FormuEvo: LLM-Guided Evolution for Discovering Solver-Efficient Mixed-Integer Programming Formulations

Haofeng Yuan[1], Jianing Peng[1], Jieyi Bi[2], Ni Zhang[3], Shiji Song[1,†], Zhiguang Cao[3]

[1]Department of Automation, BNRist, Tsinghua University
[2]College of Computing and Data Science, Nanyang Technological University
[3]School of Computing and Information Systems, Singapore Management University

yhf22@mails.tsinghua.edu.cn, pjn24@mails.tsinghua.edu.cn, jieyi001@e.ntu.edu.sg
ni.zhang.2025@phdcs.smu.edu.sg, shijis@mail.tsinghua.edu.cn, zgcao@smu.edu.sg

Abstract

Mixed-integer programming (MIP) lies at the core of operations research and industrial optimization. While large language models (LLMs) have recently shown promise in automated MIP modeling from natural language, they prioritize semantic correctness but overlook formulation strength, severely bottlenecking the efficiency of downstream solvers. We propose FormuEvo, an LLM-guided evolutionary framework for automated discovery of solver-efficient MIP formulations. FormuEvo frames MIP formulation design as evolutionary optimization over the symbolic space of MIP formulations, represented as executable modeling programs, by iteratively generating, evaluating, and selecting stronger candidates via LLM-driven crossover, mutation, and repair operations. To move beyond blind exploration, FormuEvo introduces a solver-informed diagnosis mechanism that exploits fine-grained solver statistics as verbal gradients for targeted refinement. Additionally, a structured memory abstracts prior experience into reusable modeling strategies, avoiding redundant exploration while enabling zero-shot transfer to unseen problems and bootstrapping smaller LLMs. Experiments across diverse linear and non-linear problems demonstrate that FormuEvo discovers formulations that significantly outperform both expert-designed formulations and existing LLM-based approaches, accelerating solvers by up to $5.5\times$, with distilled knowledge transferring effectively across problems and model scales.

1 Introduction

Mixed-integer programming (MIP) plays a fundamental role in decision-making across domains such as manufacturing, transportation, and management (Bixby, 2012; Wolsey, 2020; Clautiaux and Ljubić, 2025). The practical success of MIP has been largely driven by decades of advances in powerful solvers such as Gurobi (Gurobi, 2026) and COPT (Ge et al., 2022), which integrate sophisticated algorithmic components including branch-and-bound, cutting planes, and various heuristics to solve problems of growing scale and complexity.

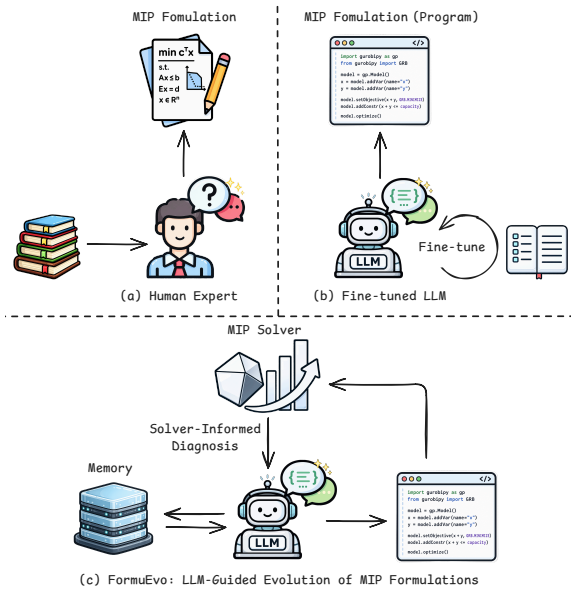

Figure 1: MIP formulation design via (a) human experts, (b) LLM fine-tuning with end-to-end generation, and (c) the LLM-guided evolution framework of FormuEvo.

A well-established principle in mathematical programming is that a single optimization problem may admit multiple mathematically equivalent MIP formulations, which guarantee the same optimal solution but can differ by orders of magnitude in computational efficiency (Wolsey, 2020). Consequently, the practical tractability of MIP depends not only on constructing a mathematically *correct* formulation, but also on designing one that is computationally *strong*, namely, whose structural properties enable MIP solvers to operate efficiently.

Despite its fundamental importance, discovering

[†]Corresponding author.
Code: https://github.com/Xyz-yuanhf/formuevo

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing, pages 5109–5135
October 24-29, 2026 ©2026 Association for Computational Linguistics

solver-efficient MIP formulations has long been regarded as one of the most challenging and expertise-intensive problems in operations research (Vielma, 2015). Even experienced experts require substantial domain knowledge and structural insights to craft a strong formulation (Fig. 1(a)). This challenge is further complicated by the fact that modeling intuitions can become outdated: with the development of modern MIP solvers, classical modeling strategies once regarded as "best practices", such as certain strengthening constraints or symmetry-breaking rules, can inadvertently interfere with the advanced presolving procedures, cut generation, and heuristic routines, leading to counter-intuitive performance degradation (Achterberg and Wunderling, 2013; Salvagnin, 2018). This motivates a new automated approach capable of discovering solver-efficient MIP formulations and adaptable to the advancing algorithmic internals of modern solvers.

Large language models (LLMs) have recently demonstrated remarkable promise in automating MIP modeling from natural language (Wang et al., 2025; Qian et al., 2025; Zhou et al., 2026). By fine-tuning based on curated formulation datasets, they have effectively bridged the gap between natural language problem descriptions and executable MIP modeling programs via end-to-end generation (Fig. 1(b)). However, their objective is *correctness* and *executability*, namely, producing a formulation that accurately encodes the target problem and can be successfully solved by a downstream solver. As a result, the formulations they generate, though correct, are often computationally naive, falling far short of strong formulations required for large-scale real-world optimization.

To address this gap, we propose **FormuEvo**, a novel LLM-guided evolutionary framework for automated discovery of solver-efficient MIP formulations. Rather than treating formulation design as single-pass generation, FormuEvo reframes it as *evolutionary optimization* over the symbolic space of MIP formulations, which are represented as executable modeling programs. Given a natural language problem description and a formulation template, FormuEvo maintains a population of candidate formulations, iteratively applying LLM-driven crossover, mutation, and repair operations guided directly by the solver performance, progressively evolving toward solver-efficient formulations (Fig. 1(c)). To move beyond blind exploration, FormuEvo introduces two key mechanisms: 1) a *solver-informed diagnosis* mechanism, where a diagnostic LLM converts fine-grained solver statistics into interpretable verbal gradients, guiding evolution toward targeted refinement, and 2) a *structured memory*, which abstracts prior experience into reusable modeling strategies, avoiding redundant exploration while further enabling zero-shot transfer to unseen problems and bootstrapping smaller LLMs by distilling a generalizable knowledge base. Our main contributions are as follows:

- We propose FormuEvo, an LLM-guided evolutionary framework for automated discovery of solver-efficient MIP formulations.
- We introduce a solver-informed diagnosis mechanism that leverages fine-grained solver statistics for targeted, solver-aware evolution.
- We develop a structured memory that distills reusable knowledge, enabling efficient search and transfer across problem and model scales.
- FormuEvo discovers non-obvious MIP formulations that outperform the state-of-the-art baselines, accelerating MIP solvers by up to $5.5\times$.

2 Related Work

2.1 MIP Modeling and Formulation

Designing strong MIP formulations is a long-standing challenge in operations research (Wolsey, 2020). Classical strengthening techniques include valid inequalities and cutting planes (Nemhauser and Wolsey, 1988; Cornuéjols, 2008), symmetry-breaking constraints (Margot, 2009), and extended variables and reformulations (Conforti et al., 2013). However, with rapid development of modern MIP solvers, classical techniques can sometimes conflict with sophisticated solver internals and lead to counter-intuitive performance degradation (Achterberg and Wunderling, 2013; Salvagnin, 2018), motivating the need for automated, adaptive approaches to formulation design.

2.2 LLM for Automated MIP Modeling

Recent advances in LLMs have automated MIP modeling from natural language through structured formulations (Ramamonjison et al., 2022), agent systems (AhmadiTeshnizi et al., 2024), or fine-tuning (Huang et al., 2025; Wang et al., 2026; Zhou et al., 2026). Despite improving domain fidelity, they suffer from two major flaws. First, they operate at the *instance* level, generating formulations with parameters tied to particular instances rather than discovering generalized, instance-agnostic formulations at the *problem* level. Second, as these

open-loop methods prioritize mathematical *correctness* rather than formulation *strength*, the generated formulations are structurally weak and inefficient for large-scale optimization.

2.3 LLM-Guided Evolutionary Search

LLM-guided evolutionary search has emerged as a powerful paradigm for automated algorithm design and heuristic discovery (Liu et al., 2024; Novikov et al., 2025). FunSearch (Romera-Paredes et al., 2024) pioneered this by using LLMs as evolutionary operators over executable code, while subsequent works (Ye et al., 2024, 2025; Zhang et al., 2026; Hou et al., 2026; Gungordu et al., 2026) extended this framework for combinatorial heuristic design. However, little work has adapted this paradigm to MIP formulation. The most closely related work, EvoCut (Yazdani et al., 2025), applies LLM-based evolution to cutting plane generation. However, EvoCut remains limited to local relaxation tightening on an existing model and lacks the capability of holistic structural redesign. In contrast, FormuEvo evolves over the global symbolic space of MIP formulations, and incorporates solver-informed diagnosis and structured memory to enable systematic and directed improvement.

3 FormuEvo

3.1 Problem Formalization

Given a problem p described in natural language, an MIP formulation can be modeled as:

$$\begin{aligned} \min_{x,y} \quad & c(x,y) \\ \text{s.t.} \quad & g_i(x,y) \le 0, \quad i = 1,\dots,m, \\ & h_j(x,y) = 0, \quad j = 1,\dots,n, \\ & x \in X \subseteq \mathbb{R}^q, \\ & y \in Y \subseteq \mathbb{Z}^r, \end{aligned} \tag{1}$$

where x and y denote continuous and integer decision variables, $c(x,y)$ is the objective function, and $g_i(x,y)$, $h_j(x,y)$ denote inequality and equality constraints, respectively. In practice, the MIP formulation is instantiated as executable programs (e.g., Python code with Gurobi) that specify these components and can be directly executed by an MIP solver. We denote by $\mathcal{F}$ the symbolic space of MIP formulations of problem p, namely, the set of all semantically correct and syntactically valid programs encoding problem p. By definition, any two formulations $f_1,\ f_2 \in \mathcal{F}$, possibly involving different structures of variables and constraints, are equivalent in the sense of optimization, which guarantees the same optimal solution for problem p.

Existing LLM-based approaches treat MIP modeling as a *satisfaction problem*, aiming to generate *any* formulation that can be successfully solved by a downstream MIP solver and produce the correct solution. However, they largely overlook a fundamental principle in mathematical optimization: although formulations in $\mathcal{F}$ are mathematically equivalent, they can induce dramatically different solver behaviors and computational efficiency. In practice, two equivalent formulations may differ by orders of magnitude in runtime due to differences in relaxation tightness, symmetry structure, and even solver-specific implementation details. Motivated by this, we reframe MIP formulation design as an *optimization problem* over the symbolic space of MIP formulations toward minimizing the downstream computational cost:

$$f^\star = \arg\min_{f \in \mathcal{F}} \phi(f), \tag{2}$$

where $\phi(f)$ evaluates the computational cost (e.g., runtime) of formulation f with a downstream MIP solver. This defines an optimization problem over a vast, discrete, and highly structured symbolic space, where the objective is non-differentiable and expensive to evaluate. Therefore, we frame the discovery of solver-efficient MIP formulations as an evolutionary search, with LLMs as operators to navigate the symbolic space of MIP formulations.

3.2 LLM-Guided Evolutionary Search

FormuEvo takes a natural language description and a formulation template f_0 as input, maintaining a population $\mathcal{P}_t = \{f_1, \dots, f_N\}$ of N candidate formulations at generation t. To effectively explore the vast and structured symbolic space $\mathcal{F}$, FormuEvo employs an evolutionary loop coordinated by five specialized LLM modules: 1) a *generator LLM* that produces new candidate formulations, 2) a *diagnostic LLM* that interprets solver feedback to guide generation, 3) a *repair LLM* that debugs failed formulations, 4) a *reflector LLM* that abstracts evolution experience into structural memories, and 5) a *distiller LLM* that distills accumulated memory into transferable knowledge for unseen problems. The overall framework is illustrated in Fig. 2.

Initialization. Given the problem description and a formulation template, the evolution starts by prompting the generator LLM to produce N candidate formulations as the initial population $\mathcal{P}_0$. The

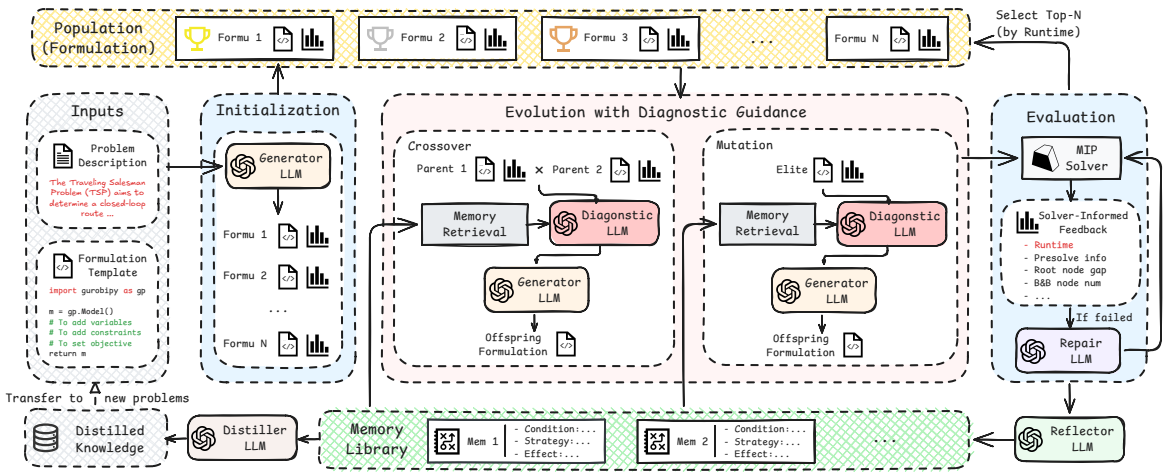

Figure 2: The overview of FormuEvo. FormuEvo iteratively evolves formulations through LLM-driven crossover, mutation, and repair toward better solver efficiency, guided by solver-informed diagnosis and structured memory.

generator is encouraged to explore diverse formulation variants, such as alternative variables, logically equivalent but structurally different constraints, as well as variations in implementation details, to prevent premature convergence to sub-optimal local regions in the symbolic space.

