EFFIOR: POST-MODEL REFINEMENT FOR EFFICIENT SOLVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have made substantial progress in automatically generating mathematical models and solver implementations for operations research problems. However, correct modeling does not necessarily imply efficient solving, as the same optimization problem may admit multiple valid formulations with substantially different computational performance. We propose EffiOR, a framework for improving existing optimization models and solver implementations rather than rebuilding them from scratch. EffiOR formalizes 29 reusable Optimization Skills across Model-Structure, Formulation-Refinement, and Solver-Implementation, enabling existing models and solver implementations to be transformed for more efficient solving while preserving the original problem semantics. Experiments on FrontierOR-179 show consistent efficiency gains, while four additional optimization-modeling benchmarks demonstrate high transformation consistency.

1 INTRODUCTION

Mathematical programming supports decision making in transportation, scheduling, resource allocation, and production planning Toth and Vigo (2014); Pinedo (2016); Katoh and Ibaraki (1988); Pochet and Wolsey (2006). Translating operational requirements into mathematical formulations and executable solver programs traditionally requires substantial operations research expertise. Recent advances in large language models (LLMs) have enabled increasingly automated modeling workflows, supported by benchmarks for natural-language optimization Ramamonjison et al. (2022); Huang et al. (2025b) and systems integrating formulation, code generation, execution, and correction Ahmaditeshnizi et al. (2024); AhmadiTeshnizi et al. (2024). Optimization-specific training, structured reasoning, and explicit verification further improve the reliability of generated models Huang et al. (2025a); Jiang et al. (2025a); Lu et al. (2025); Liu et al. (2025; 2026b); Wang et al. (2025b).

Correct and executable models can nevertheless remain computationally inefficient. Different formulations of the same optimization problem may differ substantially in model size, relaxation strength, symmetry, and variable domains. Solver implementations introduce additional differences through model construction, data access, and the organization of repeated solves. Classical optimization addresses these factors through bound tightening, valid inequalities, symmetry reduction, decomposition, and variable or constraint generation Wolsey (1998); Nemhauser and Wolsey (1988); Margot (2010); Benders (1962); Dantzig and Wolfe (1960); Gilmore and Gomory (1961); Barnhart et al. (1998). Applying such techniques requires identifying their mathematical conditions, updating the formulation and implementation consistently, and checking that the transformed solution retains its intended meaning.

Growing interest in LLM-based heuristic and algorithm design Liu et al. (2024); Ye et al. (2024), together with benchmarks targeting computational performance Sun et al. (2026); Chen et al. (2026); Lin et al. (2026); Kong et al. (2026), highlights the importance of efficient solving. Retrieval and reusable optimization knowledge also provide resources for improving modeling and solving workflows Jiang et al. (2025b); Pan et al. (2025); Liu et al. (2026a); Kong et al. (2025); Yang et al. (2026). These developments motivate a downstream task: given an existing mathematical formulation and solver implementation, identify and apply suitable transformations that improve solving efficiency while preserving the specified problem semantics. This task requires coordinated reasoning about mathematical structure, implementation choices, and execution evidence.

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

We propose **EffiOR**, a framework for systematic post-model refinement. EffiOR first constructs a semantic intermediate representation $\mathcal{IR}$ that aligns the problem description, mathematical formulation, and solver implementation at the variable, constraint, and objective levels. It organizes 29 reusable Optimization Skills into three categories: Model-Structure, Formulation-Refinement, and Solver-Implementation. To support candidate selection, EffiOR consolidates shared necessary conditions from 117 Skill-specific applicability conditions into 39 Selection Features. These features are evaluated over the current $\mathcal{IR}$ to filter candidates before Skill-specific validation. After an accepted transformation, the formulation, implementation, and $\mathcal{IR}$ are synchronized, and affected applicability decisions are revisited. Instrumented execution then compares the transformed artifacts with the frozen baseline through objective-level and stricter solver-state consistency checks, including solution recovery and feasibility verification.

We evaluate EffiOR using two upstream modeling frameworks and three backbone LLMs, with a generation-budget-matched Iterative CoT baseline. On FrontierOR-179 Kong et al. (2026), EffiOR improves Qual@2%, Qual@1%, and QTE over the original models by 3.26, 7.08, and 14.24 percentage points, respectively, averaged across the six framework–backbone settings. A complementary evaluation uses independently verified upstream outputs from four smaller modeling benchmarks: NL4Opt, MAMO EasyLP, MAMO ComplexLP, and IndustryOR. Across their framework–backbone–benchmark combinations, EffiOR achieves macro-averaged objective consistency of 94.47% and state consistency of 88.73%. Ablations with GPT-5.6 Sol (High) indicate that adding Formulation-Refinement Skills yields the largest QTE gains under both upstream frameworks, while Model-Structure Skills provide further improvements. These results support structured post-model refinement as a practical direction for improving the computational performance of automatically generated optimization models.

2 Related Work

2.1 LLM-based Automated Optimization Modeling

LLM-based optimization modeling has progressed from benchmarks for translating natural-language problems into mathematical formulations and solver outputs Ramamonjison et al. (2022); Huang et al. (2025b); Yang et al. (2025) to multi-stage systems integrating modeling, execution, correction, retrieval, and verification. OptiMUS combines formulation, code generation, execution, and debugging Ahmaditeshnizi et al. (2024); AhmadiTeshnizi et al. (2024); OptiTree introduces hierarchical modeling search Liu et al. (2025); DRoC and Opt-Miner incorporate external modeling knowledge Jiang et al. (2025b); Liu et al. (2026a); and Opt-Verifier and ORGEval explicitly examine semantic consistency Liu et al. (2026b); Wang et al. (2025b).

A parallel line improves the underlying LLM through optimization-specific training. ORLM, LLMOPT, and OptMATH construct specialized data or representations for optimization modeling Huang et al. (2025a); Jiang et al. (2025a); Lu et al. (2025), while Solver-Informed RL, StepORLM, and OR-PRM exploit solver or process-level supervision Chen et al. (2025); Zhou et al. (2026); Wang et al. (2026). Structured reasoning and optimization-specific supervision provide further improvements Wang et al. (2025a); Zhang et al. (2026), and OptMaster begins to connect formulation search with downstream solving experience Lin et al. (2026). EffiOR addresses a subsequent stage: it takes an existing formulation and solver implementation as input and transforms them for more efficient solving.

2.2 LLM-based Efficient Solving and Algorithm Design

Beyond modeling, LLMs have been used to search for efficient algorithms and heuristics. Evolution of Heuristics and ReEvo use LLM-generated proposals and reflections to guide heuristic evolution Liu et al. (2024); Ye et al. (2024), while CO-Bench and HeuriGym provide execution-based benchmarks for algorithm-design capabilities Sun et al. (2026); Chen et al. (2026).

FrontierOR evaluates whether LLMs can exploit problem structure to solve large-scale optimization instances efficiently Kong et al. (2026). Its tasks highlight that a mathematically correct formulation may still be computationally unsuitable; for example, `bront2009` admits a direct formulation, while its large column space motivates column generation Bront et al. (2009); Kong et al. (2026). Unlike open-ended algorithm search, EffiOR does not seek new heuristics from scratch. It selects

and applies established optimization transformations to an existing formulation and implementation according to their structural applicability.

2.3 Structure-Exploiting Mathematical Optimization

EffiOR's Optimization Skills are grounded in established mathematical programming techniques. Formulation-level methods include bound tightening, valid inequalities, and symmetry reduction Wolsey (1998); Nemhauser and Wolsey (1988); Margot (2010). Structural methods include Benders decomposition Benders (1962), Dantzig–Wolfe decomposition Dantzig and Wolfe (1960), column generation Gilmore and Gomory (1961), branch-and-price Barnhart et al. (1998), constraint generation and branch-and-cut Padberg and Rinaldi (1991), and Lagrangian relaxation Fisher (1981).

3 Problem Formulation and Optimization Skills

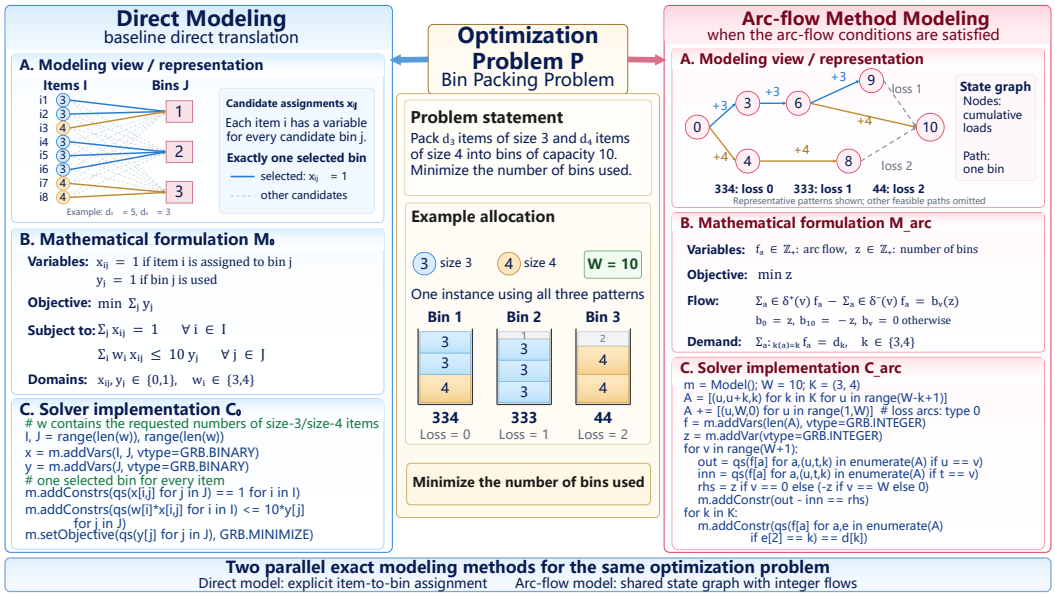


Figure 1: Two exact formulations of the same optimization problem: direct item-to-bin assignment and Arc-flow state-graph modeling, preserving the same problem semantics and objective.

3.1 Problem Definition

Existing automated optimization modeling methods aim to translate a natural-language optimization problem $\mathcal{P}$ into a mathematical formulation $\mathcal{M}$ and executable solver code $\mathcal{C}$:

$$\mathcal{P} \longrightarrow (\mathcal{M}, \mathcal{C}). \tag{1}$$

However, correctness and executability do not determine solving efficiency. The same optimization problem may admit multiple correct mathematical formulations and solver implementations with substantially different computational behavior. We therefore study the post-model problem of improving an existing pair $(\mathcal{M}, \mathcal{C})$ while preserving the semantics of $\mathcal{P}$. We define the set of mathematical formulations that correctly represent $\mathcal{P}$ as

$$\mathfrak{M}(\mathcal{P}) = \{\mathcal{M} \mid \mathcal{M} \text{ correctly represents } \mathcal{P}\}. \tag{2}$$

For a mathematical formulation $\mathcal{M}$, we further define the set of solver implementations that faithfully realize $\mathcal{M}$ as

$$\mathfrak{C}(\mathcal{M}) = \{\mathcal{C} \mid \mathcal{C} \text{ faithfully implements } \mathcal{M}\}. \tag{3}$$

Thus, a solver implementation is not considered independently of the mathematical formulation it realizes. Let $\tau(\mathcal{C})$ denote the solving cost of executing $\mathcal{C}$ for a fixed problem instance under a fixed computational environment and resource budget. The ideal post-model efficiency optimization problem is

$$(\mathcal{M}^*, \mathcal{C}^*) = \underset{\substack{\mathcal{M}\in\mathfrak{M}(\mathcal{P})\\ \mathcal{C}\in\mathfrak{C}(\mathcal{M})}}{\arg\min}\ \tau(\mathcal{C}). \tag{4}$$

Equation 4 captures the central requirement of our setting: solving efficiency may be improved by changing either the mathematical formulation or its solver implementation, but only while the resulting formulation remains correct for the original problem and the implementation remains faithful to that formulation. Directly searching over all valid mathematical formulations and their solver implementations is generally impractical. We therefore restrict this ideal space through a structured family of reusable optimization transformations.

3.2 Optimization Method Space

The ideal optimization problem defined in Section 3.1 considers all mathematically correct formulations of $\mathcal{P}$ and all solver implementations that faithfully realize them. Directly searching this space is generally impractical, since the space of valid formulations and implementations is too large to enumerate. We therefore restrict the search to transformations induced by a finite library of established optimization methods. Let

$$\mathcal{K} = \mathcal{K}_{\mathcal{M}} \cup \mathcal{K}_{\mathcal{C}}, \tag{5}$$

where $\mathcal{K}_{\mathcal{M}}$ contains methods that may modify the mathematical formulation, and $\mathcal{K}_{\mathcal{C}}$ contains methods that preserve the mathematical formulation and modify only its solver implementation.

For a model-level method $k \in \mathcal{K}_{\mathcal{M}}$, the method-induced effective transformation set is defined as

$$\mathcal{T}_k^{\mathcal{M}}(\mathcal{P}, \mathcal{M}, \mathcal{C}) = \{(\mathcal{M}', \mathcal{C}') \mid (\mathcal{M}', \mathcal{C}') \in k(\mathcal{P}, \mathcal{M}, \mathcal{C}), \mathcal{M}' \in \mathfrak{M}(\mathcal{P}), \mathcal{C}' \in \mathfrak{C}(\mathcal{M}'), \tau(\mathcal{C}') < \tau(\mathcal{C})\}. \tag{6}$$

For an implementation-level method $k \in \mathcal{K}_{\mathcal{C}}$, the mathematical formulation remains unchanged. Its effective transformation set is defined as

$$\mathcal{T}_k^{\mathcal{C}}(\mathcal{P}, \mathcal{M}, \mathcal{C}) = \{(\mathcal{M}, \mathcal{C}') \mid (\mathcal{M}, \mathcal{C}') \in k(\mathcal{P}, \mathcal{M}, \mathcal{C}), \mathcal{C}' \in \mathfrak{C}(\mathcal{M}), \tau(\mathcal{C}') < \tau(\mathcal{C})\}. \tag{7}$$

For convenience, the effective transformation set induced by any method $k \in \mathcal{K}$ is denoted as $\mathcal{T}_k(\mathcal{P}, \mathcal{M}, \mathcal{C})$. A method is considered applicable to the current state only when its effective transformation set is non-empty:

$$\mathcal{T}_k(\mathcal{P}, \mathcal{M}, \mathcal{C}) \neq \varnothing. \tag{8}$$

Otherwise, method k cannot provide an efficiency-improving transformation that preserves the correctness of the original problem under the current state. The restricted search space is defined by the set of model–implementation pairs reachable through effective transformations induced by methods in $\mathcal{K}$:

$$\mathfrak{R}_{\mathcal{K}}(\mathcal{P}, \mathcal{M}, \mathcal{C}) = \{(\mathcal{M}', \mathcal{C}') \mid (\mathcal{M}', \mathcal{C}') \text{ can be obtained by repeatedly applying methods in } \mathcal{K}\}. \tag{9}$$

The practical post-model efficiency optimization problem is therefore

$$(\mathcal{M}^*, \mathcal{C}^*) = \underset{(\mathcal{M}', \mathcal{C}')\in\mathfrak{R}_{\mathcal{K}}(\mathcal{P}, \mathcal{M}, \mathcal{C})}{\arg\min}\ \tau(\mathcal{C}'). \tag{10}$$

The central challenge is therefore to determine, for each candidate method k, whether

$$\mathcal{T}_k(\mathcal{P}, \mathcal{M}, \mathcal{C}) = \varnothing. \tag{11}$$

Explicitly constructing every possible transformation and evaluating its efficiency for each candidate method would require repeatedly performing the complete transformation and verification process. This is computationally expensive and defeats the purpose of restricting the search space. Section 4 therefore introduces a mechanism that identifies potentially effective methods before performing their complete method-specific transformation and verification.

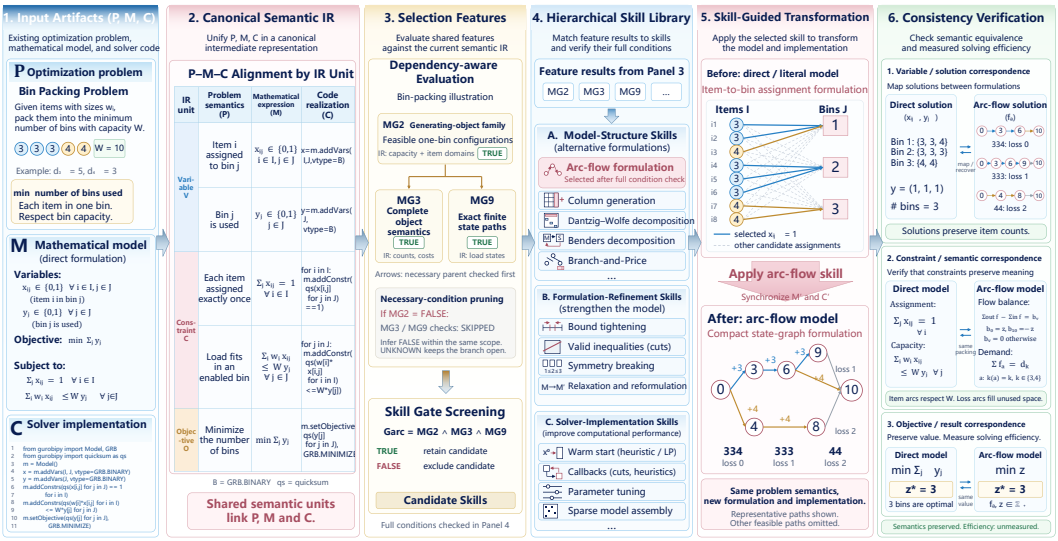


Figure 2: EffiOR pipeline from semantic alignment and Skill selection to optimization transformation and consistency verification.

4 EFFIOR FRAMEWORK

Figure 2 summarizes the overall EffiOR pipeline, from semantic alignment and Skill selection to optimization transformation and consistency verification.

4.1 SEMANTIC ALIGNMENT IR

The problem description $\mathcal{P}$, mathematical formulation $\mathcal{M}$, and solver implementation $\mathcal{C}$ express the same underlying optimization structure in different forms. We therefore construct a semantic intermediate representation $\mathcal{IR}$ that explicitly aligns their shared semantics, providing a unified state for subsequent optimization transformation.

The construction of $\mathcal{IR}$ consists of two main steps: semantic decomposition and semantic alignment. The LLM first decomposes $\mathcal{P}, \mathcal{M}, \mathcal{C}$ into three fundamental as the extracted variable-, constraint-, and objective-related semantic units, respectively. It then semantically matches units that describe the same optimization meaning and merges them into a unified unit in ($\mathcal{IR}$).

For example, consider the item-to-bin assignment decision in the direct bin-packing formulation. The LLM first decomposes the corresponding semantic units from the three representations separately: from $\mathcal{P}$, it extracts the decision "assign item i to bin j" from $\mathcal{M}$, it extracts $x_{ij} = \begin{cases} 1, & \text{if item } i \text{ is assigned to bin } j \\ 0, & \text{otherwise} \end{cases}$ from $\mathcal{C}$, and it extracts the corresponding solver declaration `x = model.addVars(I, J, vtype=GRB.BINARY, name="x")`. The LLM then determines that these three units express the same optimization decision and semantically aligns them into a unified variable-level unit as $\mathcal{IR}_{\text{var}}$. Following the same process, the LLM aligns the constraint- and objective-related semantics extracted from $\mathcal{P}$, $\mathcal{M}$, and $\mathcal{C}$ into $\mathcal{IR}_{\text{con}}$ and $\mathcal{IR}_{\text{obj}}$, respectively. Together, $\mathcal{IR}_{\text{var}}$, $\mathcal{IR}_{\text{con}}$, and $\mathcal{IR}_{\text{obj}}$ constitute the complete semantic representation $\mathcal{IR}$.

4.2 SKILL SCHEDULE

We distill 29 reusable Optimization Skills and organize them into three categories according to the type of transformation they perform:

Model-Structure: Model-Structure Skills modify the global representation or solution structure of the mathematical model. Different structural formulations usually represent alternative ways of

modeling the same problem rather than transformations that should be blindly composed. Therefore, the agent evaluates the candidate Model-Structure Skills and selects at most one applicable structural transformation for the current model.

Formulation-Refinement: Formulation-Refinement Skills improve the current mathematical formulation without replacing its overall modeling structure. Multiple refinement Skills may be applicable to the same model and can therefore be applied sequentially.

Solver-Implementation Skills: Solver-Implementation Skills preserve the finalized mathematical formulation and improve only its solver implementation. These Skills operate on $\mathcal{C}$ through transformations such as solver configuration, search initialization, callbacks, or model-construction optimization.

Thus, Skill application proceeds over an evolving optimization state rather than a static list of candidates. Whenever a Skill changes the current model or implementation, $\mathcal{IR}$ is synchronized with the new state before subsequent Skill decisions are made.

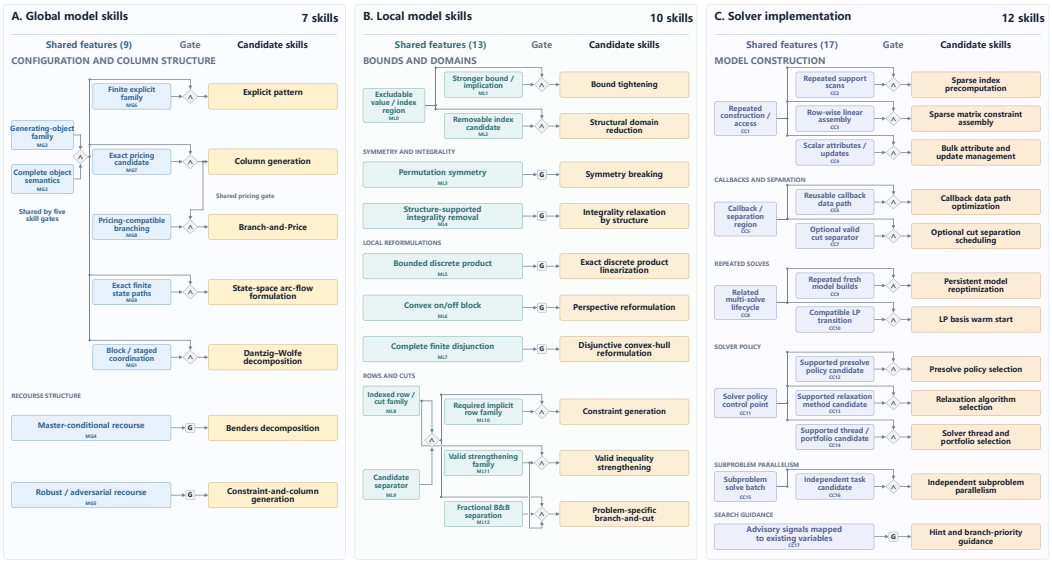


Figure 3: Mapping from shared Selection Features to candidate Optimization Skills through feature gates.

4.3 Feature-Guided Skill Selection

The 29 Optimization Skills in our library contain 117 applicability conditions in total. Directly evaluating these Skill-specific conditions for every optimization state would require substantial and repetitive LLM reasoning, particularly because many Skills rely on overlapping structural conditions. To reduce this cost, we consolidate the shared necessary conditions into 39 reusable *Selection Features*, including 9 Model-Structure, 13 Formulation-Refinement, and 17 Solver-Implementation Features. These Features are evaluated by the LLM against the semantic and structural evidence in the current $\mathcal{IR}$, providing a lightweight screening layer before the complete Skill-specific applicability check.

The screening process is further sparse: the agent does not necessarily evaluate all 39 Selection Features for every state. For each Skill, its candidate gate is formed by the conjunction of the required Features. Once any required Feature is evaluated as FALSE, the corresponding gate can no longer be satisfied, and the remaining Features needed only for that gate are skipped. Consequently, the agent evaluates only the Features that can still affect the current routing decision and performs the full applicability check only for the surviving candidate Skills. Figure 3 shows how the shared Selection Features are combined through feature gates to screen candidate Skills across the three Skill categories.

4.4 Consistency Verification

After Skill transformation, the Agent performs consistency verification between the pre- and post-transformation results. Both versions are executed on the same instance under the same computational environment. The verification covers solver status, solution existence, variable-level solution semantics, feasibility with respect to the original constraints, and objective sense and value. For workflows containing multiple externally meaningful solve episodes, semantically corresponding outcomes are compared through the $\mathcal{IR}$ mapping.

To obtain trustworthy runtime evidence, we instrument the actual solver calls with lightweight observation code while preserving the original solving logic, call arguments, and number of solver invocations. The instrumentation automatically records the state and results produced by each actual solve. The pre-transformation execution is frozen as the baseline, while the transformed implementation is executed again to produce a fresh observation. An independent verifier then compares the two, ensuring that the final consistency decision is based on actual solver executions rather than results generated or reported by the Agent itself.

This comparison remains possible even when a Skill substantially changes the formulation because $\mathcal{IR}$ is continuously updated during transformation to maintain the correspondence between the pre- and post-transformation model elements. It records how original variables are represented by new variables or structures, how constraints are replaced or reformulated, and how objective terms correspond across transformations. The verifier can therefore use these mappings to interpret the transformed solution in terms of the original model and check whether the variable-level decisions, feasibility, and objective results remain consistent.

