Reasoning in a Combinatorial and Constrained World: Benchmarking LLMs on Natural-Language Combinatorial Optimization

Xia Jiang[1,*], **Jing Chen**[2,*], **Cong Zhang**[3], **Jie Gao**[4],
Chengpeng Hu[1], **Chenhao Zhang**[5], **Yaoxin Wu**[1], **Yingqian Zhang**[1]

[1]Eindhoven University of Technology, [2]Brandenburg University of Technology,
[3]Nanyang Technological University, [4]Delft University of Technology,
[5]Southeast University

[*]Equal contribution. **Correspondence:** y.wu2@tue.nl

Abstract

While large language models (LLMs) have shown strong performance in math and logic reasoning, their ability to handle combinatorial optimization (CO)—searching high-dimensional solution spaces under hard constraints—remains underexplored. To bridge the gap, we introduce NLCO, a **N**atural **L**anguage **C**ombinatorial **O**ptimization benchmark that evaluates LLMs on end-to-end CO reasoning: given a language-described decision-making scenario, the model must output a discrete solution without writing code or calling external solvers. NLCO covers 43 CO problems and is organized using a four-layer taxonomy of variable types, constraint families, global patterns, and objective classes, enabling fine-grained evaluation. We provide solver-annotated solutions and comprehensively evaluate LLMs by feasibility, solution optimality, and reasoning efficiency. Experiments across a wide range of modern LLMs show that high-performing models achieve strong feasibility and solution quality on small instances, but both degrade as instance size grows, even if more tokens are used for reasoning. We also observe systematic effects across the taxonomy: set-based tasks are relatively easy, whereas graph-structured problems and bottleneck objectives lead to more frequent failures. The benchmark dataset and code for data generation and evaluation are publicly available [1] [2].

1 Introduction

Recent advances in large language models (LLMs) have performed remarkably across domains such as language understanding (Wang et al., 2024), programming (Lei et al., 2025), and planning (Shao et al., 2025). From arithmetic calculation to strategic decision making, LLMs are increasingly being explored as general-purpose problem solvers (Mirzadeh et al., 2025; Singhi et al., 2025; Alkayyal et al., 2026), sparking interest in the evaluation of their limits in reasoning (Plaat et al., 2025). However, most existing evaluations emphasize relatively simple reasoning competencies, such as arithmetic (Zheng et al., 2022), multi-hop question answering (QA) (Wu et al., 2025), or rule satisfaction (Zhou et al., 2025; Chen et al., 2025), which, while informative, offer limited insight into how well LLMs perform decision making under high-dimensional and constraint settings. Thus, there is a need for benchmarks that move beyond surface-level pattern recognition and probe LLMs by tasks with explicit constraints and objectives (Mondorf and Plank, 2024), so as to systematically assess their potential in more decision-support scenarios.

Combinatorial optimization (CO) offers a rigorous evaluation challenge as it requires reasoning beyond simple arithmetic. Deductive CO reasoning often involves: 1) objective/constraint modeling, 2) constructing discrete decisions (e.g., routes, schedules) that scale with instance size, 3) global consistency checking where a single violation invalidates the solution, and 4) reasoning about solution improvement and near-optimality under a combinatorial landscape. Unlike open-ended dialogue or F1-scored QA tasks, which often reward locally plausible or partially correct outputs, CO exposes the non-local nature of reasoning. Greedy or myopic solutions may trigger infeasibility or drastic objective degradation (Solomon, 1987). Critically, CO enables two-level evaluation: solutions are first judged by feasibility (as a binary indicator), and feasible ones are then ranked by objective values, yielding a fine-grained criterion. Since many CO problems are NP-hard, evaluation also naturally involves a quality-efficiency trade-off. Understanding how LLMs perform also matters beyond benchmarking: CO underlie real-world decision making, such as routing, planning, allocation (Bengio et al., 2021), so it also helps gauge LLMs' potential as

---

[1] https://huggingface.co/datasets/summer142857jiang/NLCO

[2] https://github.com/jing12e/nlco

Findings of the Association for Computational Linguistics: ACL 2026, pages 30592–30648

decision-support agents for realistic scenarios.

In this paper, we introduce NLCO, the **N**atural **L**anguage **C**ombinatorial **O**ptimization benchmark, which frames CO tasks as contextualized decision making scenarios, and LLMs are evaluated by taking as input textual CO specifications to reason out explicit discrete solutions. The solution feasibility is assessed by problem-specific constraints and optimization quality is measured by an objective. NLCO significantly differs from optimization benchmarks that primarily evaluate code writing, heuristic calling, or external solver invoking (Yang et al., 2025c; Imajuku et al., 2025; Sun et al., 2025). These indirect prompts resort to solvers, confounding model performance with external search and constraint handling, making it hard to attribute success/failure to the model's internal reasoning. We deliberately remove the usage of external solvers so that LLMs must reason over the original constrained, combinatorial solution space without tool assistance. This setup exposes failure modes (e.g., constraint violation, missing variable), facilitating diagnostic analysis rather than mere task accuracy.

To support systematic coverage and diagnostic analysis, NLCO tasks are organized using a four-layer taxonomy, which is specified by the following dimensions: variable type, constraint-family group, global constraint pattern(s), and objective class. This taxonomy enables controlled comparisons along structural dimensions (e.g., graph vs. set) and supports decomposed (and structural) evaluations by reporting metrics within each dimension. Within this framework, NLCO comprises 43 problems that span routing, scheduling, packing, etc. Problem instances are drawn from standard CO libraries and real-world scenarios. All instances are scaled to be tractable for LLMs, and rendered as textual scenarios while preserving their original problem structure. The comprehensive and well-processed instances provide a reusable resource for future LLM benchmarking.

Research Questions. The end-to-end requirement and four-layer taxonomy in NLCO enable systematic investigation on when and why LLMs succeed or fail for CO by internal reasoning capability. Accordingly, we focus on three research questions (RQs): **RQ1 (End-to-end solving capability)**: To what extent can current LLMs reason over a constrained, combinatorial solution space, solving CO problems end-to-end from textual descriptions? **RQ2 (Problem structure-aware reliability)**: How does problem structure (e.g., variable types, constraint families) influence reasoning reliability, and which, among them, primarily drive infeasible or suboptimal solutions? **RQ3 (Quality-efficiency trade-off)**: How well can LLMs balance reasoning efficacy (e.g., optimality) against efficiency (e.g., token usage) when solving NP-hard CO problems? We answer these RQs using the NLCO benchmark to probe the limits of LLMs reasoning with real-world constraints and combinatorics, and to assess their reliability as decision-support agents under verifiable constraints and resource budgets.

Benchmark	**CO**	**#Prob.**	**E2E**	**Ctx.**	**Taxonomy**
FrontierCO (Feng et al., 2025)	✓	8	✗	✗	Application domain
CO-Bench (Sun et al., 2025)	✓	36	✗	✗	Application domain
CP-Bench (Michailidis et al., 2025)	✗	N/A	✗	✗	N/A
NPHardEval (Fan et al., 2024)	✗	5	✓	✗	Complexity class
GraphArena (Tang et al., 2025)	✗	6	✓	✗	Complexity class
NLCO (this work)	✓	**43**	✓	✓	**Four-layer taxonomy**

Table 1: Comparison of NLCO with existing benchmarks. **CO** denotes whether a benchmark is primarily CO-oriented; **#Prob.** is the number of distinct CO tasks covered; **E2E** (End-to-end) specifies whether LLMs must produce solutions directly from textual input; **Ctx.** (Contextualization) marks whether instances are contextualized in natural-language scenarios (vs. fixed formats); **Taxonomy** reports how the benchmark organizes tasks for problem selection or structural evaluation.