Evaluation. Each candidate formulation $f \in \mathcal{P}_t$ is evaluated by executing it on a downstream MIP solver (e.g., Gurobi) over a set of instances of problem p. If execution fails due to compilation errors or correctness violations (by verifying whether the solved objective values match the ground-truth optimal value), the error tracebacks and solver logs are fed into a repair LLM, which attempts to fix the candidate formulation while preserving its intended modeling logic. Candidates exceeding the maximum repair budget are discarded, ensuring that all maintained formulations strictly belong to $\mathcal{F}$. Notably, since the generator LLM may produce bold yet error-prone formulation variants, the repair operations can still be valuable for exploration.

For each successfully evaluated formulation, its fitness is measured by the solver cost $\phi(f)$, computed as the shifted geometric mean (SGM) of per-instance runtimes, a standard evaluation metric for solver performance. Meanwhile, the reflector LLM abstracts the modeling decisions and solver feedback from each evaluated formulation into reusable strategies, which are stored in a structured memory library for future retrieval (Section 3.4).

Evolution. At each generation, FormuEvo preserves the top-N formulations according to their fitness. The next generation is produced through two LLM-based operators: *crossover* and *mutation*.

For crossover, two distinct parent formulations are sampled from the current population $\mathcal{P}_t$, with probability according to their fitness rank. They are first processed by the diagnostic LLM to produce a structural diagnosis (Section 3.3). Guided by this diagnosis, the generator LLM produces an offspring formulation that integrates complementary strengths or explores an orthogonal modeling direction. For mutation, a high-performing elite is selected. Similar to the crossover workflow, the diagnostic LLM analyzes the elite formulation and pinpoints specific refinement opportunities to explore its structural neighborhood. Guided by this targeted diagnosis, the generator LLM then implements directed refinements instead of blind perturbation to produce a mutated offspring.

All offspring candidates are subject to the evaluation and repair procedure described above. This process iterates for T generations, to evolve the population toward increasingly solver-efficient formulations. The best formulation across all generations is returned as the final formulation $\hat{f}^\star$.

3.3 Solver-Informed Diagnosis

A key bottleneck of evolutionary search is that it overly relies on a scalar fitness signal to guide the evolution (Romera-Paredes et al., 2024; Shi et al., 2026). In our case, the runtime fitness can indicate *whether* a formulation is slow, but provides little insight into *why* it is slow, making the search space exploration highly sample-inefficient.

To overcome this bottleneck, FormuEvo introduces a *solver-informed diagnosis* mechanism. We note that modern MIP solvers are not mere black-box evaluators; instead, they expose rich, fine-

grained internal statistics during optimization, such as presolving outcomes, relaxation quality, and branching dynamics (right of Fig. 2). These statistics reflect important solver behaviors and reveal structural bottlenecks that a single runtime cannot capture. For example, a weak relaxation may indicate excessive fractional solutions that require strengthening, while a large branch-and-bound tree may suggest a highly symmetric structure.

To exploit this fine-grained feedback for targeted, solver-informed evolution, the diagnostic LLM takes the parent or elite formulations alongside their solver statistics to produce a structured verbal diagnosis before crossover or mutation. The diagnosis identifies major computational bottlenecks, attributes them to specific structural properties, and proposes targeted directions for improvement. For instance, if parent f_i is compact but suffers from weak relaxations that induce excessive branching, while parent f_j yields stronger bounds but incurs costly presolving, the diagnostic LLM may guide the generator LLM to selectively incorporate the bound-tightening constraints of f_j where they matter most, while retaining the compact structure of f_i elsewhere. This diagnosis functions as an interpretable verbal gradient, guiding the generator LLM toward principled, solver-informed structural improvement rather than undirected exploration.

3.4 Structured Memory

We note that in long-horizon evolutionary search, LLMs are prone to redundant exploration, such as repeatedly revisiting failed modifications or forgetting effective sub-structures that can be directly retrieved for future refinements. To mitigate this issue, FormuEvo incorporates a structured memory library to abstract, organize, and store both successful and failed experiences into reusable modeling strategies for guiding future evolution steps.

Specifically, after each evaluation, the reflector LLM abstracts the modeling modifications and solver feedback between parent and offspring formulations into a structured memory entry with three fields: 1) a *condition* that describes the problem context and formulation characteristics under which the strategy was applied; 2) a *strategy* that captures the specific modeling decision or structural modification; and 3) an *effect* that summarizes its observed impact on solver behavior and optimization performance. This structured format, as illustrated in Fig. 3, enables precise retrieval and reuse of prior experience. During subsequent evolution, the diagnostic LLM retrieves relevant memory entries based on the current bottleneck conditions and incorporates them as additional context to enrich the diagnosis, leveraging prior experience to prune the search space and avoid known pitfalls.

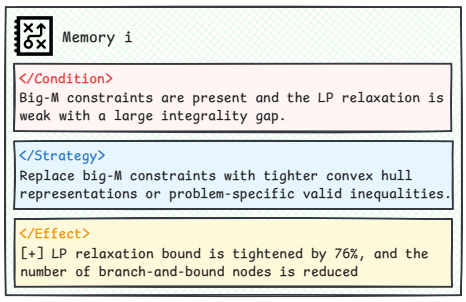

Figure 3: Example of a structured memory entry.

While the memory efficiently accelerates the search within a single problem, we note that optimization problems often share common modeling logic and structural properties (e.g., routing logic, assignment constraints), even when their mathematical formulations may appear different. Therefore, upon completion of the evolutionary search, we introduce a distiller LLM to summarize the accumulated memory library into high-level problem-agnostic knowledge by removing instance-specific details. This distilled knowledge base not only enables FormuEvo to perform efficient zero-shot transfer when tasked with entirely unseen optimization problems, but can also serve as a lightweight yet high-quality reasoning resource for bootstrapping smaller LLMs with domain expertise distilled from a more powerful model (see Section 4.3).

4 Experiments

4.1 Experiment Setup

Benchmarks. We evaluate FormuEvo on classical mixed-integer linear programming (MILP) and non-linear programming (MINLP) benchmarks, including: traveling salesman problem (TSP), job-shop scheduling problem (JSSP), bin packing problem (BPP), capacitated facility location problem (CFLP), and non-linear quadratic assignment problem (QAP). These problems span diverse optimization structures, covering routing, scheduling, packing, location, and assignment. To further assess the capability of FormuEvo beyond well-studied textbook settings, we additionally consider two challenging and less explored optimization tasks: neural network verification (NNV) (Tjeng et al., 2019) and a mathematical challenge derived from

IMO 2025 Problem 6 (Art of Problem Solving Wiki, 2025), where human-designed formulation priors are limited or absent. Following prior practices (Gleixner et al., 2021), instances for classical problems are categorized into three levels: Easy, Medium, and Hard. The Easy set consists of small instances used for training and validating in evolution, while the Medium and Hard sets contain larger instances held out to test the final performance. For the two novel optimization problems, we use Easy instances for evolution, and evaluate final performance on a separate set of Hard instances. More detailed problem descriptions and the corresponding instance splits are provided in Appendix A.

Baselines. We compare FormuEvo with two categories of baselines. First, we include both standard and state-of-the-art expert-designed formulations from the operations research literature, tracing the advancement of human expertise. For instance, TSP baselines range from the textbook MTZ formulation (Miller et al., 1960), to the stronger single-commodity flow (SCF) formulation (Gavish and Graves, 1978), up to the MCF-RLT (Balma et al., 2018), which achieves the theoretically tightest known relaxation bound, representing the current limits of human design. We also compare with recent LLM-based modeling approaches, including ORLM (Huang et al., 2025) and StepORLM (Zhou et al., 2026), to demonstrate the gap between correctness-oriented generation and efficiency-oriented evolution. Finally, we include EvoCut (Yazdani et al., 2025), an LLM-guided cutting-plane generation method, to highlight the difference between local formulation strengthening and our exploration of the broad symbolic space of MIP formulations. Additional introductions for the baseline formulations of each problem are also provided in Appendix A.

Implementation. All experiments are conducted in Python using Gurobi 10.0 as the downstream MIP solver, with a single thread and default parameters, on a server equipped with AMD EPYC 9654 processors. We use a population size of $N = 8$ and evolve for $T = 5$ generations, producing 8 offspring per generation via crossover and mutation with a mutation rate of 0.3. Memory-augmented generation is applied with probability 0.7 to avoid search overfitting toward previously explored regions. Each candidate is evaluated on 100 Easy instances, with fitness measured by SGM runtime with a 1-second shift (Gleixner et al., 2021). Failed candidates receive one repair attempt. Operator prompts are provided in Appendix B.

FormuEvo uses GPT-5.4-mini as the backbone LLM (a comparison across different backbones is reported in Section 4.5). ORLM and StepORLM use their fine-tuned LLMs, and are allowed the same number of generation attempts as FormuEvo for a fair comparison. For EvoCut, we adopt the reported formulation from their original paper on problems covered by their benchmark, noting that EvoCut uses a stronger backbone model than ours. For problems outside their benchmark suite, we rerun EvoCut using the same backbone and experimental settings as FormuEvo.

4.2 Main Results

Tables 1, 2 present results on classical MILP and MINLP benchmarks and the two novel problems, respectively. We report three metrics: 1) "Time", SGM runtime over 100 test instances and 5 independent runs, with a per-instance time limit of 600 seconds; 2) "Wins", the number of instances on which a formulation achieves the best runtime among all formulations; and 3) "Solved", the number of instances solved to optimality within the time limit.

Overall, FormuEvo discovers formulations that outperform both expert-designed and LLM-based baselines, with acceleration of up to $5.5\times$ over the best baselines. The best discovered formulations are presented in Appendix A. Moreover, FormuEvo attains the best runtime on the majority of instances across nearly all benchmarks, demonstrating robustness across diverse optimization structures. Beyond the numerical improvements, the results reveal several important insights.

First, FormuEvo successfully discovers formulations that outperform not only textbook models but also highly optimized expert-designed formulations, often by substantial margins. For instance, in TSP, while the human-designed MCF-RLT formulation has the theoretically tightest relaxation bound, it completely fails (0/100 solved) on Hard instances. This highlights a known but often overlooked fact: theoretical tightness does not necessarily translate to solver efficiency, as a heavily extended formulation can overwhelm the computation in MIP solvers. In contrast, FormuEvo is efficiency-oriented and guided by solver-informed statistics, enabling targeted optimization toward the practical computational objective of solvers.

Second, the results of ORLM and StepORLM further illustrate the gap between correctness and

Table 1: Performance on classical MILP and MINLP problems.

		Easy		Medium		Hard		
	Formulation	Time↓	Wins↑	Time↓	Wins↑	Time↓	Wins↑	Solved↑
TSP	MTZ (1960)	0.9578	0/100	6.1826	5/100	95.1315	0/100	62/100
	SCF (1978)	0.3079	3/100	1.0065	10/100	8.3315	3/100	**100/100**
	MCF-RLT (2018)	3.2729	0/100	84.1649	0/100	600.0570	0/100	0/100
	ORLM (2025)	0.9575	2/100	6.3270	0/100	92.5953	0/100	62/100
	StepORLM (2026)	1.7667	0/100	17.0081	0/100	173.8792	0/100	64/100
	EvoCut (2025)	0.5268	8/100	2.5013	6/100	13.8359	17/100	92/100
	FormuEvo (Ours)	**0.1410** (+54.2%)	**87/100**	**0.5640** (+44.0%)	**79/100**	**3.9469** (+52.6%)	**80/100**	**100/100**
JSSP	Disj. (1960)	0.3264	19/100	1.8389	9/100	17.6653	25/100	98/100
	Enh. Disj. (2016)	0.4886	1/100	7.9405	0/100	158.1492	0/100	77/100
	ORLM (2025)	600.0012	0/100	600.0032	0/100	600.0019	0/100	0/100
	StepORLM (2026)	0.3040	28/100	1.7270	22/100	18.7024	20/100	**99/100**
	EvoCut (2025)	0.3005	19/100	1.7627	20/100	18.1329	14/100	98/100
	FormuEvo (Ours)	**0.2781** (+7.5%)	**33/100**	**1.5316** (+11.3%)	**49/100**	**15.4237** (+12.7%)	**41/100**	98/100
BPP	Kant. (1960)	18.6428	0/100	191.9222	0/100	351.7605	0/100	28/100
	AF (1999)	0.2710	0/100	0.9930	1/100	4.9778	1/100	**100/100**
	VPSolver (2016)	0.0610	**53/100**	0.4064	40/100	1.4425	38/100	**100/100**
	ORLM (2025)	18.3614	0/100	287.7057	0/100	341.1659	0/100	24/100
	StepORLM (2026)	12.3430	0/100	117.0032	0/100	95.0336	1/100	42/100
	EvoCut (2025)	3.0279	0/100	26.7969	4/100	29.0496	5/100	60/100
	FormuEvo (Ours)	**0.0543** (+11.0%)	47/100	**0.3418** (+15.9%)	**55/100**	**0.8384** (+41.9%)	**55/100**	**100/100**
CFLP	Agg. (1965)	2.7178	19/100	22.5520	8/100	87.3078	6/100	96/100
	Disagg. (1989)	4.8283	0/100	31.4982	0/100	113.4126	1/100	95/100
	Enh. Disagg. (2021)	5.0126	0/100	33.1480	0/100	113.5033	1/100	95/100
	ORLM (2025)	2.8439	0/100	24.0395	0/100	88.4319	2/100	95/100
	StepORLM (2026)	3.2663	1/100	26.8000	3/100	104.8201	3/100	94/100
	EvoCut (2025)	2.7698	5/100	17.1088	14/100	59.4309	16/100	**100/100**
	FormuEvo (Ours)	**1.6258** (+40.2%)	**75/100**	**11.6952** (+31.6%)	**75/100**	**44.3447** (+25.4%)	**71/100**	**100/100**
QAP	KB Quad. (1957)	9.9323	0/100	23.7309	0/100	538.7275	0/100	28/100
	McC. Lin. (1976)	9.5458	0/100	67.9480	0/100	600.0062	0/100	0/100
	AJ Lin. (1994)	1.2154	1/100	8.0087	0/100	237.7423	0/100	97/100
	XY-KB Lin. (2006)	0.4044	28/100	1.6163	25/100	37.8213	28/100	**100/100**
	IPQAPR (2010)	**0.3929**	35/100	1.6883	26/100	40.2989	19/100	**100/100**
	ORLM (2025)*	-	-	-	-	-	-	-
	StepORLM (2026)	9.7489	0/100	68.3794	0/100	600.0061	0/100	0/100
	EvoCut (2025)	10.1303	0/100	23.9718	0/100	538.5478	0/100	28/100
	FormuEvo (Ours)	0.3936 (-0.2%)	**36/100**	**1.5648** (+3.2%)	**49/100**	**34.5034** (+8.8%)	**53/100**	**100/100**

*ORLM failed to produce a correct formulation for QAP.

efficiency. Although they successfully generate correct and executable formulations on standard MILPs, their outputs largely remain naive textbook formulations, which are weak and scale poorly in practice. Moreover, both approaches fail to produce valid formulations for the novel NNV and IMO problems. This also underscores the limited generalization of fine-tuning approaches to more complex problems beyond their training distribution. See Appendix C.5 for further discussion.