5 Experiments

We evaluate EffiOR from three perspectives: solving efficiency, transformation consistency, and the contribution of its three Skill categories. To isolate the effect of EffiOR's structured optimization mechanism from additional test-time reasoning, we further compare against a generation-budget-matched Iterative CoT baseline. Our code is available in the anonymous repository at: https://anonymous.4open.science/r/EffiORagent-4034/

5.1 Experimental Setup

Benchmarks: For solving-efficiency evaluation, we use FrontierOR-179 Kong et al. (2026), whose computationally demanding instances are sufficiently discriminative to expose differences in formulation and solving strategy.

For consistency evaluation, we use four optimization-modeling benchmarks: NL4Opt Ramamonjison et al. (2022), MAMO EasyLP and MAMO ComplexLP Huang et al. (2025b) and IndustryOR Huang et al. (2025a). These modeling-oriented benchmarks contain substantially smaller instances, where runtime differences are less discriminative; we therefore use them to evaluate whether transformations preserve independently verified upstream outputs.

Baselines and backbones: We use OptiMUS-0.3 AhmadiTeshnizi et al. (2024) and LLMOPT Jiang et al. (2025a) as upstream modeling approaches, and evaluate GPT-5.6 Sol (High), GLM-5.2, and Qwen3.7-Max as the backbone LLMs for post-model transformation. For OptiMUS, the upstream modeling stage and EffiOR use the same LLM in each setting; e.g., a GPT-generated OptiMUS model is refined by GPT, and likewise for GLM and Qwen. LLMOPT, in contrast, is a trained optimization model with fixed upstream outputs, so the three backbone LLMs are used only for the subsequent EffiOR and Iterative CoT stages. All upstream mathematical formulations $\mathcal{M}_0$ and solver implementations $\mathcal{C}_0$ are frozen before post-model optimization.

We compare three settings. **Original** directly evaluates the frozen $(\mathcal{M}_0, \mathcal{C}_0)$ without transformation. **Iterative CoT** serves as an unstructured post-model refinement baseline, repeatedly revising the same artifacts through generic reasoning. **EffiOR** applies the complete $\mathcal{IR}$-guided Optimization Skill workflow. Since no directly comparable prior system operates on the same frozen formulation–implementation pair for post-model efficiency refinement, Iterative CoT is generation-budget matched

Table 1: Main results on FrontierOR-179. All values are percentages. Obj. Cons. denotes objective-level consistency, and State Cons. denotes solver-state consistency. For LLMOPT, the backbone model applies only to the post-model refinement method and is independent of the original LLMOPT output.

		OptiMUS					LLMOPT				
Backbone	Method	Qual@2% ↑	Qual@1% ↑	QTE ↑	Obj. Cons. ↑	State Cons. ↑	Qual@2% ↑	Qual@1% ↑	QTE ↑	Obj. Cons. ↑	State Cons. ↑
GPT-5.6	Original	62.57	54.19	35.75	–	–	67.04	54.75	39.11	–	–
	Iterative CoT	69.27	60.89	48.60	86.59	71.51	70.95	56.98	42.46	84.92	77.09
	EffiOR	**75.98**	**69.83**	**56.42**	**97.21**	**95.53**	**73.18**	**68.72**	**56.98**	**96.65**	**94.41**
GLM-5.2	Original	41.34	35.75	22.91	–	–	67.04	54.75	39.11	–	–
	Iterative CoT	45.25	39.66	31.84	77.65	67.60	46.93	36.31	29.05	73.18	67.04
	EffiOR	**56.98**	**50.84**	**48.60**	**96.09**	**94.41**	**58.66**	**48.60**	**39.11**	**95.53**	**92.74**
Qwen3.7	Original	39.66	34.08	22.35	–	–	67.04	54.75	39.11	–	–
	Iterative CoT	44.69	37.43	29.05	68.16	59.78	46.37	36.87	27.37	74.30	61.45
	EffiOR	**50.84**	**48.60**	**43.02**	**93.30**	**86.03**	**48.60**	**44.13**	**39.66**	**94.97**	**83.80**

to EffiOR using the same backbone, frozen artifacts, and execution evidence, but without structured Optimization Skills or feature-guided routing.

Metrics: For solving efficiency, we follow FrontierOR and report **Qual@2%**, **Qual@1%**, and **QTE**. Qual@2% and Qual@1% measure the proportion of instances whose feasible objective is within 2% and 1%, respectively, of or better than the Gurobi reference. QTE further requires a Qual@1%-qualified solution to finish no slower than Gurobi Kong et al. (2026). For consistency evaluation, we use two gates. **Objective Consistency (Obj. Cons.)** is the loose gate and checks only whether the pre- and post-transformation objective results agree within the predefined numerical tolerance(1%). **State Consistency (State Cons.)** is the strict gate defined in Section 4.4. In addition to objective consistency, it checks solver status, solution existence, objective sense, the correspondence of transformed variables and solutions through $\mathcal{IR}$, and the feasibility of the recovered solution with respect to the original formulation. For workflows with multiple solver calls, the solve sequence is also required to remain consistent. Both metrics report the proportion of evaluated samples that pass the corresponding gate.

Implementation: All programs are evaluated under the same isolated environment using Python 3.13 and Gurobi 12, with four CPU cores available to each execution, identical memory and time limits, and disabled network access. The four-core limit specifies the maximum CPU resources available to a single program execution and does not force the solver to use exactly four threads. Final benchmark results are not fed back to Iterative CoT or EffiOR for further modification.

5.2 Main Results on FrontierOR

5.3 Main Results on FrontierOR

Table 1 compares Original, Iterative CoT, and EffiOR on FrontierOR-179. Within each framework–backbone setting, all methods start from the same frozen formulation–implementation pair. Averaged across the six settings, EffiOR improves Qual@2%, Qual@1%, and QTE over Original by 3.26, 7.08, and 14.24 percentage points, respectively.

Performance varies across upstream frameworks and backbone models. For LLMOPT with GLM-5.2 and Qwen3.7-Max, Qual@2% and Qual@1% decrease relative to Original, while QTE remains unchanged or improves slightly. Compared with Iterative CoT, EffiOR achieves higher Qual@2%, Qual@1%, and QTE in all six settings, together with higher objective and state consistency. These results support an average benefit from structured post-model refinement while showing that improvements depend on the upstream artifacts and the refinement backbone.

5.4 Consistency Evaluation Across Modeling Benchmarks

We further evaluate EffiOR on independently verified upstream outputs from four optimization-modeling benchmarks: NL4Opt, MAMO EasyLP, MAMO ComplexLP, and IndustryOR. Table 2 compares the transformed results with their frozen baselines using objective-level and state-level consistency checks.

Table 2: Consistency evaluation of EffiOR on independently verified upstream modeling outputs. Obj. Cons. is the loose objective-level gate and State Cons. is the strict solver-state gate. All values are percentages.

Framework	Backbone	Gate	NL4Opt	EasyLP	ComplexLP	IndustryOR
OptiMUS-0.3	GPT-5.6 Sol (High)	Obj. Cons.	94.57	96.95	98.25	97.96
		State Cons.	89.59	93.74	95.61	95.92
	GLM-5.2	Obj. Cons.	90.36	92.58	93.98	94.29
		State Cons.	81.33	85.15	86.75	85.71
	Qwen3.7-Max	Obj. Cons.	89.81	91.77	92.41	93.75
		State Cons.	79.62	83.78	86.08	84.38
LLMOPT	GPT-5.6 Sol (High)	Obj. Cons.	95.88	98.14	99.23	100.00
		State Cons.	91.77	96.27	98.46	97.56
	GLM-5.2	Obj. Cons.	92.59	94.75	95.38	95.12
		State Cons.	85.19	89.32	91.54	90.24
	Qwen3.7-Max	Obj. Cons.	90.53	92.54	93.85	92.68
		State Cons.	81.48	85.25	86.92	87.80

Table 3: Ablation study on FrontierOR-179 with GPT-5.6 Sol (High). All values are percentages. Obj. Cons. denotes objective-level consistency, and State Cons. denotes solver-state consistency.

	OptiMUS					LLMOPT				
Method	Qual@2% ↑	Qual@1% ↑	QTE ↑	Obj. Cons. ↑	State Cons. ↑	Qual@2% ↑	Qual@1% ↑	QTE ↑	Obj. Cons. ↑	State Cons. ↑
EffiOR	**75.98**	**69.83**	**56.42**	97.21	95.53	**73.18**	**68.72**	**56.98**	96.65	94.41
Refinement + Solver	73.18	67.60	54.75	97.21	95.53	72.07	67.04	53.07	95.53	94.97
Solver Only	61.45	54.19	35.20	93.30	89.39	68.72	55.87	38.55	96.65	95.53
Original	62.57	54.19	35.75	–	–	67.04	54.75	39.11	–	–

Across all framework–backbone–benchmark combinations, Obj. Cons. averages 94.47%, while State Cons. averages 88.73%. The mean gap, computed before rounding, is 5.75 percentage points. This gap shows that the additional state checks identify discrepancies that objective agreement alone does not detect. Under the stricter criterion, GPT-5.6 Sol (High) achieves an average State Cons. of 93.72% with OptiMUS-0.3 and 96.02% with LLMOPT across the four benchmarks.

5.5 Ablation Study

We evaluate the contribution of the three Skill categories on FrontierOR-179 with GPT-5.6 Sol (High), using both OptiMUS and LLMOPT upstream artifacts. The ablation follows the cumulative Skill schedule of EffiOR: *Solver Only* retains only Solver-Implementation Skills, *Refinement + Solver* further adds Formulation-Refinement Skills, and *EffiOR* additionally includes Model-Structure Skills.

Contribution of the Skill categories: The ablation shows a consistent pattern across both upstream frameworks. Solver-Implementation Skills alone do not improve aggregate QTE: QTE is 35.20% versus 35.75% for Original with OptiMUS, and 38.55% versus 39.11% with LLMOPT. The largest gains arise after adding Formulation-Refinement Skills, which increase QTE to 54.75% and 53.07%, respectively. Model-Structure Skills provide a further improvement, raising QTE to 56.42% with OptiMUS and 56.98% with LLMOPT.

6 Conclusion and Limitations

Conclusion: We present EffiOR, a framework for refining existing optimization models and solver implementations through reusable Optimization Skills, $\mathcal{IR}$-guided selection, and execution-based consistency checks. On FrontierOR-179, EffiOR improves Qual@2%, Qual@1%, and QTE over the original models by 3.26, 7.08, and 14.24 percentage points, respectively, averaged across six framework–backbone settings. On four additional optimization-modeling benchmarks, objective and state consistency average 94.47% and 88.73%, respectively.

Limitations: Transformation consistency does not guarantee the correctness of the upstream model itself. If the input formulation already misrepresents the original problem, preserving its semantics may also preserve that error. EffiOR therefore depends on the quality of the upstream modeling stage, and our future work will further investigate model verification and repair in this setting.

Ethics Statement

This work does not involve human subjects, animal experiments, or the collection of personally identifiable information. All datasets used in our experiments are existing optimization benchmarks and are used solely for research purposes. We are not aware of any direct privacy, safety, or discriminatory risks introduced by the experiments conducted in this work. The authors take responsibility for conducting the study in accordance with standard research ethics and integrity practices.

Reproducibility Statement

We provide detailed descriptions of the EffiOR framework, Optimization Skills, experimental settings, evaluation metrics, and verification procedures throughout the paper and appendix. The implementation, experimental configurations, and supporting materials are released through the anonymous project repository to facilitate reproduction of our results. All experiments are conducted under controlled computational settings, with the relevant solver version, resource limits, backbone models, and evaluation procedures specified in the paper.

LLM Usage

Large language models are an explicit component of the methodology evaluated in this work. GPT-5.6 Sol (High), GLM-5.2, and Qwen3.7-Max are used as backbone models in the EffiOR experiments, and their outputs are evaluated through the execution and consistency-verification procedures described in the paper.

LLMs were additionally used to assist with manuscript preparation, including language polishing, rephrasing, and LaTeX formatting.

The authors take full responsibility for all LLM-assisted content in the manuscript and experiments.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, Herman Brunborg, Shayan Talaei, Connor Lawless, and Madeleine Udell. OptiMUS-0.3: Using large language models to model and solve optimization problems at scale. *arXiv preprint arXiv:2407.19633*, 2024. URL https://arxiv.org/abs/2407.19633.

Ali Ahmaditeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 577–596. PMLR, 2024. URL https://proceedings.mlr.press/v235/ahmaditeshnizi24a.html.

Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998. doi: 10.1287/opre.46.3.316.

J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962. doi: 10.1007/BF01386316.

Andrea Bettinelli et al. A branch-and-bound algorithm for the knapsack problem with conflict graph. *INFORMS Journal on Computing*, 2017.

Bollapragada, Ghattas, and Hooker. Optimal design of truss structures by logic-based branch and cut. *Operations Research*, 49(1), 2001.

Filipe Brandão and João Pedro Pedroso. Bin packing and related problems: General arc-flow formulation with graph compression. *Computers & Operations Research*, 2016.

Juan José Miranda Bront, Isabel Méndez-Díaz, and Gustavo Vulcano. A column generation algorithm for choice-based network revenue management. *Operations Research*, 57(3):769–784, 2009. doi: 10.1287/opre.1080.0567.

Alberto Caprara, David Pisinger, and Paolo Toth. Exact solution of the quadratic knapsack problem. *INFORMS Journal on Computing*, 11(2), 1999.

Hongzheng Chen, Yingheng Wang, Yaohui Cai, Hins Hu, Jiajie Li, Shirley Huang, Chenhui Deng, Rongjian Liang, Shufeng Kong, Haoxing Ren, Samitha Samaranayake, Carla P. Gomes, and Zhiru Zhang. HeuriGym: An agentic benchmark for LLM-crafted heuristics in combinatorial optimization. In *International Conference on Learning Representations*, 2026. URL https://iclr.cc/virtual/2026/papers.html.

Yitian Chen, Jingfan Xia, Siyu Shao, DongDong Ge, and Yinyu Ye. Solver-informed RL: Grounding large language models for authentic optimization modeling. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3539. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/98555b92e5dedbdad5921049281995bc-Abstract-Conference.html.

Marilène Cherkesly et al. Branch-price-and-cut algorithms for the pickup and delivery problem with time windows and last-in-first-out loading. *Transportation Science*, 2015.

George B. Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960. doi: 10.1287/opre.8.1.101.

J. M. Valério de Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 1999.

Zeger Degraeve and Raf Jans. A new dantzig–wolfe reformulation and branch-and-price algorithm for the capacitated lot sizing problem with set up times. *Operations Research*, 55(5), 2007.

Matteo Fischetti, Ivana Ljubić, and Markus Sinnl. Redesigning benders decomposition for large-scale facility location. *Management Science*, 63(7), 2017.

Marshall L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, 1981. doi: 10.1287/mnsc.27.1.1.

Michel Gamache et al. A column generation approach for large-scale aircrew rostering problems. *Operations Research*, 1999.

P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961. doi: 10.1287/opre.9.6.849.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. ORLM: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 73(6):2986–3009, 2025a. doi: 10.1287/opre.2024.1233. URL https://doi.org/10.1287/opre.2024.1233.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. LLMs for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 2678–2710. Association for Computational Linguistics, 2025b. doi: 10.18653/v1/2025.findings-naacl.146. URL https://aclanthology.org/2025.findings-naacl.146/.

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. LLMOPT: Learning to define and solve general optimization problems from scratch. In *International Conference on Learning Representations*, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/fbe6dd68b0cf2b1d43b458d2b8ca31b0-Abstract-Conference.html.

Xia Jiang, Yaoxin Wu, Chenhao Zhang, and Yingqian Zhang. DRoC: Elevating large language models for complex vehicle routing via decomposed retrieval of constraints. In *International Conference on Learning Representations*, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/73d6c3e4b214deebbbf8256e26d2cf45-Abstract-Conference.html.

Kang and Lee. An exact algorithm for heterogeneous drone–truck routing problem. *Transportation Science*, 2021.

Naoki Katoh and Toshihide Ibaraki. *Resource Allocation Problems: Algorithmic Approaches*. Foundations of Computing. The MIT Press, 1988. ISBN 9780262090278. URL https://mitpress.mit.edu/9780262090278/resource-allocation-problems/.

Gorka Kobeaga et al. A revisited branch-and-cut algorithm for large-scale orienteering problems. *European Journal of Operational Research*, 2024.

Minwei Kong, Ao Qu, Xiaotong Guo, Wenbin Ouyang, Chonghe Jiang, Han Zheng, Yining Ma, Dingyi Zhuang, Yuhan Tang, Junyi Li, Hai Wang, Cathy Wu, and Jinhua Zhao. AlphaOPT: Formulating optimization programs with self-improving LLM experience library. *arXiv preprint arXiv:2510.18428*, 2025. URL https://arxiv.org/abs/2510.18428.

Minwei Kong, Chonghe Jiang, Ao Qu, Wenbin Ouyang, Zhaoming Zeng, Xiaotong Guo, Zhekai Li, Junyi Li, Yi Fan, Xinshou Zheng, Xi Jing, Yikai Zhang, Zhiwei Liang, Seonghoo Kim, Runqing Yang, Zijian Zhou, Sirui Li, Han Zheng, Wangyang Ying, Ou Zheng, Chonghuan Wang, Jinglong Zhao, Hanzhang Qin, Cathy Wu, Paul Pu Liang, Jinhua Zhao, and Hai Wang. FrontierOR: Benchmarking LLMs' capacity for efficient algorithm design in large-scale optimization. *arXiv preprint arXiv:2605.25246*, 2026. URL https://arxiv.org/abs/2605.25246.

Hang Lin, Yuanpeng Gao, Yuzhi Zhang, Kun Yuan, Gang Yan, Siheng Chen, Linfeng Zhang, and Weinan E. OptMaster: A DAG-based framework for formulation and heuristic discovery in optimization. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=zvbuhAxwKy.

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

Haoyang Liu, Jie Wang, Yuyang Cai, Xiongwei Han, Yufei Kuang, and Jianye Hao. OptiTree: Hierarchical thoughts generation with tree search for LLM optimization modeling. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-4027. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/aeaf9b454f4670d5bf2f02b4b6e5bffd-Abstract-Conference.html.

Haoyang Liu, Yuyang Cai, Jie Wang, Xiongwei Han, Minyang Hu, Shuqi Liu, Mingxuan Yuan, Jianye Hao, and Feng Wu. Opt-Miner: Empowering information-seeking agent with tree-guided data synthesis for optimization modeling. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026a. URL https://miralab.ai/publication/.

Haoyang Liu, Jie Wang, Boxuan Niu, Xiongwei Han, Yian Xu, Mingxuan Ye, Zijie Geng, Fangzhou Zhu, Tao Zhong, Mingxuan Yuan, and Jianye Hao. Opt-Verifier: Unleashing the power of LLMs for optimization modeling via dual-side verification. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026b. URL https://arxiv.org/abs/2605.29556.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. OptMATH: A scalable bidirectional data synthesis framework for optimization modeling. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 40769–40802. PMLR, 2025. URL https://proceedings.mlr.press/v267/lu25o.html.

François Margot. Symmetry in integer linear programming. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958–2008*, pages 647–686. Springer, 2010. doi: 10.1007/978-3-540-68279-0_17.

George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988. ISBN 9780471828198. doi: 10.1002/9781118627372. URL https://onlinelibrary.wiley.com/doi/book/10.1002/9781118627372.

Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991. doi: 10.1137/1033004.

Tianpeng Pan, Wenqiang Pu, Licheng Zhao, and Rui Zhou. Leveraging LLM agents for automated optimization modeling for SASP problems: A graph-RAG based approach. In *2025 44th Chinese Control Conference (CCC)*, pages 5476–5481, 2025. doi: 10.23919/CCC64809.2025.11178847. URL https://arxiv.org/abs/2501.18320.

Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, Cham, 5 edition, 2016. doi: 10.1007/978-3-319-26580-3. URL https://link.springer.com/book/10.1007/978-3-319-26580-3.

Yves Pochet and Laurence A. Wolsey. *Production Planning by Mixed Integer Programming*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. ISBN 9780387299594. doi: 10.1007/0-387-33477-7. URL https://link.springer.com/book/10.1007/0-387-33477-7.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pages 189–203. PMLR, 2022. URL https://proceedings.mlr.press/v220/ramamonjison23a.html.

Reference case associated with FrontierOR alfieri2020. Enhancing column generation by a machine-learning-based pricing heuristic for graph coloring. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022.

Reference case associated with FrontierOR delage2022. Logic-based benders decomposition for integrated process configuration and production planning problems. *INFORMS Journal on Computing*, 2022.

Reference case associated with FrontierOR kobayashi2021. Bilevel cutting-plane algorithm for solving cardinality-constrained mean-cvar portfolio optimization problems. *Journal of Global Optimization*, 2021.

C. C. Ribeiro and François Soumis. A column generation approach to the multiple-depot vehicle scheduling problem. *Operations Research*, 42(1), 1994.

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

Paolo Toth and Daniele Vigo, editors. *Vehicle Routing: Problems, Methods, and Applications*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2 edition, 2014. ISBN 9781611973587. doi: 10.1137/1.9781611973594. URL https://epubs.siam.org/doi/book/10.1137/1.9781611973594.

Yilin Wang, Heng Zhou, Dongxing Mao, Linjie Li, Jingru Tan, Haochen Han, Zhengyuan Yang, Alex Jinpeng Wang, and Min Li. OR-PRM: A process reward model for algorithmic problem in operations research. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/4624803ce92f6c44d8cabe085d8d9d96-Abstract-Conference.html.

Zhiyuan Wang, Bokui Chen, Yinya Huang, Qingxing Cao, Ming He, Jianping Fan, and Xiaodan Liang. ORMind: A cognitive-inspired end-to-end reasoning framework for operations research. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pages 104–131. Association for Computational Linguistics, 2025a. doi: 10.18653/v1/2025.acl-industry.10. URL https://aclanthology.org/2025.acl-industry.10/.

Zhuohan Wang, Ziwei Zhu, Ziniu Li, Congliang Chen, Yizhou Han, Yufeng Lin, Zhihang Lin, Angyang Gu, Xinglin Hu, Ruoyu Sun, and Tian Ding. ORGEval: Graph-theoretic evaluation of LLMs in optimization modeling. *arXiv preprint arXiv:2510.27610*, 2025b. URL https://arxiv.org/abs/2510.27610.

Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1998. ISBN 9780471283669.

Haochen Yang, Ke Zhao, Mengyuan Ma, Xingyu Lu, Xiangfeng Wang, and Hong Qian. OptSkills: Learning generalizable optimization skills from problem archetypes via cluster-based distillation. *arXiv preprint arXiv:2605.29829*, 2026. URL https://arxiv.org/abs/2605.29829.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. OptiBench meets ReSocratic: Measure and improve LLMs for optimization modeling. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/3deb687c44d3687ace0729e5db3b4efd-Abstract-Conference.html.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1381.

Bo Zeng and Long Zhao. Solving two-stage robust optimization problems using a column-and-constraint generation method. *Operations Research Letters*, 41(5), 2013.

Yansen Zhang, Qingcan Kang, Yujie Chen, Yufei Wang, Xiongwei Han, Tao Zhong, Mingxuan Yuan, and Chen Ma. SAC-Opt: Semantic anchors for iterative correction in optimization modeling. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2510.05115. Also accepted to the ICLR 2026 Workshop on AI for Mechanism Design and Strategic Decision Making.

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. StepORLM: A self-evolving framework with generative process supervision for operations research language models. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/0bcfb525c8f8f07ae10a93d0b2a40e00-Abstract-Conference.html.

APPENDIX CONTENTS

A Additional Experimental Details **18**

A.1 Execution-Verified Upstream Modeling Accuracy 18

A.2 Skill Invocation Statistics 18

B Optimization Skill Catalog and Detailed Case Studies **21**

B.1 Model-Structure Skills 21

B.1.1 Benders Decomposition 21

B.1.2 Dantzig–Wolfe Decomposition 23

B.1.3 Explicit Pattern Enumeration 25

B.1.4 Column Generation 27

B.1.5 Branch-and-Price 28

B.1.6 State-Space Arc-Flow Formulation 30

B.1.7 Constraint-and-Column Generation 32

B.2 Formulation-Refinement Skills 33

B.2.1 Bound Tightening 33

B.2.2 Structural Domain Reduction 35

B.2.3 Symmetry Breaking 36

B.2.4 Integrality Relaxation by Structure 37

B.2.5 Exact Discrete Product Linearization 38

B.2.6 Perspective Reformulation 40

B.2.7 Disjunctive Convex-Hull Reformulation 41

B.2.8 Constraint Generation 42

B.2.9 Valid-Inequality Strengthening 44

B.2.10 Problem-Specific Branch-and-Cut 45

B.3 Solver-Implementation Skills 47

B.3.1 Sparse Index Precomputation 47

B.3.2 Sparse Matrix Constraint Assembly 48

B.3.3 Bulk Attribute and Update Management 50

B.3.4 Presolve Policy Selection 51

B.3.5 Relaxation Algorithm Selection 53

B.3.6 Callback Data Path Optimization 54

B.3.7 Optional Cut Separation Scheduling 56

B.3.8 Persistent Model Reoptimization 57

B.3.9 LP Basis Warm Start 58

B.3.10 Independent Subproblem Parallelism 60

B.3.11 Solver Thread and Portfolio Selection 61

B.3.12 Hint and Branch Priority Guidance 63

C Verified Before-and-After Case Studies **64**

C.1 Quadratic Knapsack (`caprara1999`) 65

C.2 Graph Coloring (`alfieri2020`) 67

C.3 Bin Packing (`delorme2020`) 69

C.4 Capacitated Facility Location (`klose2005`) 71

C.5 Fixed-Charge Transportation (`roberti2015`) 74

Table 4: Execution-verified upstream modeling accuracy on the four benchmarks used in the consistency evaluation. Each entry reports the number of correct modeling outputs over the total number of benchmark instances, followed by the corresponding accuracy. The Overall column is computed over all 1,252 instances.