Related Work and Positioning. LLM reasoning benchmarks have rapidly expanded, from math word problems (Cobbe et al., 2021; Uesato et al., 2022) to logic reasoning (Liu et al., 2020; Zhu et al., 2025) and commonsense reasoning (Roewer-Després et al., 2025). Our work targets a more distinct challenge with complex CO tackled by end-to-end reasoning. As summarized in Table 1, related work largely fall into two complementary lines: 1) Optimization-centric suites, such as FrontierCO (Feng et al., 2025), CO-Bench (Sun et al., 2025), and CP-Bench (Michailidis et al., 2025), often assess LLMs as interfaces that produce code for heuristics or solver calls, making it hard to isolate internal end-to-end solving capability, diagnose reasoning failures in a structural manner, and measure reasoning efficiency; 2) Direct-output benchmarks, such as NPHardEval (Fan et al., 2024) and GraphArena (Tang et al., 2025), require solution generation but cover few problem types. By organizing tasks mainly by complexity class, they are limited in structural analysis and broad comparisons. NLCO bridges these gaps: it unifies 40+ CO tasks in contextualized scenarios under a four-layer taxonomy, and forces end-to-end solving, enabling controlled analyses of problem structure-aware reliability and quality-efficiency trade-offs. An extended related work on 1) LLMs for optimization, 2) LLM reasoning evaluations, and 3) optimization benchmark datasets is elaborated in Appendix A.

2 Preliminaries

2.1 Combinatorial Optimization

CO involves finding an optimal configuration of discrete variables that satisfies a set of constraints while minimizing (or maximizing) an objective. Formally, a CO problem can be defined as:

$$\min_{x \in \mathcal{X}} f(x) \quad \text{s.t.} \quad x \in \mathcal{F}, \tag{1}$$

where $\mathcal{X}$ is the discrete search space, $f(x)$ denotes the objective function, and $\mathcal{F} \subseteq \mathcal{X}$ encodes the feasible set satisfying all problem-specific constraints.

2.2 Problem Statement

In the context of LLM-based reasoning, we consider CO problems expressed entirely in natural language. Each instance consists of:

Input Description (D): A natural-language prompt specifying a CO problem description (i.e., an instance), explicitly or implicitly implying the variables, constraints, and optimization objectives, with an example illustrated in Figure 1(a).

Underlying Formal Model ($P = \langle \mathcal{X}, \mathcal{F}, f \rangle$): A hidden mathematical formulation associated with D. This model is used only for solution checking and scoring, and is never given to the LLM.

Solution Representation (S): A structured solution x^* is defined as the one that is feasible ($x^* \in \mathcal{F}$) and optimal for $f(x)$. It serves as the reference label for the problem instance.

Given an instance, the LLM receives only the text and output a solution $\hat{x}$, which is supposed to align with the formal constraints of P. Let $\text{Eval}(\cdot)$ be a function that measures both feasibility violations and solution optimality. At a high level, the reasoning objective can be written as $\hat{x} = \arg\min_{x \in \mathcal{X}} \mathbb{E}_{\text{prompt}(D)}[\text{Eval}(x, \mathcal{F}, f)]$, where we assume infeasibility violation and objective function are minimized.

3 NLCO Benchmark

NLCO benchmark features the first large-scale dataset for end-to-end natural language CO reasoning. More than 40 CO problems are organized under a novel taxonomy that captures diverse decision domains, constraint structures, and objective classes, as shown in Figure 1(b). For each problem, NLCO provides instances at three difficulty levels. Detailed criteria for difficulty levels of each problem are provided in Table 5.

3.1 Benchmark Construction

NLCO is constructed in four steps, including 1) Taxonomy Design (Section 3.1.1): we propose a four-layer taxonomy as a unified design space for diverse CO tasks; 2) Task Selection (Section 3.1.2): we specify a set of diverse CO problems according to taxonomy, covering heterogeneous variable domains, constraint families, and objective classes; 3) Data Generation (Section 3.1.3): we synthesize CO instances and contextualize them in natural language; 4) Evaluation Process (Section 3.2): we specify solution checking protocols and metrics to gauge solution quality and LLM reasoning ability.

3.1.1 Taxonomy Design

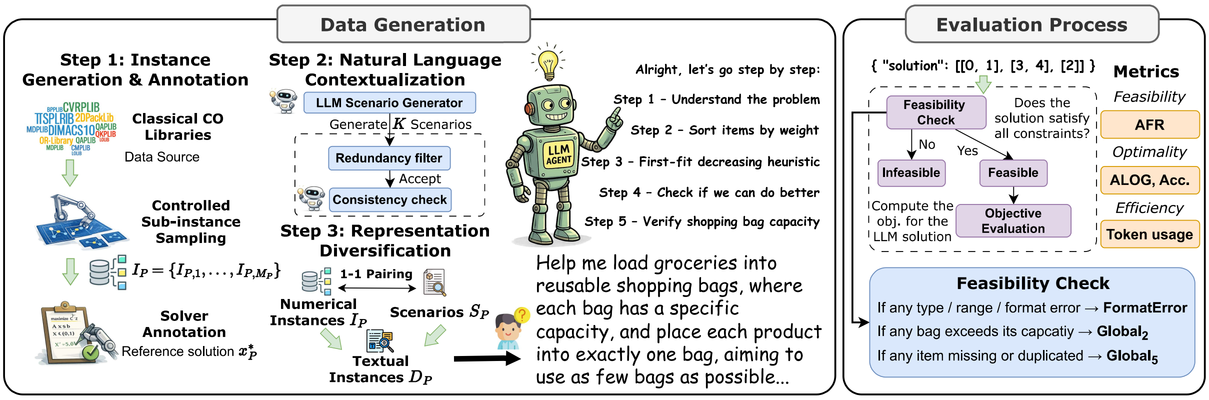
We propose a four-layer taxonomy to guide task selection and diagnose reasoning challenges across different CO problem structures. Our taxonomy, shown in Figure 1(b), builds on concepts from the constraint programming (CP), which has long studied how diverse CO problems can be expressed using a set of reusable constraint templates. We ground our taxonomy in the XCSP^3 specification (Boussemart et al., 2016), a widely used standard for combinatorial constraint modeling. We adopts it as a core principle to group CO tasks into structurally coherent families used to specify comprehensive and diverse aspects for LLM reasoning. For each $P = \langle \mathcal{X}, \mathcal{F}, f \rangle$, we assign a tuple:

$$\tau(P) = (\text{VarSort}(P), \text{Family}(P), \text{Global}(P), \text{ObjClass}(P)), \tag{2}$$

where each axis captures one aspect of the underlying CP model of a CO task, as described below.

Variable Sort (VarSort): the primary decision domain used in a canonical CP formulation, drawn from INT (integer / Boolean variables), SET (set-valued variables), and GRAPH (graph-structured variables). These sorts correspond to algebraic and set-theoretic reasoning, as well as reasoning over relational structures (e.g., paths and connectivity).

Constraint-family Group (Family): the dominant structural family governing feasibility, such



Alright, let's go step by step:

Step 1 - Understand the problem

Step 2 - Sort items by weight

Step 3 - First-fit decreasing heuristic

Step 4 - Check if we can do better

Step 5 - Verify shopping bag capacity

Help me load groceries into reusable shopping bags, where each bag has a specific capacity, and place each product into exactly one bag, aiming to use as few bags as possible...