Third, EvoCut, which strengthens fixed formulations through LLM-generated cutting planes, yields moderate improvements over baseline formulations but remains fundamentally limited by the underly-

Table 2: Performance on two novel MIP problems.

	Formulation	Time↓	Wins↑	Solved↑
NNV	Std. Formu. (2019)	69.5200	9/100	71/100
	ORLM / StepORLM	-	-	-
	EvoCut (2025)	67.6093	13/100	73/100
	FormuEvo (Ours)	**21.4139** (+68.3%)	**78/100**	**86/100**
IMO	Std. Formu. (2025)	114.0912	0/4	3/4
	ORLM / StepORLM	-	-	-
	EvoCut (2025)	99.4400	0/4	3/4
	FormuEvo (Ours)	**17.9341** (+82.0%)	**4/4**	**4/4**

Table 3: Ablation study of FormuEvo components.

	Method	Easy	Medium	Hard
TSP	Best Baseline	0.3079	1.0065	8.3315
	FormuEvo	**0.1410**	**0.5640**	**3.9469**
	w/o Memory	0.1776	0.6102	4.6649
	w/o Diagnosis	0.2308	0.7350	6.5207
JSSP	Best Baseline	0.3005	1.7270	17.6653
	FormuEvo	**0.2781**	**1.5316**	**15.4237**
	w/o Memory	0.2846	1.5698	16.3223
	w/o Diagnosis	0.3048	1.7648	18.0159

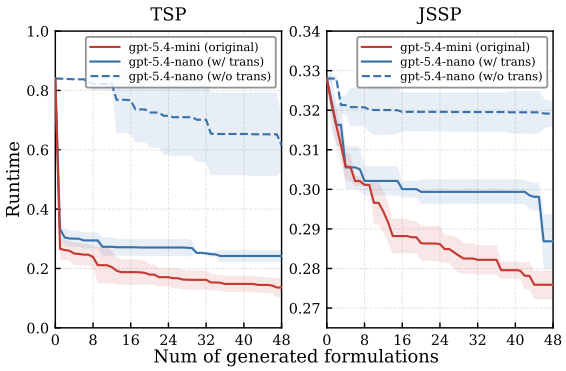

Figure 4: Transfer performance on smaller LLMs.

Table 4: Impact of different backbone LLMs.

	Method	Cost	Easy	Medium	Hard
TSP	Best Baseline	-	0.3079	1.0065	8.3315
	GPT-5.4-mini	∼ $2	0.1410	0.5640	**3.9469**
	Claude-Sonnet-4.6	∼ $10	**0.1017**	**0.4777**	4.0148
	DeepSeek-V4-Flash	∼ $0.5	0.1498	0.6283	5.3068
JSSP	Best Baseline	-	0.3005	1.7270	17.6653
	GPT-5.4-mini	∼ $3	**0.2781**	**1.5316**	**15.4237**
	Claude-Sonnet-4.6	∼ $16	0.2808	1.5319	15.8340
	DeepSeek-V4-Flash	∼ $0.8	0.3050	1.6399	16.1327

ing model structure. For instance, on BPP, EvoCut improves upon standard formulations but remains significantly weaker than both VPSolver and FormuEvo, because the primary bottleneck of BPP lies in its assignment structure rather than LP relaxation strength. Similarly, on QAP, the standard quadratic formulation defines a Birkhoff polytope that already represents the convex hull (von Neumann, 1953), and EvoCut fails to achieve any improvement through strengthening cuts. In contrast, FormuEvo breaks through this limitation by searching over the entire symbolic space of MIP formulations, enabling the discovery of structural reformulations such as alternative linearizations and variable extension that fundamentally alter the search landscape rather than merely modifying a fixed one.

4.3 Transfer Performance

We evaluate the transferability of distilled knowledge to unseen problems and smaller LLMs. Fig. 4 shows evolution trajectories using a smaller LLM (GPT-5.4-nano), with distilled knowledge aggregated from all problems except the target. Without transfer, GPT-5.4-nano exhibits substantially slower convergence and higher final runtime, struggling to move beyond the baseline formulations. In contrast, when augmented with the distilled knowledge, GPT-5.4-nano significantly narrows the gap, achieving convergence trajectories and final performance close to the larger GPT-5.4-mini model. These results suggest that the distilled knowledge can serve as a lightweight but effective reasoning source, compensating for the capacity of smaller models by providing structured domain expertise that would otherwise be difficult to infer independently, which offers a practical pathway toward strong MIP formulation discovery at lower costs.

4.4 Ablation Study on Key Components

We analyze the contributions of two key components: solver-informed diagnosis and memory, by comparing FormuEvo against two ablated variants. As shown in Table 3, removing either component leads to clear performance degradation, indicating that the diagnosis mechanism plays a crucial role in providing the directional signal for FormuEvo, while the memory reuse also facilitates the evolution by improving the search efficiency.

4.5 Impact of Different LLM Backbones

We further evaluate FormuEvo with different backbone LLMs (Table 4). All three consistently outperform the best baselines on both problems, with only modest differences in formulation quality and

generalization, where no backbone model consistently dominates across all settings. These results indicate that the gains stem primarily from the evolutionary framework and are robust to the choice of backbone. We adopt GPT-5.4-mini as the default because it offers the best overall trade-off between performance and API cost.

Additional analyses on different downstream solvers, backbone LLMs, and evolutionary hyperparameters are provided in Appendix C.2 - C.4.

5 Conclusions

We presented FormuEvo, an LLM-guided evolutionary framework for automatically discovering solver-efficient MIP formulations, which incorporates solver-informed diagnosis and structured memory for directed and systematic improvement. Experiments show that FormuEvo successfully discovers formulations that significantly outperform state-of-the-art baselines, accelerating modern MIP solvers by up to $5.5\times$. Notably, FormuEvo is transferable to unseen problems and smaller LLMs by distilling a generalizable knowledge base, demonstrating its generalizability and applicability.

Limitations

FormuEvo focuses on discovering static MIP formulations that can be directly solved by off-the-shelf general-purpose MIP solvers, which represents the most commonly adopted approach in real-world practice. However, for many large-scale or highly complex optimization problems, advanced problem-specific solution approaches in operations research often rely on dynamic relaxation or decomposition algorithms, such as column generation, which iteratively adds variables by solving a pricing sub-problem, and Benders decomposition, which relaxes the original problem and then alternates between a master problem and sub-problems to progressively tighten the formulation. These methods typically require iterative interactions between master problems and sub-problems, along with dynamic generation of variables and constraints during the solving process. In such approaches, *formulation* design and *algorithm* development are inherently coupled: the effectiveness of the formulation depends on the decomposition strategy, while the decomposition procedure itself is tightly determined by the underlying formulation structure. Extending FormuEvo beyond static formulations to jointly evolve reformulations and decomposition-based solution algorithms is a promising but substantially more challenging direction. We leave this extension for future work.

Acknowledgments

This work was done during the first author's internship at Singapore Management University. This work is supported in part by the BNRist Project (No: BNR2024TD03003), the National Natural Science Foundation of China (No: 42327901), and the Program of China Scholarship Council (No: 202506210230). This research is supported by the National Research Foundation Singapore under the AI Singapore Programme (AISG Award No: AISG3-RPGV-2025-017). We would also like to thank the AI4Opt group for helpful discussions, and the reviewers for their valuable feedback.

References

Tobias Achterberg and Roland Wunderling. 2013. Mixed integer programming: Analyzing 12 years of progress. In *Facets of Combinatorial Optimization*, pages 449–481. Springer.

Warren P Adams and Terri A Johnson. 1994. Improved linear programming-based lower bounds for the quadratic assignment problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 16:43–77.

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

Art of Problem Solving Wiki. 2025. 2025 IMO Problem 6. https://artofproblemsolving.com/wiki/index.php/2025_IMO_Problems/Problem_6. Accessed: May 2026.

Pasquale Avella, Maurizio Boccia, Sara Mattia, and Fabrizio Rossi. 2021. Weak flow cover inequalities for the capacitated facility location problem. *European Journal of Operational Research*, 289(2):485–494.

Michel Louis Balinski. 1965. Integer programming: Methods, uses, computations. *Management Science*, 12(3):253–313.

Ali Balma, Safa Ben Salem, Mehdi Mrad, and Talel Ladhari. 2018. Strong multi-commodity flow formulations for the asymmetric traveling salesman problem. *European Journal of Operational Research*, 271(1):72–79.

Robert E Bixby. 2012. A brief history of linear and mixed-integer programming computation. In *Optimization Stories*, pages 107–121.

Filipe Brandao and Joao Pedro Pedroso. 2016. Bin packing and related problems: General arc-flow formulation with graph compression. *Computers & Operations Research*, 69:56–67.

François Clautiaux and Ivana Ljubić. 2025. Last fifty years of integer linear programming: A focus on recent practical advances. *European Journal of Operational Research*, 324(3):707–731.

Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. 2013. Extended formulations in combinatorial optimization. *Annals of Operations Research*, 204(1):97–143.

Gérard Cornuéjols. 2008. Valid inequalities for mixed integer linear programs. *Mathematical Programming*, 112(1):3–44.

Bezalel Gavish and Stephen C Graves. 1978. The travelling salesman problem and related problems. *MIT Operations Research Center*.

Dongdong Ge, Qi Huangfu, Zizhuo Wang, Jian Wu, and Yinyu Ye. 2022. Cardinal optimizer (COPT) user guide. *arXiv preprint arXiv:2208.14314*.

Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. 2021. MIPLIB 2017: Data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3):443–490.

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

Gurobi. 2026. Gurobi Optimizer Reference Manual. https://www.gurobi.com. Accessed: May 2026.

Christopher Hojny, Mathieu Besançon, Ksenia Bestuzheva, Sander Borst, João Dionísio, Johannes Ehls, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Adrian Göß, Alexander Hoen, Jacob von Holly-Ponientzietz, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Marco Lübbecke, Stephen J. Maher, and 15 others. 2025. The SCIP optimization suite 10.0. Technical report, Optimization Online.

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

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

Leonid V Kantorovich. 1960. Mathematical methods of organizing and planning production. *Management Science*, 6(4):366–422.

Tjalling C Koopmans and Martin Beckmann. 1957. Assignment problems and the location of economic activities. *Econometrica: Journal of the Econometric Society*, pages 53–76.

Wen-Yang Ku and J Christopher Beck. 2016. Mixed integer programming models for job shop scheduling: A computational analysis. *Computers & Operations Research*, 73:165–173.

Janny MY Leung and Thomas L Magnanti. 1989. Valid inequalities and facets of the capacitated plant location problem. *Mathematical Programming*, 44(1):271–291.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning*.

Alan S Manne. 1960. On the job-shop scheduling problem. *Operations Research*, 8(2):219–223.

François Margot. 2009. Symmetry in integer linear programming. *50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art*, pages 647–686.

Garth P McCormick. 1976. Computability of global solutions to factorable nonconvex programs: Part I—convex underestimating problems. *Mathematical Programming*, 10(1):147–175.

Clair E Miller, Albert W Tucker, and Richard A Zemlin. 1960. Integer programming formulation of traveling salesman problems. *Journal of the ACM*, 7(4):326–329.

George L Nemhauser and Laurence A Wolsey. 1988. *Integer and Combinatorial Optimization*, volume 18. Wiley New York.

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

Cheng Qian, Hongyi Du, Hongru Wang, Xiusi Chen, Yuji Zhang, Avirup Sil, Chengxiang Zhai, Kathleen McKeown, and Heng Ji. 2025. ModelingAgent: Bridging LLMs and mathematical modeling for real-world challenges. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 1599–1633.

Rindra Ramamonjison, Haley Li, Timothy Yu, Shiqi He, Vishnu Rengan, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. 2022. Augmenting operations research with auto-formulation of optimization models from problem descriptions. In *Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 29–62.

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

Domenico Salvagnin. 2018. Symmetry breaking inequalities from the schreier-sims table. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 521–529. Springer.

Yiding Shi, Jianan Zhou, Wen Song, Jieyi Bi, Yaoxin Wu, Zhiguang Cao, and Jie Zhang. 2026. Generalizable heuristic generation through LLMs with meta-optimization. In *International Conference on Learning Representations*.

Marius M Solomon. 1987. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265.

Eric Taillard. 1993. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285.

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. 2019. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*.

JM Valério de Carvalho. 1999. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86(0):629–659.

Juan Pablo Vielma. 2015. Mixed integer linear programming formulation techniques. *SIAM Review*, 57(1):3–57.

John von Neumann. 1953. A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games II*, 28:5–12.

Yilin Wang, Heng Zhou, Dongxing Mao, Linjie Li, Jingru Tan, Haochen Han, Zhengyuan Yang, Alex Jinpeng Wang, and Min Li. 2026. OR-PRM: A process reward model for algorithmic problem in operations research. In *International Conference on Learning Representations*.

Zhiyuan Wang, Bokui Chen, Yinya Huang, Qingxing Cao, Ming He, Jianping Fan, and Xiaodan Liang. 2025. ORMind: A cognitive-inspired end-to-end reasoning framework for operations research. In *Annual Meeting of the Association for Computational Linguistics: Industry Track*, pages 104–131.

Laurence A Wolsey. 2020. *Integer Programming*. John Wiley & Sons.

Yong Xia and Ya-Xiang Yuan. 2006. A new linearization method for quadratic assignment problems. *Optimisation Methods and Software*, 21(5):805–818.

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

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. 2024. ReEvo: Large language models as hyper-heuristics with reflective evolution. *Advances in Neural Information Processing Systems*, 37:43571–43608.

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

Huizhen Zhang, Cesar Beltran-Royo, and Miguel Constantino. 2010. Effective formulation reductions for the quadratic assignment problem. *Computers & Operations Research*, 37(11):2007–2016.

Ni Zhang, Zhiguang Cao, Jianan Zhou, Cong Zhang, and Yew-Soon Ong. 2026. An agentic framework with LLMs for solving complex vehicle routing problems. In *International Conference on Learning Representations*.

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. 2026. StepORLM: A self-evolving framework with generative process supervision for operations research language models. In *International Conference on Learning Representations*.

A Problems and Formulations

This appendix details the benchmark problems used in our experiments, including the descriptions, standard formulations, baselines, and the best formulations discovered by FormuEvo. Detailed instance splits in the experiments are shown in Table 5.

Table 5: Instance splits for experiments.