Method	Backbone	NL4Opt	EasyLP	ComplexLP	IndustryOR	Overall
OptiMUS-0.3	GPT-5.6 Sol (High)	221/289 (76.47%)	591/652 (90.64%)	114/211 (54.03%)	49/100 (49.00%)	**975/1252 (77.88%)**
OptiMUS-0.3	GLM-5.2	166/289 (57.44%)	431/652 (66.10%)	83/211 (39.34%)	35/100 (35.00%)	715/1252 (57.11%)
OptiMUS-0.3	Qwen3.7-Max	157/289 (54.33%)	413/652 (63.34%)	79/211 (37.44%)	32/100 (32.00%)	681/1252 (54.39%)
LLMOPT	–	243/289 (84.08%)	590/652 (90.49%)	130/211 (61.61%)	41/100 (41.00%)	**1004/1252 (80.19%)**

A Additional Experimental Details

A.1 Execution-Verified Upstream Modeling Accuracy

Before applying EffiOR, we evaluate the correctness of the original modeling outputs produced by the two upstream modeling approaches, OptiMUS-0.3 and LLMOPT. This evaluation determines the set of independently verified upstream samples used in the subsequent consistency experiment. We consider four optimization-modeling benchmarks: NL4Opt, MAMO EasyLP, MAMO ComplexLP, and IndustryOR, containing 289, 652, 211, and 100 instances, respectively, for a total of 1,252 instances.

For OptiMUS-0.3, we separately evaluate three backbone models: GPT-5.6 Sol (High), GLM-5.2, and Qwen3.7-Max. For each setting, the modeling accuracy is computed as

$$\text{Modeling Accuracy} = \frac{N_{\text{correct}}}{N_{\text{total}}} \times 100\%,$$

where N_{correct} denotes the number of upstream modeling outputs that pass the original correctness verification before any EffiOR transformation is applied.

Table 4 reports both the number of correct samples and the corresponding modeling accuracy.

The results show substantial variation in the quality of the frozen upstream modeling outputs. OptiMUS-0.3 achieves an overall modeling accuracy of 77.88% with GPT-5.6 Sol (High), compared with 57.11% using GLM-5.2 and 54.39% using Qwen3.7-Max. LLMOPT produces 1,004 correct models among the 1,252 evaluated instances, corresponding to an overall modeling accuracy of 80.19%.

These original modeling accuracies are distinct from the consistency metrics reported in the main experiments. In particular, the subsequent consistency evaluation is conditioned on the upstream model already being correct. Therefore, for a given framework–backbone–benchmark setting, the denominator of the consistency metric is the corresponding N_{correct} reported in Table 4, rather than the total number of benchmark instances. Specifically,

$$\text{Consistency} = \frac{N_{\text{correct after transformation}}}{N_{\text{correct before transformation}}} \times 100\%.$$

This separation allows the consistency experiment to measure whether EffiOR preserves the correctness of an initially valid modeling result, independently of the upstream modeling method's original success rate.

A.2 Skill Invocation Statistics

To better understand how EffiOR operates beyond the aggregate performance metrics, we examine the Skills that are actually invoked during optimization on FrontierOR-179. We report the invocation profiles separately for Model Skills and Solver-Implementation Skills under the two upstream modeling frameworks, OptiMUS and LLMOPT. Since different Skills address different structural or implementation opportunities, a single task may activate more than one Skill.

Table 5: Actual invocation statistics of Model Skills on FrontierOR-179. Count denotes the number of tasks in which the corresponding Skill is invoked.

		OptiMUS		LLMOPT	
Category	**Skill**	**Count**	**Rate (%)**	**Count**	**Rate (%)**
Model-Structure	Benders Decomposition	11	6.15	10	5.59
	Explicit Pattern Enumeration	8	4.47	7	3.91
	Column Generation	6	3.35	6	3.35
	Branch-and-Price	5	2.79	4	2.23
	State-Space Arc-Flow	4	2.23	3	1.68
	Constraint-and-Column Generation	1	0.56	1	0.56
	Dantzig–Wolfe Decomposition	0	0.00	1	0.56
Formulation-Refinement	Structural Domain Reduction	73	40.78	66	36.87
	Bound Tightening	60	33.52	59	32.96
	Symmetry Breaking	38	21.23	36	20.11
	Valid Inequality Strengthening	26	14.53	26	14.53
	Constraint Generation	26	14.53	26	14.53
	Exact Discrete Product Linearization	14	7.82	14	7.82
	Problem-Specific Branch-and-Cut	3	1.68	1	0.56
	Perspective Reformulation	3	1.68	6	3.35
	Disjunctive Convex Hull Reformulation	3	1.68	3	1.68
	Integrality Relaxation by Structure	2	1.12	0	0.00

Model-Skill invocation patterns. The Model-Skill distributions are highly similar across the two upstream frameworks. The dominant transformations in both cases are Formulation-Refinement Skills rather than global Model-Structure transformations. Structural Domain Reduction is the most frequently invoked Model Skill, appearing in 73 OptiMUS tasks and 66 LLMOPT tasks. Bound Tightening follows closely with 60 and 59 invocations, while Symmetry Breaking is activated in 38 and 36 tasks, respectively. These three transformations therefore constitute the most common mathematical interventions performed by EffiOR.

This pattern suggests that a substantial fraction of the generated models are already structurally recognizable as valid optimization formulations, but still contain unnecessary decision domains, loose bounds, or redundant symmetric representations. Such deficiencies can often be addressed without replacing the overall formulation architecture. The similar invocation frequencies under OptiMUS and LLMOPT further indicate that these opportunities are not artifacts of one specific upstream modeling framework.

A second group of frequently used techniques consists of strengthening and dynamic-constraint transformations. Valid Inequality Strengthening and Constraint Generation are each invoked on 26 tasks for both OptiMUS and LLMOPT. Exact Discrete Product Linearization is invoked on 14 tasks under both frameworks. Compared with domain reduction or bound tightening, these transformations require more specific mathematical structures, but their repeated occurrence shows that they remain broadly relevant across multiple problem families in FrontierOR.

In contrast, Model-Structure Skills are activated much less frequently. Benders Decomposition is used on 11 OptiMUS tasks and 10 LLMOPT tasks, Explicit Pattern Enumeration on 8 and 7 tasks, and Column Generation on 6 tasks under both frameworks. Branch-and-Price and State-Space Arc-Flow are even more selective. This difference is expected because these transformations change the global representation or solving architecture rather than locally strengthening an existing formulation. Their applicability therefore requires more restrictive structural conditions, such as decomposable recourse, a configuration-based representation, a pricing problem, or an exploitable state-space structure.

The close agreement between OptiMUS and LLMOPT is particularly visible for Column Generation, Valid Inequality Strengthening, Constraint Generation, Exact Discrete Product Linearization, and Disjunctive Convex Hull Reformulation, whose invocation counts are identical under the two frameworks. The main differences are relatively small: Structural Domain Reduction is used seven more times for OptiMUS, whereas Perspective Reformulation appears three more times for LLMOPT. Overall, the results indicate that the Model-Skill selector is primarily responding to properties of the optimization formulation rather than to framework-specific generation patterns.

Solver-Implementation Skill invocation patterns. The Solver-Implementation Skills exhibit a markedly more concentrated distribution than the Model Skills. The two

Table 6: Actual invocation statistics of the 12 Solver-Implementation Skills on FrontierOR-179.

		OptiMUS		LLMOPT	
ID	**Solver-Implementation Skill**	**Count**	**Rate (%)**	**Count**	**Rate (%)**
B1	`sparse_index_precomputation`	38	21.23	21	11.73
B2	`sparse_matrix_constraint_assembly`	124	69.27	117	65.36
B3	`bulk_attribute_and_update_management`	133	74.30	78	43.58
P1	`presolve_policy_selection`	3	1.68	6	3.35
A1	`relaxation_algorithm_selection`	2	1.12	8	4.47
S2	`hint_and_branch_priority_guidance`	8	4.47	4	2.23
C1	`callback_data_path_optimization`	18	10.06	15	8.38
C2	`optional_cut_separation_scheduling`	1	0.56	0	0.00
R1	`persistent_model_reoptimization`	11	6.15	16	8.94
R2	`lp_basis_warm_start`	3	1.68	3	1.68
H1	`independent_subproblem_parallelism`	7	3.91	11	6.15
H2	`solver_thread_and_portfolio_selection`	3	1.68	5	2.79

model-construction transformations, `sparse_matrix_constraint_assembly` and `bulk_attribute_and_update_management`, account for the largest number of code-level interventions under both upstream frameworks. For OptiMUS, they are invoked in 124 and 133 tasks, corresponding to 69.27% and 74.30% of FrontierOR-179. For LLMOPT, the corresponding counts are 117 and 78 tasks. This shows that a large fraction of the implementation optimization opportunities arise before or during model construction itself, rather than exclusively inside the solver's search procedure.

The high frequency of sparse matrix and constraint assembly is especially stable across frameworks: it appears in 69.27% of OptiMUS outputs and 65.36% of LLMOPT outputs. This suggests that inefficient row-by-row or scalar-oriented construction patterns are common in solver programs generated by both upstream frameworks. Bulk attribute and update management is even more frequent for OptiMUS, appearing in 74.30% of tasks, but decreases to 43.58% for LLMOPT. This is the largest framework-level difference among the Solver-Implementation Skills and indicates that the two upstream systems differ more strongly in how they manipulate solver objects and attributes than in how they assemble sparse constraints.

Sparse index precomputation forms a second construction-related pattern. It is invoked in 38 OptiMUS tasks but only 21 LLMOPT tasks. Together, the B1–B3 results show that implementation inefficiency is frequently associated with how the mathematical model is materialized in Python: unnecessary dense index generation, repeated scalar construction, and repeated attribute or model updates provide substantially more optimization opportunities than most low-level solver-control mechanisms.

Among the solver-stage Skills, callback data-path optimization is the most frequently used, with 18 invocations for OptiMUS and 15 for LLMOPT. This indicates that callback-enabled programs frequently contain avoidable data-access or repeated-computation overhead once callbacks are present. Persistent model reoptimization is also observed repeatedly, with 11 OptiMUS and 16 LLMOPT invocations. The higher LLMOPT count suggests that LLMOPT-generated programs more often expose repeated-solve structures for which rebuilding the entire model can be replaced by controlled reuse of an existing solver model.

The remaining Solver-Implementation Skills are substantially more selective. Hint and branch-priority guidance appears in 8 OptiMUS tasks and 4 LLMOPT tasks; independent-subproblem parallelism appears in 7 and 11 tasks, respectively; and solver thread and portfolio selection appears in only 3 and 5 tasks. Presolve policy selection, relaxation algorithm selection, LP-basis warm starts, and optional cut-separation scheduling are similarly rare. These Skills depend on relatively specific execution structures or solver-state requirements and therefore should not be expected to activate broadly across the benchmark.

There are also meaningful differences between the two upstream frameworks. OptiMUS produces more opportunities for sparse index precomputation, bulk attribute/update management, hint and branch-priority guidance, and callback data-path optimization. LLMOPT, in contrast, produces more cases requiring presolve policy selection, relaxation algorithm selection, persistent-model reoptimization, independent-subproblem parallelism, and solver-resource configuration. These differences show that two upstream systems can generate mathematically similar optimization models while still producing solver implementations with substantially different computational structures.

Relationship between Model and Solver-Implementation Skills. Taken together, the two invocation profiles reveal that EffiOR performs different types of intervention at different stages of the optimization pipeline. At the mathematical level, the most common modifications are Structural Domain Reduction, Bound Tightening, and Symmetry Breaking. At the implementation level, the most common modifications concern efficient construction and manipulation of the solver model, particularly sparse constraint assembly and bulk operations.

This distinction is important because mathematical and implementation inefficiencies are not interchangeable. A formulation may already have a reasonable mathematical structure but still be realized inefficiently in code; conversely, an efficiently written solver program may implement a formulation with unnecessarily large domains, weak bounds, or exploitable symmetry. The observed invocation profiles therefore support the hierarchical design of EffiOR: Model Skills and Solver-Implementation Skills target complementary sources of inefficiency rather than duplicate the same transformation.

Finally, invocation frequency should be interpreted as evidence of how often a particular optimization opportunity is encountered, rather than as a direct measure of its performance contribution. Highly frequent Skills such as Structural Domain Reduction or sparse matrix assembly represent broadly recurring opportunities, whereas less frequently activated Skills such as Benders Decomposition, Column Generation, or independent-subproblem parallelism apply only when stronger structural conditions are satisfied. Consequently, this analysis complements the aggregate FrontierOR results and the ablation study by showing which components of EffiOR are actually exercised across the benchmark and how their usage differs between upstream modeling frameworks.

B Optimization Skill Catalog and Detailed Case Studies

B.1 Model-Structure Skills

The following seven Skills modify the global representation or solution structure of the mathematical model while preserving the semantics of the original optimization problem.

B.1.1 Benders Decomposition

Trigger Conditions. Apply `benders_decomposition` when the decision variables admit a master–recourse partition in which fixing the master variables makes the remaining optimization problem substantially easier, decomposable, or continuously solvable. A valid application requires all of the following structural properties:

1. The variables can be partitioned into master variables y and recourse variables x such that the recourse problem depends on y through a known affine or otherwise certifiable linking structure.
2. For every relevant fixed y, the recourse value $Q(y)$ can be evaluated or bounded by a tractable subproblem. In the classical case, the recourse subproblem is a linear program with a valid dual; integer or logic-based recourse requires a separate exact value or logic certificate.
3. Feasibility and optimality information from the recourse problem can be converted into globally valid inequalities in the master space. A timeout, a heuristic incumbent, or an uncertified dual object is not sufficient.
4. Replacing the explicit recourse block by a master approximation reduces repeated structure or avoids constructing a large monolithic model.

Do not use this transformation when fixing y does not simplify the remaining problem, when valid master cuts cannot be derived, or when an integer recourse problem would be incorrectly replaced by the dual of its LP relaxation.

Mathematical Principle. For a minimization problem of the form

$$\min_{y \in Y} \; f^\top y + Q(y),$$

with

$$Q(y) = \min_{x \geq 0} \left\{ q^\top x : Wx \geq h - Ty \right\},$$

classical Benders decomposition replaces the explicit recourse variables x in the master by an auxiliary variable θ satisfying $\theta \geq Q(y)$. The dual recourse set is

$$\mathcal{D} = \left\{ \pi \geq 0 : W^\top \pi \leq q \right\}.$$

For every dual-feasible extreme point $\pi^r \in \mathcal{D}$, weak duality yields the globally valid optimality cut

$$\theta \geq (\pi^r)^\top (h - Ty).$$

If the recourse problem is infeasible, an admissible dual ray r yields a feasibility cut that removes only master decisions for which the recourse system is infeasible. The master therefore evolves as

$$\min_{y,\theta} \; f^\top y + \theta$$

subject to the original master constraints and the accumulated Benders cuts. The method is exact when the cut-generation mechanism is complete for the chosen recourse class and lower and upper bounds are maintained separately.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A facility-location problem chooses which candidate facilities to open and assigns every customer to an open facility. Opening facility j incurs fixed cost f_j, and serving customer i from facility j incurs assignment cost c_{ij}. The objective is to minimize opening plus assignment cost while assigning every customer exactly once. The large assignment block is continuous once the facility-opening vector is fixed, which creates the required master–recourse structure Fischetti et al. (2017).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let $y_j \in \{0,1\}$ indicate whether facility j is open and let $x_{ij} \geq 0$ denote the fraction of customer i assigned to facility j. A compact formulation is

$$\min \quad \sum_{j \in J} f_j y_j + \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}, \tag{12}$$

$$\text{s.t.} \quad \sum_{j \in J} x_{ij} = 1 \qquad \forall i \in I, \tag{13}$$

$$x_{ij} \leq y_j \qquad \forall i \in I,\ j \in J, \tag{14}$$

$$y_j \in \{0,1\}, \quad x_{ij} \geq 0. \tag{15}$$

The baseline code explicitly materializes the full allocation block:

```
m = gp.Model()
y = m.addVars(J, vtype=GRB.BINARY, name="open")
x = m.addVars(I, J, lb=0.0, name="assign")
m.addConstrs(gp.quicksum(x[i,j] for j in J) == 1 for i in I)
m.addConstrs(x[i,j] <= y[j] for i in I for j in J)
m.setObjective(
    gp.quicksum(f[j]*y[j] for j in J) +
    gp.quicksum(c[i,j]*x[i,j] for i in I for j in J),
    GRB.MINIMIZE)
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Keep only y in the master and replace the explicit customer-allocation block by recourse variables θ_i. For a fixed $\bar{y}$, customer i solves

$$Q_i(\bar{y}) = \min_{x_i \geq 0} \left\{ \sum_j c_{ij} x_{ij} : \sum_j x_{ij} = 1,\ x_{ij} \leq \bar{y}_j \right\}.$$

Writing $-x_{ij} \geq -\bar{y}_j$, a dual representation uses a free variable α_i for the assignment equality and $\beta_{ij} \geq 0$ for the upper-bound rows:

$$\max \; \alpha_i - \sum_j \beta_{ij} \bar{y}_j, \qquad \alpha_i - \beta_{ij} \leq c_{ij}.$$

Every certified dual solution therefore generates

$$\theta_i \geq \alpha_i^r - \sum_j \beta_{ij}^r y_j.$$

The code changes from one monolithic model to a master plus repeatedly solved customer recourse problems; only violated certified cuts are appended to the master.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The Benders master is

$$\min \quad \sum_{j \in J} f_j y_j + \sum_{i \in I} \theta_i, \tag{16}$$

$$\text{s.t.} \quad \theta_i \geq \alpha_i^r - \sum_{j \in J} \beta_{ij}^r y_j \qquad \forall i \in I,\ r \in \mathcal{R}_i, \tag{17}$$

$$y_j \in \{0,1\}, \quad \theta_i \geq \underline{Q}_i. \tag{18}$$

A direct outer-loop implementation is:

```
master = gp.Model()
y = master.addVars(J, vtype=GRB.BINARY, name="open")
theta = master.addVars(I, lb=theta_lb, name="theta")
master.setObjective(gp.quicksum(f[j]*y[j] for j in J) + theta.sum())

while True:
    master.optimize()
    y_bar = {j: y[j].X for j in J}
    violated = False
    for i in I:
        cert = solve_assignment_recourse_and_dual(i, y_bar, c)
        if cert.infeasible:
            add_benders_feasibility_cut(master, y, cert)
            violated = True
        elif theta[i].X + eps < cert.value:
            master.addConstr(
                theta[i] >= cert.alpha -
                gp.quicksum(cert.beta[j] * y[j] for j in J))
            violated = True
    if not violated:
        break
```

B.1.2 DANTZIG–WOLFE DECOMPOSITION

Trigger Conditions. Apply `dantzig_wolfe_decomposition` when removing a small set of linking constraints decomposes the model into meaningful independent blocks. The transformation is appropriate when each block has a nontrivial local feasible set, the original coupling can be represented through aggregate block coefficients, and local solutions can be reconstructed exactly. A typical structural signature is a block-angular matrix in which local constraints are block diagonal and only a limited number of rows couple blocks. The reformulation is useful when the convex hull of each local block is stronger or more compactly represented by its extreme points or integer configurations. Do not use the method if the proposed linking rows do not actually separate the problem, if cross-block dependencies would be dropped, or if the local representation has no identifiable benefit.

Mathematical Principle. Consider

$$\min \quad \sum_{i \in I} c_i^\top x_i, \tag{19}$$

$$\text{s.t.} \quad \sum_{i \in I} A_i x_i \leq b, \tag{20}$$

$$x_i \in X_i, \qquad i \in I, \tag{21}$$

where the first row block contains the linking constraints and X_i contains all local constraints for block i. Let $\{x_i^p : p \in P_i\}$ denote the relevant extreme points or integer configurations of X_i. Represent

$$x_i = \sum_{p \in P_i} x_i^p \lambda_{ip}, \qquad \sum_{p \in P_i} \lambda_{ip} = 1, \qquad \lambda_{ip} \geq 0.$$

Substitution yields the Dantzig–Wolfe master

$$\min \quad \sum_i \sum_{p \in P_i} c_{ip} \lambda_{ip}, \tag{22}$$

$$\text{s.t.} \quad \sum_i \sum_{p \in P_i} a_{ip} \lambda_{ip} \leq b, \tag{23}$$

$$\sum_{p \in P_i} \lambda_{ip} = 1 \qquad \forall i, \tag{24}$$

$$\lambda_{ip} \geq 0, \tag{25}$$

where $c_{ip} = c_i^\top x_i^p$ and $a_{ip} = A_i x_i^p$. The reformulation changes the representation of each local block but retains all linking constraints. Column generation is an optional solution mechanism for a large P_i; it is not part of the Dantzig–Wolfe transformation itself.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A capacitated lot-sizing problem plans production for multiple items across periods. Each item has demand, setup decisions, production quantities, and inventory balance. Items are locally independent except that they consume a shared production capacity in each period. Fixing the shared capacity rows separates the model into independent item-level planning blocks Degraeve and Jans (2007).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let x_{it} be production, y_{it} setup, and I_{it} ending inventory. A compact model is

$$\min \quad \sum_{i,t} \left(f_{it} y_{it} + c_{it} x_{it} + h_{it} I_{it} \right), \tag{26}$$

$$\text{s.t.} \quad I_{i,t-1} + x_{it} = d_{it} + I_{it} \qquad \forall i, t, \tag{27}$$

$$x_{it} \leq M_{it} y_{it} \qquad \forall i, t, \tag{28}$$

$$\sum_i \left(p_i x_{it} + s_i y_{it} \right) \leq C_t \qquad \forall t, \tag{29}$$

$$y_{it} \in \{0, 1\}, \quad x_{it}, I_{it} \geq 0. \tag{30}$$

```
x = m.addVars(I, T, lb=0.0, name="prod")
y = m.addVars(I, T, vtype=GRB.BINARY, name="setup")
inv = m.addVars(I, T, lb=0.0, name="inv")

m.addConstrs(inv[i,t-1] + x[i,t] == demand[i,t] + inv[i,t]
             for i in I for t in T if t > 0)
m.addConstrs(x[i,t] <= M[i,t] * y[i,t] for i in I for t in T)
m.addConstrs(gp.quicksum(proc[i]*x[i,t] + st[i]*y[i,t] for i in I)
             <= cap[t] for t in T)
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Remove the explicit item-period decision block from the master. For each item i, define P_i as the complete set of feasible item plans satisfying inventory balance, setup logic, and item-specific restrictions. For plan p, precompute its cost c_{ip} and period-capacity consumption a_{itp}. Replace (x_{it}, y_{it}, I_{it}) in the master by one plan-selection variable λ_{ip}. The item balance and setup constraints move into the definition of a valid plan; the shared capacity rows remain explicitly in the master. Code generation therefore changes from directly creating item-period variables to constructing or querying item-plan objects and then using their precomputed master coefficients.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The master becomes

$$\min \quad \sum_i \sum_{p \in P_i} c_{ip} \lambda_{ip}, \tag{31}$$

$$\text{s.t.} \quad \sum_{p \in P_i} \lambda_{ip} = 1 \qquad \forall i, \tag{32}$$

$$\sum_i \sum_{p \in P_i} a_{itp} \lambda_{ip} \le C_t \qquad \forall t, \tag{33}$$

$$\lambda_{ip} \ge 0. \tag{34}$$

If integer recovery is required, the chosen plan representation and the master integrality treatment must provide it explicitly.

```
plans = {i: build_complete_item_plans(i, data) for i in I}
keys = [(i,p) for i in I for p in plans[i]]
lam = m.addVars(keys, lb=0.0, name="lambda")

m.addConstrs(
    gp.quicksum(lam[i,p] for p in plans[i]) == 1
    for i in I)
m.addConstrs(
    gp.quicksum(plans[i][p].capacity[t] * lam[i,p]
                for i in I for p in plans[i]) <= cap[t]
    for t in T)
m.setObjective(
    gp.quicksum(plans[i][p].cost * lam[i,p]
                for i in I for p in plans[i]),
    GRB.MINIMIZE)
```

B.1.3 EXPLICIT PATTERN ENUMERATION

Trigger Conditions. Apply `explicit_pattern_enumeration` when a repeated local decision can be represented by a finite family of feasible configurations and that complete family can be enumerated within the available construction and storage budget. The local feasibility test must be exact, every configuration index must have a justified finite domain, and every original feasible solution must admit a decomposition into the enumerated configurations. The method is especially suitable for repeated cutting, packing, machine-mode, shift, or process configurations with identical local rules. Do not use it when only a sampled or heuristic pattern pool is available, when the family-size estimate exceeds the budget, or when the existing model already contains the same complete family.