{ "solution": [[0, 1], [3, 4], [2]] }

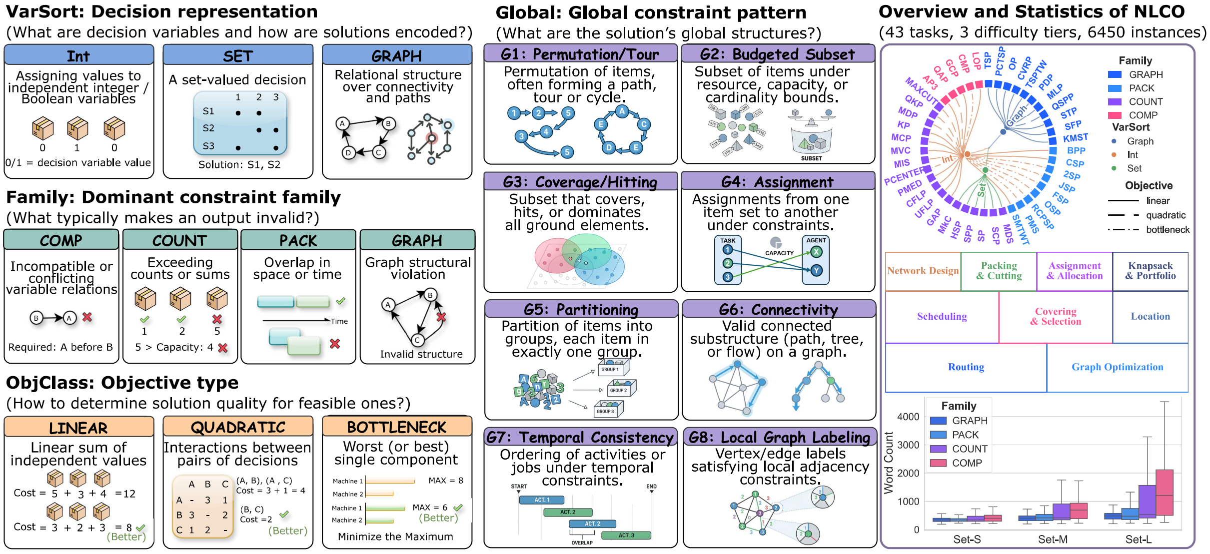
Feasibility Check

If any type / range / format error → **FormatError**

If any bag exceeds its capcatiy → **Global$_2$**

If any item missing or duplicated → **Global$_5$**

(a) NLCO data generation and evaluation process (bin packing shown as an example).



(b) NLCO four-layer taxonomy and dataset overview.

Figure 1: Overview of the NLCO benchmark.

as COMP (comparison constraints such as orderings and inequalities); COUNT (counting and summation constraints); PACK (packing and scheduling constraints); and GRAPH (graph-defined constraints encoded over discrete variables).

Global Constraint Pattern(s) (Global): the global constraints that a valid solution must satisfy. Instead of recording all fine-grained CP global constraints (e.g., `allDifferent`), which reflect solver-level modeling choices but are overly detailed for analyzing LLM behavior, we group them into a small set of patterns used to summarize feasible solutions. Concretely, we use eight patterns, such as Permutation/Tour, Budgeted Subset, etc, which are presented in the middle part of Figure 1(b); each CO task is associated with one or more patterns, indicating which global structural requirements are central to its feasibility. We detail formal definitions and the mapping from classical CP global constraints to patterns in Appendix B.

Global(P) plays a dual role in NLCO. On the modeling side, it groups standard CP global constraints into shared structural types (e.g., mapping `circuit` and `path` to permutation/tour), keeping a direct link to canonical formulations. On the evaluation side, it determines pattern-specific feasibility checks for LLM outputs. For the bin packing example in Figure 1(a), we parse the output as a nested list and flag infeasibility if the solution is malformed, exceeds any bin's capacity, or misses/duplicates items. This pattern-aware validation enables fine-grained diagnosis. In particular, by recording which patterns are violated, we can pinpoint global structures (e.g., partitioning or connectivity) that LLMs most often fail to satisfy.

Objective Class (ObjClass): the cost or utility functions, grouped into linear, quadratic, and bottleneck (min–max / max–min) objectives.

We stress that the taxonomy is a coarse but interpretable abstraction rather than a complete classification of CO problems. Details of the taxonomy and tasks under $\tau(\cdot)$ are provided in Appendix B.

3.1.2 Task Selection

In the absence of a universally accepted "gold-standard" task set, we treat $\tau(\cdot)$ as a design space and instantiate the task suite to ensure broad coverage of decision domains and constraint structures.

Decision Domain Coverage. We ensure that all three variable types in VarSort are integrated into different CO tasks, so that evaluation can be conducted across heterogeneous decision domains, comparing their effects in LLM reasoning.

Constraint Coverage. Meanwhile, we target a wide range of CO tasks with various constraints. Within each constraint family (COMP, COUNT, PACK, GRAPH), we include multiple CO tasks whose CP formulations pertain to different global constraint patterns. On top of that, we can still specify tasks by their applications (routing, scheduling, etc.). Such taxonomy-based task selections favorably organize the task suite, so as to readily diagnose LLM reasoning from distinct dimensions (variable type, constraint family, etc.)

Difficulty and LLM Solvability. Since complexity of NP-hard CO often exponentially grows against problem sizes, we offer a natural scalability axis: increasing the problem size expands the combinatorial space and thus yields harder reasoning challenges. We scale each problem into three tiers: Set-S, Set-M, and Set-L, based on its size-related parameters and structural factors (e.g., number of nodes, graph density). Intuitively, Set-S instances are small enough so that the solution space could be sufficiently explored; Set-M typically requires heuristic reasoning or stochastic exploration for performance enhancement; Set-L further enlarges the combinatorial solution space, making feasibility and efficiency by reasoning more intractable. We generate 50 instances per tier for each CO task.

With the above criteria, NLCO covers 43 tasks, 3 difficulty tiers, and 6450 instances, spanning domains of network design, packing, cutting, assignment, allocation, knapsack, portfolio, scheduling, covering, selection, location, routing, and graph optimization problems, as shown in the right panel of Figure 1(b). The word count for textual instances ranges from 190 to 10022, indicating difficulties of language understanding and reasoning. More taxonomy details are offered in Table 4 of Appendix B.

3.1.3 Data Generation

As shown in Figure 1(a), the data generation process encompasses three steps, including 1) instance generation and annotation, 2) natural-language contextualization, and 3) representation diversification.

Instance Generation and Solver Annotation. Instead of synthesizing CO instances using specific data distributions (e.g., uniform distributions) (Kool et al., 2018; Fan et al., 2024; Hua et al., 2026), we construct our dataset by extracting and adapting instances from well-established optimization libraries (e.g., TSPLIB (Reinelt, 1991)) that are sourced from realistic decision-making scenarios. As such, we aim to obtain a set of numerical instances $\mathcal{I}_P = \{I_{P,1}, I_{P,2}, \ldots, I_{P,M_P}\}$ for each CO task P. CO instances in libraries are often defined at scales (e.g., with hundreds of nodes) beyond the feasibility of LLM reasoning (e.g.,exceeding the model's context budget or inducing an intractably large search space). Hence, we generate sub-instances $I_{P,j}$ by controlled sampling (e.g., randomly sampling 20 nodes from a 200-node graph). This empowers different levels of tractability for end-to-end reasoning but also preserves the characteristics of the original CO tasks.