Problem	Category	Configuration
TSP	Easy	30 cities
	Medium	50 cities
	Hard	100 cities
JSSP	Easy	8 jobs, 8 machines
	Medium	10 jobs, 10 machines
	Hard	12 jobs, 12 machines
BPP	Easy	100 items, 150 capacity
	Medium	150 items, 300 capacity
	Hard	200 items, 400 capacity
CFLP	Easy	100 facilities, 200 customers
	Medium	200 facilities, 400 customers
	Hard	300 facilities, 600 customers
QAP	Easy	8 facilities / locations
	Medium	9 facilities / locations
	Hard	10 facilities / locations
NNV	Easy	MLP dim [20, 80, 10]
	Hard	MLP dim [40, 160, 20]
IMO	Easy	Grid size $\{3, 4, 5, 6\}$
	Hard	Grid size $\{7, 8, 9, 10\}$

A.1 Traveling Salesman Problem (TSP)

Problem description

The traveling salesman problem aims to determine an optimal Hamiltonian cycle (route) to visit a set of n cities exactly once and return to the starting city. The problem is defined on a complete graph where each pair of cities (i, j) is associated with a travel distance (or cost) c_{ij}. The decision involves sequencing the visits to ensure that every city is entered exactly once and exited exactly once. The selected route must form a single continuous tour covering all n cities. Any disconnected, smaller loops (subtours) that do not include all cities are strictly prohibited. The objective is to minimize the total travel distance of the complete tour.

Standard MIP formulation

- $V = \{1, 2, \ldots, n\}$: the set of n cities.
- $C = V \setminus \{1\}$: the set of cities except city 1.
- c_{ij}: travel distance from city i to city j.
- x_{ij}: decision variable that equals 1 if and only if the route travels directly from node i to node j.
- u_i: auxiliary ordering variable used for subtour elimination, where $u_1 = 1$.

Below is the standard MTZ formulation for TSP:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n} \sum_{j=1,\, j\neq i}^{n} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{j=1,\, j\neq i}^{n} x_{ij} = 1, \quad \forall i \in V, \\
& \sum_{i=1,\, i\neq j}^{n} x_{ij} = 1, \quad \forall j \in V, \\
& u_i - u_j \leq (n-1)(1 - x_{ij}),\ \forall i, j \in C,\ i \neq j, \\
& x_{ij} \in \{0, 1\}, \quad \forall i, j \in V,\ i \neq j, \\
& 2 \leq u_i \leq n, \quad \forall i \in C.
\end{aligned}
$$

Baseline formulations

- MTZ: The standard formulation using continuous auxiliary sequencing variables to enforce a strictly increasing visit order for subtour elimination, as described above.
- SCF: A flow-based formulation that eliminates subtours by tracking a single continuous flow where the depot dispatches $n - 1$ units and each subsequent city consumes exactly one unit.
- MCF-RLT: An advanced flow-based formulation using reformulation-linearization technique, which linearizes higher-order products of flow and precedence variables, achieving the theoretically tightest known relaxation bound.
- ORLM: Generated by ORLM, same as MTZ.
- StepORLM: Generated by StepORLM, based on MTZ with explicit self-loop elimination.
- EvoCut: A strengthened MTZ with lifted ordering constraints, envelope bounds, and elimination of 2-cycles and depot-associated short cycles.

FormuEvo formulation

- $\bar{c}_{ij} = c_{ij} + c_{ji}$: bidirectional distance of (i, j).
- f_{ij}: non-negative flow variable on arc (i, j), defined for $i, j \in C$, $i \neq j$.

The FormuEvo formulation preserves the global connectivity guarantee of SCF formulation while selectively incorporating small lifted DFJ subtour elimination cuts as local strengthening constraints:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n} \sum_{j=1,\, j\neq i}^{n} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{j=1,\, j\neq i}^{n} x_{ij} = 1, \quad \forall i \in V,
\end{aligned}
$$

$$\sum_{i=1,\, i\neq j}^{n} x_{ij} = 1, \quad \forall j \in V,$$
$$x_{ij} + x_{ji} \leq 1, \quad \forall i, j \in V,\ i < j,$$
$$\sum_{i\in S} \sum_{j\in S,\, j\neq i} x_{ij} \leq |S| - 1, \quad \forall S \in \mathcal{F},$$
$$(n-1)\cdot x_{1j} + \sum_{i\in C\setminus\{j\}} f_{ij} - \sum_{k\in C\setminus\{j\}} f_{jk} = 1, \quad \forall j \in C,$$
$$x_{ij} \leq f_{ij} \leq (n-2)\cdot x_{ij}, \quad \forall i, j \in C,\ i \neq j,$$
$$x_{ij} \in \{0,1\}, \quad \forall i, j \in V,\ i \neq j,$$
$$f_{ij} \geq 0, \quad \forall i, j \in C,\ i \neq j.$$

The subtour elimination family $\mathcal{F}$ is defined as the set of all unique collection consisting of:

- Depot 3-cuts: $\{1, a, b\}, \forall a, b \in C,\ a < b$.
- Local 3-cuts: For each $i \in C$, let $N_K(i)$ be the $K = \min(4, n-2)$ nearest neighbors of i w.r.t. $\bar{c}_{ij}$. For every pair $\{a, b\} \subseteq N_K(i)$, add the set $\{i, a, b\}$ if $|\{i, a, b\}| < n$.
- Depot 4-cuts (if $n > 4$): Let $N_K(1)$ be the $K = \min(6, n-1)$ nearest neighbors of the depot w.r.t. $\bar{c}_{1j}$. For every triple $\{a, b, c\} \subseteq N_K(1)$, add the set $\{1, a, b, c\}$ if $|\{1, a, b, c\}| < n$.
- Local 4-cuts (if $n > 4$): For each $i \in C$ and every triple $\{a, b, c\} \subseteq N_K(i)$, $K = \min(4, n-2)$, add the set $\{i, a, b, c\}$ if $|\{i, a, b, c\}| < n$.

A.2 Job-Shop Scheduling Problem (JSSP)

Problem description

The job-shop scheduling problem aims to determine the optimal scheduling of a set of n jobs on a set of m machines. Each job i consists of m operations, where each operation must be processed on a distinct machine. The processing order of these operations is predefined for each job, representing its technological order. Each machine can process at most one operation at a time, and preemption is strictly prohibited. The decision involves determining the exact start time of each operation such that both machine capacity constraints and job precedence constraints are satisfied. The objective is to minimize the makespan, defined as the completion time of the last operation across all jobs.

Standard MIP formulation

- $J = \{1, \ldots, n\}$: the set of n jobs.
- $M = \{1, \ldots, m\}$: the set of m machines (also the number of operations per job).
- $\mu_{ij} \in M$: the machine that processes the j-th operation of job i.
- $\pi_i(k) \in M$: the index of the operation in job i that executes on machine k, i.e., $\mu_{i,\pi_i(k)} = k$.
- p_{ij}: processing time of j-th operation of job i.
- $z_{i_1 i_2 k}$: binary variable that equals 1 if and only if job i_1 is processed before job i_2 on machine k.
- s_{ij}: decision variable representing the start time of the j-th operation of job i.
- $C_{\max}$: decision variable representing makespan.
- L: a large enough constant, set to $\sum_{i\in J,\, j\in M} p_{ij}$.

Below is the disjunctive formulation for JSSP:

$$\min C_{\max}$$
$$\text{s.t. } s_{i,j+1} \geq s_{ij} + p_{ij},\ \forall i \in J,\ \forall j \in M \setminus \{m\},$$
$$s_{i_1,\pi_{i_1}(k)} + p_{i_1,\pi_{i_1}(k)} \leq s_{i_2,\pi_{i_2}(k)} + L\cdot(1 - z_{i_1 i_2 k}),\ \forall i_1, i_2 \in J,\ i_1 < i_2,\ k \in M,$$
$$s_{i_2,\pi_{i_2}(k)} + p_{i_2,\pi_{i_2}(k)} \leq s_{i_1,\pi_{i_1}(k)} + L\cdot z_{i_1 i_2 k}, \quad \forall i_1, i_2 \in J,\ i_1 < i_2,\ k \in M,$$
$$C_{\max} \geq s_{ij} + p_{ij}, \quad \forall i \in J,\ j \in M,$$
$$z_{i_1 i_2 k} \in \{0,1\},\ \forall i_1, i_2 \in J,\ i_1 < i_2,\ k \in M.$$
$$s_{ij} \geq 0, \quad \forall i \in J,\ j \in M,$$
$$C_{\max} \geq 0.$$

Baseline formulations

- Disj.: The standard disjunctive formulation using big-M constraints for non-overlapping requirements, as described above.
- Enh. Disj.: An enhanced disjunctive formulation that replaces standard big-M inequalities with continuous surplus variables to explicitly track and bound the idle time between operations.
- ORLM: Generated by ORLM, a time-indexed formulation that enforces the precedence and non-overlapping requirements using time constraints.
- StepORLM: Generated by StepORLM, same as the disjunctive formulation.
- EvoCut: A strengthened disjunctive formulation that tightens the linear relaxation with static valid inequalities derived from maximum job lengths and machine-level critical-path bounds.

FormuEvo formulation

- $\underline{s}_{ij} = \sum_{l=1}^{j-1} p_{il}$: lower bound on s_{ij}
- $\sigma_{ij} = \sum_{l=j}^{m} p_{il}$: tail processing time from the j-th operation of job i onward.
- $\bar{s}_{ij} = \max(\underline{s}_{ij}, \hat{C} - \sigma_{ij})$: upper bound on s_{ij}, where $\hat{C}$ is a heuristic makespan.
- $\mu_i = (\mu_{i1}, \ldots, \mu_{im})$: the machine assignment vector of job i.

- $p_i = (p_{i1}, \ldots, p_{im})$: the processing time vector of job i.
- $\omega_j = m - j + 1$: symmetry-breaking weight for operation j.
- $y_{i_1 i_2 k}$: binary disjunctive variable on machine k, defined only for *free* pairs (pairs whose ordering cannot be inferred from bounds).
- $L_{abk} = \bar{s}_{a,\pi_a(k)} + p_{a,\pi_a(k)} - \underline{s}_{b,\pi_b(k)}$: pair-specific tight big-M for precedence $a \to b$.

The FormuEvo formulation strengthens the disjunctive model through tight variable bounds, tail-based makespan constraints, symmetry-breaking constraints, and bound-driven fixing of resolvable disjunctions to reduce binary variables.

$$
\begin{aligned}
\min\ & C_{\max} \\
\text{s.t.}\ & s_{i,j+1} \geq s_{i,j} + p_{ij},\ \forall i \in J,\ \forall j \in M \setminus \{m\}, \\
& C_{\max} \geq s_{i,j} + \sigma_{ij}, \quad \forall i \in J,\ \forall j \in M, \\
& \sum_{j=1}^{m} \omega_j\, s_{i_1,j} \leq \sum_{j=1}^{m} \omega_j\, s_{i_2,j}, \\
& \qquad \forall\, i_1, i_2 \in J \text{ with } \mu_{i_1} = \mu_{i_2},\ p_{i_1} = p_{i_2}, \\
& s_{i_1,\pi_{i_1}(k)} + p_{i_1,\pi_{i_1}(k)} \leq s_{i_2,\pi_{i_2}(k)}, \\
& \qquad \forall k \in M,\ \forall (i_1, i_2) \in \mathcal{P}_k, \\
& s_{i_1,\pi_{i_1}(k)} + p_{i_1,\pi_{i_1}(k)} \leq s_{i_2,\pi_{i_2}(k)} + \\
& L_{i_1 i_2 k} \cdot (1 - y_{i_1 i_2 k}), \forall k \in M,\ \forall (i_1, i_2) \in \mathcal{Y}_k, \\
& s_{i_2,\pi_{i_2}(k)} + p_{i_2,\pi_{i_2}(k)} \leq s_{i_1,\pi_{i_1}(k)} + \\
& \qquad L_{i_2 i_1 k} \cdot y_{i_1 i_2 k}, \forall k \in M,\ \forall (i_1, i_2) \in \mathcal{Y}_k, \\
& \underline{s}_{ij} \leq s_{ij} \leq \bar{s}_{ij}, \quad \forall i \in J,\ \forall j \in M, \\
& y_{i_1 i_2 k} \in \{0, 1\}, \quad \forall k \in M,\ \forall (i_1, i_2) \in \mathcal{Y}_k.
\end{aligned}
$$

where $\mathcal{P}_k$ and $\mathcal{Y}_k$ are defined as follows. For each machine k, consider the operations processed on k and sort them lexicographically by $(\underline{s}_{ij} + p_{ij},\ \bar{s}_{ij})$. For every ordered pair (i_1, j_1) and (i_2, j_2) in this order, if $\underline{s}_{i_1,j_1} + p_{i_1,j_1} > \bar{s}_{i_2,j_2}$, then (i_2, i_1) is added to a fixed precedence relation. Its transitive closure defines a partial order on machine k. Pairs resolved by this order are placed in $\mathcal{P}_k$, while the remaining unresolved pairs form $\mathcal{Y}_k$ and retain the binary big-M disjunction. The heuristic makespan $\hat{C}$ is initialized from the best schedule produced by constructive heuristics, including serial list scheduling and priority dispatch rules.

A.3 Bin Packing Problem (BPP)

Problem description

The bin packing problem aims to determine the optimal assignment of a set of n discrete items into a set of identical bins. Each bin is homogeneous with a fixed, deterministic maximum weight capacity W. Each item i has a predefined weight w_i, and all item weights are assumed to be positive integers. The packing decision involves determining exactly which bin each item is placed into. Every item must be assigned to exactly one bin. For any bin that is used, the cumulative weight of all items assigned to it cannot exceed W. Sufficient bins are assumed to be available, bounded by a maximum possible number m (typically $m = n$). The objective is to minimize the total number of bins actively used to accommodate all n items.

Standard MIP formulation

Notation

- $I = \{1, 2, \ldots, n\}$: the set of n items.
- $B = \{1, 2, \ldots, m\}$: the set of m available bins.
- w_i: weight of item i.
- W: maximum weight capacity of each bin.
- x_{ij}: binary variable that equals 1 if and only if item i is placed into bin j.
- y_j: binary variable that equals 1 if and only if bin j is actively used.

Below is the Kantorovich formulation for BPP:

$$
\begin{aligned}
\min\ & \sum_{j=1}^{m} y_j \\
\text{s.t.}\ & \sum_{j=1}^{m} x_{ij} = 1, \quad \forall i \in I, \\
& \sum_{i=1}^{n} w_i x_{ij} \leq W y_j, \quad \forall j \in B, \\
& x_{ij} \in \{0, 1\}, \quad \forall i \in I,\ \forall j \in B, \\
& y_j \in \{0, 1\}, \quad \forall j \in B.
\end{aligned}
$$

Baseline formulations

- Kant.: The Kantorovich assignment formulation, as described above.
- AF: A pseudo-polynomial formulation that reformulates BPP as a flow network problem.
- VPSolver: An AF-style formulation with graph compression and side constraints.
- ORLM: Generated by ORLM, same as Kant.
- StepORLM: Generated by StepORLM, based on Kantorovich formulation with symmetry-breaking constraints and objective bounds.
- EvoCut: A strengthened Kantorovich variant that breaks symmetry via lexicographical ordering inequalities, continuous weight bounds, and index-based assignment restrictions for sorted items.