Mathematical Principle. Let a local pattern be an integer vector $a^p = (a_{1p}, \ldots, a_{mp})$ satisfying a local resource system

$$Ra^p \le r, \qquad a^p \in \mathbb{Z}_+^m.$$

If the feasible family

$$P = \{p : a^p \text{ satisfies the complete local rules}\}$$

is finite and manageable, enumerate P before optimization and freeze every coefficient a_{ip}. A master then chooses only pattern multiplicities n_p:

$$\min \quad \sum_{p \in P} c_p n_p, \tag{35}$$

$$\text{s.t.} \quad \sum_{p \in P} a_{ip} n_p \ge d_i \qquad \forall i, \tag{36}$$

$$n_p \in \mathbb{Z}_+. \tag{37}$$

This removes endogenous pattern-composition variables and avoids products between pattern design and pattern use. Exactness depends on complete enumeration and a decoder from selected patterns back to the original decisions.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A production-configuration instance repeatedly chooses feasible local configurations to satisfy product demand. Each configuration contains integer quantities of several products and must obey a common capacity rule. The audited small instance has six product dimensions and a finite complete configuration family; exhaustive enumeration produces 291 nonzero feasible patterns, making explicit enumeration preferable to keeping configuration composition endogenous Reference case associated with FrontierOR delage2022 (2022).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. A fragile endogenous-pattern model can use a_{ip} as the quantity of product i inside pattern slot p and n_p as the number of times slot p is used:

$$\min \quad \sum_{p \in P} n_p, \tag{38}$$

$$\text{s.t.} \quad \sum_{i} \ell_i a_{ip} \le C \qquad \forall p, \tag{39}$$

$$\sum_{p} a_{ip} n_p \ge d_i \qquad \forall i, \tag{40}$$

$$a_{ip}, n_p \in \mathbb{Z}_+. \tag{41}$$

The coverage term $a_{ip}n_p$ is bilinear and the arbitrary slot set P does not by itself guarantee completeness.

```
pattern_qty = m.addVars(P, I, vtype=GRB.INTEGER, lb=0)
use = m.addVars(P, vtype=GRB.INTEGER, lb=0)
for p in P:
    m.addConstr(gp.quicksum(length[i]*pattern_qty[p,i] for i in I) <=
    capacity)
# coverage would contain pattern_qty[p,i] * use[p]
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Move pattern composition outside the optimization model. Enumerate every nonzero integer vector a satisfying the complete configuration rules, canonicalize and deduplicate it, and freeze the resulting coefficients. Remove all a_{ip} decision variables and their local-capacity rows from the master; keep only integer pattern-use variables n_p. The bilinear terms disappear because a_{ip} becomes data. Code generation changes from creating pattern-composition variables to running a deterministic enumeration procedure before model construction.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. With the complete family P fixed, use

$$\min \quad \sum_{p \in P} n_p, \tag{42}$$

$$\text{s.t.} \quad \sum_{p \in P} a_{ip} n_p \ge d_i \qquad \forall i, \tag{43}$$

$$n_p \in \mathbb{Z}_+ \qquad \forall p. \tag{44}$$

```
patterns = enumerate_all_feasible_patterns(lengths, capacity)
patterns = canonicalize_and_deduplicate(patterns)
assert len(patterns) == 291   # audited small instance only

use = m.addVars(range(len(patterns)), vtype=GRB.INTEGER,
                lb=0, name="use")
m.addConstrs(
    gp.quicksum(patterns[p][i] * use[p] for p in range(len(patterns)))
    >= demand[i] for i in I)
m.setObjective(use.sum(), GRB.MINIMIZE)
```

B.1.4 COLUMN GENERATION

Trigger Conditions. Apply `column_generation` when the complete LP master contains a prohibitively large family of variables, each variable represents a well-defined feasible local object, and reduced-cost minimization over the complete column universe can be solved or bounded exactly. The pricing problem must incorporate the dual coefficients of every active master row that contributes to reduced cost. The method is appropriate when columns are routes, rosters, schedules, patterns, or other combinatorial objects that are too numerous to enumerate explicitly. Do not use column generation if only a heuristic column pool is available but an LP-optimality certificate is required, or if all relevant columns already fit comfortably in memory.

Mathematical Principle. Consider the full LP master

$$\min \quad \sum_{r \in R} c_r \lambda_r, \tag{45}$$

$$\text{s.t.} \quad \sum_{r \in R} a_{ir} \lambda_r = b_i \qquad \forall i, \tag{46}$$

$$\lambda_r \geq 0. \tag{47}$$

For a restricted set $R' \subset R$, solve the restricted master and obtain dual variables π_i. The reduced cost of any legal column $r \in R$ is

$$\bar{c}_r = c_r - \sum_i \pi_i a_{ir}.$$

The pricing problem minimizes $\bar{c}_r$ over the *complete* legal column space. A column with $\bar{c}_r < 0$ is added to the restricted master. When a valid pricing lower bound proves

$$\min_{r \in R} \bar{c}_r \geq -\varepsilon,$$

the current restricted-master solution is optimal for the full LP. Integer optimality does not follow automatically; it requires complete enumeration, Branch-and-Price, or another exact integer mechanism.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. An aircrew rostering problem must cover required activities with legal crew rosters. A roster is a complete legal sequence satisfying duty, rest, and operational rules, and has a known cost. The set of legal personalized rosters is enormous, while reduced-cost roster generation can be posed as a structured pricing problem Gamache et al. (1999).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let R be the set of all legal rosters, $a_{ir} = 1$ if roster r covers required activity i, and c_r its cost:

$$\min \quad \sum_{r \in R} c_r \lambda_r, \tag{48}$$

$$\text{s.t.} \quad \sum_{r \in R} a_{ir} \lambda_r = 1 \qquad \forall i, \tag{49}$$

$$\lambda_r \geq 0. \tag{50}$$

A direct implementation would first enumerate the entire family:

```
all_rosters = enumerate_all_legal_rosters(data)
lam = master.addVars(all_rosters, lb=0.0, name="roster")
master.addConstrs(
    gp.quicksum(a[i,r] * lam[r] for r in all_rosters) == 1
    for i in I)
master.setObjective(
    gp.quicksum(cost[r] * lam[r] for r in all_rosters), GRB.MINIMIZE)
```

(3) How the Mathematical Formulation and Code Are Modified. Replace the full column set R by a dynamic subset R'. Keep the mathematical definition of the full master unchanged, but construct only the current restricted master. After each LP solve, read the dual vector π and solve

$$\min_{r \in R} \left(c_r - \sum_i \pi_i a_{ir} \right)$$

over all legal rosters. Add every validated negative-reduced-cost roster needed by the policy, then reoptimize. Code therefore changes from one-shot full enumeration to an iterative *solve–price–add* loop with a pricing certificate.

(4) Modified Mathematical Formulation and Code. At iteration k, solve

$$\min \quad \sum_{r \in R_k} c_r \lambda_r, \tag{51}$$

$$\text{s.t.} \quad \sum_{r \in R_k} a_{ir} \lambda_r = 1 \qquad \forall i, \tag{52}$$

$$\lambda_r \geq 0, \tag{53}$$

then solve exact pricing over R. Terminate only after certifying that no omitted legal column has negative reduced cost.

```
R = seed_feasible_rosters(data)
master, cover_rows, lam = build_continuous_rmp(R)

while True:
    master.optimize()
    pi = {i: cover_rows[i].Pi for i in I}
    price = solve_exact_roster_pricing(pi, data)
    if price.lower_bound >= -eps:
        break
    for r in validated_negative_columns(price, pi):
        add_roster_column(master, lam, r, cover_rows)
```

B.1.5 Branch-and-Price

Trigger Conditions. Apply `branch_and_price` when the integer model is naturally expressed with an implicit column family, root column generation alone does not certify the required integer optimum, and exact pricing remains available after branching. The branching rule must partition the full integer feasible set and its restrictions must be enforceable both on existing columns and on future columns generated by pricing. A valid implementation therefore requires an explicit node lifecycle: solve the node restricted master, price to a certified full-node LP bound, fathom or branch, and propagate all active branching decisions into descendant pricing problems. Do not treat "root column generation followed by an integer solve over the generated columns" as exact Branch-and-Price; that procedure is only a restricted-master heuristic unless all omitted columns are proved irrelevant.

Mathematical Principle. Let the integer master be

$$\min \quad \sum_{p \in P} c_p \lambda_p, \tag{54}$$

$$\text{s.t.} \quad A\lambda \geq b, \tag{55}$$

$$\lambda_p \in \mathbb{Z}_+. \tag{56}$$

At branch-and-bound node n, let B_n denote all branch restrictions inherited from the root. The node LP is still defined over the complete column universe:

$$\min\{c^\top \lambda : A\lambda \geq b,\ B_n \lambda \leq d_n,\ \lambda \geq 0\}.$$

Column generation solves this LP without explicit enumeration. Its pricing problem must include dual contributions from both the original master rows and B_n. Only after pricing certifies that no

improving node-feasible column exists is the resulting LP value a valid node lower bound. If the node solution is fractional, a pricing-compatible branch rule creates child partitions whose union contains every integer solution represented at the parent. Exactness follows from complete node pricing, valid branching partitions, and exhaustive search or certified fathoming.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A one-dimensional bin-packing problem has item types with sizes w_i and demands d_i. A feasible packing pattern describes the multiplicity of every item type placed in one bin of capacity C. The pattern family is too large to enumerate, so root LP column generation is natural; however, the LP solution may remain fractional and an exact integer optimum is required de Carvalho (1999).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. For every feasible pattern $p \in P$, let a_{ip} be the number of items of type i in the pattern and let λ_p denote the number of bins using it:

$$\min \quad \sum_{p \in P} \lambda_p, \tag{57}$$

$$\text{s.t.} \quad \sum_{p \in P} a_{ip} \lambda_p \geq d_i \qquad \forall i, \tag{58}$$

$$\lambda_p \in \mathbb{Z}_+. \tag{59}$$

A non-exact shortcut first runs root column generation and then changes only the generated variables to integer:

```
R = run_column_generation_at_root()
for v in R.values():
    v.VType = GRB.INTEGER
restricted_master.optimize()   # exact only over the restricted set R
```

This can return a good integer solution but does not prove that an omitted pattern could not improve it.

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Retain the same full integer master, but replace the single restricted-MIP solve by an explicit branch-and-bound tree. Each node stores branch rows B_n, its active column set, and inherited bounds. Before using a node LP value for fathoming, solve column generation under B_n until pricing certifies the full node LP. If fractional, branch on a rule that can be represented in the pricing problem, such as compatible assignment/pairing decisions or other problem-specific disjunctions. The key code change is therefore architectural: pricing is no longer a root preprocessing step; it becomes part of every node evaluation.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. For each node n, solve the restricted representation of

$$L_n = \min \quad \sum_{p \in P_n} c_p \lambda_p, \tag{60}$$

$$\text{s.t.} \quad \sum_{p \in P_n} a_{ip} \lambda_p \geq d_i \qquad \forall i, \tag{61}$$

$$B_n \lambda \leq d_n, \tag{62}$$

$$\lambda_p \geq 0, \tag{63}$$

while pricing over the complete set of patterns satisfying B_n. Node n may be fathomed only after L_n is certified for the full node column space.

```
queue = [root_node()]
incumbent = GRB.INFINITY

while queue:
```

```
    node = queue.pop()
    rmp = build_node_rmp(node.columns, node.branch_rows)
    cg = price_node_to_certification(rmp, node.branch_rows, data)

    if cg.full_node_bound >= incumbent - eps:
        continue
    if cg.solution_is_integer:
        incumbent = update_incumbent(cg, incumbent)
    else:
        left, right = pricing_compatible_branch(cg, node)
        queue.extend([left, right])
```

B.1.6 STATE-SPACE ARC-FLOW FORMULATION

Trigger Conditions. Apply `state_space_arc_flow_formulation` when the problem contains repeated local configurations that can be represented exactly by paths in a finite state graph. The following problem properties should hold:

1. A repeated local object exists, such as one bin packing, one cutting pattern, one machine schedule, or another reusable configuration.
2. A finite state s summarizes all information required to determine the feasibility and contribution of future local decisions. A transition has the form $s' = T(s, a)$ for local decision a.
3. Every required feasible configuration maps to a source-to-sink path and every accepted path decodes to a feasible original configuration.
4. Different configurations share states or transitions, so graph representation can avoid repeated structure.
5. The exact reachable graph $G = (V, A)$ fits the construction, memory, and solution budget. Heuristic pruning or width restriction must not be presented as an exact reformulation.

Mathematical Principle. Let $\mathcal{P}$ be the set of feasible local configurations. Build a state graph $G = (V, A)$ so that each transition $a = (u, v, j)$ encodes a local action j and each valid source-to-sink path corresponds to one $p \in \mathcal{P}$. Integer arc-flow variables f_a count how many selected configurations use each transition. Internal states satisfy flow conservation,

$$\sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a, \qquad v \in V \setminus \{s, t\},$$

and side constraints count the contribution of labeled arcs to global demand. The transformation is

explicit repeated configurations $\implies$ shared state-space network flow.

Graph compression is exact only if the merged states preserve all future feasibility, cost, linking coefficients, and the path decoder.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. Items are partitioned into types $j \in J$. Type j has size w_j and demand d_j. All bins have capacity C. Every item must be packed, no bin may exceed capacity, and the number of used bins is minimized. A single bin is governed by the same local capacity rule regardless of its label; its construction can be described by the accumulated used capacity. This creates an exact finite-state representation when the reachable capacity graph is manageable Brandão and Pedroso (2016).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let B be a sufficiently large candidate-bin set. Define $x_{jb} \in \mathbb{Z}_+$ as the number of type-j items assigned to bin b and $y_b \in \{0, 1\}$

as the bin-use indicator:

$$\min \quad \sum_{b\in B} y_b, \tag{64}$$

$$\text{s.t.} \quad \sum_{b\in B} x_{jb} = d_j \qquad \forall j \in J, \tag{65}$$

$$\sum_{j\in J} w_j x_{jb} \le C y_b \qquad \forall b \in B, \tag{66}$$

$$x_{jb} \in \mathbb{Z}_+, \quad y_b \in \{0,1\}. \tag{67}$$

```
x = m.addVars(types, bins, vtype=GRB.INTEGER, lb=0, name="x")
y = m.addVars(bins, vtype=GRB.BINARY, name="y")
m.addConstrs(
    gp.quicksum(x[j,b] for b in bins) == demand[j]
    for j in types)
m.addConstrs(
    gp.quicksum(weight[j]*x[j,b] for j in types) <= capacity*y[b]
    for b in bins)
m.setObjective(gp.quicksum(y[b] for b in bins), GRB.MINIMIZE)
```

(3) How the Mathematical Formulation and Code Are Modified. Remove the explicit candidate-bin index. Use the accumulated load u as the state. For every item type j, add an item arc (u, v, j) whenever

$$v = u + w_j \le C.$$

If a configuration ends before capacity C, add an exact loss/completion arc to the common sink. Replace x_{jb} by integer arc-flow variables f_{uvj} and replace $\sum_b y_b$ by a total source-to-sink flow variable z. Per-bin capacity constraints disappear because infeasible capacity transitions are absent from the graph. Their structural role is replaced by valid-arc construction and flow conservation:

x_{jb}	$\Longrightarrow$	f_{uvj}
candidate bins	$\Longrightarrow$	capacity states
per-bin capacity rows	$\Longrightarrow$	flow conservation
$\sum_b y_b$	$\Longrightarrow$	z.

The code therefore constructs the graph first and creates variables only on its arcs.

(4) Modified Mathematical Formulation and Code. Let $A = A^{\text{item}} \cup A^{\text{loss}}$ and

$$A_j = \{(u, v, j) : v = u + w_j,\ v \le C\}.$$

The exact arc-flow model is

$$\min \quad z, \tag{68}$$

$$\text{s.t.} \quad \sum_{a\in\delta^+(0)} f_a - \sum_{a\in\delta^-(0)} f_a = z, \tag{69}$$

$$\sum_{a\in\delta^-(v)} f_a = \sum_{a\in\delta^+(v)} f_a \qquad \forall v \in V \setminus \{0, C\}, \tag{70}$$

$$\sum_{a\in\delta^-(C)} f_a - \sum_{a\in\delta^+(C)} f_a = z, \tag{71}$$

$$\sum_{(u,v,j)\in A_j} f_{uvj} = d_j \qquad \forall j \in J, \tag{72}$$

$$f_a \in \mathbb{Z}_+, \quad z \in \mathbb{Z}_+. \tag{73}$$

```
states, item_arcs, loss_arcs = build_arc_flow_graph(weight, capacity)
arcs = item_arcs + loss_arcs
```

```
f = m.addVars(arcs, vtype=GRB.INTEGER, lb=0, name="flow")
z = m.addVar(vtype=GRB.INTEGER, lb=0, name="num_bins")

m.addConstr(outflow(0, f) - inflow(0, f) == z)
for v in states:
    if v not in (0, capacity):
        m.addConstr(inflow(v, f) == outflow(v, f))
m.addConstr(inflow(capacity, f) - outflow(capacity, f) == z)
for j in types:
    m.addConstr(gp.quicksum(f[a] for a in item_arcs[j]) == demand[j])
m.setObjective(z, GRB.MINIMIZE)
```

B.1.7 Constraint-and-Column Generation

Trigger Conditions. Apply `constraint_and_column_generation` to a two-stage robust optimization model when first-stage decisions are made before uncertainty is revealed, recourse adapts after the uncertainty realization, the full scenario expansion is too large, and a valid adversarial oracle can identify a worst-case scenario or certify a bound over the complete uncertainty set. The uncertainty set and information timing must already be part of the problem; they must not be invented from missing data. The recourse block for one scenario must be well defined and scenario-specific. Do not use C&CG for a stochastic expectation by silently replacing it with a maximum, and do not claim exact robust optimality from a merely local adversarial solve.

Mathematical Principle. A canonical two-stage robust problem is

$$\min_{y \in Y} \left\{ c^\top y + \max_{u \in \mathcal{U}} Q(y, u) \right\},$$

where

$$Q(y, u) = \min_{x} \{ q^\top x : Wx \geq h(u) - Ty,\ x \in X(u) \}.$$

Instead of expanding all $u \in \mathcal{U}$, maintain a finite discovered scenario set S. The restricted robust master contains y, an epigraph variable η, and a separate recourse block x^s for each $s \in S$:

$$\begin{aligned} \min \quad & c^\top y + \eta, && && (74)\\ \text{s.t.} \quad & Wx^s \geq h(u^s) - Ty && \forall s \in S, && (75)\\ & \eta \geq q^\top x^s && \forall s \in S. && (76) \end{aligned}$$

For the current $\bar{y}$, an adversarial subproblem searches the full $\mathcal{U}$ for the worst recourse realization u^*. If it violates the restricted master, append both u^* and a new scenario-specific recourse block. The master gives a lower bound and the evaluated robust objective of a complete feasible first-stage solution gives an upper bound.

Before-and-After Case Study.

(1) Natural Language Problem. A robust facility-location and transportation problem chooses facilities before demand is known. After uncertain demand is revealed, shipment quantities adapt to serve customers subject to the opened capacity. The objective minimizes first-stage opening cost plus the worst-case recourse transportation cost over a prescribed uncertainty set Zeng and Zhao (2013).

(2) Original Mathematical Formulation and Code. An extensive robust representation can be written as

$$\begin{aligned} \min \quad & f^\top y + \eta, && && (77)\\ \text{s.t.} \quad & x^u \in X(y, u) && \forall u \in \mathcal{U}_{\text{ext}}, && (78)\\ & \eta \geq q^\top x^u && \forall u \in \mathcal{U}_{\text{ext}}, && (79)\\ & y \in Y, && && (80) \end{aligned}$$

where $\mathcal{U}_{\text{ext}}$ denotes the relevant extreme scenarios. A direct implementation creates one recourse block for every scenario:

```
for u in enumerate_all_extreme_scenarios(U):
    x[u] = add_transport_recourse(master, y, u)
    master.addConstr(eta >= transport_cost(x[u]))
```

(3) How the Mathematical Formulation and Code Are Modified. Replace the full scenario index set by a dynamic subset S. Keep one independent recourse block per discovered scenario. Solve the restricted robust master to obtain $\bar{y}$, then solve an exact adversarial problem over the full uncertainty set. When a violating scenario u^* is found, append u^* and its recourse variables and rows to S. Unlike ordinary column generation, the generated object is an uncertainty realization together with a recourse block, not a reduced-cost column.

(4) Modified Mathematical Formulation and Code. At iteration k use

$$\min \quad f^\top y + \eta, \tag{81}$$
$$\text{s.t.} \quad x^s \in X(y, u^s) \qquad \forall s \in S_k, \tag{82}$$
$$\eta \geq q^\top x^s \qquad \forall s \in S_k, \tag{83}$$
$$y \in Y. \tag{84}$$

The update loop is:

```
S = [initial_feasible_scenario(data)]
LB, UB = -GRB.INFINITY, GRB.INFINITY

while UB - LB > eps:
    master = build_location_transport_master(S)
    master.optimize()
    y_bar = read_first_stage(master)
    LB = master.ObjVal

    worst = solve_exact_adversarial_subproblem(y_bar, data)
    UB = min(UB, first_stage_cost(y_bar) + worst.recourse_value)
    if worst.violates_master_bound(eps):
        S.append(worst.scenario)
    else:
        break
```

B.2 Formulation-Refinement Skills

The following ten Skills refine the current mathematical formulation without replacing its overall modeling structure.

B.2.1 Bound Tightening

Trigger Conditions. Apply `bound_tightening` when the current formulation contains loose finite or infinite variable bounds, or coarse global implication constants, and strictly stronger valid bounds can be derived from the original constraints, data, or certified auxiliary optimization. The derivation must be valid for every original feasible solution; observed incumbent values, default guesses, or unsupported business assumptions are not sufficient. The transformation is especially relevant to Big-M implications, time windows, resource limits, and variables whose natural bounds are much tighter than generic model-wide constants. If a proposed bound is not strictly stronger or is already supplied equivalently without an explicit modeling benefit, no meaningful patch is present.

Mathematical Principle. For an implication

$$x = 1 \implies a^\top z \leq b,$$

a Big-M formulation is

$$a^\top z \leq b + M(1 - x).$$

Any valid M must satisfy

$$M \geq \max\{a^\top z - b : z \in Z,\ x = 0\}.$$

A smaller valid value strengthens the continuous relaxation while leaving the active implication unchanged. More generally, replacing

$$L_j \leq z_j \leq U_j$$

by certified bounds

$$L'_j \leq z_j \leq U'_j, \qquad L'_j \geq L_j,\ U'_j \leq U_j,$$

preserves the original feasible set if every feasible solution already satisfies the new bounds. Bound tightening can therefore reduce LP slack, improve propagation, and shrink derived Big-M coefficients without changing the underlying decision semantics.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A pickup-and-delivery routing problem with time windows schedules service times along selected arcs. If arc (i, j) is used, service at j cannot begin before service at i plus service duration and travel time. Each node has an earliest and latest feasible service-start time. A single global horizon-based Big-M is valid but can be much larger than necessary Cherkesly et al. (2015).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let $x_{ij} \in \{0, 1\}$ select arc (i, j) and let t_i be the service-start time with

$$e_i \leq t_i \leq \ell_i.$$

The baseline precedence row is

$$t_j \geq t_i + s_i + \tau_{ij} - M(1 - x_{ij}),$$

where one global M may be chosen as the planning horizon.

```
BIG_M = horizon
for i, j in feasible_arcs:
    m.addConstr(
        t[j] >= t[i] + service[i] + travel[i,j]
        - BIG_M * (1 - x[i,j]))
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. When $x_{ij} = 0$, the precedence row must be redundant for every $t_i \leq \ell_i$ and $t_j \geq e_j$. Therefore the smallest nonnegative safe arc-specific constant from these bounds is

$$M_{ij} = \max\{0,\ \ell_i + s_i + \tau_{ij} - e_j\}.$$

Replace the single M by M_{ij} in each corresponding row. The active case $x_{ij} = 1$ is unchanged; only the inactive allowance is tightened. In code, derive and store one constant per arc from the same time-window data that defines the service-time bounds.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The strengthened time rows are

$$e_i \leq t_i \leq \ell_i \qquad \forall i, \tag{85}$$

$$t_j \geq t_i + s_i + \tau_{ij} - M_{ij}(1 - x_{ij}) \qquad \forall (i, j) \in A, \tag{86}$$

$$M_{ij} = \max\{0, \ell_i + s_i + \tau_{ij} - e_j\}. \tag{87}$$

```
M = {}
for i, j in feasible_arcs:
    M[i,j] = max(
        0.0,
        latest[i] + service[i] + travel[i,j] - earliest[j])
    m.addConstr(
        t[j] >= t[i] + service[i] + travel[i,j]
        - M[i,j] * (1 - x[i,j]))
```

B.2.2 STRUCTURAL DOMAIN REDUCTION

Trigger Conditions. Apply `structural_domain_reduction` when compatibility, reachability, time-window, capacity, or other exact structural conditions prove that some nominal decision indices can never appear in any feasible original solution. The removal predicate must be based on complete semantic information and must prove impossibility, not merely high cost or low empirical frequency. All expressions, adjacency lists, callbacks, and decoders that use the affected index set must be updated consistently. Do not classify a mere sparse-loop rewrite over the same feasible domain as domain reduction; the method removes decisions that are mathematically impossible.

