

All experiments are conducted on our workstation with AMD 3970X, RTX 3090, and 128GB memory.

Table 14: Improvement on the CORLAT dataset when integrated with different component combinations.

Method	P	B	C	P+B	P+C	B+C	P+B+C
ReEvo	12.34%	3.73%	11.68%	12.42%	14.20%	12.09%	14.49%
Integrated-ML (config2)	**23.61%**	**13.42%**	**25.27%**	25.36%	**29.23%**	**27.18%**	30.71%
Jolly-MILP (Ours)	19.55%	13.04%	18.79%	**26.18%**	27.41%	26.50%	**33.99%**

J More Experiments and Analysis

J.1 Standard Deviation Results.

We conduct the experiments 5 times with different random seeds. Table 13 reports the mean improvement in solving time and PD integral, together with the corresponding standard deviations. Notably, Jolly-MILP consistently outperforms both integrated ML-based methods and LLM-based approaches on the easy, medium, and hard datasets, demonstrating robust effectiveness across a wide range of problem difficulties. The relatively small variances indicate that the performance gains of Jolly-MILP are stable across runs and confirm that the observed improvements are not due to random fluctuations.

J.2 Different Component Combinations.

We further analyze the performance of integrating different combinations of solver components on the CORLAT dataset. The presolving, branching, and cut selection components are denoted by **P**, **B**, and **C**, respectively. The results are summarized in Table 14. We observe that integrating multiple components consistently leads to better performance than using any single component in isolation. Moreover, for a given method, combining with a component that performs well individually typically yields greater improvements than combining with a weaker component. However, when all components are integrated, diminishing returns are evident: the improvement achieved by jointly integrating three components is substantially smaller than the sum of the improvements obtained from each component individually. Notably, Jolly-MILP significantly alleviates this diminishing-gain effect. Specifically, while the ML-based baselines outperform Jolly-MILP when each of the three components is considered individually, the performance gap narrows once two components are integrated, with ML-based methods and Jolly-MILP achieving comparable results. When all three components are integrated, Jolly-MILP surpasses the ML-based approaches. This behavior can be attributed to the fact that our framework explicitly models and leverages the synergistic interactions among components: each component is aware of the algorithms used by the others and adapts accordingly during evolution. Furthermore, our single-component variants consistently outperform ReEvo, indicating that making information from other components available during the evolutionary process provides more informative search directions and facilitates the discovery of higher-quality algorithms. This result further supports the importance of joint optimization in enhancing overall solver performance.

J.3 Synergistic Interaction Analysis.

A core premise of the Jolly-MILP framework is that the performance of an MILP solver emerges not merely from the independent quality of its components, but from their synergistic interactions. Our experimental results provide strong empirical evidence for this hypothesis. In Fig. 15, we present an example of the Coordinator's output to further illustrate the mechanism of the synergistic interaction. The Coordinator highlights several causal chains that create tight coupling between the components:

Presolve→Branching/Cut Selection. Presolve determines the initial search landscape. Overly aggressive reduction, while beneficial for problem size, can prune the set of "informative" fractional variables, thereby starving both cut selection (reducing the pool of good cut candidates) and branching (weakening the discriminative power of its feature amplifiers).

Cut Selection→Branching. The cuts added directly alter the LP relaxation, changing variable fractionalities and constraint slacks. Well-chosen cuts sharpen the features (e.g., slack distance) that branching relies on, creating a positive feedback loop. Conversely, dense or numerically unstable cuts can degrade LP solve times and corrupt the feature space.

The Coordinator resolves these dependencies not by optimizing each component for its isolated task, but by steering their evolution toward mutually supportive configurations. For **presolve**, evolution is directed toward policies that balance aggressive reduction with the *preservation of branching-relevant variables*. This shifts the objective from mere constraint/variable elimination to *structure-aware filtering* that curates a high-quality input for downstream components. For **cut selection**, evolution is guided to prioritize cuts that improve *branching discrimination and LP stability simultaneously*. This moves beyond cut strength toward a holistic view of how each cut influences the subsequent search. For **branching**, evolution is focused on rules that effectively *amplify the feature signals prepared by the upstream components*.

In our experiments, when each component is integrated into the solver in isolation (with other components fixed to default), its performance is slightly inferior to specialized ML models. However, when all three components are jointly integrated, Jolly-MILP achieves better overall performance than any configuration of these individually stronger ML components. This performance inversion is attributed to positive synergistic interactions between the components.

Table 15: Improvement and Search Time (in brackets) on Item Placement.

Eval. Time Limit (s) \ Pop. Size	2	3	5	10
30	1.97% (0.4h)	2.86% (0.6h)	3.31% (1.1h)	4.52% (1.9h)
60	2.66% (0.8h)	3.95% (1.3h)	4.92% (2.0h)	6.13% (4.2h)
100	3.58% (1.4h)	4.36% (2.1h)	6.49% (3.6h)	9.86% (7.4h)
300	3.84% (4.2h)	5.28% (6.2h)	8.73% (9.7h)	11.01% (19.6h)

J.4 Further Discussion on Computational Cost

To provide a comprehensive understanding of the practical deployment of Jolly-MILP, we analyze the computational overhead from two dimensions: CPU compute and LLM API consumption. We focus on Item Placement, the most resource-intensive task in our experiments, to illustrate the upper bound of the costs involved.