FormuEvo formulation

Notation

- $g = \gcd(W, w_1, \ldots, w_n)$: greatest common divisor (GCD) of W and all weights w_i.
- $\mathcal{W} = \{\bar{w}_1, \ldots, \bar{w}_K\}$: set of distinct item weights after GCD scaling (W and w_i are divided by g).
- b_w: the demand quantity of items with scaled weight $w \in \mathcal{W}$.
- B_0: the number of items whose weight equals the capacity W, which exactly fill a bin, and are pre-packed and excluded from the problem.
- $\mathcal{S} = \{0 = s_0 < s_1 < \cdots < s_T \leq W\}$: the sorted set of valid cumulative bin loads (nodes in the flow network).
- $x_{w,s}$: item flow variable representing the number of items of weight w placed into bins that currently have an accumulated load of state s.
- ℓ_t: loss flow variable representing the unused bin capacity between states s_t and s_{t+1}.
- z: variable representing the number of bins required to pack all items lighter than W.

The FormuEvo formulation constructs a directed acyclic graph where valid residual capacities are compressed via subset-sum filtering. Loss arcs (ℓ_t) ensure flow conservation without requiring explicit bin indexing, while z and ℓ_t are naturally integers.

$$
\begin{aligned}
\min \;\; & B_0 + z \\
\text{s.t.} \;\; & \sum_{w:\, w \in \mathcal{S}} x_{w,0} = z, \\
& \sum_{w:\, s_t - w \in \mathcal{S}} x_{w,\, s_t - w} + \ell_{t-1} = \sum_{w:\, s_t + w \in \mathcal{S}} x_{w, s_t} + \ell_t, \\
& \qquad\qquad \forall t = 1, \ldots, T-1, \\
& \sum_{w:\, W - w \in \mathcal{S}} x_{w,\, W-w} + \ell_{T-1} = z, \\
& \sum_{s:\, s + w \in \mathcal{S}} x_{w,s} = b_w, \quad \forall w \in \mathcal{W}, \\
& \underline{z} \leq z \leq \overline{z}, \\
& x_{w,s} \in \mathbb{Z}_{\geq 0}, \quad \forall w \in \mathcal{W},\ s \in \mathcal{S},\ s + w \in \mathcal{S}, \\
& \ell_t \geq 0, \quad \forall t = 1, \ldots, T-1.
\end{aligned}
$$

where $\underline{z}, \overline{z}$ are valid lower and upper bounds obtained from capacity-based bounds and a first-fit decreasing heuristic. The state set $\mathcal{S}$ is generated by subset-sum reachability over the scaled item sizes. Starting from 0, every reachable load $s + w$ with $s \in \mathcal{S}$ and $w \in \mathcal{W}$ is iteratively added whenever $s + w \leq W$. Only transitions between reachable states are retained in the compressed network.

A.4 Capacitated Facility Location Problem (CFLP)

Problem description

The capacitated facility location problem aims to determine the optimal subset of facilities to open from a set of m candidate locations and the corresponding shipment quantities to satisfy the demands of n customers. Each facility i incurs a fixed opening cost f_i and has a maximum supply capacity C_i. Each customer j has a demand d_j that must be fully met. It is assumed that the total available capacity is sufficient to satisfy total demand. Shipping one unit from location i to customer j incurs a transportation cost c_{ij}, and a facility can supply customers only if it is opened. The decision involves selecting which facilities to open and determining the exact shipment quantity from each opened facility to each customer such that all demands are met without exceeding any facility's capacity. The objective is to minimize the total cost, comprising the sum of fixed opening costs and variable transportation costs.

Standard MIP formulation

Notation

- $I = \{1, 2, \ldots, m\}$: the set of m locations.
- $J = \{1, 2, \ldots, n\}$: the set of n customers.
- f_i: fixed cost of opening facility i.
- C_i: maximum supply capacity of facility i.
- d_j: demand of customer j.
- c_{ij}: unit transportation cost from facility i to customer j.
- y_i: binary variable that equals 1 if and only if facility i is opened.
- x_{ij}: continuous variable representing the shipment quantity from facility i to customer j.

The standard MIP formulation is as follows.

$$
\begin{aligned}
\min \;\; & \sum_{i=1}^{m} f_i y_i + \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \\
\text{s.t.} \;\; & \sum_{i=1}^{m} x_{ij} = d_j, \quad \forall j \in J, \\
& \sum_{j=1}^{n} x_{ij} \leq C_i y_i, \quad \forall i \in I, \\
& y_i \in \{0, 1\}, \quad \forall i \in I, \\
& x_{ij} \geq 0, \quad \forall i \in I,\ \forall j \in J.
\end{aligned}
$$

Baseline formulations

- Agg.: The aggregated formulation using facility-level capacity constraints, as described above.

- Disagg.: A disaggregated formulation that replaces the capacity constraint with per-customer linking inequalities for a tighter LP relaxation.
- ORLM: Generated by ORLM, same as the aggregated formulation.
- StepORLM: Generated by StepORLM, based on the aggregated formulation with explicitly decomposed demand constraints.
- EvoCut: A strengthened disaggregated formulation with tightened variable upper bounds, single-customer coverage inequalities, global capacity bounds, minimum-open-facility bounds, and subset coverage cuts for the largest customers.

FormuEvo formulation

Notation

- $u_{ij} = \min(C_i,\, d_j)$: an upper bound on x_{ij}.
- $\mathcal{L}_j = \{i \in I \mid C_i \geq d_j\}$: set of large facilities whose individual capacity can satisfy demand d_j.

FormuEvo tightens the formulation through per-arc upper bounds, global and per-customer cover inequalities, minimum-open-count constraints, and symmetry breaking based on dominance.

$$
\begin{aligned}
\min\ & \sum_{i=1}^{m} f_i y_i + \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij} x_{ij} \\
\text{s.t.}\ & \sum_{i=1}^{m} x_{ij} = d_j, \quad \forall j \in J, \\
& \sum_{j=1}^{n} x_{ij} \leq C_i y_i, \quad \forall i \in I, \\
& \sum_{i=1}^{m} C_i y_i \geq \sum_{j=1}^{n} d_j, \\
& \sum_{i \in \mathcal{L}_j} C_i y_i \geq d_j - \sum_{i \in I \setminus \mathcal{L}_j} \min(C_i, d_j),\ \forall j \in J, \\
& \sum_{i=1}^{m} y_i \geq \left\lceil \frac{\sum_{j=1}^{n} d_j}{\max_{i \in I} C_i} \right\rceil, \\
& y_k \leq y_i, \quad \forall\, i, k \in I \text{ with } C_i \geq C_k,\ f_i \leq f_k, \\
& \qquad\qquad \text{and } c_{ij} \leq c_{kj}, \forall j \in J, \\
& 0 \leq x_{ij} \leq u_{ij}, \quad \forall i \in I,\ \forall j \in J, \\
& y_i \in \{0,1\}, \quad \forall i \in I.
\end{aligned}
$$

A.5 Quadratic Assignment Problem (QAP)

Problem description

The quadratic assignment problem aims to determine the optimal one-to-one assignment of a set of n facilities to a set of n distinct locations. Each facility must be assigned to exactly one location, and each location must be occupied by exactly one facility, establishing a strict bijective mapping. There is a deterministic flow f_{ij} between every pair of facilities i and j, and a deterministic distance d_{kl} between every pair of locations k and l. The decision involves determining a bijective assignment between facilities and locations. The objective is to minimize the total interaction cost, computed as the sum of flow–distance products induced by all assigned facility pairs.

Standard MIP formulation

Notation

- $N = \{1, \ldots, n\}$: the set of facilities / locations.
- f_{ij}: flow (interaction weight) between facility i and facility j.
- d_{kl}: distance between location k and location l.
- x_{ik}: binary variable that equals 1 if and only if facility i is assigned to location k.

Below is the standard Koopmans–Beckmann binary quadratic formulation:

$$
\begin{aligned}
\min\ & \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} f_{ij}\, d_{kl}\, x_{ik}\, x_{jl} \\
\text{s.t.}\ & \sum_{k=1}^{n} x_{ik} = 1, \quad \forall i \in N, \\
& \sum_{i=1}^{n} x_{ik} = 1, \quad \forall k \in N, \\
& x_{ik} \in \{0,1\}, \quad \forall i, k \in N.
\end{aligned}
$$

Baseline formulations

- K-B Quad.: The standard Koopmans–Beckmann binary quadratic formulation, as described above.
- McC. Lin.: A classical linearization that introduces auxiliary variables y_{ijkl} to represent products $x_{ik}x_{jl}$ and enforces equivalence via standard McCormick envelope inequalities.
- AJ Lin.: The Adams–Johnson linearization that lifts continuous y_{ijkl} variables with tight linking constraints, yielding a stronger relaxation.
- XY-KB Lin.: The Xia–Yuan linearization that precomputes minimum-assignment lower and upper bounds, and then applies auxiliary variables with tightening lower-bound constraints.
- IPQAPR: A compact linearization that reduces the dimension of y_{ijkl} by exploiting symmetry and enforces cross-block consistency constraints.
- StepORLM: Generated by StepORLM, same as McC.

- EvoCut: Same as K-B formulation, since the Birkhoff polytope already represents the convex hull of the assignment constraints.

FormuEvo formulation

Notation

- z_{ij}^{kl}: lifted continuous variable in $[0,1]$, representing $x_{ik}x_{jl}$ for $i<j$ and $k \neq l$.
- $\mathcal{G}_F$: set of unordered facility pairs (i,j) with $i<j$ whose flow rows and columns are identical, i.e., $f_{i,\cdot} = f_{j,\cdot}$ and $f_{\cdot,i} = f_{\cdot,j}$.
- $\mathcal{G}_D$: set of unordered location pairs (k,l) with $k<l$ whose distance rows and columns are identical, i.e., $d_{k,\cdot} = d_{l,\cdot}$ and $d_{\cdot,k} = d_{\cdot,l}$.

FormuEvo uses a transportation-style linearization that lifts each unordered facility pair (i,j) into a separate z-block. Cross-block consistency is enforced by anchor-slice coupling, and symmetry is broken via cumulative prefix inequalities.

$$
\begin{aligned}
\min\ & \sum_{i=1}^{n}\sum_{k=1}^{n} f_{ii}d_{kk}\, x_{ik} + \\
& \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{n}\sum_{l=1,\, l\neq k}^{n} \left(f_{ij}d_{kl} + f_{ji}d_{lk}\right) z_{ij}^{kl} \\
\text{s.t.}\ & \sum_{k=1}^{n} x_{ik} = 1, \quad \forall i \in N, \\
& \sum_{i=1}^{n} x_{ik} = 1, \quad \forall k \in N, \\
& \sum_{l=1,\, l\neq k}^{n} z_{ij}^{kl} = x_{ik}, \quad \forall i<j,\ k \in N, \\
& \sum_{k=1,\, k\neq l}^{n} z_{ij}^{kl} = x_{jl}, \quad \forall i<j,\ l \in N, \\
& \sum_{j=i+1}^{n} z_{ij}^{kl} + \sum_{j=1}^{i-1} z_{ji}^{lk} = x_{ik}, \\
& \qquad \forall i \in N,\ k,l \in N,\ k \neq l, \\
& \sum_{r=1}^{t} x_{ir} \geq \sum_{r=1}^{t} x_{jr}, \\
& \qquad \forall \{i,j\} \in \mathcal{G}_F,\ i<j,\ t \in N, \\
& \sum_{r=1}^{t} x_{rk} \geq \sum_{r=1}^{t} x_{rl}, \\
& \qquad \forall \{k,l\} \in \mathcal{G}_D,\ k<l,\ t \in N, \\
& x_{ik} \in \{0,1\}, \quad \forall i,k \in N, \\
& 0 \leq z_{ij}^{kl} \leq 1, \quad \forall i<j,\ k,l \in N,\ k \neq l.
\end{aligned}
$$

A.6 Neural Network Verification (NNV)

Problem description

The neural network verification problem aims to determine the minimum adversarial distortion required to induce a targeted misclassification in a pre-trained multi-layer perceptron (MLP) classifier. Given a correctly classified input sample $x_0 \in [0,1]^{n_0}$ with true label λ, the goal is to find a perturbed input $x \in [0,1]^{n_0}$ that causes the network to predict a specified wrong label $\mu \neq \lambda$, while minimizing the ℓ_1 distance between x and x_0. The network consists of L layers, where each hidden layer $k \in \{1,\ldots,L-1\}$ applies a linear transformation followed by a ReLU activation, and the output layer L applies a linear transformation only. The adversarial condition requires that the output logit of μ is at least that of λ. The objective is to minimize the ℓ_1 distortion $\sum_i |x_i - x_{0,i}|$.

Standard MIP formulation

Notation

- n_0: dimension of the input space.
- L: total number of (hidden + output) layers.
- n_k: number of neurons in layer k ($k = 1,\ldots,L$).
- $W^{(k)} \in \mathbb{R}^{n_k \times n_{k-1}}$: weight matrix of layer k.
- $b^{(k)} \in \mathbb{R}^{n_k}$: bias vector of layer k.
- $x_0 \in [0,1]^{n_0}$: original input, classified as λ.
- $\mu \neq \lambda$: target adversarial label.
- x_i: perturbed input component, $x_i \in [0,1]$.
- d_i: auxiliary variable for linearizing $|x_i - x_{0,i}|$.
- $\hat{z}_j^{(k)}$: pre-activation of neuron j in layer k.
- $z_j^{(k)}$: post-activation of neuron j in layer k.
- $a_j^{(k)} \in \{0,1\}$: ReLU indicator for neuron j in layer k; $a=0$ encodes the active branch ($\hat{z} \geq 0$), $a=1$ encodes the inactive branch ($\hat{z} \leq 0$).
- $M_j^{(k)}$: per-neuron big-M bound obtained via interval arithmetic over $x \in [0,1]^{n_0}$.

$$
\begin{aligned}
\min\ & \sum_{i=1}^{n_0} d_i \\
\text{s.t.}\ & d_i \geq x_i - x_{0,i}, \quad d_i \geq x_{0,i} - x_i, \\
& \qquad \forall i = 1,\ldots,n_0, \\
& \hat{z}_j^{(k)} = \sum_{i=1}^{n_{k-1}} W_{ji}^{(k)} z_i^{(k-1)} + b_j^{(k)}, \\
& \qquad \forall k = 1,\ldots,L,\ j = 1,\ldots,n_k, \\
& z_j^{(k)} \leq \hat{z}_j^{(k)} + M_j^{(k)} a_j^{(k)}, \\
& \qquad \forall k = 1,\ldots,L-1,\ j = 1,\ldots,n_k, \\
& z_j^{(k)} \geq \hat{z}_j^{(k)},
\end{aligned}
$$

$$\forall k = 1, \dots, L-1,\ j = 1, \dots, n_k,$$

$$z_j^{(k)} \le M_j^{(k)}(1 - a_j^{(k)}),$$

$$\forall k = 1, \dots, L-1,\ j = 1, \dots, n_k,$$

$$z_j^{(k)} \ge 0,\ \forall k = 1, \dots, L-1,\ j = 1, \dots, n_k,$$

$$\hat{z}_\mu^{(L)} \ge \hat{z}_\lambda^{(L)},$$

$$0 \le x_i \le 1, \quad \forall i = 1, \dots, n_0,$$

$$a_j^{(k)} \in \{0, 1\},$$

$$\forall k = 1, \dots, L-1,\ j = 1, \dots, n_k,$$

where $z^{(0)} = x$ (i.e., $z_i^{(0)} = x_i$), $z^{(L)} = \hat{z}^{(L)}$ (the output layer has no ReLU).