For each generated instance $I_{P,j} \in \mathcal{I}_P$, we utilize a domain-specific solver to obtain its (near-)optimal solution $x^*_{P,j}$. This solution is stored as the reference label associated with a task $P = \langle \mathcal{X}, \mathcal{F}, f \rangle$, and is used to assess reasoning results. More details of instance generation and solver annotation for each task are specified in Appendix C.

Natural-language Contextualization. After generating and annotating numerical CO instances, we further contextualize each instance into a natural language description that embeds the problem within real-world decision-making scenarios (e.g., logistics planning or task scheduling). Similar to the work on dialogue generation (Finch and Choi, 2024) and optimization problem generation (Huang et al., 2025a), we begin by generating a set of scenarios through calling an LLM, and employ an iterative generate-filter-verify procedure for better diversity and accuracy, as detailed below.

Given a problem P, our goal is to construct a set of linguistically diverse yet semantically equivalent scenarios $\mathcal{S}_P = \{s_{P,1}, \ldots, s_{P,N_P}\}$, where each scenario gives an everyday narrative of P while preserving its objective and constraints. To this end, in each iteration, an LLM proposes K candidate scenarios for P, which are subsequently verified

and filtered to enforce correctness and diversity.
Semantic Redundancy Filtering. To promote scenario diversity and avoid near-duplicate scenarios, we apply an embedding-based novelty criterion. We encode all textual scenarios using a SentenceTransformer (`all-MiniLM-L6-v2`), which produces an embedding $\mathbf{e}(\cdot)$ for each scenario. For each new candidate c, we compute its cosine similarity with each previously accepted scenario $s \in \mathcal{S}_P$. If $\max_{s \in \mathcal{S}_P} \cos\big(\mathbf{e}(c), \mathbf{e}(s)\big) > 0.7$, the candidate is discarded as a semantic paraphrase.
Problem-Consistency Verification. The semantically diverse candidate is further verified by an LLM-based checker conditioned on the formal specification of P. By doing so, we ensure that the scenario still preserves the original problem structure without altering the CO task.

The generate-filter-verify procedure continues till $|\mathcal{S}_P| \geq M_P$. Finally, for each numerical instance $I_{P,j} \in \mathcal{I}_P$, we associate it with exactly one scenario $s_{P,j} \in \mathcal{S}_P$, resulting in a textual instance $d_{P,j} = (I_{P,j}, s_{P,j})$. This procedure ensures that all instances are contextualized by coherent and semantically valid natural-language descriptions.
Representation Diversification. To reflect variation in user input, we randomly sample $d_{P,j}$ to create instance variants that change only data presentation while preserving the same optimization instance. Concretely, we apply simple transformations to $d_{P,j}$: 1) **Format changes**, presenting the same data inputs in natural language, CSV, JSON, or Markdown table; and 2) **Indexing changes**, using different ways to label variables, such as number-based indices or alphabetical labels. These diverse inputs further involve influence of different formats in LLM reasoning, which is seldom considered in current benchmarks. Further implementation details, used prompts, and examples are provided in Appendix D.

3.2 Evaluation Process

As shown in Figure 1(a), we evaluate LLM solutions along 3 dimensions: feasibility, optimality, and inference efficiency. Given a textual instance $d_{P,i}$ of a CO problem $P = \langle \mathcal{X}, \mathcal{F}, f \rangle$, the model produces a textual solution, which we parse into a numerical decision $\hat{x}_{P,i} \in \mathcal{X}$. We assess feasibility by checking $\hat{x}_{P,i}$ against $\mathcal{F}$ with global-pattern validation. Unparseable outputs or constraint violations are deemed infeasible. For feasible ones, we compute their objectives $f(\hat{x}_{P,i})$ for comparison.
Core Metrics. We report four primary metrics. 1) **Average feasibility rate (AFR)**: fraction of instances, for which the LLM output is feasible (i.e., it meets all constraints). 2) **Average accuracy (Acc.)**: fraction of instances, for which the output is feasible and achieves the reference objective, i.e., $f(\hat{x}_{P,i}) = f^*_{P,i}$ (within a small tolerance). 3) **Average log optimality gap (ALOG)**: among feasible instances, we measure the gap of objective values to the optimum. The gaps are averaged in log-space to reduce the influence of heavy-tailed errors. We compute the relative gap $\mathrm{Gap}_{P,i} = \frac{f(\hat{x}_{P,i}) - f^*_{P,i}}{\max\{|f^*_{P,i}|, \epsilon\}}$ and report $\log(1 + \mathrm{Gap})$ averaged over feasible instances. 4) **Token usage**: we track average output tokens as a simple proxy for LLM reasoning cost.

4 Experiments

4.1 Experimental Setup

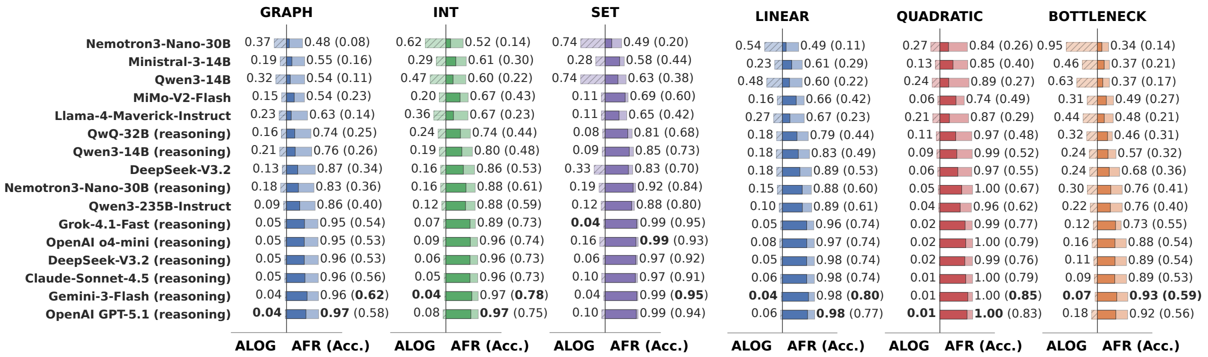
We use NLCO to evaluate eight **open-weighted LLMs:** Qwen3-14B (Yang et al., 2025a), Ministral-3-14B (Mistral AI, 2025), NVIDIA Nemotron3-Nano-30B-A3B (NVIDIA, 2025), QwQ-32B (Qwen Team, 2025), Llama-4-Maverick (17Bx128E)-Instruct (Meta AI, 2025), Qwen3-235B-A22B-Instruct-2507 (Yang et al., 2025a), Xiaomi MiMo-V2-Flash (Xiaomi, 2025), DeepSeek-V3.2 (Liu et al., 2025), and five **proprietary LLMs:** OpenAI o4-mini (OpenAI, 2025b), GPT-5.1 (OpenAI, 2025a), Anthropic Claude-Sonnet-4.5 (Anthropic, 2025), xAI Grok-4.1-Fast (xAI, 2025), and Gemini-3-Flash (Google DeepMind, 2025). We cover both standard chat LLMs and explicit reasoning LLMs. For models that offer separate the two variants, we use notation '(reasoning)' to denote its reasoning version. Detailed experiment configurations are provided in Appendix E.