Execution Efficiency and Hardware Requirements. The evolution process for Item Placement (which is hard and the **most resource-intensive task** in the experiments) requires approximately **19.6 hours**. It is important to emphasize that Jolly-MILP is CPU-only for both symbolic search and inference. This distinguishes Jolly-MILP from traditional deep learning-based approaches, which typically necessitate heavy GPU dependency. For comparison, training separate ML models for this task exceeds **25 hours on GPUs**, with over **100 hours** required for label collection on the same hardware as Jolly-MILP.

LLM API Consumption. Regarding token usage, the process consumes approximately 595k input tokens and 159k output tokens. The monetary cost varies depending on the choice of the backbone model. Using a SOTA model (e.g., GPT-5.4), the total cost is approximately $3.6. Utilizing a high-performance alternative like DeepSeek-V3.2, the cost is approximately $0.23. Moreover, under fixed algorithmic hyperparameters, these API costs remain relatively constant across any given dataset, providing a predictable budget for large-scale experiments.

Flexible Trade-off: Search Time vs. Performance. Furthermore, we conduct a cost-ablation study. We set different algorithmic hyperparameters, including **Evaluation Time Limit** and **Algorithm Population Size**, resulting in varying search times. We report the respective improvements in Table 15. While the full search yields peak performance, a notable improvement of 4.92% (comparable to the ML baseline) can be achieved in just 2.0 hours, highlighting the framework's ability to rapidly discover high-performing heuristics.

Table 16: Improvement and Search Time (in brackets) on Multiple Knapsack.

Updated Component Num.	1 (Ours)	2 (P+C)	2 (P+B)	2 (C+B)	3 (Full)
Improvement (Time)	33.46% (0.9h)	35.86% (10.6h)	34.51% (11.5h)	34.96% (11.2h)	36.38% (>60h)

J.5 Further Discussion on Evaluation Scheme

The adoption of a template-based evaluation scheme in Jolly-MILP attempts to provide a deliberate and strategic trade-off between computational tractability and the discovery of cross-component synergies.

To provide a better comprehension, we conduct an experiment on the evaluation strategy to compare the utilized single-component updates against multi-component joint sampling. The results are reported in Table 16. As shown, the full joint sampling of all solver components might theoretically explore a larger search space, our empirical analysis reveals that such a method is practically **prohibitive**, incurring costs **exceeding 60 hours** on an easy task, while yielding only **marginal performance gains**, suggesting that the Coordinator-guided single-component updates already capture most of the exploitable synergy.

Our template-based approach utilizes the **Coordinator's memory** to navigate the complex interdependencies between solver components without requiring exhaustive joint sampling. Jolly-MILP maintains a diverse pool of candidate rules, ensuring that sub-optimal but potentially promising candidates are retained rather than prematurely discarded. For instance, if a high-quality Presolving rule (P) is generated but initially proves incompatible with the current state-of-the-art Branching rule (B), we do not abandon P. Instead, the Coordinator will guide the generator to iteratively evolve a new Branching rule (B') designed to be compatible with P. Once the performance of the synergistic pair (P, B') surpasses the current configuration, the new logic is successfully integrated into the solver.

Table 17: Improvement and Training Time (in brackets) Comparison

Model \ Dataset	MK	MIS	Item Placement
Joint-ML	32.89% (~9h)	33.16% (~39h)	9.52% (~200h)
Jolly-MILP	**33.46% (0.9h)**	**33.68% (4.0h)**	**11.01% (19.6h)**

J.6 Comparison with Distribution-Aware Joint-Aligned ML Components

As discussed in Sec. 1, fully co-training the heterogeneous ML components of an integrated MILP solver is impractical: each component relies on a different supervision paradigm: imitation labels for branching, expert-derived rankings for cut selection, and reinforcement signals for presolving, which prevents a unified end-to-end objective. Nevertheless, to verify that the performance gap reported in Table 1 is **not merely an artifact of inter-component distribution shift across separately trained ML models**, we construct a stronger reference baseline, denoted **Joint-ML**, via a reduced-cost sequential alignment procedure.

Joint-ML is trained in a cascade that explicitly aligns each downstream component with the actual data distribution induced by its upstream counterparts:

- **Stage 1: Presolving.** The presolving model (L2P) is trained as in its original paper and integrated into SCIP.
- **Stage 2: Cut selection.** With the trained presolving model active during data collection, we re-generate the labels required for cut selection, and train the cut model (HEM) on this presolve-aware distribution.
- **Stage 3: Branching.** With both the trained presolving and cut-selection models active, we re-generate branching labels and train the branching model (Symb4CO) on the resulting distribution.