Baseline formulations

- Std. Formu.: The standard formulation as above.
- EvoCut: A strengthened variant that adds triangle relaxation inequalities, tightening the LP relaxation over the big-M envelope.

FormuEvo formulation

Notation

- $d_i^+,\ d_i^-$: split ℓ_1 perturbation components, with $d_i^+ \in [0,\ 1 - x_{0,i}]$, $d_i^- \in [0,\ x_{0,i}]$; the perturbed input is $x_i = x_{0,i} + d_i^+ - d_i^-$ and the distortion is $d_i^+ + d_i^- = |x_i - x_{0,i}|$.
- $l_j^{(k)}$: pre-computed lower bounds on $\hat{z}_j^{(k)}$.
- $u_j^{(k)}$: pre-computed upper bounds on $\hat{z}_j^{(k)}$.

The formulation uses the split ℓ_1 variables d_i^+, d_i^- and per-neuron bounds $l_j^{(k)}, u_j^{(k)}$.

$$\min \sum_{i=1}^{n_0} \left(d_i^+ + d_i^-\right)$$

$$\text{s.t. } 0 \le d_i^+ \le 1 - x_{0,i},\ 0 \le d_i^- \le x_{0,i},$$

$$\forall i = 1, \dots, n_0,$$

$$z_i^{(0)} = x_{0,i} + d_i^+ - d_i^-, \quad \forall i = 1, \dots, n_0,$$

$$\hat{z}_j^{(k)} = \sum_{i=1}^{n_{k-1}} W_{ji}^{(k)} z_i^{(k-1)} + b_j^{(k)},$$

$$\forall k = 1, \dots, L,\ j = 1, \dots, n_k,$$

$$z_j^{(k)} \ge \hat{z}_j^{(k)},$$

$$\forall k = 1, \dots, L-1,\ j = 1, \dots, n_k,$$

$$z_j^{(k)} \le \hat{z}_j^{(k)} - l_j^{(k)} a_j^{(k)},$$

$$\forall k = 1, \dots, L-1,\ j = 1, \dots, n_k,$$

$$z_j^{(k)} \le u_j^{(k)}(1 - a_j^{(k)}),$$

$$\forall k = 1, \dots, L-1,\ j = 1, \dots, n_k,$$

$$z_j^{(k)} \ge 0,\ \forall k = 1, \dots, L-1,\ j = 1, \dots, n_k,$$

$$\hat{z}_\mu^{(L)} \ge \hat{z}_\lambda^{(L)},$$

$$d_i^+,\ d_i^- \ge 0, \quad \forall i = 1, \dots, n_0,$$

$$a_j^{(k)} \in \{0, 1\},$$

$$\forall k = 1, \dots, L-1,\ j = 1, \dots, n_k.$$

A.7 IMO 2025 Problem 6 (IMO)

Problem description

The IMO 2025 Problem 6 asks to cover an $n \times n$ grid of unit squares using non-overlapping axis-aligned rectangular tiles, such that exactly one unit square remains uncovered (a "hole") in each row and each column. This problem can be modeled as an MIP. The objective is to minimize the total number of rectangles used. Each rectangle spans a contiguous interval of columns over a contiguous interval of rows. The decision involves selecting the hole positions and determining how rectangles are placed so that every non-hole cell is covered exactly once. The formulation additionally tracks the vertical continuation of horizontal intervals across adjacent rows, allowing rectangles to be represented compactly through interval propagation and flow-balance relations.

Standard MIP formulation

Notation

- n: grid size ($n \times n$).
- $R = \{1, \dots, n\}$: the set of row indices.
- $C = \{1, \dots, n\}$: the set of column indices.
- $\mathcal{I} = \{(a, b) \mid 1 \le a \le b \le n\}$: the set of all contiguous column intervals.
- $h_{ij} \in \{0, 1\}$: equals 1 if and only if the hole in row i is located at column j.
- $x_{iab} \in \{0, 1\}$: equals 1 if and only if interval (a, b) is active on row i, i.e., columns $[a, \dots, b]$ are covered by the same rectangle on row i.
- $s_{iab} \in \{0, 1\}$: equals 1 if and only if a rectangle with horizontal span (a, b) starts at row i.
- $t_{iab} \in \{0, 1\}$: equals 1 if and only if a rectangle with horizontal span (a, b) ends at row i.

$$\min \sum_{i=1}^{n} \sum_{(a,b) \in \mathcal{I}} s_{iab}$$

$$\text{s.t. } \sum_{j=1}^{n} h_{ij} = 1, \quad \forall i \in R,$$

$$\sum_{i=1}^{n} h_{ij} = 1, \quad \forall j \in C,$$

$$\sum_{(a,b)\in\mathcal{I}:\, a\le j\le b} x_{iab} + h_{ij} = 1, \quad \forall i \in R,\ j \in C,$$
$$x_{1ab} - s_{1ab} = 0, \quad \forall (a,b) \in \mathcal{I},$$
$$x_{iab} - x_{i-1,ab} - s_{iab} + t_{i-1,ab} = 0, \quad \forall i = 2,\dots,n,\ (a,b) \in \mathcal{I},$$
$$x_{nab} - t_{nab} = 0, \quad \forall (a,b) \in \mathcal{I},$$
$$h_{ij} \in \{0,1\}, \quad \forall i \in R,\ j \in C,$$
$$x_{iab},\ s_{iab},\ t_{iab} \in \{0,1\},\ \forall i \in R,\ (a,b) \in \mathcal{I}.$$

Baseline formulations

- Std. Formu.: The standard formulation as above.
- EvoCut: A variant with cuts enforcing mandatory new interval starts when hole movement disrupts coverage continuity across rows.

FormuEvo formulation

Notation

- $\mathcal{R} = \{(i_1, i_2, a, b) \mid 1 \le i_1 \le i_2 \le n,\ (a,b) \in \mathcal{I}\}$: the set of all axis-aligned subrectangles.
- $y_{i_1 i_2 ab} \in \{0,1\}$: equals 1 if and only if a rectangle spanning rows i_1 through i_2 and columns a through b is selected.
- $c_{ij} \in [0,1]$: variable representing the number of selected rectangles covering cell (i,j).
- $\mathcal{R}_{ij}^{\mathrm{TL}} = \{(i_1, i_2, a, b) \in \mathcal{R} \mid i_1 = i,\ a = j\}$: rectangles whose top-left corner is at (i,j).
- $\mathcal{R}_{ij}^{\mathrm{TR}} = \{(i_1, i_2, a, b) \in \mathcal{R} \mid i_1 = i,\ b = j-1\}$ $(j > 1)$: rectangles whose top row is i and rightmost column is $j-1$.
- $\mathcal{R}_{ij}^{\mathrm{BL}} = \{(i_1, i_2, a, b) \in \mathcal{R} \mid i_2 = i-1,\ a = j\}$ $(i > 1)$: rectangles whose bottom row is $i-1$ and leftmost column is j.
- $\mathcal{R}_{ij}^{\mathrm{BR}} = \{(i_1, i_2, a, b) \in \mathcal{R} \mid i_2 = i-1,\ b = j-1\}$ $(i, j > 1)$: rectangles whose bottom-right corner is at $(i-1, j-1)$.

FormuEvo formulation directly models each possible axis-aligned rectangle as a binary variable, then reconstructs cell occupancy via a 2D difference array. Strong aggregated cuts enforce exact cover by projecting onto rows, columns, and the total area.

$$\min \sum_{(i_1,i_2,a,b)\in\mathcal{R}} y_{i_1 i_2 ab}$$
$$\text{s.t.} \sum_{j=1}^{n} h_{ij} = 1, \quad \forall i \in R,$$
$$\sum_{i=1}^{n} h_{ij} = 1, \quad \forall j \in C,$$
$$c_{ij} = c_{i-1,j} + c_{i,j-1} - c_{i-1,j-1} + \sum_{\mathcal{R}_{ij}^{\mathrm{TL}}} y_{i_1 i_2 ab} - \sum_{\mathcal{R}_{ij}^{\mathrm{TR}}} y_{i_1 i_2 ab} - \sum_{\mathcal{R}_{ij}^{\mathrm{BL}}} y_{i_1 i_2 ab} + \sum_{\mathcal{R}_{ij}^{\mathrm{BR}}} y_{i_1 i_2 ab}, \quad \forall i \in R,\ j \in C,$$
$$c_{ij} + h_{ij} = 1, \quad \forall i \in R,\ j \in C,$$
$$\sum_{(i_1,i_2,a,b)\in\mathcal{R}:\, i_1\le i\le i_2} (b - a + 1)\, y_{i_1 i_2 ab} = n - 1, \quad \forall i \in R,$$
$$\sum_{(i_1,i_2,a,b)\in\mathcal{R}:\, a\le j\le b} (i_2 - i_1 + 1)\, y_{i_1 i_2 ab} = n - 1, \quad \forall j \in C,$$
$$\sum_{(i_1,i_2,a,b)\in\mathcal{R}} (i_2 - i_1 + 1)(b - a + 1)\, y_{i_1 i_2 ab} = n(n-1),$$
$$y_{i_1 i_2 ab} \in \{0,1\}, \quad \forall (i_1, i_2, a, b) \in \mathcal{R},$$
$$c_{ij} \in \{0,1\}, \quad \forall i \in R,\ j \in C.$$

where $c_{0,j} = c_{i,0} = c_{0,0} = 0$, and all summations over $\mathcal{R}_{ij}^{\cdot}$ involving out-of-bounds indices are defined to be zero. The reconstruction of c_{ij} follows the standard 2D difference-array scheme: each rectangle variable $y_{i_1 i_2 ab}$ induces a unit increment over its support via corner updates $+1$ at (i_1, a), -1 at $(i_1, b{+}1)$, -1 at $(i_2{+}1, a)$, and $+1$ at $(i_2{+}1, b{+}1)$. Consequently, the resulting prefix-sum transformation yields $c_{ij} = 1$ if and only if cell (i,j) lies within exactly one selected rectangle.

B Implementation Details of FormuEvo

Algorithm 1 outlines the framework of FormuEvo. Below, we present a reference of our prompts used in FormuEvo. All complete prompts are available in our code repository at https://github.com/Xyz-yuanhf/formuevo.

Generator LLM for Initialization

For initialization, we use the prompt for the generator LLM to produce the initial population.

System Prompt

You are an expert in mathematical optimization, mixed-integer programming (MIP), and Gurobi modeling. You specialize in translating the problem description and its preliminary Gurobi models into mathematically correct, high-efficiency, and numerically stable Gurobi models implemented in Python using 'gurobipy'.
The user will provide:

- Problem description: The real-world logic and constraints of the optimization problem to be modeled.

Algorithm 1 Pseudo-code of FormuEvo

Require: Problem description p, formulation template f_0, population size N, generations T, mutation rate ρ, memory usage rate γ, external knowledge base $\mathcal{B}_{\text{init}}$ (optional).

Ensure: Final MIP formulation $\hat{f}^\star$, distilled knowledge base $\mathcal{B}$.

Initialize population $\mathcal{P}_0 \leftarrow \varnothing$
Initialize memory library $\mathcal{M} \leftarrow \mathcal{B}_{\text{init}}$ (if $\mathcal{B}_{\text{init}}$ is available, $\varnothing$ otherwise)
// Initialization: generate initial population
while $|\mathcal{P}_0| < N$ **do**
  $f_{\text{init}} \leftarrow \textsc{GeneratorLLM}(p,\ f_0)$ *// Initial formulation*
  $\mathcal{P}_0 \leftarrow \mathcal{P}_0 \cup \{\textsc{EvaluateAndRepair}(f_{\text{init}})\}$ *// Evaluate and repair*
end while
// Update best formulation
$\hat{f}^\star \leftarrow \arg\min_{f \in \mathcal{P}_0} \phi(f)$
// Evolution: repeat for T generations
for $t = 1$ **to** T **do**
  $\mathcal{O} \leftarrow \varnothing$ *// Offspring pool*
  while $|\mathcal{O}| < N$ **do**
    // Crossover operation
    Sample two parents $f_i,\ f_j$ from $\mathcal{P}_{t-1}$
    // Diagnosis of parents
    if $u < \gamma$ where $u \sim U(0,1)$ **then**
      $d_{ij} \leftarrow \textsc{DiagnosticLLM}(f_i,\ \text{stats}(f_i),\ f_j,\ \text{stats}(f_j),\ \mathcal{M})$ *// Diagnosis of parents*
    else
      $d_{ij} \leftarrow \textsc{DiagnosticLLM}(f_i,\ \text{stats}(f_i),\ f_j,\ \text{stats}(f_j))$ *// Without memory*
    end if
    $f_o \leftarrow \textsc{GeneratorLLM}(p,\ f_i,\ f_j,\ d_{ij})$ *// Offspring by crossover*
    $\mathcal{O} \leftarrow \mathcal{O} \cup \{\textsc{EvaluateAndRepair}(f_o)\}$ *// Evaluate and repair*
    $\mathcal{M} \leftarrow \textsc{ReflectorLLM}(f_i,\ \text{stats}(f_i),\ f_j,\ \text{stats}(f_j),\ f_o,\ \text{stats}(f_o),\ \mathcal{M})$ *// Update memory*
    // Mutation operation
    if $|\mathcal{O}| < N$ **and** $u < \rho$ where $u \sim U(0,1)$ **then**
      Sample elite f_e from $\mathcal{P}_{t-1}$
      if $u < \gamma$ where $u \sim U(0,1)$ **then**
        $d_e \leftarrow \textsc{DiagnosticLLM}(f_e,\ \text{stats}(f_e),\ \mathcal{M})$ *// Diagnosis of elite*
      else
        $d_e \leftarrow \textsc{DiagnosticLLM}(f_e,\ \text{stats}(f_e))$ *// Without memory*
      end if
      $f_o \leftarrow \textsc{GeneratorLLM}(p,\ f_e,\ d_e)$ *// Offspring by mutation*
      $\mathcal{O} \leftarrow \mathcal{O} \cup \{\textsc{EvaluateAndRepair}(f_o)\}$ *// Evaluate and repair*
      $\mathcal{M} \leftarrow \textsc{ReflectorLLM}(f_e,\ \text{stats}(f_e),\ f_o,\ \text{stats}(f_o),\ \mathcal{M})$ *// Update memory*
    end if
  end while
  // Select top-N individuals for next generation
  $\mathcal{P}_t \leftarrow \textsc{TopN}(\mathcal{P}_{t-1} \cup \mathcal{O},\ N)$
  // Update best formulation
  $\hat{f}^\star \leftarrow \arg\min(\phi(\hat{f}^\star),\ \min_{f \in \mathcal{O}} \phi(f))$
end for
// Knowledge distillation for transfer
$\mathcal{B} \leftarrow \textsc{DistillerLLM}(\mathcal{M})$ *// Distill knowledge from memory*
return $\hat{f}^\star,\ \mathcal{B}$

- Python template: A Python 'gurobipy' function template. You must preserve the function name and input arguments exactly. You may refactor the internal implementation freely.
- Memory library (if available): A collection of experience triplets (condition → strategy → effect) distilled from past evolution experiments.