4.2 Overall Performance

To answer **RQ1** on **end-to-end capability**, Table 2 reports NLCO results, evaluating feasibility (AFR), optimality (Acc. and ALOG), and inference cost (tok.). Unlike prior reports that LLMs rarely solve CO (Fan et al., 2024; Tang et al., 2025), we find that frontier LLMs can often translate textual specifications into globally feasible decisions: on Set-S, the best models reach near-saturated feasibility (up to **98.9%** AFR) and achieve high exact-optimality rates (up to **91.6%** Acc.), demonstrating strong end-to-end reasoning on small CO instances. However, this capability does not scale smoothly with problem size. As instances grow from Set-S to Set-L, feasibility and optimality deteriorate sharply

Model	Set-S				Set-M				Set-L			
	AFR ↑	Acc. ↑	ALOG ↓	tok. ↓	AFR ↑	Acc. ↑	ALOG ↓	tok. ↓	AFR ↑	Acc. ↑	ALOG ↓	tok. ↓
Open-weight Models												
Qwen3-14B	70.4	38.7	0.1734	2492	58.1	17.5	0.5359	3053	48.9	7.8	0.7255	3328
Ministral-3-14B	72.0	47.7	0.0979	10326	58.7	25.4	0.2656	14325	46.6	12.0	0.4292	15643
Nemotron3-Nano-30B	64.2	26.8	0.2627	2208	50.4	9.6	0.6302	3663	37.4	4.0	0.8820	4283
Llama-4-Maverick-Instruct	78.8	42.8	0.1467	1471	65.1	19.0	0.3111	1730	52.7	7.9	0.4314	2009
Qwen3-235B-Instruct	95.3	80.1	0.0248	6701	87.6	54.9	0.0933	8848	79.7	36.3	0.2207	9988
DeepSeek-V3.2	93.4	72.1	0.0427	2613	86.1	47.7	0.1466	3510	78.5	31.1	0.3361	4019
MiMo-V2-Flash	75.0	58.4	0.0711	13733	62.8	37.4	0.1732	21751	54.3	24.7	0.2763	25501
Qwen3-14B (reasoning)	90.7	69.9	0.0427	8929	79.1	42.4	0.1834	11576	69.4	25.5	0.3298	12351
Nemotron3-Nano-30B (reasoning)	94.2	80.1	0.0360	11708	88.2	54.2	0.1328	21797	79.7	38.2	0.3355	26449
QwQ-32B (reasoning)	89.9	68.5	0.0533	11484	73.8	39.1	0.1998	14541	61.1	19.6	0.3517	15472
DeepSeek-V3.2 (reasoning)	98.7	88.2	0.0071	9262	96.8	69.1	0.0403	15484	92.9	54.1	0.1238	19403
Proprietary Models												
Grok-4.1-Fast (reasoning)	98.3	88.7	0.0067	6582	93.6	70.9	0.0312	12389	83.7	53.4	0.1441	16641
Claude-Sonnet-4.5 (reasoning)	98.7	89.0	0.0061	10755	96.7	70.4	0.0393	16802	93.2	53.0	0.1282	20590
OpenAI o4-mini (reasoning)	98.9	89.3	0.0058	9544	96.5	70.7	0.0471	19118	92.5	53.1	0.2102	23426
OpenAI GPT-5.1 (reasoning)	98.9	90.9	0.0056	6573	98.0	72.8	0.0462	12823	95.5	57.2	0.1716	16646
Gemini-3-Flash (reasoning)	98.8	91.6	0.0068	16993	97.5	76.8	0.0203	24325	95.4	60.9	0.0899	27700

Table 2: Performance on NLCO across three difficulty tiers. AFR and Acc. are reported in percentages (%). **Legend:** 🟩: Best results achieved by open-weight LLMs; 🟦: Best results achieved by proprietary LLMs.



(a)

(b)

Figure 2: Aggregated performance across (a) VarSort and (b) ObjClass dimensions in NLCO taxonomy.

across models (AFR/Acc. decrease and ALOG increases), while token usage increases consistently. This coupled trend indicates that larger combinatorial search spaces still pose a substantial challenge for LLMs: models tend to expend more inference-time computation yet are increasingly unable to maintain feasibility and reach optimal solutions.

Smaller LLMs generally struggle to solve CO problems reliably even on easier tiers: Nemotron3-Nano-30B attains only 64.2%/50.4%/37.4% AFR on Set-S/M/L with low Acc. and relatively large ALOG. Similarly, Qwen3-14B drops from 70.4% AFR on Set-S to 48.9% on Set-L, while Acc. falls from 38.7% to 7.8%. The results suggest that limited model capacity substantially constrains CO reasoning, especially as the combinatorial space grows. We further evaluate LLMs by performance profiles (Dolan and Moré, 2002), and report average per-task results, as presented in Appendix F.

4.3 Aggregated Analysis

To answer **RQ2** on **structure-aware reliability**, we aggregate results across taxonomy dimensions. As shown in Figure 2, model performance exhibits heterogeneity across variable types (VarSort) and objective classes (ObjClass), indicating that failures are not uniform but tied to problem structure.

Impact of VarSort and ObjClass. Across frontier LLMs, SET tasks are the easiest, with high feasibility and markedly higher exact-optimality than other sorts. In comparison, GRAPH problems are more error-prone, likely because they require maintaining global relational consistency rather than making relatively local, enumerable choices. Objective form also induces performance swings. Linear and quadratic objectives are reliable: frontier LLMs reach high Acc. (e.g., GPT-5.1: 77% and 83%). Notably, bottleneck tasks remain challenging, e.g., GPT-5.1 drops to 56% Acc. The consistent degra-

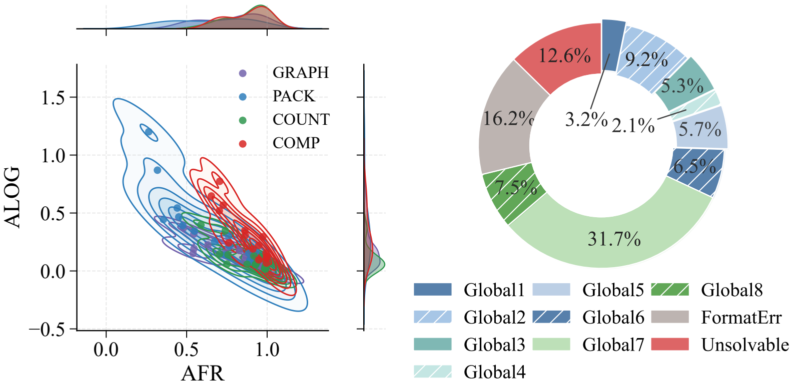


Figure 3: AFR–ALOG joint distribution by family (left) and infeasibility mode distribution (right). *FormatError* is raised when the solution cannot be parsed with missing fields or non-finite numbers. *Unsolvable* denotes instances that remain unsolved under the token limit.