This cascaded scheme is a deliberate compromise between full joint optimization (which would require a nested data-regeneration loop and is computationally prohibitive) and naive integration (which our Integrated-ML baselines in Table 1 already represent). It provides a **fair upper-bound proxy** for distribution-aware ML training under realistic compute budgets.

Table 17 reports the comparison on easy tasks (Multiple Knapsack, Maximum Independent Set) and a representative hard task (Item Placement). The numbers in brackets denote the **end-to-end training/search time**, including label collection.

On the easy datasets, Joint-ML achieves comparable performance with Jolly-MILP, and on the hard Item Placement dataset, it is 1.49% worse than Jolly-MILP. Also, the cost asymmetry is severe. Joint-ML requires 10× more time, with dependency on GPUs. In contrast, Jolly-MILP is CPU-only and produces transferable, warm-startable strategies (Tables 6 and 4), allowing the search cost to be amortized across new instances and problem classes, which is unavailable to retrained ML pipelines.

Table 18: Improvement of Jolly-MILP vs. MILPTune

Model \ Dataset	MK	MIS	SC	CORLAT	MIK	MIRP
MILPTune	23.18%	12.91%	25.13%	24.68%	5.76%	10.43%
Jolly-MILP	33.46%	33.68%	35.70%	33.99%	7.21%	18.15%

J.7 More Learn-to-Config Baselines

To further evaluate the effectiveness of Jolly-MILP, we expanded our comparative analysis beyond the parameter-tuning method (SMAC3). Specifically, we include a comparison with MILPTune [21], a specialized learning framework designed for automatic MILP solver configuration by learning problem similarities.

As demonstrated in Table 18, Jolly-MILP consistently and significantly outperforms MILPTune across all evaluated datasets, including MK, MIS, SC, CORLAT, MIK, and MIRP. The substantial performance gap observed between the two models highlights a fundamental difference in methodology: Parameter-tuning methods like MILPTune are typically constrained to the predefined parameter space of the solver. These methods are limited to optimizing the usage and thresholds of existing, hard-coded rules provided by the solver's developers. In contrast, Jolly-MILP operates within a significantly more expressive design space by generating entirely new symbolic policies tailored to the specific structure of a problem. Rather than merely adjusting existing flags, our framework discovers novel algorithmic logic that does not exist in the original solver code, allowing it to surpass the theoretical performance ceiling of standard configuration tools.

K Broader Impacts

Jolly-MILP brings positive societal impacts by enhancing the efficiency and interpretability of MILP solvers, benefiting industries like logistics and manufacturing through optimized resource allocation. It also democratizes access to advanced optimization tools, fostering broader AI adoption. While the shift towards AI-driven solutions may lead to some changes in job roles and reliance on proprietary technologies, these challenges are manageable with proper oversight, ensuring that the overall benefits of improved efficiency and accessibility outweigh potential drawbacks.

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 clearly state the paper's main contributions: proposing Jolly-MILP, the first framework to jointly optimize presolving, cut selection, and branching via an LLM-guided evolutionary process with a Coordinator and hierarchical Controller. Extensive experiments on nine datasets consistently demonstrate superior performance over integrated ML and existing LLM-based methods, exactly matching the claims. These claims are fully supported by the methodology (Sec. 2) and experimental results (Sec. 3).

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

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

   Answer: [Yes]

   Justification: The limitation is discussed.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

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 include theoretical results.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

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: The paper provides extensive details for reproducibility, including dataset descriptions and generation parameters (Appendix H), hyperparameters, evaluation metrics (Appendix B.3), algorithmic pseudocode (Appendix G), prompt templates for Generators, Analyzer, Coordinator, and Controller (Appendix C), feature specifications, baseline training and configuration settings (Appendix I), and hardware specifications.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

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: [Yes]

   Justification: To bridge the gap until the code release, we have included the full set of Prompt Templates and system instructions in Appendix. This allows reviewers to verify the logic behind our LLM-guided strategy generation. All datasets are standard in the community. Detailed configurations are documented in Appendix.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

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 paper specifies data splits (Appendix H), hyperparameters (population sizes, number of offspring, depth intervals, time limit, etc.), evaluation metrics (solving time, PD integral; Appendix B.3), and detailed prompts for LLM-based generation (Appendix C). For ML baselines, it describes training sample sizes and validation-based selection (Appendix I).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

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: [Yes]

Justification: The paper reports that experiments were run 5 times with different random seeds. In Appendix J.1, Table 11 presents the mean improvement percentages along with standard deviations for all datasets and baselines.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

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: The paper specifies the compute hardware in Appendix I.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

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: We have read the NeurIPS Code of Ethics, and We are sure that our research conforms to it.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.

- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

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

    Answer: [Yes]

    Justification: Discussed in Appendix K.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

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 paper poses no such risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

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: The models and datasets used in our paper are properly cited.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

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

Answer: [Yes]

Justification: As described in the paper.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

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 nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

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: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

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: This paper proposes an LLM-guided framework for exact MILP solvers. The usage of LLMs is described in detail.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.