Mathematical Principle. Suppose a dense model creates variables x_k for $k \in K$. If a predicate $P(k)$ proves

$$\forall x \in \mathcal{F}:\ x_k = 0 \qquad \text{whenever } P(k) = \text{false},$$

then define the reduced domain

$$K' = \{k \in K : P(k) = \text{true}\}.$$

Replacing x_K by $x_{K'}$ and interpreting removed decisions as fixed to zero preserves the complete feasible set after projection. The benefit comes from reducing variable count, nonzeros, and downstream combinatorial choices before optimization. The proof predicate may combine multiple exact conditions, but every removed tuple must be covered by the proof.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A heterogeneous drone–truck routing problem allows a drone to fly between candidate locations only when the destination is compatible with that drone, the flight duration fits its endurance, and the move can respect the relevant timing restrictions. A dense Cartesian product of drones and location pairs contains many sorties that are structurally impossible Kang and Lee (2021).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let $x_{dij} \in \{0,1\}$ indicate drone d flying from i to j. A dense formulation can create all triples and then force known-impossible sorties to zero:

$$x_{dij} \in \{0,1\} \qquad \forall d \in D,\ i,j \in V, \tag{88}$$

$$x_{dij} = 0 \qquad \forall (d,i,j) \text{ violating endurance or compatibility.} \tag{89}$$

```
drone_arc = m.addVars(D, V, V, vtype=GRB.BINARY, name="drone_arc")
for d, i, j in itertools.product(D, V, V):
    if flight_time[d,i,j] > endurance[d]:
        m.addConstr(drone_arc[d,i,j] == 0)
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Define an eligibility predicate before variable construction. For example, retain (d,i,j) only if $i \neq j$, customer j is compatible with drone d, the flight fits endurance, and the timing test proves that the transition can reach j within its allowed window. Let

$$E = \{(d,i,j) : P_{\text{compat}} \wedge P_{\text{endurance}} \wedge P_{\text{time}}\}.$$

Remove all x_{dij} with $(d,i,j) \notin E$ rather than creating them and adding zero-fixing rows. Rewrite every incoming/outgoing incidence sum over E.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The reduced model defines only

$$x_{dij} \in \{0,1\}, \qquad (d,i,j) \in E,$$

with routing and conservation expressions built from the reduced adjacency sets

$$\delta_E^+(d,i) = \{j : (d,i,j) \in E\}, \quad \delta_E^-(d,j) = \{i : (d,i,j) \in E\}.$$

```
eligible = [
    (d,i,j)
    for d in D for i in V for j in V
    if i != j
    and compatible[d,j]
    and flight_time[d,i,j] <= endurance[d]
    and time_window_reachable(d, i, j)
]
drone_arc = m.addVars(eligible, vtype=GRB.BINARY, name="drone_arc")

outgoing = {(d,i): [j for dd,ii,j in eligible if dd == d and ii == i]
            for d in D for i in V}
```

B.2.3 Symmetry Breaking

Trigger Conditions. Apply `symmetry_breaking` when a nontrivial permutation of labels leaves every constraint, variable domain, objective coefficient, and relevant output meaning unchanged. The repeated labels must represent truly interchangeable resources, colors, bins, machines, or other entities. A valid symmetry-breaking rule must preserve at least one representative from every equivalence orbit that matters to the original problem. Do not infer symmetry from similar variable names alone: capacities, costs, availability, hidden linking constraints, or label-sensitive output semantics can destroy interchangeability. Multiple symmetry-breaking rules may be imposed together only when their joint canonicalization is proved feasible for every orbit.

Mathematical Principle. Let a group Γ act on the feasible set $\mathcal{F}$ such that for all $\gamma \in \Gamma$,

$$x \in \mathcal{F} \Longrightarrow \gamma(x) \in \mathcal{F}, \qquad c^\top \gamma(x) = c^\top x.$$

The orbit of x is

$$\mathcal{O}(x) = \{\gamma(x) : \gamma \in \Gamma\}.$$

Symmetry breaking adds constraints defining a canonical subset $\mathcal{C} \subseteq \mathcal{F}$ such that

$$\mathcal{O}(x) \cap \mathcal{C} \neq \varnothing \qquad \forall x \in \mathcal{F}.$$

The labeled feasible set becomes smaller, but at least one objective-equivalent representative of every meaningful solution orbit remains. Typical canonicalization rules order activation variables, restrict first use of labels, or fix a known symmetric structure to a prescribed label order.

Before-and-After Case Study.

(1) Natural Language Problem. A graph-coloring model assigns one color to each vertex and minimizes the number of used colors. Color names have no intrinsic meaning: permuting all color labels in a feasible coloring produces the same partition of vertices and the same objective value. This creates a large family of equivalent labeled solutions Reference case associated with FrontierOR alfieri2020 (2022).

(2) Original Mathematical Formulation and Code. Let $x_{vc} = 1$ if vertex v uses color c, and let $y_c = 1$ if color c is used. A compact model is

$$\min \quad \sum_{c \in C} y_c, \tag{90}$$

$$\text{s.t.} \quad \sum_{c \in C} x_{vc} = 1 \qquad \forall v \in V, \tag{91}$$

$$x_{uc} + x_{vc} \leq y_c \qquad \forall \{u, v\} \in E,\ c \in C, \tag{92}$$

$$x_{vc} \leq y_c \qquad \forall v, c, \tag{93}$$

$$x_{vc}, y_c \in \{0, 1\}. \tag{94}$$

```
x = m.addVars(V, C, vtype=GRB.BINARY, name="assign")
y = m.addVars(C, vtype=GRB.BINARY, name="used")
m.addConstrs(gp.quicksum(x[v,c] for c in C) == 1 for v in V)
m.addConstrs(x[u,c] + x[v,c] <= y[c]
             for u,v in E for c in C)
m.addConstrs(x[v,c] <= y[c] for v in V for c in C)
m.setObjective(y.sum(), GRB.MINIMIZE)
```

(3) How the Mathematical Formulation and Code Are Modified. Choose a canonical order for color labels. First enforce monotone color activation,

$$y_c \geq y_{c+1},$$

so lower-numbered colors are used before higher-numbered colors. With vertices ordered $0, 1, \ldots$, a first-use rule can additionally forbid vertex v from using a color whose index exceeds v under the chosen zero-based convention. These constraints remove color-label permutations without changing the underlying vertex partition represented by at least one labeling. The code adds only canonicalization rows or fixed upper bounds; the objective and original coloring constraints remain unchanged.

(4) Modified Mathematical Formulation and Code. Augment the original model by

$$y_c \geq y_{c+1} \qquad c = 0, \ldots, |C| - 2, \tag{95}$$

$$x_{vc} = 0 \qquad c > v, \tag{96}$$

provided the first-use ordering is valid for the chosen vertex ordering. The complete model keeps the original coloring rows and objective.

```
m.addConstrs(y[c] >= y[c+1] for c in range(len(C)-1))
for v in V:
    for c in C:
        if c > v:
            x[v,c].UB = 0
```

B.2.4 Integrality Relaxation by Structure

Trigger Conditions. Apply `integrality_relaxation_by_structure` when a subset of integer variables belongs to a polyhedron that is provably integral under the actual constraints, right-hand sides, and bounds of the current model. Typical certificates arise from total unimodularity, bipartite assignment polytopes, network-flow incidence matrices with integral supplies, or a valid partial-integrality theorem. The proof must cover every side constraint touching the proposed variables; visual similarity to a network or one observed integral LP solution is not enough. Do not relax integrality if fractional right-hand sides, coupling rows, nonlinear objectives, or additional bounds invalidate the theorem.

Mathematical Principle. Let

$$P = \{x : Ax = b,\ \ell \leq x \leq u\}$$

with A totally unimodular and b, ℓ, u integral. Then every extreme point of P is integral. For a linear objective, an LP optimum therefore exists at an integral extreme point. Integer declarations on x are unnecessary:

$$x \in \mathbb{Z}^n \quad \Longrightarrow \quad x \in \mathbb{R}^n$$

without changing the optimal value, provided all constraints defining the relevant polyhedron remain within the integrality certificate. Removing unnecessary integer domains can eliminate branch-and-bound decisions while preserving the ability to recover an integral optimum.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A vehicle-scheduling substructure routes an integral number of vehicles through a time-space network. Nodes represent trip or depot events, arcs represent feasible vehicle transitions, and each node has an integral supply or demand. When no additional side constraint destroys the network structure, the node–arc incidence matrix is totally unimodular, so integer arc-flow declarations are redundant Ribeiro and Soumis (1994).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. For directed network $G = (N, A)$, let x_a be the number of vehicles on arc a and let $b_v \in \mathbb{Z}$ be the node supply:

$$\min \quad \sum_{a \in A} c_a x_a, \tag{97}$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a = b_v \qquad \forall v \in N, \tag{98}$$

$$0 \le x_a \le u_a, \quad x_a \in \mathbb{Z} \qquad \forall a \in A. \tag{99}$$

```
flow = m.addVars(A, vtype=GRB.INTEGER, lb=0, name="flow")
m.addConstrs(
    outflow(v, flow) - inflow(v, flow) == supply[v]
    for v in N)
m.setObjective(gp.quicksum(cost[a] * flow[a] for a in A), GRB.MINIMIZE)
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Verify that the complete constraint matrix involving the proposed flow variables is the directed node–arc incidence matrix plus integral simple bounds, and that all node supplies and capacities are integral. Then remove only the integrality declarations for the covered flow variables. No flow-balance equation, objective coefficient, or bound changes. Variables outside the certificate remain integer. In code, the patch changes the variable type from `GRB.INTEGER` to `GRB.CONTINUOUS`; it does not add rounding or post-hoc repair.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The relaxed model is

$$\min \quad \sum_{a \in A} c_a x_a, \tag{100}$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a = b_v \qquad \forall v, \tag{101}$$

$$0 \le x_a \le u_a, \qquad x_a \in \mathbb{R} \qquad \forall a. \tag{102}$$

By total unimodularity and integral data, an optimal integral extreme point exists.

```
flow = m.addVars(A, vtype=GRB.CONTINUOUS, lb=0, name="flow")
m.addConstrs(
    outflow(v, flow) - inflow(v, flow) == supply[v]
    for v in N)
m.setObjective(gp.quicksum(cost[a] * flow[a] for a in A), GRB.MINIMIZE)
```

B.2.5 EXACT DISCRETE PRODUCT LINEARIZATION

Trigger Conditions. Apply `exact_discrete_product_linearization` when a nonlinear product contains at least one binary or bounded integer factor, all required bounds are finite and valid, and an exact mixed-integer linear representation provides a concrete modeling benefit. The transformation must cover the actual factor domains and preserve the integer feasible set exactly. Do not apply a binary-product formula to two genuinely continuous variables, do not invent missing bounds, and do not expand a large discrete encoding without checking its model-size budget. Native quadratic handling may be preferable when the exact linearization would be larger with no structural advantage.

Mathematical Principle. For binary variables $x_i, x_j \in \{0,1\}$ and product

$$z_{ij} = x_i x_j,$$

the convex hull of the four feasible binary points is described by the Fortet inequalities

$$z_{ij} \le x_i, \tag{103}$$
$$z_{ij} \le x_j, \tag{104}$$
$$z_{ij} \ge x_i + x_j - 1, \tag{105}$$
$$z_{ij} \ge 0. \tag{106}$$

When x_i and x_j are binary these constraints force $z_{ij} = x_i x_j$. A quadratic term $q_{ij} x_i x_j$ can therefore be replaced exactly by the linear term $q_{ij} z_{ij}$. Similar exact encodings exist for binary–bounded-continuous or bounded-integer products, but the linking inequalities and number of auxiliary variables must match the actual domain.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A quadratic knapsack problem selects items with profits and weights under a capacity limit. In addition to individual profits, selecting a pair of items may create an interaction profit or penalty. These pair interactions produce products of binary selection variables in the objective Caprara et al. (1999).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let $x_i \in \{0,1\}$ select item i. A quadratic formulation is

$$\max \quad \sum_i p_i x_i + \sum_{(i,j)\in E_Q} q_{ij} x_i x_j, \tag{107}$$
$$\text{s.t.} \quad \sum_i w_i x_i \le C, \tag{108}$$
$$x_i \in \{0,1\}. \tag{109}$$

```
x = m.addVars(N, vtype=GRB.BINARY, name="x")
m.addConstr(gp.quicksum(weight[i]*x[i] for i in N) <= capacity)
obj = gp.quicksum(profit[i]*x[i] for i in N)
obj += gp.quicksum(q[i,j]*x[i]*x[j] for i,j in interacting_pairs)
m.setObjective(obj, GRB.MAXIMIZE)
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. For every nonzero pair interaction (i,j), introduce one auxiliary variable z_{ij} representing $x_i x_j$. Remove the quadratic product from the objective and replace it with z_{ij}. Add the three Fortet linking inequalities for each pair. No auxiliary is needed for a zero interaction coefficient. The code changes from a quadratic objective expression to a linear objective plus sparse pair-linking rows.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The exact MILP is

$$\max \quad \sum_i p_i x_i + \sum_{(i,j)\in E_Q} q_{ij} z_{ij}, \tag{110}$$
$$\text{s.t.} \quad \sum_i w_i x_i \le C, \tag{111}$$
$$z_{ij} \le x_i \qquad \forall (i,j) \in E_Q, \tag{112}$$
$$z_{ij} \le x_j \qquad \forall (i,j) \in E_Q, \tag{113}$$
$$z_{ij} \ge x_i + x_j - 1 \qquad \forall (i,j) \in E_Q, \tag{114}$$
$$x_i \in \{0,1\}, \quad z_{ij} \in \{0,1\}. \tag{115}$$

```
x = m.addVars(N, vtype=GRB.BINARY, name="x")
z = m.addVars(interacting_pairs, vtype=GRB.BINARY, name="pair")
m.addConstr(gp.quicksum(weight[i]*x[i] for i in N) <= capacity)
for i, j in interacting_pairs:
    m.addConstr(z[i,j] <= x[i])
    m.addConstr(z[i,j] <= x[j])
    m.addConstr(z[i,j] >= x[i] + x[j] - 1)
m.setObjective(
    gp.quicksum(profit[i]*x[i] for i in N) +
    gp.quicksum(q[i,j]*z[i,j] for i,j in interacting_pairs),
    GRB.MAXIMIZE)
```

B.2.6 PERSPECTIVE REFORMULATION

Trigger Conditions. Apply `perspective_reformulation` when a binary variable activates a convex cost or convex feasible set, the inactive state forces the associated continuous variable to a known value (typically zero after a valid shift), and the closed perspective of the convex term has an exact representation supported by the target solver. The active function must be convex in the epigraph direction being minimized. Do not apply the method to an arbitrary binary–continuous product, to a nonconvex active term, or when the continuous variable may remain nonzero while the selector is inactive. If the pipeline requires a strict MILP, a nonlinear perspective should be used only when an accepted exact linear representation is available.

Mathematical Principle. For an on/off epigraph

$$z \in \{0,1\}, \qquad Lz \le x \le Uz,$$

with active convex function $f(x)$ and epigraph variable t, the naive constraint

$$t \ge f(x)$$

ignores the selector in the continuous relaxation. The closed perspective replaces it by

$$t \ge z f(x/z)$$

for $z > 0$, together with the proper closure at $z = 0$. For $f(x) = x^2$, the perspective epigraph is

$$x^2 \le zt, \qquad t \ge 0,\ z \ge 0.$$

At binary z, this preserves the original on/off set exactly: $z = 0$ forces $x = 0$, while $z = 1$ recovers $t \ge x^2$. For fractional z, the relaxation is substantially stronger than the naive epigraph.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A cardinality-constrained investment model decides whether an asset or activity is selected and how much continuous amount is allocated to it. Allocation is zero when the selector is off, and an active allocation incurs a convex quadratic epigraph cost. The binary selector and quadratic epigraph form an on/off convex structure suitable for a perspective reformulation Reference case associated with FrontierOR kobayashi2021 (2021).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. For each i, let $z_i \in \{0,1\}$ be the selection variable, $0 \le x_i \le U_i z_i$ the amount, and t_i the quadratic epigraph. A baseline model contains

$$0 \le x_i \le U_i z_i, \tag{116}$$

$$t_i \ge x_i^2, \tag{117}$$

$$z_i \in \{0,1\}, \quad t_i \ge 0. \tag{118}$$

The objective can include $\sum_i q_i t_i$ together with the remaining portfolio terms.

```
z = m.addVars(N, vtype=GRB.BINARY, name="select")
x = m.addVars(N, lb=0.0, name="amount")
t = m.addVars(N, lb=0.0, name="epi")
m.addConstrs(x[i] <= U[i] * z[i] for i in N)
for i in N:
    m.addQConstr(t[i] >= x[i] * x[i])
```

(3) How the Mathematical Formulation and Code Are Modified. Keep the selector z_i, amount x_i, activation bound, and epigraph variable t_i. Replace only the naive active epigraph $t_i \geq x_i^2$ by the closed perspective constraint

$$x_i^2 \leq z_i t_i.$$

At integer z_i the feasible set is unchanged; at fractional z_i the new inequality links the epigraph directly to the activation level and removes weak fractional points. The code patch replaces one quadratic constraint per item while retaining the remaining model unchanged.

(4) Modified Mathematical Formulation and Code. The local on/off block becomes

$$0 \leq x_i \leq U_i z_i, \tag{119}$$

$$x_i^2 \leq z_i t_i, \tag{120}$$

$$z_i \in \{0,1\}, \quad t_i \geq 0. \tag{121}$$

```
z = m.addVars(N, vtype=GRB.BINARY, name="select")
x = m.addVars(N, lb=0.0, name="amount")
t = m.addVars(N, lb=0.0, name="epi")
m.addConstrs(x[i] <= U[i] * z[i] for i in N)
for i in N:
    m.addQConstr(x[i] * x[i] <= z[i] * t[i], name=f"persp_{i}")
```

B.2.7 Disjunctive Convex-Hull Reformulation

Trigger Conditions. Apply `disjunctive_convex_hull_reformulation` when the model contains a finite logical union of polyhedral branches, the number of branches is manageable, each branch has a complete explicit description, and valid finite branch bounds or an appropriate common-recession analysis are available. The method is most useful when a Big-M representation of the same disjunction has a weak relaxation. Do not use it when the branch family is too large, when branch constraints are incomplete, or when unbounded branches make the proposed scaled formulation invalid.

Mathematical Principle. Suppose

$$x \in \bigcup_{k \in K} P_k, \qquad P_k = \{x : A_k x \leq b_k,\ L_k \leq x \leq U_k\}.$$

Introduce selectors $z_k \in \{0,1\}$ with $\sum_k z_k = 1$ and disaggregated copies x^k. The hull formulation uses

$$x = \sum_{k \in K} x^k, \tag{122}$$

$$A_k x^k \leq b_k z_k \qquad \forall k, \tag{123}$$

$$L_k z_k \leq x^k \leq U_k z_k \qquad \forall k, \tag{124}$$

$$\sum_k z_k = 1. \tag{125}$$

For binary selectors, projection onto x recovers the original union. Under the bounded-branch assumptions, relaxing z gives the convex hull of the local disjunction, which is generally stronger than a generic Big-M encoding.

Before-and-After Case Study.

(1) Natural Language Problem. A discrete truss-design model chooses one structural design mode or cross-section alternative for a member. Each mode imposes its own bounded linear restrictions on local design and force variables. Exactly one mode must be active, so the local feasible region is a finite union of polyhedra Bollapragada et al. (2001).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let z_k select branch k and let $A_k x \le b_k$ be the branch constraints. A Big-M model is

$$\sum_{k \in K} z_k = 1, \tag{126}$$

$$A_k x \le b_k + M_k(1 - z_k) \qquad \forall k, \tag{127}$$

$$z_k \in \{0, 1\}. \tag{128}$$

```
z = m.addVars(K, vtype=GRB.BINARY, name="mode")
m.addConstr(z.sum() == 1)
for k in K:
    for row in branch_rows[k]:
        m.addConstr(row.A @ x <= row.b + M[row.id] * (1 - z[k]))
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Remove the branch-specific Big-M rows. Create one local copy x^k per branch and aggregate the original variable through $x = \sum_k x^k$. Scale each branch's row and finite bounds by z_k. The selector variables remain, but they now activate the branch copy directly instead of relaxing a common x through a large constant. Code changes from adding Big-M slacks to building disaggregated variables and scaled branch systems.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The hull formulation is

$$\sum_{k} z_k = 1, \tag{129}$$

$$x = \sum_{k} x^k, \tag{130}$$

$$A_k x^k \le b_k z_k \qquad \forall k, \tag{131}$$

$$L_k z_k \le x^k \le U_k z_k \qquad \forall k, \tag{132}$$

$$z_k \in \{0, 1\}. \tag{133}$$

```
z = m.addVars(K, vtype=GRB.BINARY, name="mode")
xk = {k: m.addMVar(n, lb=-GRB.INFINITY, name=f"x_{k}") for k in K}
m.addConstr(z.sum() == 1)
for j in range(n):
    m.addConstr(x[j] == gp.quicksum(xk[k][j] for k in K))
for k in K:
    m.addConstr(branch_A[k] @ xk[k] <= branch_b[k] * z[k])
    add_scaled_branch_bounds(m, xk[k], z[k], L[k], U[k])
```

B.2.8 CONSTRAINT GENERATION

Trigger Conditions. Apply `constraint_generation` when the mathematical model contains a very large or exponential family of *defining feasibility constraints*, explicit construction of the entire family is impractical, and violated members can be separated reliably from candidate solutions. The family must remain part of the original mathematical specification even if only a subset is materialized initially. For exact optimization, every accepted integer incumbent must be checked by a complete separator so that termination implies feasibility with respect to the full row family. A fractional separator may be added for performance, but it cannot replace the mandatory incumbent-feasibility check. Do not classify optional strengthening inequalities as constraint generation when the omitted rows are not required for original feasibility.

Mathematical Principle. Consider

$$\min \quad c^\top x, \tag{134}$$

$$\text{s.t.} \quad x \in X_0, \tag{135}$$

$$g_k(x) \le 0 \qquad \forall k \in \mathcal{K}, \tag{136}$$

where $\mathcal{K}$ is too large to enumerate. Constraint generation maintains a subset $\mathcal{K}_t \subset \mathcal{K}$ and repeatedly solves

$$\min\{c^\top x : x \in X_0,\ g_k(x) \le 0\ \forall k \in \mathcal{K}_t\}.$$

Given solution $\bar{x}$, a separator solves or implements

$$\max_{k \in \mathcal{K}} g_k(\bar{x}).$$

If the maximum is positive, one or more violated constraints are added. If a complete separator proves no violated row exists, $\bar{x}$ is feasible for the full original formulation. In a MIP callback, defining rows omitted from the initial model are naturally enforced with lazy constraints at incumbent solutions.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. An orienteering problem chooses a profitable subset of vertices and a route starting and ending at a depot within a travel budget. Degree constraints can make selected arcs look locally consistent while still allowing disconnected subtours that do not contain the depot. Connectivity therefore requires a large family of subset constraints, but only a small fraction is typically violated by any incumbent Kobeaga et al. (2024).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let $x_{ij} \in \{0,1\}$ indicate selected directed arc (i,j) and $y_i \in \{0,1\}$ indicate whether customer i is visited. A representative formulation contains

$$\max \quad \sum_{i \in V \setminus \{0\}} p_i y_i, \tag{137}$$

$$\text{s.t.} \quad \sum_j x_{ij} = y_i \qquad \forall i \ne 0, \tag{138}$$

$$\sum_j x_{ji} = y_i \qquad \forall i \ne 0, \tag{139}$$

$$\sum_{i,j} t_{ij} x_{ij} \le T, \tag{140}$$

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \ge y_k \qquad \forall \varnothing \ne S \subseteq V \setminus \{0\},\ k \in S. \tag{141}$$

The final row family excludes disconnected selected components. Full enumeration would resemble:

```
for S in every_nontrivial_subset(customers):
    for k in S:
        m.addConstr(
            gp.quicksum(x[i,j] for i in S for j in V if j not in S)
            >= y[k])
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Keep the complete connectivity family in the mathematical definition, but do not construct it initially. Build only the objective, degree equations, budget, and other compact rows. At every integer incumbent, recover the selected arc graph, find any connected component S not containing the depot, choose a selected $k \in S$, and add the corresponding connectivity row as a lazy constraint. The mathematical feasible region is unchanged; only the row-construction policy changes from eager enumeration to exact separation.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The mathematical model remains

$$\max \quad \sum_{i \ne 0} p_i y_i, \tag{142}$$

$$\text{s.t.} \quad \text{degree, budget, and linking constraints,} \tag{143}$$

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \ge y_k \qquad \forall \varnothing \ne S \subseteq V \setminus \{0\},\ k \in S, \tag{144}$$

but the exponential family is represented implicitly by its separator:

```
def callback(model, where):
    if where == GRB.Callback.MIPSOL:
        xbar = model.cbGetSolution(x)
        ybar = model.cbGetSolution(y)
        for S in disconnected_selected_components(xbar, ybar, depot=0):
            k = next(i for i in S if ybar[i] > 0.5)
            model.cbLazy(
                gp.quicksum(x[i,j] for i in S for j in V if j not in S)
                >= y[k])

m.Params.LazyConstraints = 1
m.optimize(callback)
```

B.2.9 VALID-INEQUALITY STRENGTHENING

Trigger Conditions. Apply `valid_inequality_strengthening` when the current formulation is already correct but its continuous relaxation admits fractional points that can be removed by a provably valid, nonredundant inequality derived from identifiable problem structure. The inequality must be satisfied by every original integer-feasible solution and should provide a stronger relaxation than rows already present. Typical sources include knapsack covers, conflict cliques, capacity arguments, lot-sizing structure, and other known polyhedral relations. Do not use this Skill to add a missing defining constraint, to restate an already implied row, or merely to enable generic solver cuts.

Mathematical Principle. Let

$$X = \{x \in \mathbb{Z}^n : Ax \le b\}$$

be the original integer feasible set and let

$$P = \{x \in \mathbb{R}^n : Ax \le b\}$$

be its relaxation. A valid inequality

$$\alpha^\top x \le \beta$$

must satisfy

$$\alpha^\top x \le \beta \qquad \forall x \in X.$$

Adding it changes the relaxation to

$$P' = P \cap \{x : \alpha^\top x \le \beta\}$$

with

$$\mathrm{conv}(X) \subseteq P' \subseteq P.$$

Hence integer feasibility and the original objective are preserved, while fractional feasible points may be removed. The gain depends on the strength and nonredundancy of the added family.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A knapsack problem with a conflict graph selects profitable items under a capacity limit. In addition to the knapsack capacity, certain item pairs are incompatible. Pairwise conflict rows are correct, but when several items form a clique in the conflict graph, the LP relaxation can still distribute fractional value across many mutually incompatible items Bettinelli et al. (2017).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let $x_i \in \{0,1\}$ indicate whether item i is selected and let E be the conflict-edge set:

$$\max \quad \sum_{i \in V} p_i x_i, \tag{145}$$

$$\text{s.t.} \quad \sum_{i \in V} w_i x_i \le C, \tag{146}$$

$$x_i + x_j \le 1 \qquad \forall \{i,j\} \in E, \tag{147}$$

$$x_i \in \{0,1\}. \tag{148}$$

```
x = m.addVars(V, vtype=GRB.BINARY, name="take")
m.addConstr(gp.quicksum(weight[i]*x[i] for i in V) <= capacity)
for i, j in conflict_edges:
    m.addConstr(x[i] + x[j] <= 1)
m.setObjective(gp.quicksum(profit[i]*x[i] for i in V), GRB.MAXIMIZE)
```

(3) How the Mathematical Formulation and Code Are Modified. Find a certified clique C in the conflict graph. Because every pair of items in C conflicts, any integer-feasible solution can select at most one member of C. Add the clique inequality

$$\sum_{i \in C} x_i \leq 1.$$

The pairwise rows and original knapsack model remain intact. The new row is not required to define integer feasibility—the pairwise constraints already do that—but it cuts fractional points such as $x_i = 1/2$ on several clique vertices. Code therefore adds a selected family of clique rows after verifying the graph witness and avoiding dominated duplicates.

(4) Modified Mathematical Formulation and Code. The strengthened model is

$$\max \quad \sum_i p_i x_i, \tag{149}$$

$$\text{s.t.} \quad \sum_i w_i x_i \leq C, \tag{150}$$

$$x_i + x_j \leq 1 \qquad \forall \{i,j\} \in E, \tag{151}$$

$$\sum_{i \in C} x_i \leq 1 \qquad \forall C \in \mathcal{C}, \tag{152}$$

$$x_i \in \{0,1\}, \tag{153}$$

where $\mathcal{C}$ is the selected certified clique family.