Your task is:
- Generate a better (i.e., mathematically correct, high-efficiency, and numerically stable) Gurobi model than the preliminary one.
- Your generated model should be implemented in Python 'gurobipy' and follow the standard Gurobi modeling practices. Ensure all variables, constraints, and objective functions are well-defined.
- Your generated model must be mathematically correct and designed to solve as efficiently as possible under Gurobi's default parameters (do not rely on parameter tuning).
- To achieve this you should apply advanced modeling techniques where appropriate, including but not limited to {example_strengthing_techniques}.
- You should produce clean, well-structured, and executable Python code using 'gurobipy'. Keep code concise but readable.
- You must provide a concise explanation of the optimization ideas and key improvements in the 'idea' section. Focus on your enhancement motivation and why the new formulation is computationally superior.

Critical constraints:
- Do not change the function signature.
- Do not rely on solver parameter tuning. However, setting parameters strictly required for the model to be accepted by Gurobi is allowed if your formulation necessitates it.
- Do not assume missing data unless clearly implied; make minimal necessary assumptions.
- If ambiguity exists, choose the most standard and computationally efficient formulation.
- Calling 'model.optimize()' or any other solving/execution functions is strictly forbidden within the function. The function must return the model object before any optimization is performed.
- Do not use Gurobi callbacks in any form. If your formulation would naturally rely on callbacks, you must replace it with a static, callback-free alternative.

Response format:
- Return only a valid JSON object in the exact structure below ({detailed_format_instruction}):

```
{
"code": "```python\n<A Python (gurobipy) function implementing the improved Gurobi model>\n```",
"idea": "A concise but technically clear idea of key modeling improvements and any non-trivial reformulations"
}
```

User Prompt

Problem description:
{problem_description}

Python template:
{python_template}

Memory library (if available):
{memory_library}

Diagnostic LLM

We present the prompt of the diagnostic LLM for mutation. The prompt for crossover follows the same structure but operates on a formulation pair.

System Prompt

You are an expert in mathematical optimization, mixed-integer programming (MIP), Gurobi modeling, and structural evolution. Your role is to examine the solver performance profile of a given formulation and produce a concise, actionable diagnostic report that will guide a downstream mutation agent in evolving this formulation into a computationally superior variant.

The user will provide:
- Problem description: The real-world logic and constraints of the optimization problem to be modeled.
- Current formulation data: You will receive its Python 'gurobipy' code and the mathematical idea behind it.
- Solver performance profile, including:
  num_vars: Number of variables in the model.
  num_constrs: Number of constraints in the model.
  root_bound: Average relaxation bound at root node.
  root_gap: Average root relaxation gap (%) = |opt_obj - root_bound| / |opt_obj| * 100.
  node_count: Average number of Branch-and-Bound nodes explored.
  presolve_rows_removed: Average number of rows (constraints) removed by Gurobi presolve.
  presolve_cols_removed: Average number of columns (variables) removed by Gurobi presolve.
  presolve_bounds_changed: Average number of variable bounds changed by Gurobi presolve.
- Memory library (if available): A collection of experience triplets (condition → strategy → effect) distilled from past evolution experiments.

Your task is:
- Perform a bottleneck identification: Determine the primary computational bottleneck of the current formulation. Use, but not limited to, the following diagnostic logic: {example_diagnostic_logic}
- Identify secondary bottlenecks (if possible): After identifying the primary bottleneck, note any secondary issues that could become the new bottleneck once the primary is resolved. This helps mutation avoid shifting the problem without actually improving it.
- Report a concise mutation guidance: Based on your diagnosis, produce specific, actionable recommendations for mutation. Your recommendations should: State the single most impactful mutation direction. Provide 1-3 specific MIP modeling techniques to apply, ranked by expected impact. Indicate what the mutation should preserve from the current formulation. Suggest the expected trade-off. Warn against the potential risks of the recommendation. You can retrieve the memory library for entries whose condition matches the diagnosed conditions.

Output format:
Return a structured diagnostic report in the format:
Formulation profile:
- Size: [num_vars] variables, [num_constrs] constraints
- Relaxation: root_gap = [value]%
- Branching: node_count = [value]
- Presolve: [rows_removed] rows removed, [cols_removed] cols removed, [bounds_changed] bounds changed
- Overall assessment: [one-sentence summary of where time is being spent]

Primary bottleneck:
- Category: [weak relaxation / excessive branching / per-node cost / loose bounds / model size]
- Evidence: [cite specific metric values]

Secondary bottleneck(s):
- Category: [description, or "none identified."]

Mutation recommendations (priority-ranked):
- The most impactful technique with specific detail
- Second technique
- Third technique, if applicable

Preserve (do not degrade):
- Aspects of current formulation that are strong and must be maintained

Expected trade-offs:
- What metrics should improve and what might worsen

Risk warnings:
- Potential pitfalls of the mutation, or "none identified."

User Prompt

Problem description:
{problem_description}

Current formulation:
{current_formulation_data}

Runtime:
{runtime_profile}

Solver profile:
{solver_profile}

Memory library (if available):
{memory_library}

Crossover and Mutation

We employ four types of crossover operators: complement, hybrid, intersection, and min-violation, and four types of mutation operators: general, exploratory, searching, and polyhedral, representing different interaction relationships and evolutionary focuses. Here we present the prompt of general mutation, and other operators are similar in structure, available in our code repository.

System Prompt

You are an expert in mathematical optimization, mixed-integer programming (MIP), and Gurobi modeling. You specialize in evolving a baseline formulation through rigorous mathematical refinement, polyhedral lifting, and innovative paradigm shifts, transforming it into a mathematically correct and substantially more efficient model implemented in Python using 'gurobipy'.

The user will provide:
- Problem description: The real-world logic and constraints of the optimization problem to be modeled.
- Python template: A Python 'gurobipy' function template. You must preserve the function name and input arguments exactly. You may refactor the internal implementation freely.
- Current formulation data: You will receive its Python 'gurobipy' code, the mathematical idea behind it, and its evaluation fitness, where a lower fitness indicates a computationally superior model, e.g., lower average runtime.
- Diagnostic report (if available): A structured analysis of the current model's solver performance profile, identifying the computational bottlenecks and priority mutation directions.

Your task is:
- Perform a general mutation. You must analyze the provided model and propose a structurally refined version of it.
- Retain the core mathematical paradigm of the original model, but aggressively seek local improvements to optimize its computational efficiency.
- Identify and eliminate redundant constraints, unnecessary variables, or slight symmetries.
- Improve the structural implementation. For instance, tighten variable bounds, compute tighter big-M coefficients based on problem data, or rewrite logical conditions using more efficient 'gurobipy' features.
- Your mutated model should act as an optimized, polished evolution of its predecessor, strictly maintaining mathematical correctness while accelerating the branch-and-bound process.
- To achieve this you should apply advanced modeling techniques where appropriate, including but not limited to {example_strengthing_techniques}.
- You should produce clean, well-structured, and executable Python code using 'gurobipy'. Keep code concise but readable.
- You must provide a concise explanation of the optimization ideas and key improvements in the 'idea' section. Focus on your enhancement motivation and why the new formulation is computationally superior.

Critical constraints:
- Do not change the function signature.
- Do not rely on solver parameter tuning. However, setting parameters strictly required for the model to be accepted by Gurobi is allowed if your formulation necessitates it.
- Do not assume missing data unless clearly implied; make minimal necessary assumptions.
- If ambiguity exists, choose the most standard and computationally efficient formulation.
- Calling 'model.optimize()' or any other solving/execution functions is strictly forbidden within the function. The function must return the model object before any optimization is performed.
- Do not use Gurobi callbacks in any form. If your formulation would naturally rely on callbacks, you must replace it with a static, callback-free alternative.

Response format:
- Return only a valid JSON object in the exact structure below ({detailed_format_instruction}):

```
{
"code": "```python\n<A Python (gurobipy) function implementing the improved Gurobi model>\n```",
"idea": "Explain the specific local structural improvements and why they reduce solving time"
}
```

User Prompt

Problem description:
{problem_description}

Python template:
{python_template}

Current formulation:

{current_formulation_data}

Runtime:
{runtime_profile}

Solver profile:
{solver_profile}

Diagnostic report (if available):
{diagnostic_report}

Memory Reflection

System Prompt

You are an expert in mathematical optimization, mixed-integer programming (MIP), Gurobi modeling, and algorithmic analysis. Your role is to distill actionable, transferable insights and knowledge from MIP formulation evolutions by analyzing parent-offspring relationships and extracting concise experience triplets.
The user will provide:
- Problem description: The real-world logic and constraints of the optimization problem to be modeled.
- Parent formulation data: For each parent formulation, you will receive its Python 'gurobipy' code, the mathematical idea behind it, its evaluation fitness (i.e., average runtime), and its solver performance profiles.
- Diagnostic report: the diagnostic analysis that guided this evolution step.
- Offspring formulation data: Python 'gurobipy' code and the mathematical idea of the generated offspring formulation.
- Evolution outcome: Fitness (average runtime) and solver performance profiles for the generated offspring formulation, in the same format as the parent(s).

Your task is: Extract exactly experience triplets from this evolution step. Each triplet captures a reusable lesson in the form:
- Condition: The observable situation before the modification, described in terms of solver profile symptoms and/or structural problem features. Conditions must be generalizable. Do not reference specific problem names, variable names, or instance sizes. Use solver metric patterns and structural descriptors only.
- Strategy: The specific modeling action taken, described as a concrete, reproducible technique. Strategies must be precise enough that another modeler could apply them to a different problem exhibiting the same Condition.
- Effect: The outcome and mechanistic explanation, state whether performance improved or degraded, cite the key metric changes, and briefly explain why.

Guidelines for high-quality triplets:
- Be concise: Each field should be 1-2 sentences maximum. Avoid verbose explanations.
- Be generalizable: Triplets should apply beyond the specific problem instance. Abstract away problem-specific details into structural patterns.
- Capture both successes and failures: Failed strategies (offspring worse than parent) are equally valuable, as long as they prevent repeating mistakes.
- Be mechanistic: The Effect field must explain the solver mechanism, not just report metric changes.
- One lesson per triplet: If the evolution involved multiple simultaneous changes, decompose into separate triplets where possible, attributing each metric change to its most likely cause.
- Mark outcome clearly: Start the Effect field with [+] for improvements or [-] for degradations.

Response format:
- Your response must consist of ONLY the triplets in the exact format below. No preamble, no greeting, no commentary.
- Return 1-3 triplets. Use exactly the format shown. Number each triplet.

```
- Condition: [generalizable situation descriptor]
- Strategy: [concrete modeling action]
- Effect: [+/-] [metric changes and mechanistic explanation]
```

User Prompt

Problem description:
{problem_description}

Parent formulation:
{parent_formulation_data}

Parent runtime and solver profile:
{parent_runtime_solver_profile}

Diagnostic report:
{diagnostic_report}

Offspring formulation:
{offspring_formulation_data}

Offspring runtime and solver profile:
{offspring_runtime_solver_profile}

Knowledge Distillation

System Prompt

You are an expert in mathematical optimization, mixed-integer programming (MIP), Gurobi modeling, and algorithmic analysis. Your role is to distill multiple memory libraries (may be problem-specific) into a single, unified, general-purpose knowledge base of MIP formulation principles that are transferable to any new optimization problem.
The user will provide: Problem-specific memory libraries: Multiple memory libraries, each accumulated from evolutionary MIP formulation improvement on a specific problem type. Each library contains experience triplets with Condition, Strategy, and Effect. The libraries may come from diverse problem domains (e.g., scheduling, routing, packing, facility location, etc.)
Your task is to produce a general transferable knowledge base by performing the following:
- Cross-problem validation: Identify entries that appear (with consistent Effects) across multiple problem-specific libraries. These represent robust, problem-independent MIP modeling principles and should be prioritized and preserved with high fidelity.
- Abstraction: For entries that reference residual problem-specific structure (even if already somewhat general), further abstract the Condition to the broadest valid scope.
- Consolidation: Merge entries from different libraries that express the same underlying principle. Even if worded differently or applied in different problem contexts, unify them into a single canonical entry if the

Condition-Strategy-Effect logic is equivalent. Combine metric evidence from multiple sources.

- Conflict resolution: If the same Condition-Strategy pair shows positive Effect in some libraries but negative Effect in others, analyze whether the divergence can be explained by a distinguishable sub-condition (e.g., strategy works for sparse constraint matrices but fails for dense ones). If so, split into two entries with refined, non-overlapping Conditions.
- Pruning: The final knowledge base should contain at most 30 entries. Remove entries that are too narrow to plausibly transfer or trivial or obvious.

Response format:

- Your response must consist of only the triplets in the exact format below. No preamble, no greeting, no commentary.
- Number entries sequentially with IDs starting from 1.
- Use exactly the format below for each entry.

```
[KNOWLEDGE BASE START]
ID: 1
- Condition: [generalizable situation descriptor]
- Strategy: [concrete modeling action]
- Effect: [+/-] [metric changes and mechanistic explanation]
ID: 2
- Condition: [generalizable situation descriptor]
- Strategy: [concrete modeling action]
- Effect: [+/-] [metric changes and mechanistic explanation]
...
[KNOWLEDGE BASE END]
```

User Prompt

Memory libraries:
{memory_libraries}

C Additional Results and Discussions

C.1 Statistical Significance Analysis

We conduct paired Wilcoxon signed-rank tests on per-instance runtimes to compare FormuEvo with the best baselines over each problem and difficulty category (Table 6). For most problems such as TSP, CFLP, and the NNV and IMO tasks, FormuEvo achieves statistically significant improvements ($p < 0.01$) across all categories. Notably, significance generally increases with instance difficulty, suggesting that the advantages of evolved formulations are more evident in challenging problem instances, where formulation strength critically impacts relaxation quality and branching behavior.