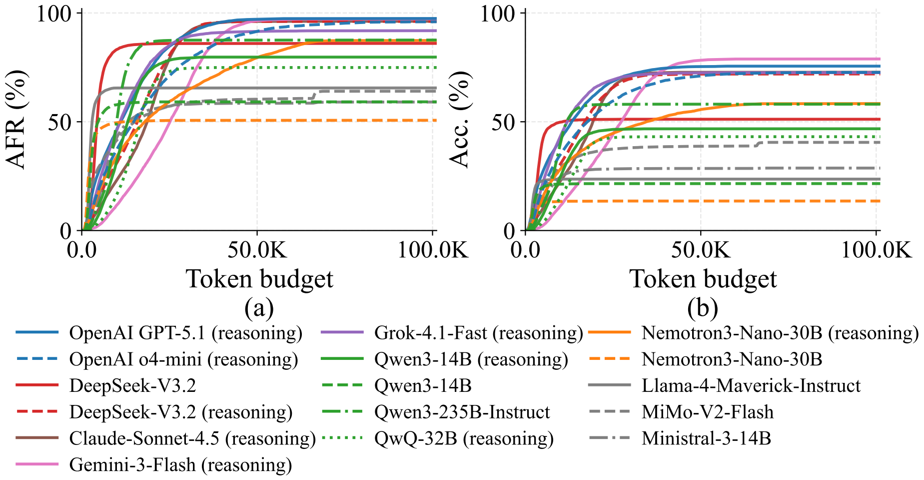


Figure 4: LLM data profiles across all difficulty tiers. (a) AFR vs. Token budget; (b) Acc. vs. Token budget.

dation suggests a characteristic failure mode: when the objective is dominated by a worst-case component, local errors or incomplete global verification can disproportionately harm optimality, and models struggle to identify the optimal solution reliably.

Constraint Families and Global Patterns. To better diagnose the drivers of infeasibility and suboptimality, we pool outputs from all LLMs and break down performance by Family and global patterns. Figure 3 (left) plots the joint distribution of AFR and ALOG across families, revealing structure dependence: PACK exhibits the largest variance and the heaviest failure tail (low AFR with high ALOG), whereas COUNT and GRAPH concentrate near high AFR and low ALOG. Figure 3 (right) further attributes infeasible cases to specific patterns, which are highly concentrated in a few groups, most notably Global$_7$ (31.7%) and Global$_2$ (9.2%), followed by mid-sized patterns (e.g., Global$_{8/6/5}$). Finally, Unsolvable accounts for 12.6% of failures, indicating that a non-trivial portion of instances exceed the inference budget and remain unsolved under token limits. Per-model infeasibility mode is analyzed in Appendix F.

4.4 Reasoning Cost Analysis

From Table 2, we can find that the high-performing LLMs (e.g., Gemini-3-Flash) achieve a high AFR and relatively low ALOG, but at the cost of extremely high token usage. To better answer **RQ3** for the **quality-efficiency trade-off** of reasoning, we introduce data profile (Moré and Wild, 2009), a standard benchmarking tool in optimization.

More precisely, for each instance i and model m, let $t_{i,m}$ be the token cost of the model output. We fix a success criterion (i.e., the solution is feasible or optimal). We define the fraction of instances solved within a budget B tokens as $d_m(B) =$

Type	Model	Acc.	tok.
Non-reasoning	Qwen3-14b	39.58	2276
	Ministral-14b	40.60	2511
	Nemotron3-Nano-30b	35.26	3515
	Llama-4-Maverick-Instruct	53.07	1430
	Qwen3-235b-instruct	69.26	2209
	DeepSeek-V3.2	66.98	1828
	Mimo-V2-Flash	41.72	7482
Reasoning	Qwen3-14b (reasoning)	51.21	11816
	Nemotron3-Nano-30b (reasoning)	41.44	6655
	QwQ-32b (reasoning)	54.70	11921
	DeepSeek-V3.2 (reasoning)	76.19	4689
	Grok-4.1-fast (reasoning)	67.26	5462
	Claude-sonnet-4.5 (reasoning)	80.90	5798
	o4-mini (reasoning)	56.47	7555
	GPT-5.1 (reasoning)	76.88	4343
	Gemini-3-flash (reasoning)	**83.77**	20651

Table 3: Performance comparison of different LLMs on solver-calling code generation.

$\frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\big[t_{i,m} \leq B \ \wedge\ \text{success}(i,m)\big]$ (N is the number of instances). In our settings, $d_m(B)$ for feasibility and optimality naturally translates to two metrics used in the paper, i.e., AFR and Acc. Plotting $d_m(B)$ against B yields a curve where higher is better: a higher curve means the model solves more instances under the same token budget, and a curve that rises earlier indicates better efficiency.

As shown in Figure 4, feasibility increases rapidly with budget and then saturates, suggesting that extra tokens mainly help with constraint satisfaction and self-checking up to a "knee" region, after which returns diminish. Optimality increases more gradually and plateaus well below feasibility, indicating that finding a valid solution is easier than reliably improving it to high quality. Across models, reasoning-enabled variants typically gain faster at small budgets (higher token efficiency) and reach higher plateaus. We also see that some models (e.g., DeepSeek-V3.2) attain high feasibility with modest budgets but convert additional tokens into little optimality improvement, implying that longer generations often boost validity more than objective value. Finally, although Gemini-3-Flash achieves the best overall results, it consistently uses more tokens than other top-tier models (e.g., DeepSeek), highlighting the need for more efficient reasoning.

We further analyze inference-time compute scaling via Best-of-N sampling in Appendix F.

4.5 Solver-calling Capability Benchmarking

Since solver code generation for optimization is important in real-world scenarios, we conduct experiments by prompting LLMs to write Gurobi code to solve the NLCO problems, and the results on Set-L are summarized in Table 3, where we use an objective tolerance of 1e-3 to judge the accuracy.

While the ability to write solver-calling code significantly improves results compared to the end-to-end setting, models continue to exhibit notable heterogeneity in both accuracy and reasoning efficiency. More specifically, models with exceptionally strong coding foundations like Claude-Sonnet-4.5 and Gemini-3-Flash perform significantly better. Meanwhile, even with the assistance of world-class solvers, model performance on NLCO is still not saturated (i.e., the best-performing LLM achieves 83.77% accuracy). This confirms that NLCO is not only a test of internal logic but also a suitable benchmarking framework for the coding and mathematical modeling capabilities of LLMs.

5 Conclusion

We introduce NLCO benchmark for assessing LLM reasoning on a broad range of CO tasks. NLCO organizes 43 problems under a four-layer taxonomy and tests LLMs as end-to-end solvers that produce decisions directly from natural-language scenarios. Our empirical study suggests that current LLMs often succeed on small instances, but achieving reliable optimality on larger ones remains challenging, especially for graph-structured and bottleneck objectives, and extra tokens do not fully close the gap. Thus, LLM-based decision support for real CO workflows will require methods that better integrate reasoning over global constraints and deliver more efficient inference. We hope NLCO serves as a reusable testbed to witness more reliable and efficient LLM reasoning for language-based CO.

Limitations

Our work has several limitations that point to important directions for future research:

Scope of tasks. Although NLCO covers over 40 CO problems and spans multiple families, it remains a finite sample of the broader landscape of constrained and combinatorial decision-making. Moreover, to keep end-to-end inference tractable for current LLMs, instances are scaled to moderate sizes. As a result, observed performance may not extrapolate to substantially larger instances where classical solvers typically become essential.

Robustness of natural-language specification. NLCO instances are embedded into contextualized textual scenarios, but they are still generated from underlying formal problems with controlled templates and parameterizations. This design improves coverage and reproducibility, yet it may not fully capture the diversity and ambiguity of real-world specifications (e.g., missing constraints, underspecified objectives, or ambiguous requirements). Evaluating robustness to noisy and user-authored problem statements remains an open direction.

Generalization across prompting/agentic strategies. We intentionally evaluate models under a specific prompting and inference protocol. The design choice aims to isolate and assess the inherent reasoning ability of LLMs on CO problems, rather than performance gains introduced by task-specific prompting strategies, tool use, self-verification, or agentic workflows. While alternative prompting strategies or tool-augmented setups may further improve performance, our results reflect controlled end-to-end reasoning ability under a standardized evaluation setting. Building on NLCO, future work can investigate how prompt engineering, tools, or agentic solvers interact with LLM reasoning to address CO problems.