```
for C in certified_maximal_cliques(conflict_graph):
    if len(C) >= 3 and not already_dominated(C):
        m.addConstr(
            gp.quicksum(x[i] for i in C) <= 1,
            name=f"clique_{clique_id(C)}")
```

B.2.10 Problem-Specific Branch-and-Cut

Trigger Conditions. Apply `problem_specific_branch_and_cut` when a correct integer model has a weak relaxation and the problem admits one or more reusable, globally integer-valid inequality families with practical separation procedures. The added cuts must exploit identifiable problem structure rather than generic solver parameters. Every user cut added globally must be valid for all original integer-feasible solutions, independent of the current branch-and-bound node. If defining feasibility constraints are omitted from the base model, they must be handled separately by a complete lazy-constraint checker; optional strengthening cuts cannot replace mandatory feasibility enforcement. Do not classify generic branch-and-bound, generic solver cuts, or a callback with no structural cut family as this Skill.

Mathematical Principle. Let the base MIP be

$$\min\{c^\top x : x \in X \cap \mathbb{Z}^n\},$$

with LP relaxation P. Suppose a problem-specific family

$$\alpha^{k\top} x \leq \beta^k, \qquad k \in \mathcal{K},$$

is valid for every integer point in X but too large to add completely. At selected branch-and-bound nodes, solve a separation problem on the current LP solution $\bar{x}$ to find

$$k \in \mathcal{K}: \quad \alpha^{k\top} \bar{x} > \beta^k + \varepsilon.$$

Add the violated row as a global user cut. The branch-and-bound tree remains the solver's native tree; the custom component strengthens node relaxations. For example, in an undirected homogeneous CVRP with positive demands and vehicle capacity Q, a rounded-capacity inequality is

$$x(\delta(S)) \geq 2 \left\lceil \frac{d(S)}{Q} \right\rceil,$$

provided all assumptions and the undirected edge convention hold. A different routing model requires its own valid family and constants.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A pickup-and-delivery problem with time windows and LIFO loading chooses vehicle routes that must satisfy paired pickup–delivery service, timing, capacity, and loading-order semantics. A compact or column-based core model can be correct yet have a weak relaxation. Problem-specific inequalities derived from the pickup–delivery, time-window, capacity, and LIFO structure can be separated during branch-and-bound to tighten fractional node solutions Cherkesly et al. (2015).

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. Let X_{core} denote the complete integer feasible set defined by the core routing, pairing, timing, capacity, and LIFO constraints, and let x denote its routing variables. The baseline model is

$$\begin{aligned} \min \quad & c^\top x, && (154)\\ \text{s.t.} \quad & x \in X_{\text{core}}, && (155)\\ & x \in \{0,1\}^{|A|}. && (156) \end{aligned}$$

It relies on generic branch-and-bound after construction:

```
m, x = build_pdptwl_core_model(data)
m.optimize()    # no problem-specific cut separator
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Identify a certified family $\mathcal{K}$ of globally valid PDPTWL inequalities. For each $k \in \mathcal{K}$, write

$$\alpha^{k\top} x \leq \beta^k,$$

and implement a separator that searches for violations in an optimal node relaxation. The full mathematical strengthened model contains all such inequalities, but the implementation generates only violated members. Add optional strengthening cuts with `cbCut` at `MIPNODE`; if the base model also omits any defining feasibility rows, enforce those independently with `cbLazy` at `MIPSOL`. The solver's native branching remains unchanged.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The strengthened model is

$$\begin{aligned} \min \quad & c^\top x, && && (157)\\ \text{s.t.} \quad & x \in X_{\text{core}}, && && (158)\\ & \alpha^{k\top} x \leq \beta^k && \forall k \in \mathcal{K}, && (159)\\ & x \in \{0,1\}^{|A|}. && && (160) \end{aligned}$$

The cut family is generated dynamically:

```
def callback(model, where):
    if where == GRB.Callback.MIPNODE:
        if model.cbGet(GRB.Callback.MIPNODE_STATUS) != GRB.OPTIMAL:
            return
        xrel = model.cbGetNodeRel(x)
        for cut in separate_valid_pdptwl_cuts(xrel, data):
            model.cbCut(cut.expr <= cut.rhs)
```

```
    elif where == GRB.Callback.MIPSOL:
        xinc = model.cbGetSolution(x)
        # Required only if defining rows are intentionally delayed.
        enforce_complete_original_feasibility(model, xinc, data)

m.Params.PreCrush = 1
m.optimize(callback)
```

B.3 Solver-Implementation Skills

The following twelve Skills preserve the finalized mathematical formulation and modify only its solver implementation. For every Skill in this category,

$$\mathcal{M}' = \mathcal{M}, \qquad \mathcal{C} \to \mathcal{C}'.$$

These Skills optimize model construction, solver configuration, callback execution, model lifecycle, warm-start state, parallel execution, or search guidance without changing the intended feasible set, objective semantics, or correctness contract.

B.3.1 Sparse Index Precomputation

Skill ID: `sparse_index_precomputation`.

Trigger Conditions. Apply `sparse_index_precomputation` when model construction repeatedly scans the same support relation to recover local subsets, while the support identity and the indexed part of the filtering predicate remain stable over the reuse interval. The transformation is appropriate when a reusable adjacency, incidence, membership, or key-to-record map can replace repeated full-table filtering without changing the variable or constraint domain. Preserve record multiplicity, parallel records, self-loops, and any residual predicate that still depends on changing data. Do not use this Skill to remove mathematically feasible variables or constraints; such a change is a Model Skill rather than a code-level indexing optimization. If the support changes, an explicit invalidation or rebuild rule is required.

Solver-Implementation Principle. Let K be a fixed support table and let a family of construction operations repeatedly require

$$K(q) = \{k \in K : P_{\text{static}}(q,k) \wedge P_{\text{dynamic}}(q,k)\}.$$

The baseline repeatedly evaluates P_{static} over all K. Precomputation builds

$$I(q) = \{k \in K : P_{\text{static}}(q,k)\}$$

once and then evaluates only

$$K(q) = \{k \in I(q) : P_{\text{dynamic}}(q,k)\}.$$

The mathematical model is unchanged:

$$\boxed{M' = M}, \qquad \boxed{C \to C'}.$$

Only the Python construction path changes from repeated full scans to indexed access. Exactness requires the indexed support to contain exactly the same record IDs, with the same multiplicity and coefficient semantics, as the baseline.

Before-and-After Case Study.

(1) Natural Language Problem. A multi-commodity flow problem routes several commodities through a directed network. Each commodity has a flow-balance equation at every node, and each physical arc has a shared capacity. The mathematical model is sparse because each balance row touches only arcs incident to its node, but a literal Python implementation may repeatedly scan the complete arc table to discover those incident arcs.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\min \quad \sum_{k \in K} \sum_{a \in A} c_a x_{ka}, \tag{161}$$

$$\text{s.t.} \quad \sum_{a \in \delta^+(i)} x_{ka} - \sum_{a \in \delta^-(i)} x_{ka} = b_{ki} \qquad \forall k \in K,\ i \in N, \tag{162}$$

$$\sum_{k \in K} x_{ka} \leq u_a \qquad \forall a \in A, \tag{163}$$

$$x_{ka} \geq 0 \qquad \forall k, a. \tag{164}$$

The baseline implementation is:

```
# Baseline: every balance row scans the complete arc table.
for k in commodities:
    for i in nodes:
        outgoing = gp.quicksum(
            x[k, a] for a, u, v, cost in arcs if u == i)
        incoming = gp.quicksum(
            x[k, a] for a, u, v, cost in arcs if v == i)
        m.addConstr(outgoing - incoming == balance[k, i])

for a, u, v, cost in arcs:
    m.addConstr(gp.quicksum(x[k, a] for k in commodities) <= capacity[a])
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Keep every variable, coefficient, balance equation, capacity row, and objective term unchanged. Before row construction, scan the arc table once and build outgoing and incoming arc-ID lists. Balance rows then iterate only over their indexed incidence lists. Parallel arcs remain separate because the index stores arc IDs rather than endpoint pairs; self-loops appear in both lists and cancel exactly as in the baseline.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The modified mathematical formulation is identical to the original:

$$\boxed{\min \sum_{k,a} c_a x_{ka} \quad \text{s.t. the same flow-balance, capacity, and domain constraints.}}$$

Thus $M' = M$; only the support-access implementation changes. The modified implementation is:

```
out_arcs = {i: [] for i in nodes}
in_arcs = {i: [] for i in nodes}
for a, u, v, cost in arcs:
    out_arcs[u].append(a)
    in_arcs[v].append(a)

for k in commodities:
    for i in nodes:
        outgoing = gp.quicksum(x[k, a] for a in out_arcs[i])
        incoming = gp.quicksum(x[k, a] for a in in_arcs[i])
        m.addConstr(outgoing - incoming == balance[k, i])

for a, u, v, cost in arcs:
    m.addConstr(gp.quicksum(x[k, a] for k in commodities) <= capacity[a])
```

B.3.2 SPARSE MATRIX CONSTRAINT ASSEMBLY

Skill ID: `sparse_matrix_constraint_assembly`.

Trigger Conditions. Apply `sparse_matrix_constraint_assembly` when a sizable linear constraint block is already mathematically fixed, its coefficients/senses/RHS values can be assembled numerically outside solver objects, and row-by-row Python expression construction or API insertion is a meaningful cost. An explicit semantic row order and column order must be available so that the sparse matrix, variable list, senses, and RHS vector are aligned. Mixed linear row senses are allowed. Do not absorb indicator, quadratic, general, nonlinear, or callback-generated constraints into a linear matrix block by dropping their semantics. Parallel numeric preparation is a separate execution mode and is admissible only when coefficient generation is independent and model insertion remains in the owning process.

Solver-Implementation Principle. Suppose the linear block is

$$Ax \bowtie b,$$

with semantic row IDs R, column IDs J, sparse coefficient set

$$\mathcal{N} = \{(r, j, A_{rj}) : A_{rj} \neq 0\}.$$

The baseline constructs one `LinExpr` and submits one row at a time. The patch instead assembles a sparse numeric representation of A, fixes an ordered object vector x_J, and submits the same block through a matrix API. Algebraically,

$$(A, \bowtie, b, x_J)_{\text{before}} = (A, \bowtie, b, x_J)_{\text{after}},$$

so

$$\boxed{M' = M}, \qquad C \to C'.$$

Duplicate triplets must be aggregated to the same coefficient and no new dropping tolerance may be introduced.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A continuous allocation model contains a sparse linear block with capacity, demand, and balance rows. The variable objects already exist, but the baseline constructs each row independently in Python even though all coefficients, senses, and right-hand sides are available as arrays.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\begin{aligned}
\min \quad & x_1 + x_2 + x_3, && && (165)\\
\text{s.t.} \quad & 2x_1 + x_2 \leq 9, && && (166)\\
& x_1 + x_2 \geq 3, && && (167)\\
& x_1 - x_2 + x_3 = 1, && && (168)\\
& 0 \leq x_j \leq 10 && j = 1, 2, 3. && (169)
\end{aligned}$$

The baseline implementation is:

```
rows = {}
for r, row_id in enumerate(ROW_IDS):
    terms = [(c, a) for rr, c, a in TERMS if rr == r]
    expr = gp.LinExpr(
        [a for c, a in terms],
        [variables[COL_IDS[c]] for c, a in terms])
    if SENSE[r] == "<":
        relation = expr <= float(RHS[r])
    elif SENSE[r] == ">":
        relation = expr >= float(RHS[r])
    else:
        relation = expr == float(RHS[r])
    rows[row_id] = model.addConstr(relation, name=row_id)
```

(3) How the Mathematical Formulation and Code Are Modified. Freeze semantic column order `COL_IDS` and row order `ROW_IDS`. Convert the same coefficient triplets to a supported sparse matrix, sum duplicate positions, align the variable-object list with matrix columns, and submit the same sense/RHS arrays. No row is added, removed, weakened, or strengthened; only the construction mechanism changes.

(4) Modified Mathematical Formulation and Code. The mathematical model remains exactly

$$\boxed{Ax \bowtie b, \quad 0 \le x \le 10,}$$

with the same objective. Hence $M' = M$. The modified implementation is:

```
rr, cc, aa = zip(*TERMS)
matrix = coo_matrix(
    (np.asarray(aa, dtype=np.float64), (rr, cc)),
    shape=(len(ROW_IDS), len(COL_IDS))).tocsr()
matrix.sum_duplicates()
matrix.sort_indices()

ordered_vars = [variables[key] for key in COL_IDS]
model.addMConstr(matrix, ordered_vars, SENSE, RHS)

# If downstream code needs semantic handles/names, remap only the rows
# inserted by this block rather than assuming every model row belongs to
    it.
```

B.3.3 Bulk Attribute and Update Management

Skill ID: `bulk_attribute_and_update_management`.

Trigger Conditions. Apply `bulk_attribute_and_update_management` when an identifiable family of Gurobi objects is accessed or modified through many scalar attribute operations, or when redundant synchronization calls occur in a hot construction/reoptimization path. The object-to-semantic-ID mapping, conditional update subset, and attribute availability must be known. Solution-dependent attributes require an appropriate solved state; LP duals and bases require their own availability conditions. Do not change which objects are updated, their values, the output schema, or read-after-write semantics. Callback data access is outside this Skill and belongs to `callback_data_path_optimization`.

Solver-Implementation Principle. For object family $O = (o_1, \dots, o_n)$ and attribute values $v = (v_1, \dots, v_n)$, replace

$$o_i.a \leftarrow v_i \quad (i = 1, \dots, n)$$

or repeated scalar reads

$$(v_1, \dots, v_n) = (o_1.a, \dots, o_n.a)$$

by aligned bulk operations

$$\texttt{setAttr}(a, O, v), \qquad v = \texttt{getAttr}(a, O).$$

The semantic mapping $o_i \leftrightarrow \mathrm{id}_i$ is fixed explicitly. Redundant `update()` calls may be removed only when the next operation already supplies the required synchronization. Therefore

$$\boxed{M' = M}$$

and only Python/API interaction changes.

Before-and-After Case Study.

(1) Natural Language Problem. A continuous allocation LP is solved repeatedly while one demand RHS changes between episodes. After each solve, the program returns business-keyed primal variable values and row duals. The baseline performs each RHS write and each solution read through scalar object attributes.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\begin{aligned} \min \quad & x_E + 2x_W, && (170)\\ \text{s.t.} \quad & x_E + x_W \geq d, && (171)\\ & x_E \leq 3, && (172)\\ & x_E, x_W \geq 0. && (173) \end{aligned}$$

The baseline implementation is:

```
for delta in EPISODES:
    for row_id, value in delta.items():
        rows[row_id].RHS = value

    model.optimize()
    require_lp_optimum(model)

    results.append({
        "objective": float(model.ObjVal),
        "x": {key: float(x[key].X) for key in VAR_IDS},
        "pi": {key: float(rows[key].Pi) for key in ROW_IDS},
    })
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Keep the same changing RHS, solve sequence, status guard, primal outputs, and dual outputs. Build stable ordered object lists once, batch only the rows that actually change, and map bulk-read arrays back to the same semantic keys. The patch does not alter variable bounds, objective coefficients, tolerances, or the mathematical episode sequence.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. For every episode e,

$$M'_e = M_e,$$

including the same demand value d_e. The optimization problem and output contract are unchanged. The modified implementation is:

```
ordered_vars = [x[key] for key in VAR_IDS]
ordered_rows = [rows[key] for key in ROW_IDS]

for delta in EPISODES:
    changed_ids = tuple(delta)
    model.setAttr(
        "RHS",
        [rows[key] for key in changed_ids],
        [delta[key] for key in changed_ids])

    model.optimize()
    require_lp_optimum(model)

    x_values = model.getAttr("X", ordered_vars)
    pi_values = model.getAttr("Pi", ordered_rows)
    results.append({
        "objective": float(model.ObjVal),
        "x": dict(zip(VAR_IDS, map(float, x_values))),
        "pi": dict(zip(ROW_IDS, map(float, pi_values))),
    })
```

B.3.4 PRESOLVE POLICY SELECTION

Skill ID: `presolve_policy_selection`.

Trigger Conditions. Apply `presolve_policy_selection` when measured or otherwise attributable preprocessing behavior motivates a controlled comparison of supported presolve policies. Relevant evidence includes substantial presolve cost, aggregation-related fill, weak useful reduction, or numerical behavior plausibly connected to presolve. Recover the effective baseline configuration first; automatic presolve is a legitimate baseline and may remain the final selection. Model size or the number of removed rows alone is not evidence that presolve is harmful. The Skill changes only a declared presolve policy and must not simultaneously rewrite the formulation or unrelated search settings.

Solver-Implementation Principle. Let M denote the fixed mathematical model and let

$$\mathcal{S}(M;\rho)$$

denote the solver execution under presolve policy ρ. This Skill searches a small supported candidate set

$$\rho \in \mathcal{R}$$

under a fixed correctness/output contract and a predeclared performance metric. It selects a policy only after measurement:

$$\rho^* \in \arg\min_{\rho\in\mathcal{R}} T_{\text{declared}}(M;\rho)$$

subject to the required numerical and certificate conditions. The optimization model itself is unchanged:

$$\boxed{M' = M}.$$

There is no universal winning presolve level.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A small binary selection MILP contains several layers of equality-linked alias variables. These aliases are intentionally part of the model and must not be deleted by the Code Skill. They can nevertheless cause nontrivial preprocessing, so different supported presolve policies may be evaluated without changing the original formulation.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\begin{aligned}
\min \quad & 3x_0 + 5x_1 + 2x_2 + 4x_3, && && (174)\\
\text{s.t.} \quad & a_j^1 = x_j && j = 0,\dots,3, && (175)\\
& a_j^\ell = a_j^{\ell-1} && j = 0,\dots,3,\ \ell = 2,3, && (176)\\
& \sum_{j=0}^{3} a_j^3 \geq 2, && && (177)\\
& x_j \in \{0,1\}, \quad 0 \leq a_j^\ell \leq 1. && && (178)
\end{aligned}$$

The baseline implementation is:

```
model = build_model(effective_parameters)
try:
    # Preserve the supplied effective baseline, e.g. Presolve=-1 (
    automatic).
    model.optimize()
    result = solve_and_extract(model)
finally:
    model.dispose()
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Do not delete the alias variables or equality rows. Snapshot the actual effective presolve-related settings, choose a small installed-version-supported candidate set, and run fresh equivalent trials. The candidate changes only the selected presolve control. Acceptance is based on the declared end-to-end metric plus original-model feasibility/certificate checks, not on presolved row counts alone.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The modified mathematical formulation is exactly the same alias-linked MILP:

$$\boxed{M' = M}.$$

Only the execution parameter ρ changes. The modified implementation is:

```
model = build_model(effective_parameters)
try:
    model.Params.Presolve = selected_presolve
    model.optimize()
    result = solve_and_extract(model)
finally:
    model.dispose()

# selected_presolve is a controlled-trial result, not a universal default.
```

B.3.5 RELAXATION ALGORITHM SELECTION

Skill ID: `relaxation_algorithm_selection`.

Trigger Conditions. Apply `relaxation_algorithm_selection` when LP or root-relaxation work is relevant to the declared performance objective and alternative installed-version-supported relaxation algorithms can be compared without changing the mathematical problem. Record the model class, effective `Method` and any relevant `NodeMethod`/`Crossover` settings, solve lifecycle, repeated modifications, and downstream requirements. Row duals, reduced costs, a basic solution, and a transferable basis are distinct requirements. Do not silently relax a MIP, rewrite a decomposition method, or assume a particular simplex/barrier method is always superior.

Solver-Implementation Principle. For a fixed LP

$$\min\{c^\top x : Ax \bowtie b,\ \ell \le x \le u\},$$

Gurobi may solve the same relaxation through different supported algorithms $a \in \mathcal{A}$. This Skill selects

$$a^* \in \mathcal{A}$$

under fixed model content and fixed downstream output requirements. For repeated restricted-master solves, the complete episode sequence must also remain fixed. Thus

$$\boxed{M' = M}, \qquad \boxed{\text{solve algorithm } a \to a'}.$$

Iteration counts are algorithm-specific diagnostics; the decision criterion is the declared workflow-level metric and correctness contract.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A restricted master LP starts with two covering columns and receives additional columns in later episodes. Pricing or downstream logic needs valid row duals after every optimal LP solve. The same model and column sequence are replayed while only the LP solution algorithm is changed.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\min \quad \sum_{p \in R_e} c_p \lambda_p, \tag{179}$$

$$\text{s.t.} \quad \sum_{p \in R_e} a_{ip} \lambda_p \ge d_i \qquad i = 1, \dots, m, \tag{180}$$

$$\lambda_p \ge 0 \qquad p \in R_e, \tag{181}$$

where R_e is the same cumulative column set in episode e. The baseline implementation is:

```
snapshots = solve_sequence(
    effective_parameters=effective_parameters,
    method_override=None,
    require_basis=require_basis)
# Existing Method/Crossover policy is preserved.
```

(3) How the Mathematical Formulation and Code Are Modified. Keep the restricted-master rows, columns, coefficients, episode order, persistence policy, and downstream dual/reduced-cost checks fixed. Compare a small supported algorithm set against the effective baseline. If a candidate requires crossover to satisfy a basis requirement, declare that coupled choice explicitly and charge its cost. Do not interpret a different optimal dual vector as a semantic error when the dual is still valid for the required downstream operation.

(4) Modified Mathematical Formulation and Code. For every episode,

$$M'_e = M_e.$$

Only the selected relaxation algorithm and any explicitly coupled, contract-required phase setting differ. The modified implementation is:

```
snapshots = solve_sequence(
    effective_parameters=effective_parameters,
    method_override=selected_method,
    require_basis=require_basis)

# Keep only a measured policy that preserves primal feasibility,
# valid dual/reduced-cost semantics, and any required basis output.
```

B.3.6 Callback Data Path Optimization

Skill ID: `callback_data_path_optimization`.

Trigger Conditions. Apply `callback_data_path_optimization` when an existing callback spends meaningful time on repeated scalar callback queries, rebuilding stable key-to-object mappings, or recomputing genuinely invariant support structures. The callback event, solver-state guard, queried variable family, and semantic mapping must be explicit. Use event-specific callback APIs: incumbent values at an appropriate `MIPSOL` path and node-relaxation values only at an appropriate `MIPNODE` path with the required status guard. Do not skip a mandatory feasibility check, change the separator, or cache solution-dependent objects as if they were invariant. Scheduling optional separation less often is a different Skill, `optional_cut_separation_scheduling`.

Solver-Implementation Principle. Let a callback event require values

$$v = (v_1, \ldots, v_n)$$

for a fixed semantic variable family $X = (x_1, \ldots, x_n)$. The baseline performs n callback queries and reconstructs the map each time. The optimized data path prepares the ordered family once and uses one supported batched event query:

$$v = \texttt{cbGetSolution}(X)$$

or the corresponding event-specific node-relaxation operation. The detector

$$D(v)$$

and all generated lazy constraints/cuts remain unchanged. Hence

$$\boxed{M' = M}$$

and the callback's mathematical obligations are unchanged.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A symmetric traveling-salesman model includes degree constraints explicitly and enforces subtour elimination lazily. At each integer incumbent, the callback must inspect the incumbent edge set, detect disconnected tours, and add all required subtour-elimination constraints. The baseline fetches each edge value separately.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\min \quad \sum_{\{i,j\}\in E} c_{ij}x_{ij}, \tag{182}$$

$$\text{s.t.} \quad \sum_{j:\{i,j\}\in E} x_{ij} = 2 \qquad \forall i \in V, \tag{183}$$

$$\sum_{\{i,j\}\subseteq S} x_{ij} \le |S| - 1 \qquad \forall \varnothing \neq S \subsetneq V, \tag{184}$$

$$x_{ij} \in \{0,1\}. \tag{185}$$

The subtour family remains part of the mathematical model even when enforced lazily. The baseline implementation is:

```
def callback(model, where):
    if where != GRB.Callback.MIPSOL:
        return

    # Baseline: one callback query per edge.
    xbar = {e: model.cbGetSolution(x[e]) for e in EDGES}
    for S in disconnected_components(xbar):
        model.cbLazy(
            gp.quicksum(x[e] for e in internal_edges(S)) <= len(S) - 1)

m.Params.LazyConstraints = 1
m.optimize(callback)
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Freeze the stable edge-key order and corresponding Var list before optimization. At each mandatory `MIPSOL` event, issue one aligned bulk incumbent query, reconstruct the same edge-value map, and pass it to the unchanged connected-component detector and lazy-row generator. Event coverage, exception semantics, and mandatory feasibility checking remain unchanged.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The TSP formulation, including the complete subtour family, is unchanged:

$$\boxed{M' = M}.$$

Only the callback value-fetch path is modified. The modified implementation is:

```
EDGE_KEYS = tuple(EDGES)
EDGE_VARS = [x[e] for e in EDGE_KEYS]

def callback(model, where):
    if where != GRB.Callback.MIPSOL:
        return

    values = model.cbGetSolution(EDGE_VARS)
    xbar = dict(zip(EDGE_KEYS, values))
    for S in disconnected_components(xbar):
        model.cbLazy(
            gp.quicksum(x[e] for e in internal_edges(S)) <= len(S) - 1)

m.Params.LazyConstraints = 1
m.optimize(callback)
```

B.3.7 Optional Cut Separation Scheduling

Skill ID: `optional_cut_separation_scheduling`.