For BPP and QAP, the improvements on Easy instances are less significant. We attribute this to the "floor effect": modern MIP solvers can solve small instances within extremely short runtimes, and the performance is more sensitive to runtime variance across instances. As the problem scale and complexity grow (Medium and Hard), the advantages of FormuEvo become increasingly significant, demonstrating the strong scalability and upper-bound capabilities of FormuEvo, while further highlighting the importance of solver-efficient formulation for large-scale and more challenging optimization problems.

Table 6: Wilcoxon signed-rank test p-values.

Problem	Easy	Medium	Hard
TSP	1.88E-17	2.79E-13	4.13E-18
JSSP	7.83E-02	1.13E-03	2.44E-02
BPP	2.82E-01	9.29E-02	2.50E-02
CFLP	4.98E-13	3.03E-10	2.20E-08
QAP	4.44E-01	7.93E-02	3.05E-02
NNV	-	-	1.42E-14
IMO	-	-	3.91E-03

C.2 Robustness Across Different Solvers

To evaluate the robustness and adaptability of FormuEvo across downstream MIP solvers, we replace Gurobi in our main experiments with two alternative solvers: COPT (Ge et al., 2022) and SCIP (Hojny et al., 2025), and rerun the evolution process under each solver. The results are reported in Tables 7 and 8. Here, FormuEvo denotes the formulations evolved directly under the corresponding solver, while FormuEvo-G denotes the formulations evolved under Gurobi (i.e., the one obtained in our main experiments) and then directly adapted to COPT and SCIP, respectively.

Overall, FormuEvo consistently achieves significant improvements on different downstream solvers, demonstrating that the proposed evolutionary framework is not restricted to a specific optimization backend. Meanwhile, we observe substantial variation in the relative performance of different formulations across solvers (e.g., MTZ and MCF-RLT, FormuEvo and FormuEvo-G). This suggests that formulation quality is inherently solver-dependent, and the most effective formulation for one solver may not remain optimal under another optimization backend, due to the algorithmic differences among modern MIP solvers. These results highlight a key advantage of FormuEvo over static formulation generation, and also demonstrate the necessity of our solver-aware optimization that adaptively exploits solver-informed feedback to discover formulations that better align with the algorithmic internals of the downstream solver.

Table 7: Performance based on COPT solver.

		Easy		Medium		Hard		
	Formulation	Time↓	Wins↑	Time↓	Wins↑	Time↓	Wins↑	Solved↑
TSP	MTZ (1960)	5.5116	0/100	79.9319	0/100	517.6449	0/100	8/100
	SCF (1978)	0.8488	12/100	5.6820	15/100	129.9687	16/100	95/100
	MCF-RLT (2018)	3.8573	0/100	56.2487	0/100	626.8963	0/100	1/100
	ORLM (2025)	5.5235	0/100	80.0451	0/100	517.7556	0/100	8/100
	StepORLM (2026)	4.4893	0/100	66.7571	0/100	479.0096	0/100	13/100
	EvoCut (2025)	3.7323	0/100	27.5892	0/100	258.2871	7/100	41/100
	FormuEvo-G	0.7488	16/100	5.3816	12/100	129.7755	15/100	**96/100**
	FormuEvo (Ours)	**0.4517**	**72/100**	**3.0088**	**73/100**	**47.8586**	**62/100**	**96/100**
JSSP	Disj. (1960)	1.5502	12/100	9.5882	11/100	82.3530	10/100	**97/100**
	Enh. Disj. (2016)	2.5150	3/100	14.1189	2/100	117.2248	3/100	93/100
	ORLM (2025)	601.0423	0/100	600.7678	0/100	601.7053	0/100	0/100
	StepORLM (2026)	1.6431	11/100	8.9512	9/100	77.5248	14/100	96/100
	EvoCut (2025)	1.2713	27/100	8.0894	26/100	72.8686	20/100	96/100
	FormuEvo-G	1.5798	9/100	8.5912	14/100	77.1278	21/100	**97/100**
	FormuEvo	**1.1779**	**38/100**	**6.7091**	**38/100**	**67.7242**	**32/100**	**97/100**

Table 8: Performance based on SCIP solver.

		Easy		Medium		Hard		
	Formulation	Time↓	Wins↑	Time↓	Wins↑	Time↓	Wins↑*	Solved↑
TSP	MTZ (1960)	4.7167	0/100	62.3060	1/100	563.1162	12/100	11/100
	SCF (1978)	4.4918	1/100	44.2718	4/100	600.0065	0/100	0/100
	MCF-RLT (2018)	67.1733	0/100	600.0136	0/100	600.2085	0/100	0/100
	ORLM (2025)	4.7421	2/100	62.0028	3/100	563.5875	12/100	11/100
	StepORLM (2026)	5.4227	3/100	84.3814	2/100	591.2279	13/100	3/100
	EvoCut (2025)	3.6788	5/100	42.7763	10/100	555.9330	15/100	6/100
	FormuEvo-G	3.4371	11/100	35.1599	13/100	523.7598	19/100	9/100
	FormuEvo	**2.8490**	**78/100**	**31.8396**	**67/100**	**498.6678**	**29/100**	**38/100**
JSSP	Disj. (1960)	13.0998	1/100	84.2068	1/100	491.3035	2/100	27/100
	Enh. Disj. (2016)	14.3031	2/100	67.3851	1/100	405.8829	1/100	39/100
	ORLM (2025)	600.0982	0/100	600.0586	0/100	600.0538	0/100	0/100
	StepORLM (2026)	14.874	2/100	98.2472	1/100	486.7095	0/100	28/100
	EvoCut (2025)	7.4703	19/100	58.4963	5/100	460.8576	3/100	30/100
	FormuEvo-G	6.5541	17/100	36.4157	14/100	341.2865	2/100	47/100
	FormuEvo	**4.1672**	**59/100**	**17.7541**	**78/100**	**114.5116**	**92/100**	**94/100**

*For most TSP Hard instances, SCIP fails to solve to optimality within the time limit, making the "Wins" metric less indicative.

C.3 Generalization to Public Benchmarks

We further evaluate the scalability and generalization performance of FormuEvo on large-scale public benchmark instances that are more challenging than the instances used during evolution. For TSP, we test on the 200-city instances from the Solomon benchmark suite (Solomon, 1987). For JSSP, we test on the most challenging 100×20 instances from the Taillard benchmark (Taillard,

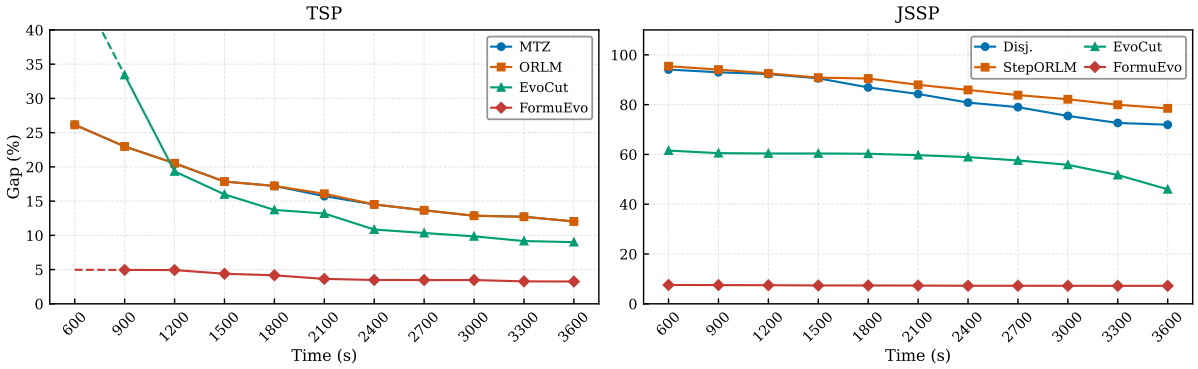

Figure 5: Performance comparisons on public benchmark instances. Left: Solomon benchmark of TSP (Solomon, 1987). Right: Taillard benchmark of JSSP (Taillard, 1993).

1993). These public benchmarks, which are more difficult than the Hard instances used in the main experiments, are recognized as challenging large-scale optimization instances for solver evaluation.

Since these instances are generally intractable to solve to optimality within practical time limits, we report the primal-dual gap curves over solving time, where smaller gaps indicate better optimization performance (Fig. 5). The results demonstrate that although FormuEvo is evolved using only small instances, the formulations discovered by FormuEvo generalize effectively to benchmark instances with substantially larger scales and different distributions. In particular, the tighter formulations produced by FormuEvo enable the solver to identify high-quality feasible solutions more quickly, thereby reducing the primal-dual gap at a significantly faster rate than baselines.

C.4 Analysis of Evolutionary Search Hyperparameters

We conduct a sensitivity analysis of evolutionary search hyperparameters on TSP, including mutation rate ρ and memory usage rate γ. As shown in Table 9, FormuEvo achieves the best overall performance with $\rho = 0.3$ and $\gamma = 0.7$, which is selected as our default setting. Notably, a large memory usage rate (e.g., $\gamma = 1.0$) does not always lead to better performance, suggesting that occasional memory-free generation helps maintain search diversity and prevents excessive reliance on previously explored patterns.

C.5 Discussion with Existing LLM-based Modeling Approaches

Finally, we provide a discussion about the comparison between FormuEvo and existing LLM-based modeling approaches (e.g., ORLM, StepORLM).

Table 9: Analysis of evolutionary hyperparameters.

Param.		Easy	Medium	Hard
ρ	0.0	0.1579	0.6128	4.1886
	0.3	**0.1410**	**0.5640**	3.9469
	0.5	0.1903	0.6451	4.3993
	0.7	0.1862	0.6472	4.4488
	1.0	0.1456	0.5719	**3.9436**
γ	0.0	0.1776	0.6102	4.6649
	0.3	0.1770	0.5791	4.4793
	0.5	0.1445	0.5809	4.4923
	0.7	**0.1410**	**0.5640**	**3.9469**
	1.0	0.1780	0.5987	4.3957

Optimization capability and continual improvement. Existing LLM-based modeling approaches primarily focus on semantic correctness and syntactical executability through supervised and reinforcement fine-tuning. Although their training data cover a broad range of optimization settings, including linear, integer, and non-linear programming tasks (e.g., Fig. 7 of ORLM (Huang et al., 2025)), the capability of these models is largely fixed once fine-tuning is completed. Even when multiple samples are generated during inference, the resulting formulations exhibit very limited diversity (as in our main results): correct generations collapse to nearly the same textbook formulations, while more diverse and efficiency-oriented instructions always lead to incorrect or infeasible results or even semantically meaningless outputs.

In contrast, FormuEvo frames formulation design itself as an explicit optimization problem over the symbolic space of MIP formulations. Instead of relying solely on the static capability of pre-

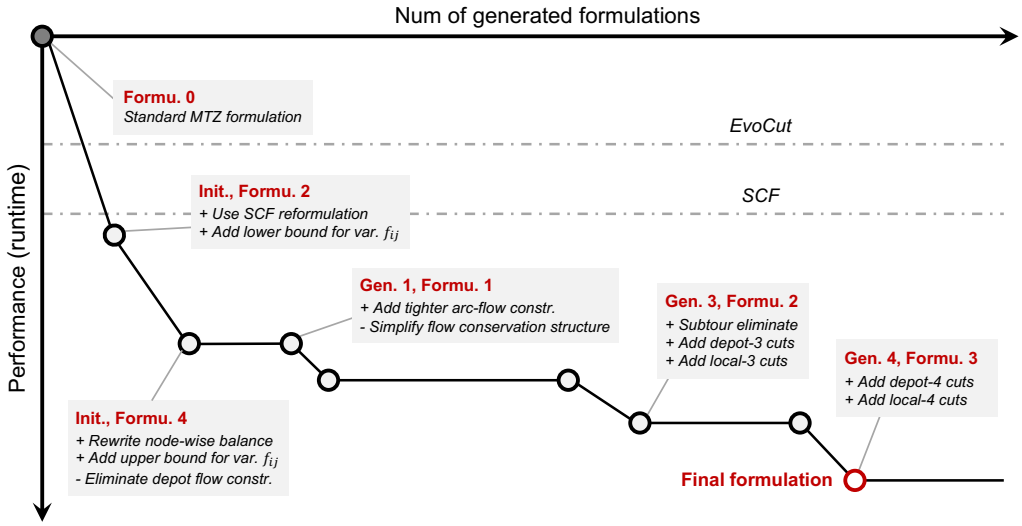

Figure 6: An illustration of the evolutionary process of FormuEvo (on TSP).

trained models, FormuEvo continuously improves candidate formulations through iterative evolutions guided by solver feedback. As shown in our results and Fig. 6, even with a small evolutionary budget, FormuEvo can discover high-quality MIP formulations that outperform both expert-designed formulations and recent LLM-based approaches. These results suggest that the formulation design benefits more from the evolutionary search and solver-informed optimization than from static formulation generation alone.

Offline and online computational costs. Existing fine-tuned modeling approaches rely on extensive curated datasets, long training times, and substantial GPU resources. Their online inference additionally incurs per-instance overhead, since formulations are generated independently for each optimization instance, even though one may attempt to recover a problem-level formulation via post-hoc aggregation using another LLM.

In contrast, FormuEvo requires no task-specific fine-tuning and can be implemented entirely via lightweight API-based interaction with general-purpose LLMs in CPU-only environments. In our experiments, the entire evolutionary process typically requires only one to several hours, depending on the complexity of the target problem and the number of evaluation instances, which is substantially lower than the time of data collection and training required for large-scale fine-tuning. Moreover, FormuEvo operates at the problem level rather than the instance level. Once a strong formulation is evolved for a problem family, it can be directly reused on unseen instances with essentially no additional optimization cost, making FormuEvo particularly attractive in scenarios where many instances have shared underlying optimization structures.

Generality and transferability. Existing fine-tuned modeling approaches are typically specialized to particular modeling languages, prompting formats, and solver interfaces seen during training, making them extremely sensitive to distribution shifts and implementation-specific overfitting. For example, in ORLM and StepORLM, we find that even minor modifications from their prompt templates can lead to substantial degradation or complete failure of the generated formulations. In contrast, FormuEvo operates on top of general-purpose foundation LLMs and treats optimization formulations as symbolic programs that can be flexibly modified and evaluated. Therefore, its optimization capability is not tied to a specific LLM backbone, modeling language, solver backend, or benchmark distribution (Appendix C.2 - C.4). The evolution framework and distilled knowledge can naturally extend across different optimization paradigms, modeling interfaces, and solver implementations, suggesting broader applicability and stronger transferability in practical optimization settings.

D The Use of LLMs

LLMs are used as key methodological components for evolutionary search in our framework. We also used LLMs to assist code writing and text polishing, while the core ideas and the manuscript itself were conceived, prepared, and finalized by the authors.