References

Henrik Abgaryan, Tristan Cazenave, and Ararat Harutyunyan. 2025. Starjob: Dataset for llm-driven job shop scheduling. *arXiv preprint arXiv:2503.01877*.

Ali Ahmaditeshnizi, Wenzhi Gao, and Madeleine Udell. 2024. OptiMUS: Scalable optimization modeling with (mi) lp solvers and large language models. In *International Conference on Machine Learning*, pages 577–596.

Majd Alkayyal, Simon Malberg, and Georg Groh. 2026. An llm-based decision support system for strategic decision-making. In *Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track and Demo Track*, pages 460–464.

Anthropic. 2025. Claude sonnet 4.5 system card. Technical report, Anthropic.

David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. 2014. *Benchmarking for Graph Clustering and Partitioning*, pages 73–82. Springer New York, New York, NY.

Egon Balas and Matthew J Saltzman. 1991. An algorithm for the three-index assignment problem. *Operations Research*, 39(1):150–161.

John E Beasley. 1985. A note on solving large p-median problems. *European Journal of Operational Research*, 21(2):270–273.

John E Beasley. 1990. Or-library: distributing test problems by electronic mail. *Journal of the operational research society*, 41(11):1069–1072.

Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. 2010. Global constraint catalog.

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

Chandra Bhagavatula, Ronan Le Bras, Chaitanya Malaviya, Keisuke Sakaguchi, Ari Holtzman, Hannah Rashkin, Doug Downey, Wen tau Yih, and Yejin Choi. 2020. Abductive commonsense reasoning. In *International Conference on Learning Representations*.

Jieyi Bi, Yining Ma, Jianan Zhou, Wen Song, Zhiguang Cao, Yaoxin Wu, and Jie Zhang. 2024. Learning to handle complex constraints for vehicle routing problems. *Advances in Neural Information Processing Systems*, 37:93479–93509.

Ole Bilde and Jakob Krarup. 1977. Sharp lower bounds and efficient algorithms for the simple plant location problem. In *Annals of Discrete Mathematics*, volume 1, pages 79–97. Elsevier.

Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. 2016. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *arXiv:1611.03398*.

Rainer E Burkard, Stefan E Karisch, and Franz Rendl. 1997. Qaplib–a quadratic assignment problem library. *Journal of Global optimization*, 10(4):391–403.

Jianghao Chen, Zhenlin Wei, Zhenjiang Ren, Ziyong Li, and Jiajun Zhang. 2025. LR^{2}Bench: Evaluating long-chain reflective reasoning capabilities of large language models via constraint satisfaction problems. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 6006–6032.

Jingxiao Chen, Ziqin Gong, Lvda Chen, Minghuan Liu, Jun Wang, Yong Yu, and Weinan Zhang. 2024. Looking ahead to avoid being late: Solving hard-constrained traveling salesman problem. In *Proceedings of the 2024 6th International Conference on Distributed Artificial Intelligences*, pages 1–12.

Paul C Chu and John E Beasley. 1997. A genetic algorithm for the generalised assignment problem. *Computers & Operations Research*, 24(1):17–23.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, and Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Francesca Da Ros, Michael Soprano, Luca Di Gaspero, and Kevin Roitero. 2025. Large language models for combinatorial optimization: A systematic review. *arXiv preprint arXiv:2507.03637*.

Maxence Delorme, Manuel Iori, and Silvano Martello. 2018. Bpplib: a library for bin packing and cutting stock problems. *Optimization Letters*, 12(2):235–250.

Elizabeth D Dolan and Jorge J Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213.

Emanuel Falkenauer. 1996. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics*, 2(1):5–30.

Lizhou Fan, Wenyue Hua, Lingyao Li, Haoyang Ling, and Yongfeng Zhang. 2024. NPHardEval: Dynamic benchmark on reasoning ability of large language models via complexity classes. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4092–4114.

Shengyu Feng, Weiwei Sun, Shanda Li, Ameet Talwalkar, and Yiming Yang. 2025. A comprehensive evaluation of contemporary ML-based solvers for combinatorial optimization. *arXiv:2505.16952*.

James D. Finch and Jinho D. Choi. 2024. Diverse and effective synthetic data generation for adaptable zero-shot dialogue state tracking. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 12527–12544.

Google DeepMind. 2025. Gemini 3 flash - model card. Technical report, Google.

Shuhan Guo, Nan Yin, James Kwok, and Quanming Yao. 2025. Nested-refinement metamorphosis: Reflective evolution for efficient optimization of networking problems. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 17398–17429.

Martin Hoefer. 2006. Ufllib: A collection of benchmark instances for the uncapacitated facility location problem. `https://resources.mpiinf.mpg.de/departments/d1/projects/benchmarks/UflLib/`.

Kaj Holmberg, Mikael Rönnqvist, and Di Yuan. 1999. An exact algorithm for the capacitated facility location problems with single sourcing. *European Journal of Operational Research*, 113(3):544–559.

Chuanbo Hua, Federico Berto, Zhikai Zhao, Jiwoo Son, Changhyun Kwon, and Jinkyoo Park. 2026. USPR: Learning a unified solver for profiled routing. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

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

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. 2025b. 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.

Zhen Huang, Zengzhi Wang, Shijie Xia, Xuefeng Li, Haoyang Zou, Ruijie Xu, Run-Ze Fan, Lyumanshan Ye, Ethan Chern, and Yixin Ye, et al. 2024. Olympicarena: Benchmarking multi-discipline cognitive reasoning for superintelligent ai. *Advances in Neural Information Processing Systems*, 37:19209–19253.

Zixiao Huang, Lifeng Guo, Wenhao Li, Junjie Sheng, Chuyun Shen, Haosheng Chen, Bo Jin, Changhong Lu, and Xiangfeng Wang. 2025c. Graphthought: Graph combinatorial optimization with thought generation. *arXiv preprint arXiv:2502.11607*.

Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. 2025. ALE-bench: A benchmark for long-horizon objective-driven algorithm engineering. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Manuel Iori, Vinícius Loti de Lima, Silvano Martello, and Michele Monaci. 2022. 2DPackLib: a two-dimensional cutting and packing library. *Optimization Letters*, 16(2):471–480.

Xia Jiang, Yaoxin Wu, Minshuo Li, Zhiguang Cao, and Zhang Yingqian. 2025a. Large language models as end-to-end combinatorial optimization solvers. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Xia Jiang, Yaoxin Wu, Chenhao Zhang, and Yingqian Zhang. 2025b. DRoC: Elevating large language models for complex vehicle routing via decomposed retrieval of constraints. In *The Thirteenth International Conference on Learning Representations*.

Raka Jovanovic. 2023. Qkplib. `https://data.mendeley.com/datasets/82pxy6yv49/1`.

Serdar Kadioglu, Yuri Malitsky, and Meinolf Sellmann. 2012. Non-model-based search guidance for set partitioning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 493–498.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30.

Gorka Kobeaga, María Merino, and Jose A. Lozano. 2018. An efficient evolutionary algorithm for the orienteering problem. *Computers & Operations Research*, 90:42–59.

Wouter Kool, Herke van Hoof, and Max Welling. 2018. Attention, learn to solve routing problems! In *International Conference on Learning Representations*.

Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A. Ehinger. 2025. Planning-driven programming: A large language model programming workflow. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Markus Leitner, Ivana Ljubić, Michael Luipersbeck, Martin Prossegger, and Maximilian Resch. 2014. New real-world instances for the steiner tree problem in graphs. Technical report, Institute for Operations Research, University of Vienna.