Trigger Conditions. Apply `optional_cut_separation_scheduling` only when an existing fractional cut separator is provably optional for correctness and its cumulative callback cost is relevant to the declared solve objective. Mandatory lazy feasibility checks and required stopping-certificate oracles must first be separated from optional strengthening. The cut formulas, validity domain, violation tolerance, and submission semantics are frozen; this Skill changes only when optional separation is executed. Do not use it when mandatory and optional logic cannot be distinguished, and do not infer that fewer calls or fewer cuts necessarily imply a faster solve.

Solver-Implementation Principle. Let $S(\bar{x})$ be an existing optional separator that returns globally valid strengthening inequalities. If it is invoked at every eligible event $e = 1, 2, \ldots$, the baseline cost is roughly

$$\sum_e T_S(e).$$

A scheduling policy $\sigma(e) \in \{0, 1\}$ changes only whether optional work runs:

$$S(\bar{x}_e) \text{ is executed iff } \sigma(e) = 1.$$

All mandatory feasibility checks execute regardless of σ. Because optional cuts do not define the original feasible set,

$$\boxed{M' = M}.$$

The tradeoff is computational: less separation overhead may produce weaker node relaxations and more downstream search.

Before-and-After Case Study.

(1) Natural Language Problem. A binary selection problem contains explicit pairwise conflict constraints, one additional defining conflict enforced lazily at integer incumbents, and a family of clique inequalities used only to strengthen fractional node relaxations. The clique separator is mathematically optional because the pairwise conflicts already define the same integer incompatibilities.

(2) Original Mathematical Formulation and Code.

$$\max \quad \sum_{i \in V} p_i x_i, \tag{186}$$

$$\text{s.t.} \quad x_i + x_j \le 1 \qquad \forall \{i, j\} \in E, \tag{187}$$

$$x_r + x_s \le 1 \qquad \text{(defining row enforced lazily if delayed)}, \tag{188}$$

$$x_i \in \{0, 1\}. \tag{189}$$

Optional clique inequalities

$$\sum_{i \in C} x_i \le 1$$

strengthen the relaxation but are not required to define the integer feasible set when C is already a conflict clique. The baseline implementation is:

```
def callback(model, where):
    if where == GRB.Callback.MIPSOL:
        enforce_mandatory_lazy_conflict(model)

    elif where == GRB.Callback.MIPNODE:
        if model.cbGet(GRB.Callback.MIPNODE_STATUS) != GRB.OPTIMAL:
            return
        # Baseline: run optional separation at every eligible node event.
        add_violated_optional_clique_cuts(model)
```

(3) How the Mathematical Formulation and Code Are Modified. Leave the mandatory `MIPSOL` path untouched. After the original `MIPNODE` eligibility/status guard, place a scheduling decision around optional clique separation only. A period of one reproduces the baseline. Preserve root behavior for the first controlled frequency experiment, reset the counter at the original solve boundary, and freeze the selected period before final evaluation.

(4) Modified Mathematical Formulation and Code. The original selection problem and the validity of every optional clique inequality are unchanged:

$$\boxed{M' = M}.$$

Only the runtime schedule for attempting optional cuts changes. The modified implementation is:

```
eligible_event = 0

def callback(model, where):
    global eligible_event

    if where == GRB.Callback.MIPSOL:
        # Mandatory path is never scheduled away.
        enforce_mandatory_lazy_conflict(model)

    elif where == GRB.Callback.MIPNODE:
        if model.cbGet(GRB.Callback.MIPNODE_STATUS) != GRB.OPTIMAL:
            return

        nodecnt = int(model.cbGet(GRB.Callback.MIPNODE_NODCNT))
        if nodecnt == 0:              # preserve root policy
            add_violated_optional_clique_cuts(model)
            return

        eligible_event += 1
        if eligible_event % period == 0:
            add_violated_optional_clique_cuts(model)
```

B.3.8 Persistent Model Reoptimization

Skill ID: `persistent_model_reoptimization`.

Trigger Conditions. Apply `persistent_model_reoptimization` when a workflow solves a sequence of related optimization episodes by repeatedly rebuilding a largely unchanged model, while each episode can be represented exactly as explicit supported deltas to a retained model. The initial state, episode order, and every RHS/coefficient/objective/bound/row/column change must be identifiable. Do not use persistence when fresh rebuilds intentionally provide required independence, when branch-local changes cannot be restored safely, or when a single solve is the whole workflow. Explicit basis transfer is not required; that is the separate `lp_basis_warm_start` Skill.

Solver-Implementation Principle. Let episode e solve model

$$M_e = M_0 \oplus \Delta_1 \oplus \cdots \oplus \Delta_e.$$

The rebuild baseline constructs M_e from scratch for every e. Persistent reoptimization instead keeps one owning model and applies only the new delta:

$$M_{e-1} \xrightarrow{\Delta_e} M_e \xrightarrow{\texttt{optimize}} .$$

The required invariant is episode-wise equivalence:

$$\boxed{M_e^{\text{persistent}} = M_e^{\text{fresh}} \quad \forall e}.$$

The higher-level algorithm and mathematical sequence do not change; only model lifecycle and state reuse change.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A restricted master LP begins with two initial columns and then receives two batches of additional columns. The baseline rebuilds the complete cumulative restricted master before every solve. Because every episode differs only by newly added columns, the same sequence can instead be represented by a persistent model with explicit column-addition deltas.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\min \quad \sum_{p \in R_e} c_p \lambda_p, \tag{190}$$

$$\text{s.t.} \quad \sum_{p \in R_e} a_{ip} \lambda_p \geq d_i \qquad \forall i, \tag{191}$$

$$\lambda_p \geq 0 \qquad \forall p \in R_e, \tag{192}$$

where $R_e = R_0 \cup \Delta R_1 \cup \cdots \cup \Delta R_e$. The baseline implementation is:

```
active = list(INITIAL)
results = []

for episode, delta in enumerate(((),) + DELTAS):
    active.extend(delta)
    model, rows, variables = build_shell()
    try:
        add_patterns(model, rows, variables, active)
        results.append(snapshot(model, episode, variables))
    finally:
        model.dispose()
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Split the builder into an invariant row shell plus explicit column additions. Construct the initial column set once, retain semantic row/column handles, and before episode e add exactly ΔR_e. Solve at the same episode boundaries as the baseline. Parameters, callbacks, output schema, and mathematical column coefficients remain unchanged.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. For every episode,

$$\boxed{R_e^{\text{persistent}} = R_e^{\text{fresh}} \quad \Longrightarrow \quad M_e^{\text{persistent}} = M_e^{\text{fresh}}.}$$

The modified implementation is:

```
model, rows, variables = build_shell()
results = []
try:
    add_patterns(model, rows, variables, INITIAL)

    for episode, delta in enumerate(((),) + DELTAS):
        add_patterns(model, rows, variables, delta)
        results.append(snapshot(model, episode, variables))
finally:
    model.dispose()
```

B.3.9 LP BASIS WARM START

Skill ID: `lp_basis_warm_start`.

Trigger Conditions. Apply `lp_basis_warm_start` when a recreated LP (or another compatible target LP) can receive a complete previously available simplex basis through stable semantic row and column identities. First determine whether retaining the same model is preferable; if so,

`persistent_model_reoptimization` may avoid explicit export/import. For conservative basis transfer, the constraint matrix, row senses, identities, and compatible domains must match, with explicitly understood permitted changes such as RHS or objective coefficients. Do not copy only a matching subset, fabricate basis statuses, or treat a primal incumbent as a basis.

Solver-Implementation Principle. For LP

$$M^{(e)} = \min\{c_e^\top x : Ax \bowtie b_e,\ \ell \le x \le u\},$$

a simplex basis can be represented by variable and row status vectors

$$B_e = (\mathrm{VBasis}_e, \mathrm{CBasis}_e).$$

When the next recreated LP has a compatible algebraic structure, map B_e by semantic IDs and import it before solving $M^{(e+1)}$. The target optimization problem remains

$$\boxed{M'_{e+1} = M_{e+1}};$$

only the solver's initial simplex state changes. Basis validity, solver acceptance, and runtime benefit are separate questions.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A sequence repeatedly recreates the same allocation LP while the demand RHS changes slightly. Variable creation order may differ between recreations, so positional basis copying is unsafe. Stable semantic variable and row IDs permit complete basis export and re-import when the matrix and domains remain compatible.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\begin{aligned} \min \quad & x_A + 2x_B, && (193)\\ \text{s.t.} \quad & x_A + x_B \ge d_e, && (194)\\ & x_A \le 4, && (195)\\ & x_A, x_B \ge 0. && (196) \end{aligned}$$

The baseline implementation is:

```
for episode, demand in enumerate(RHS_SEQUENCE):
    model, x, rows = build_lp(demand, reverse_order=bool(episode % 2))
    try:
        model.optimize()          # cold recreated LP
        require_optimum(model)
        collect_result(model, x, rows)
    finally:
        model.dispose()
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. After each appropriate basic LP solve, export complete `VBasis` and `CBasis` maps keyed by semantic IDs. Fully construct the next LP and apply its true RHS/objective changes first. Verify complete ID coverage and conservative compatibility, then assign both status arrays in aligned bulk form. If import fails after partial assignment, use a demonstrably clean fallback such as a fresh target build.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The episode model is unchanged:

$$\boxed{\min x_A + 2x_B \text{ s.t. } x_A + x_B \ge d_e,\ x_A \le 4,\ x \ge 0.}$$

Only its initial simplex basis is supplied. The modified implementation is:

```
saved = None

for episode, demand in enumerate(RHS_SEQUENCE):
```

```
    model, x, rows = build_lp(demand, reverse_order=bool(episode % 2))
    try:
        if saved is not None:
            vb_old, cb_old = saved
            model.setAttr(
                "VBasis",
                [x[key] for key in VAR_IDS],
                [vb_old[key] for key in VAR_IDS])
            model.setAttr(
                "CBasis",
                [rows[key] for key in ROW_IDS],
                [cb_old[key] for key in ROW_IDS])

        model.optimize()
        require_optimum(model)

        vb = model.getAttr("VBasis", [x[key] for key in VAR_IDS])
        cb = model.getAttr("CBasis", [rows[key] for key in ROW_IDS])
        saved = (dict(zip(VAR_IDS, vb)), dict(zip(ROW_IDS, cb)))
    finally:
        model.dispose()
```

B.3.10 INDEPENDENT SUBPROBLEM PARALLELISM

Skill ID: `independent_subproblem_parallelism`.

Trigger Conditions. Apply `independent_subproblem_parallelism` when a batch of optimization work units is independent conditional on a frozen input snapshot, so that the units can run concurrently without changing the original dependency graph. Typical examples are scenario subproblems for one fixed master decision or separate problem instances. Every worker must own its solver environment/models; only plain serializable data and results cross process boundaries. License/concurrency permission, CPU allocation, process-start behavior, and memory budget must be known. Do not parallelize future iterations whose inputs depend on current results, and do not pass live solver objects across processes.

Solver-Implementation Principle. Suppose iteration t fixes state y_t and requires

$$r_s = F_s(y_t), \qquad s \in S,$$

where the F_s share no mutable solver state and no task s depends on another task's result. Serial execution computes

$$F_1(y_t), F_2(y_t), \ldots, F_{|S|}(y_t)$$

sequentially. Parallel execution evaluates the same set concurrently, then applies a barrier:

$$\{r_s : s \in S\} \longrightarrow y_{t+1}.$$

Thus the algorithmic dependency

$$y_t \to \{F_s\}_{s \in S} \to y_{t+1}$$

is preserved exactly, and each subproblem satisfies

$$\boxed{M'_{t,s} = M_{t,s}}.$$

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A decomposition-style workflow evaluates several recourse LPs for the same frozen current master decision. The next master state is computed only after all current recourse results have returned. Recourse problems within one iteration are independent, but iterations themselves are sequential.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE. For fixed y_t, each scenario s solves

$$Q_s(y_t) = \min_{x_s}\{q_s^\top x_s : W_s x_s \geq h_s - T_s y_t,\ x_s \geq 0\}.$$

The next state is computed by a deterministic aggregation rule

$$y_{t+1} = G\big(y_t, \{Q_s(y_t), \text{required certificates}_s\}_{s\in S}\big).$$

The baseline implementation is:

```
for iteration in range(num_iterations):
    tasks = make_tasks(iteration, master_snapshot)

    # Baseline: same current-iteration subproblems, executed serially.
    returned = [solve_task(task) for task in tasks]
    ordered = validate_and_order_results(tasks, returned)

    # Dependency barrier.
    master_snapshot = next_master(master_snapshot, ordered)
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Keep task IDs, frozen master snapshots, subproblem builders, result schema, and aggregation rule unchanged. Dispatch only the current iteration's independent tasks to worker processes. Every worker creates/disposes its own environment and model. Validate the exact result-ID set and snapshot identity, reorder deterministically when required, and wait for the complete batch before computing the next master state. Coordinate worker count with per-solver thread allocations.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. Every scenario solves the same recourse model:

$$\boxed{Q'_s(y_t) = Q_s(y_t)}.$$

The same barrier precedes y_{t+1}, so the dependency graph is unchanged. The modified implementation is:

```
with ProcessPoolExecutor(
        max_workers=workers,
        mp_context=mp.get_context("spawn")) as pool:

    for iteration in range(num_iterations):
        tasks = make_tasks(iteration, master_snapshot)

        returned = list(pool.map(solve_task, tasks))
        ordered = validate_and_order_results(tasks, returned)

        # No task for iteration t+1 is created before this barrier.
        master_snapshot = next_master(master_snapshot, ordered)
```

B.3.11 SOLVER THREAD AND PORTFOLIO SELECTION

Skill ID: `solver_thread_and_portfolio_selection`.

Trigger Conditions. Apply `solver_thread_and_portfolio_selection` when one solve has an attributable optimization workload and solver-internal parallelism can be compared under an explicit CPU, memory, license, and application-semantics budget. Distinguish cooperative threads for one search from a concurrent portfolio of internal searches. Recover effective baseline settings, surrounding outer parallelism, callback requirements, and output contract. Do not use a portfolio mode that invalidates required callback semantics (for example, a required lazy-constraint path that the chosen concurrent mode does not support). More threads are not assumed to be better; single-thread or no-change policies are valid outcomes.

Solver-Implementation Principle. For fixed mathematical model M, let a resource policy be

$$\rho = (T, P, \ldots),$$

where T is the total solver thread allocation and P denotes an applicable portfolio/concurrent setting. The Skill compares compatible ρ under the same allocation:

$$\rho^* = \arg\min_{\rho \in \mathcal{R}} T_{\text{declared}}(M; \rho)$$

subject to memory, license, callback, and output constraints. If outer workers w are also active, the total requested solver threads must respect the job budget rather than maximizing w and T independently. In all cases,

$$\boxed{M' = M}.$$

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A callback-free maximum-independent-set MIP is solved repeatedly under a known job-level CPU allocation. The baseline already has an explicit resource policy. Candidate policies change only Gurobi's internal thread count and, when compatible, the number of concurrent MIP searches.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\max \quad \sum_{i \in V} x_i, \tag{197}$$

$$\text{s.t.} \quad x_i + x_j \leq 1 \qquad \forall \{i, j\} \in E, \tag{198}$$

$$x_i \in \{0, 1\} \qquad \forall i \in V. \tag{199}$$

The baseline implementation is:

```
settings = baseline_settings.copy()
with gp.Model("independent_set") as model:
    x = build_mis(model, EDGES)
    model.Params.Threads = settings["Threads"]
    model.Params.ConcurrentMIP = settings.get("ConcurrentMIP", 1)
    model.optimize()
    result = extract_and_validate(model, x)
```

(3) HOW THE MATHEMATICAL FORMULATION AND CODE ARE MODIFIED. Keep the MIS model, time/certificate contract, seed policy, callbacks (none in this fixture), and total job allocation fixed. Compare supported internal resource policies without silently adding hardware. When a portfolio comparison is performed, hold total `Threads` fixed so the experiment changes resource distribution rather than total CPU. Reject any candidate that violates application callback semantics or job-level resource constraints.

(4) MODIFIED MATHEMATICAL FORMULATION AND CODE. The independent-set model is unchanged:

$$\boxed{M' = M}.$$

Only the solver-internal resource policy differs. The modified implementation is:

```
settings = candidate_settings.copy()
if settings["Threads"] > allocated_threads:
    raise ValueError("thread budget exceeded")

with gp.Model("independent_set") as model:
    x = build_mis(model, EDGES)
    model.Params.Threads = settings["Threads"]
    model.Params.ConcurrentMIP = settings.get("ConcurrentMIP", 1)
    model.optimize()
    result = extract_and_validate(model, x)

# Keep the candidate only after repeated job-level measurement.
```

B.3.12 HINT AND BRANCH PRIORITY GUIDANCE

Skill ID: `hint_and_branch_priority_guidance`.

Trigger Conditions. Apply `hint_and_branch_priority_guidance` when a MIP has traceable, deployment-time-available advisory information for a specific variable family and the acquisition cost of that information can be charged fairly. Separate `VarHintVal`/`VarHintPri` from `BranchPriority`; a MIP `Start` is another mechanism and is not interchangeable with either. Existing starts, hints, and priorities must be inspected before applying a merge/replacement policy. Do not use hidden reference optima, evaluation-answer leakage, untraceable predictions, hard bound fixing, or unsupported attribute mappings. Unknown predictions may remain unspecified.

Solver-Implementation Principle. Let binary decision family y have an advisory prediction

$$\hat{y}_i$$

with confidence score q_i, and optionally a distinct branching importance score r_i. The patch supplies these as search guidance:

$$\texttt{VarHintVal}(y_i) = \hat{y}_i, \qquad \texttt{VarHintPri}(y_i) = q_i,$$

and/or

$$\texttt{BranchPriority}(y_i) = r_i.$$

These attributes do not add constraints or fix variables. Therefore

$$\boxed{M' = M},$$

while the solver's search trajectory may change. Hint prediction and branch-priority scoring are separate candidate mechanisms and must be evaluated separately unless the combined policy is explicitly the tested unit.

Before-and-After Case Study.

(1) NATURAL LANGUAGE PROBLEM. A capacitated facility-location model decides which facilities to open and how to ship customer demand. Before solving, a deterministic data-derived heuristic can provide advisory opening values, while a separate capacity-impact score can rank opening variables for branching. Neither signal changes feasibility.

(2) ORIGINAL MATHEMATICAL FORMULATION AND CODE.

$$\min \quad \sum_{i \in F} f_i y_i + \sum_{i \in F} \sum_{j \in C} c_{ij} x_{ij}, \tag{200}$$

$$\text{s.t.} \quad \sum_{i \in F} x_{ij} = d_j \qquad \forall j \in C, \tag{201}$$

$$\sum_{j \in C} x_{ij} \le U_i y_i \qquad \forall i \in F, \tag{202}$$

$$y_i \in \{0,1\}, \quad x_{ij} \ge 0. \tag{203}$$

The baseline implementation is:

```
model, y, shipment = build_model()

# Baseline: do not add new guidance.
# Preserve any pre-existing starts/hints/priorities according to the
# experiment's baseline policy.
model.optimize()
result = validate_solution(model, y, shipment)
```

(3) How the Mathematical Formulation and Code Are Modified. Derive the hint and branch-impact signals from legitimate input data before final evaluation, with deterministic tie-breaking and measured preparation cost. Snapshot existing guidance attributes. Apply either hints-only, branch-only, or an explicitly named combined mode to the intended opening-variable family. No constraint, variable bound, objective coefficient, or feasibility tolerance changes.

(4) Modified Mathematical Formulation and Code. The capacitated facility-location model is unchanged:

$$\boxed{M' = M}.$$

The candidate changes only advisory search attributes. The modified implementation is:

```
model, y, shipment = build_model()

if mode in ("hint", "combined"):
    for i in FACILITIES:
        y[i].VarHintVal = hint_value[i]
        y[i].VarHintPri = hint_confidence[i]

if mode in ("branch", "combined"):
    for i in FACILITIES:
        y[i].BranchPriority = branch_priority[i]

model.optimize()
result = validate_solution(model, y, shipment)

# Signal-generation time is included in the declared total budget.
```

C Verified Before-and-After Case Studies

The following five cases provide complementary, executed examples of exact product linearization, integrality relaxation, symmetry reduction, valid inequalities, and solver-implementation changes. They were selected from the 179-entry FrontierOR archive to illustrate distinct mechanisms, rather than to estimate average performance. Each original formulation is a direct translation of its supplied problem description and schema; the case identifier names the benchmark entry, not a claim to reproduce a publication's original formulas. The semantic intermediate representation $\mathcal{IR}$ aligns each change with its implementation.

Reading the comparisons. For each case, the left column is the original formulation or code and the right column is the final retained version. Teal marks added or changed model terms and code. The listings contain complete `build_model(data)` functions; line wrapping and blank alignment preserve the executed Python abstract syntax trees. The common runner handles optimization and solution extraction. Code spanning two panels continues at the displayed line number.

C.1 Quadratic Knapsack (caprara1999)

Natural-Language Problem. Choose items for one capacity-limited knapsack. Each item has a positive integer weight and an individual profit. A symmetric nonnegative profit matrix also rewards selecting pairs of items. Total profit is the sum of all matrix entries whose row and column items are both selected; consequently, each off-diagonal selected pair contributes twice its matrix coefficient. Maximize profit without exceeding capacity.

Sets and Parameters. Let $I = \{0, \dots, n-1\}$ index items, $w_i > 0$ be weights, and B be capacity, with $\max_i w_i \le B < \sum_i w_i$. Profits satisfy $p_{ij} = p_{ji} \ge 0$. Let $E = \{(i,j) : 0 \le i < j < n,\ p_{ij} \ne 0\}$ denote nonzero unordered profit pairs.

Decision Variables. The original and transformed models both use binary selections x_i. The transformed model additionally uses continuous $z_{ij} \in [0,1]$, $(i,j) \in E$, to represent the products $x_i x_j$.

Original Formulation $\mathcal{M}_0$

Direct translation

$$\begin{aligned} \max \quad & \sum_{i \in I} \sum_{j \in I} p_{ij} x_i x_j \\ \text{s.t.} \quad & \sum_{i \in I} w_i x_i \le B, \\ & x_i \in \{0,1\} \quad (i \in I). \end{aligned} \tag{204}$$

Retained Formulation $\mathcal{M}_1$

Teal indicates the transformation

$$\begin{aligned} \max \quad & \sum_{i \in I} p_{ii} x_i \\ & + 2 \sum_{(i,j) \in E} p_{ij} z_{ij} \\ \text{s.t.} \quad & \sum_{i \in I} w_i x_i \le B, \\ & x_i \in \{0,1\} \quad (i \in I), \\ & 0 \le z_{ij} \le 1, \\ & z_{ij} \le x_i, \quad z_{ij} \le x_j, \\ & z_{ij} \ge x_i + x_j - 1, \\ & (i,j) \in E. \end{aligned} \tag{205}$$

Quadratic Knapsack: Code Comparison

Original Code $\mathcal{C}_0$

Comparison lines 1–46; blank lines align edits

```
import gurobipy as gp
from gurobipy import GRB


def build_model(data):
    n = data["n"]
    weights = data["weights"]
    profits = data["profit_matrix"]
    capacity = data["capacity"]
    model = gp.Model(
        'quadratic_knapsack_lite'
        'ral')
    x = {i: model.addVar(
        vtype=GRB.BINARY,
        name=f"x[{i}]") for i in
        range(n)}









    model.addConstr(gp.quicksum(
        weights[i] * x[i] for i in
        range(n)) <= capacity,
        name="capacity")









    model.setObjective(gp.quicksum(
        profits[i][j] * x[i] * x[j]
        for i in range(n) for j in
        range(n)), GRB.MAXIMIZE)

    return model
```

Retained Code $\mathcal{C}_1$

Comparison lines 1–46; teal marks changed code

```
import gurobipy as gp
from gurobipy import GRB


def build_model(data):
    n = data["n"]
    weights = data["weights"]
    profits = data["profit_matrix"]
    capacity = data["capacity"]
    model = gp.Model(
        'quadratic_knapsack_line'
        'arized')
    x = {i: model.addVar(
        vtype=GRB.BINARY,
        name=f"x[{i}]") for i in
        range(n)}
    pairs = [(i, j) for i in range(n)
        for j in range(i + 1, n) if
        profits[i][j] != 0]
    z = {(i, j): model.addVar(lb=0.0,
        ub=1.0, vtype=GRB.CONTINUOUS,
        name=f"z[{i},{j}]") for i, j
        in pairs}
    model.addConstr(gp.quicksum(
        weights[i] * x[i] for i in
        range(n)) <= capacity,
        name="capacity")
    for i, j in pairs:
        model.addConstr(z[i, j] <=
            x[i],
            name=f"product_upper_i["
            f"{i},{j}]")
        model.addConstr(z[i, j] <=
            x[j],
            name=f"product_upper_j["
            f"{i},{j}]")
        model.addConstr(z[i, j] >=
            x[i] + x[j] - 1,
            name=f"product_lower["
            f"{i},{j}]")
    model.setObjective(gp.quicksum(
        profits[i][i] * x[i] for i in
        range(n)) + gp.quicksum(2 *
        profits[i][j] * z[i, j] for i,
        j in pairs), GRB.MAXIMIZE)
    return model
```

Correspondence. The left listing implements Equation 204; the right listing implements Equation 205.

C.2 GRAPH COLORING (ALFIERI2020)

Natural-Language Problem. Assign a color to every vertex of an undirected graph so that adjacent vertices receive different colors. Minimize the number of colors used. Vertices without incident edges must also receive a color and activate its use indicator.

Sets and Parameters. Let $G = (V, E)$ have n vertices. Preserve all edge-endpoint labels and add fresh integer labels for unnamed isolated vertices until $|V| = n$. A greedy coloring in increasing vertex-label order supplies a feasible palette of size k; let $K = \{0, \dots, k-1\}$. Every color has unit use cost.

Decision Variables. Binary x_{vc} assigns vertex v to color c. The original y_c is binary. Only the domain of y_c is changed to the continuous interval $[0, 1]$ in the transformed model; all x_{vc} remain binary.

Original Formulation $\mathcal{M}_0$

Direct translation

$$
\begin{aligned}
\min \quad & \sum_{c \in K} y_c \\
\text{s.t.} \quad & \sum_{c \in K} x_{vc} = 1 \quad (v \in V), \\
& x_{uc} + x_{vc} \le 1 \\
& \qquad (\{u, v\} \in E,\ c \in K), \\
& x_{vc} \le y_c \quad (v \in V, c \in K), \\
& x_{vc} \in \{0, 1\} \quad (v \in V, c \in K), \\
& y_c \in \{0, 1\} \quad (c \in K).
\end{aligned}
\tag{206}
$$

Retained Formulation $\mathcal{M}_1$

Teal indicates the transformation

$$
\begin{aligned}
\min \quad & \sum_{c \in K} y_c \\
\text{s.t.} \quad & \sum_{c \in K} x_{vc} = 1 \quad (v \in V), \\
& x_{uc} + x_{vc} \le 1 \\
& \qquad (\{u, v\} \in E,\ c \in K), \\
& x_{vc} \le y_c \quad (v \in V, c \in K), \\
& x_{vc} \in \{0, 1\} \quad (v \in V, c \in K), \\
& y_c \in [0, 1] \quad (c \in K).
\end{aligned}
\tag{207}
$$

Graph Coloring: Code Comparison

Original Code $\mathcal{C}_0$

Comparison lines 1–54; blank lines align edits

```
import gurobipy as gp
from gurobipy import GRB

def build_model(data):
    n = int(data['num_nodes'])
    edges = [tuple(e) for e in
        data['edges']]
    vertices = set(v for e in edges
        for v in e)
    if len(vertices) > n:
        raise ValueError(
            'More edge endpoints tha'
            'n vertices')
    fresh = 0
    while len(vertices) < n:
        if fresh not in vertices:
            vertices.add(fresh)
        fresh += 1
    V = sorted(vertices)
    adjacency = {v: set() for v in V}
    for u, v in edges:
        adjacency[u].add(v)
        adjacency[v].add(u)
    greedy = {}
    for v in V:
        used = {greedy[u] for u in
            adjacency[v] if u in
            greedy}
        c = 0
        while c in used:
            c += 1
        greedy[v] = c
    K = range(1 + max(greedy.values(),
        default=-1))
    model = gp.Model(
        'literal_graph_coloring')
    x = model.addVars(V, K,
        vtype=GRB.BINARY, name='x')
    y = model.addVars(K,
        vtype=GRB.BINARY, name='y')

    model.addConstrs((gp.quicksum(x[v,
        c] for c in K) == 1 for v in
        V), name='assign')
    model.addConstrs((x[u,c] + x[v,c]
        <= 1 for u,v in edges for c in
        K), name='edge')
    model.addConstrs((x[v,c] <= y[c]
        for v in V for c in K),
        name='use')
    model.setObjective(gp.quicksum(
        y[c] for c in K),
        GRB.MINIMIZE)
    return model
```

Retained Code $\mathcal{C}_1$

Comparison lines 1–54; teal marks changed code

```
import gurobipy as gp
from gurobipy import GRB

def build_model(data):
    n = int(data['num_nodes'])
    edges = [tuple(e) for e in
        data['edges']]
    vertices = set(v for e in edges
        for v in e)
    if len(vertices) > n:
        raise ValueError(
            'More edge endpoints tha'
            'n vertices')
    fresh = 0
    while len(vertices) < n:
        if fresh not in vertices:
            vertices.add(fresh)
        fresh += 1
    V = sorted(vertices)
    adjacency = {v: set() for v in V}
    for u, v in edges:
        adjacency[u].add(v)
        adjacency[v].add(u)
    greedy = {}
    for v in V:
        used = {greedy[u] for u in
            adjacency[v] if u in
            greedy}
        c = 0
        while c in used:
            c += 1
        greedy[v] = c
    K = range(1 + max(greedy.values(),
        default=-1))
    model = gp.Model(
        'literal_graph_coloring')
    x = model.addVars(V, K,
        vtype=GRB.BINARY, name='x')
    y = model.addVars(K, lb=0, ub=1,
        vtype=GRB.CONTINUOUS,
        name='y')
    model.addConstrs((gp.quicksum(x[v,
        c] for c in K) == 1 for v in
        V), name='assign')
    model.addConstrs((x[u,c] + x[v,c]
        <= 1 for u,v in edges for c in
        K), name='edge')
    model.addConstrs((x[v,c] <= y[c]
        for v in V for c in K),
        name='use')
    model.setObjective(gp.quicksum(
        y[c] for c in K),
        GRB.MINIMIZE)
    return model
```

Correspondence. The left listing implements Equation 206; the right listing implements Equation 207.

C.3 BIN PACKING (DELORME2020)

Natural-Language Problem. Pack all demanded item copies into identical bins of integer capacity c. Type t has width w_t and demand d_t. Every copy must be assigned exactly once, each bin's total width must fit its capacity, and the objective is to minimize the number of opened bins.

Sets and Parameters. Let $\mathcal{T} = \{0, \dots, m-1\}$ index types, $0 < w_t < c$ be integer widths, and $d_t \in \mathbb{Z}_{\geq 0}$ be demands. Set $N = \sum_t d_t = n$ and $\mathcal{B} = \{0, \dots, N-1\}$. Since each copy fits alone, N candidate bins suffice.

Decision Variables. Both formulations use integer $a_{tb} \in \{0, \dots, d_t\}$ for the number of copies of type t assigned to bin b, and binary y_b to indicate that bin b is opened.

Original Formulation $\mathcal{M}_0$

Direct translation

$$\begin{aligned} \min \quad & \sum_{b \in \mathcal{B}} y_b \\ \text{s.t.} \quad & \sum_{b \in \mathcal{B}} a_{tb} = d_t \quad (t \in \mathcal{T}), \\ & \sum_{t \in \mathcal{T}} w_t a_{tb} \leq c y_b \quad (b \in \mathcal{B}), \\ & a_{tb} \in \mathbb{Z}_{\geq 0}, \quad a_{tb} \leq d_t \\ & \qquad (t \in \mathcal{T}, b \in \mathcal{B}), \\ & y_b \in \{0, 1\} \quad (b \in \mathcal{B}). \end{aligned} \tag{208}$$

Retained Formulation $\mathcal{M}_1$

Teal indicates the transformation

$$\begin{aligned} \min \quad & \sum_{b \in \mathcal{B}} y_b \\ \text{s.t.} \quad & \sum_{b \in \mathcal{B}} a_{tb} = d_t \quad (t \in \mathcal{T}), \\ & \sum_{t \in \mathcal{T}} w_t a_{tb} \leq c y_b \quad (b \in \mathcal{B}), \\ & a_{tb} \in \mathbb{Z}_{\geq 0}, \quad a_{tb} \leq d_t \\ & \qquad (t \in \mathcal{T}, b \in \mathcal{B}), \\ & y_b \in \{0, 1\} \quad (b \in \mathcal{B}), \\ & y_b \geq y_{b+1} \\ & \qquad (b = 0, \dots, N-2). \end{aligned} \tag{209}$$

Bin Packing: Code Comparison

Original Code $\mathcal{C}_0$

Comparison lines 1–36; blank lines align edits

```
import gurobipy as gp
from gurobipy import GRB

def build_model(data):
    T = range(int(data['m']))
    demands = data['demands']
    weights = data['weights']
    capacity = data['c']
    N = sum(demands)
    if N != data['n']:
        raise ValueError(
            'Total item count disagr'
            'ees with demands')
    B = range(N)
    model = gp.Model(
        'literal_bin_packing')
    a = model.addVars(T, B,
        vtype=GRB.INTEGER, lb=0, ub={(
        t,b): demands[t] for t in T
        for b in B}, name='a')
    y = model.addVars(B,
        vtype=GRB.BINARY, name='y')
    model.addConstrs((gp.quicksum(a[t,
        b] for b in B) == demands[t]
        for t in T), name='demand')
    model.addConstrs((gp.quicksum(
        weights[t]*a[t,b] for t in T)
        <= capacity*y[b] for b in B),
        name='capacity')
    model.setObjective(gp.quicksum(
        y[b] for b in B),
        GRB.MINIMIZE)




    return model
```

Retained Code $\mathcal{C}_1$

Comparison lines 1–36; teal marks changed code

```
import gurobipy as gp
from gurobipy import GRB

def build_model(data):
    T = range(int(data['m']))
    demands = data['demands']
    weights = data['weights']
    capacity = data['c']
    N = sum(demands)
    if N != data['n']:
        raise ValueError(
            'Total item count disagr'
            'ees with demands')
    B = range(N)
    model = gp.Model(
        'literal_bin_packing')
    a = model.addVars(T, B,
        vtype=GRB.INTEGER, lb=0, ub={(
        t,b): demands[t] for t in T
        for b in B}, name='a')
    y = model.addVars(B,
        vtype=GRB.BINARY, name='y')
    model.addConstrs((gp.quicksum(a[t,
        b] for b in B) == demands[t]
        for t in T), name='demand')
    model.addConstrs((gp.quicksum(
        weights[t]*a[t,b] for t in T)
        <= capacity*y[b] for b in B),
        name='capacity')
    model.setObjective(gp.quicksum(
        y[b] for b in B),
        GRB.MINIMIZE)
    model.addConstrs((y[b] >= y[b+1]
        for b in range(max(0, N-1))),
        name='bin_order')
    return model
```

Correspondence. The left listing implements Equation 208; the right listing implements Equation 209.

C.4 Capacitated Facility Location (klose2005)

Natural-Language Problem. Choose plants to open and allocate every customer's demand across open plants. Demand may be split continuously among plants. Opening incurs a fixed cost; shipping cost for a customer–plant pair is specified for serving that customer's entire demand and scales with the assigned fraction. Meet every demand while respecting plant capacities and minimize shipping plus opening costs.

Sets and Parameters. Let $I = \{0, \dots, n_c - 1\}$ index customers and $J = \{0, \dots, n_p - 1\}$ index plants. Demand is $d_i \geq 0$, capacity is $q_j > 0$, fixed cost is $f_j \geq 0$, and $s_{ij} \geq 0$ is the shipping cost per entire customer demand. Define $D = \sum_i d_i$ and $q_{\max} = \max_j q_j$.

Decision Variables. Both formulations use continuous fractions $x_{ij} \in [0, 1]$ and binary opening variables y_j. The shipping term is $s_{ij}x_{ij}$, without multiplying by d_i again. The links $x_{ij} \leq y_j$ also apply when $d_i = 0$.

Original Formulation $\mathcal{M}_0$

Direct translation

$$
\begin{aligned}
\min \quad & \sum_{i \in I} \sum_{j \in J} s_{ij} x_{ij} \\
& + \sum_{j \in J} f_j y_j \\
\text{s.t.} \quad & \sum_{j \in J} x_{ij} = 1 \quad (i \in I), \\
& \sum_{i \in I} d_i x_{ij} \leq q_j y_j \quad (j \in J), \\
& x_{ij} \leq y_j \quad (i \in I, j \in J), \\
& x_{ij} \in [0, 1] \quad (i \in I, j \in J), \\
& y_j \in \{0, 1\} \quad (j \in J).
\end{aligned}
\tag{210}
$$

Retained Formulation $\mathcal{M}_1$

Teal indicates the transformation

$$
\begin{aligned}
\min \quad & \sum_{i \in I} \sum_{j \in J} s_{ij} x_{ij} \\
& + \sum_{j \in J} f_j y_j \\
\text{s.t.} \quad & \sum_{j \in J} x_{ij} = 1 \quad (i \in I), \\
& \sum_{i \in I} d_i x_{ij} \leq q_j y_j \quad (j \in J), \\
& x_{ij} \leq y_j \quad (i \in I, j \in J), \\
& x_{ij} \in [0, 1] \quad (i \in I, j \in J), \\
& y_j \in \{0, 1\} \quad (j \in J), \\
& \sum_{j \in J} y_j \geq \lceil D / q_{\max} \rceil \,.
\end{aligned}
\tag{211}
$$

Capacitated Facility Location: Code Comparison

Original Code $\mathcal{C}_0$

Comparison lines 1–44; blank lines align edits

```

import gurobipy as gp
from gurobipy import GRB

def build_model(data):
    I = range(int(
        data['num_customers']))
    J = range(int(data['num_plants']))
    demand = data['customers']['demands']
    capacity = data['plants']['capacities'
  ]
    fixed = data['plants']['fixed_costs']
    shipping = data['shipping_costs']
    model = gp.Model(
        'literal_splittable_faci'
        'lity_location')
    x = model.addVars(I, J, lb=0,
        ub=1, vtype=GRB.CONTINUOUS,
        name='x')
    y = model.addVars(J,
        vtype=GRB.BINARY, name='y')
    model.addConstrs((gp.quicksum(x[i,
        j] for j in J) == 1 for i in
        I), name='serve')
    model.addConstrs((gp.quicksum(
        demand[i]*x[i,j] for i in I)
        <= capacity[j]*y[j] for j in
        J), name='capacity')
    model.addConstrs((x[i,j] <= y[j]
        for i in I for j in J),
        name='open_link')
```

Retained Code $\mathcal{C}_1$

Comparison lines 1–44; teal marks changed code

```
import numpy as np
from scipy.sparse import coo_matrix
from fractions import Fraction
import math
import gurobipy as gp
from gurobipy import GRB

def build_model(data):
    I = range(int(
        data['num_customers']))
    J = range(int(data['num_plants']))
    demand = data['customers']['demands']
    capacity = data['plants']['capacities'
  ]
    fixed = data['plants']['fixed_costs']
    shipping = data['shipping_costs']
    model = gp.Model(
        'literal_splittable_faci'
        'lity_location')
    x = model.addVars(I, J, lb=0,
        ub=1, vtype=GRB.CONTINUOUS,
        name='x')
    y = model.addVars(J,
        vtype=GRB.BINARY, name='y')
    columns = [x[i,j] for i in I for j
        in J] + [y[j] for j in J]
    nj = len(J)
    offset = len(I)*nj
    (row_coordinates,
        column_coordinates,
        coefficients) = [], [], []
    (senses, right_sides,
        row_names) = [], [], []
    def push(entries, sense,
        right_side, name):
        r = len(row_names)
        for column, coefficient in entries
  :
            row_coordinates.append(r)
            column_coordinates.append(
                column)
            coefficients.append(
                coefficient)
        senses.append(sense)
        right_sides.append(right_side)
        row_names.append(name)
```

Capacitated Facility Location: Code Comparison (continued)

Original Code $\mathcal{C}_0$

Comparison lines 45–87; blank lines align edits

```
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82      model.setObjective(gp.quicksum(
83          shipping[i][j]*x[i,j] for i in
84          I for j in J) + gp.quicksum(
85          fixed[j]*y[j] for j in J),
86          GRB.MINIMIZE)
87      return model
```

Retained Code $\mathcal{C}_1$

Comparison lines 45–87; teal marks changed code

```
45      for i in I:
46          push(((i*nj+j, 1) for j in J),
47              '=', 1, f'serve[{i}]')
48      for j in J:
49          push([(i*nj+j, demand[i]) for
50              i in I] + [(offset+j,
51              -capacity[j])], '<', 0,
52              f'capacity[{j}]')
53      for i in I:
54          for j in J:
55              push([(i*nj+j, 1), (
56                  offset+j, -1)], '<',
57                  0,
58                  f'open_link[{i},{j}]')
59      total_demand = sum(Fraction(str(
60          v)) for v in demand)
61      maximum_capacity = max(Fraction(
62          str(v)) for v in capacity)
63      required_open = math.ceil(
64          total_demand /
65          maximum_capacity)
66      push(((offset+j, 1) for j in J),
67          '>', required_open,
68          'minimum_open_count')
69      matrix = coo_matrix((np.asarray(
70          coefficients, dtype=float), (
71          row_coordinates,
72          column_coordinates)),
73          shape=(len(row_names), len(
74              columns))).tocsr()
75      matrix.sum_duplicates()
76      matrix.sort_indices()
77      model.addMConstr(matrix, columns,
78          np.asarray(senses,
79          dtype='<U1'), np.asarray(
80          right_sides, dtype=float),
81          name=row_names)
82      model.setObjective(gp.quicksum(
83          shipping[i][j]*x[i,j] for i in
84          I for j in J) + gp.quicksum(
85          fixed[j]*y[j] for j in J),
86          GRB.MINIMIZE)
87      return model
```

Correspondence. The left listing implements Equation 210; the right listing implements Equation 211. The right listing includes both the minimum-open-count row and the retained sparse-matrix construction. All demand, capacity, activation, and strengthening rows are inserted in the same matrix.

C.5 FIXED-CHARGE TRANSPORTATION (ROBERTI2015)

Natural-Language Problem. Ship every source's supply to meet each destination's demand exactly in a complete bipartite network. Each route has a per-unit transportation cost and a nonnegative fixed charge whenever used. A route's capacity is the smaller of its source supply and destination demand. Minimize variable transportation costs plus activation charges.

Sets and Parameters. Let $I = \{0, \dots, m-1\}$ index sources and $J = \{0, \dots, n-1\}$ index destinations. Positive supplies s_i and demands d_j satisfy $\sum_i s_i = \sum_j d_j$. Let $u_{ij} = \min\{s_i, d_j\}$, with unit costs c_{ij} and fixed costs $f_{ij} \geq 0$. For each source, sort routes by (u_{ij}, j); take the longest prefix T_i whose cumulative capacity is strictly below s_i.

Decision Variables. Both formulations use continuous shipment quantities $q_{ij} \in [0, u_{ij}]$ and binary activations y_{ij}. An unused route can be closed without increasing cost. All route-indexed bounds and domains below apply to every $(i, j) \in I \times J$.

Original Formulation $\mathcal{M}_0$

Direct translation

$$
\begin{aligned}
\min \quad & \sum_{i \in I} \sum_{j \in J} \left(c_{ij} q_{ij} + f_{ij} y_{ij} \right) \\
\text{s.t.} \quad & \sum_{j \in J} q_{ij} = s_i \quad (i \in I), \\
& \sum_{i \in I} q_{ij} = d_j \quad (j \in J), \\
& q_{ij} \leq u_{ij} y_{ij}, \\
& 0 \leq q_{ij} \leq u_{ij}, \quad q_{ij} \in \mathbb{R}, \\
& y_{ij} \in \{0, 1\}.
\end{aligned}
\tag{212}
$$

Retained Formulation $\mathcal{M}_1$

Teal indicates the transformation

$$
\begin{aligned}
\min \quad & \sum_{i \in I} \sum_{j \in J} \left(c_{ij} q_{ij} + f_{ij} y_{ij} \right) \\
\text{s.t.} \quad & \sum_{j \in J} q_{ij} = s_i \quad (i \in I), \\
& \sum_{i \in I} q_{ij} = d_j \quad (j \in J), \\
& q_{ij} \leq u_{ij} y_{ij}, \\
& 0 \leq q_{ij} \leq u_{ij}, \quad q_{ij} \in \mathbb{R}, \\
& y_{ij} \in \{0, 1\}, \\
& \sum_{j \in J \setminus T_i} y_{ij} \geq 1 \quad (i \in I).
\end{aligned}
\tag{213}
$$

Fixed-Charge Transportation: Code Comparison

Original Code $\mathcal{C}_0$

Comparison lines 1–32; blank lines align edits

```
import gurobipy as gp
from gurobipy import GRB

def build_model(data):
    I = range(int(
        data['num_sources']))
    J = range(int(data['num_sinks']))
    supply = data['supply']
    demand = data['demand']
    capacity = data['capacity']
    fixed = data['fixed_cost']
    variable = data['variable_cost']
    if any(fixed[i][j] < 0 for i in I
        for j in J):
        raise ValueError(
            'Fixed charges must be n'
            'onnegative costs')
    model = gp.Model(
        'literal_fixed_charge_tr'
        'ansport')
    q = model.addVars(I, J, lb=0, ub={
        (i,j): capacity[i][j] for i in
        I for j in J},
        vtype=GRB.CONTINUOUS,
        name='q')
    y = model.addVars(I, J,
        vtype=GRB.BINARY, name='y')
    model.addConstrs((gp.quicksum(q[i,
        j] for j in J) == supply[i]
        for i in I), name='supply')
    model.addConstrs((gp.quicksum(q[i,
        j] for i in I) == demand[j]
```

Retained Code $\mathcal{C}_1$

Comparison lines 1–32; teal marks changed code

```
import gurobipy as gp
from gurobipy import GRB

def build_model(data):
    I = range(int(
        data['num_sources']))
    J = range(int(data['num_sinks']))
    supply = data['supply']
    demand = data['demand']
    capacity = data['capacity']
    fixed = data['fixed_cost']
    variable = data['variable_cost']
    if any(fixed[i][j] < 0 for i in I
        for j in J):
        raise ValueError(
            'Fixed charges must be n'
            'onnegative costs')
    model = gp.Model(
        'literal_fixed_charge_tr'
        'ansport')
    q = model.addVars(I, J, lb=0, ub={
        (i,j): capacity[i][j] for i in
        I for j in J},
        vtype=GRB.CONTINUOUS,
        name='q')
    y = model.addVars(I, J,
        vtype=GRB.BINARY, name='y')
    model.addConstrs((gp.quicksum(q[i,
        j] for j in J) == supply[i]
        for i in I), name='supply')
    model.addConstrs((gp.quicksum(q[i,
        j] for i in I) == demand[j]
```

Fixed-Charge Transportation: Code Comparison (continued)

Original Code $\mathcal{C}_0$

Comparison lines 33–63; blank lines align edits

```
        for j in J), name='demand')
    model.addConstrs((q[i,j] <=
        capacity[i][j]*y[i,j] for i in
        I for j in J),
        name='route_use')
    model.setObjective(gp.quicksum(
        variable[i][j]*q[i,j] +
        fixed[i][j]*y[i,j] for i in I
        for j in J), GRB.MINIMIZE)
                                                
```