Haibing Li and Andrew Lim. 2001. A metaheuristic for the pickup and delivery problem with time windows. In *Proceedings 13th IEEE international conference on tools with artificial intelligence*, pages 160–167.

Yunxin Li, Baotian Hu, Haoyuan Shi, Wei Wang, Longyue Wang, and Min Zhang. 2024. Visiongraph: leveraging large multimodal models for graph theory problems in visual context. In *International Conference on Machine Learning*, pages 27903–27919.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, and Chen Dong, et al. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

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

Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. 2020. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. *arXiv preprint arXiv:2007.08124*.

Rafael Martí, Abraham Duarte, Anna Martínez-Gavara, and Jesús Sánchez-Oro. 2021. The mdplib 2.0 library of benchmark instances for diversity problems. `https://www.uv.es/rmarti/paper/mdp.html`.

Rafael Martí, Juan J Pantrigo, Abraham Duarte, and Eduardo G Pardo. 2013. Branch and bound for the cutwidth minimization problem. *Computers & Operations Research*, 40(1):137–149.

Rafael Martí, Gerhard Reinelt, and Abraham Duarte. 2012. A benchmark library and a comparison of heuristic methods for the linear ordering problem. *Computational optimization and applications*, 51(3):1297–1317.

Meta AI. 2025. The llama 4 model card. Technical report, Meta. Model variant: Llama-4-Maverick.

Kostis Michailidis, Dimos Tsouros, and Tias Guns. 2025. CP-Bench: Evaluating large language models for constraint modelling. In *28th European Conference on Artificial Intelligence*.

Seyed Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2025. GSM-symbolic: Understanding the limitations of mathematical reasoning in large language models. In *The Thirteenth International Conference on Learning Representations*.

Mistral AI. 2025. Ministral 3 14b instruct 2512 (model card). `https://huggingface.co/mistralai/Ministral-3-14B-Instruct-2512`.

Philipp Mondorf and Barbara Plank. 2024. Beyond accuracy: Evaluating the reasoning behavior of large language models-a survey. In *First Conference on Language Modeling*.

Jorge J Moré and Stefan M Wild. 2009. Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*, 20(1):172–191.

Mahdi Mostajabdaveh, Timothy Tin Long Yu, Samarendra Chandan Bindu Dash, Rindra Ramamonjison, Jabo Serge Byusa, Giuseppe Carenini, Zirui Zhou, and Yong Zhang. 2025. Evaluating llm reasoning in the operations research domain with orqa. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24902–24910.

Shiwen Ni, Guhong Chen, Shuaimin Li, Xuanang Chen, Siyi Li, Bingli Wang, Qiyao Wang, Xingjian Wang, Yifan Zhang, and Liyang Fan, et al. 2025. A survey on large language model benchmarks. *arXiv preprint arXiv:2508.15361*.

NVIDIA. 2025. Nemotron 3 nano: Open, efficient mixture-of-experts hybrid mamba-transformer model for agentic reasoning. Technical report.

OpenAI. 2025a. Gpt-5.1 model documentation.

OpenAI. 2025b. Openai o3 and o4-mini system card. Technical report, OpenAI.

Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. 2013. A constraint solver based on abstract domains. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013*, pages 434–454.

David Pisinger. 2005. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284.

Aske Plaat, Annie Wong, Suzan Verberne, Joost Broekens, Niki Van Stein, and Thomas Bäck. 2025. Multi-step reasoning with large language models, a survey. *ACM Computing Surveys*, 58(6).

Qwen Team. 2025. QwQ-32B: Embracing the power of reinforcement learning. `https://qwenlm.github.io/blog/qwq-32b/`.

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

Varshini Reddy, Rik Koncel-Kedziorski, Viet Dac Lai, Michael Krumdick, Charles Lovering, and Chris Tanner. 2024. Docfinqa: A long-context financial reasoning dataset. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 445–458.

Jean-Charles Régin. 2010. Global constraints: A survey. In *Hybrid optimization: The ten years of CPAIOR*, pages 63–134. Springer.

Gerhard Reinelt. 1991. TSPLIB—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384.

François Roewer-Després, Jinyue Feng, Zining Zhu, and Frank Rudzicz. 2025. ACCORD: Closing the commonsense measurability gap. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3799–3829.

Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of constraint programming*. Elsevier.

Borzou Rostami, André Chassein, Michael Hopf, Davide Frey, Christoph Buchheim, Federico Malucelli, and Marc Goerigk. 2018. The quadratic shortest path problem: complexity, approximability, and solution methods. *European Journal of Operational Research*, 268(2):473–485.

Ankit Satpute, Noah Gießing, André Greiner-Petter, Moritz Schubotz, Olaf Teschke, Akiko Aizawa, and Bela Gipp. 2024. Can llms master math? investigating large language models on math stack exchange. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, page 2316–2320.

Zijian Shao, Jiancan Wu, Weijian Chen, and Xiang Wang. 2025. Personal travel solver: A preference-driven LLM-solver system for travel planning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Freda Shi, Mirac Suzgun, Markus Freitag, Xuezhi Wang, Suraj Srivats, Soroush Vosoughi, Hyung Won Chung, Yi Tay, Sebastian Ruder, Denny Zhou, Dipanjan Das, and Jason Wei. 2023. Language models are multilingual chain-of-thought reasoners. In *The Eleventh International Conference on Learning Representations*.

Nishad Singhi, Hritik Bansal, Arian Hosseini, Aditya Grover, Kai-Wei Chang, Marcus Rohrbach, and Anna Rohrbach. 2025. When to solve, when to verify: Compute-optimal problem solving and generative verification for LLM reasoning. In *Second Conference on Language Modeling*.

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

Weiwei Sun, Shengyu Feng, Shanda Li, and Yiming Yang. 2025. CO-Bench: Benchmarking language model agents in algorithm search for combinatorial optimization. *arXiv:2504.04310*.

Eric Taillard. 1993. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285.

Shunji Tanaka and Mituhiko Araki. 2008. A branch-and-bound algorithm with lagrangian relaxation to minimize total tardiness on identical parallel machines. *International Journal of Production Economics*, 113(1):446–458.

Jianheng Tang, Qifan Zhang, Yuhan Li, Nuo Chen, and Jia Li. 2025. Grapharena: Evaluating and exploring large language models on graph computation. In *The Thirteenth International Conference on Learning Representations*.

Theodore Tsiligirides. 1984. Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, 35(9):797–809.

Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. 2017. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. 2022. Solving math word problems with process-and outcome-based feedback. *arXiv:2211.14275*.

Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. 2023. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems*, 36:30840–30861.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. 2024. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. In *Advances in Neural Information Processing Systems*.

Chengwei Wei, Bin Wang, Jung-jae Kim, Guimei Liu, and Nancy Chen. 2025. Coinmath: Harnessing the power of coding instruction for math llm. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 786–797.

Niels A. Wouda, Leon Lan, and Wouter Kool. 2024. PyVRP: a high-performance VRP solver package. *INFORMS Journal on Computing*, 36(4):943–955.

Jian Wu, Linyi Yang, Zhen Wang, Manabu Okumura, and Yue Zhang. 2025. CofCA: A STEP-WISE counterfactual multi-hop QA benchmark. In *The Thirteenth International Conference on Learning Representations*.

xAI. 2025. Grok 4.1 model card.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. 2024. Chain-of-experts: When LLMs meet complex operations research problems. In *The Twelfth International Conference on Learning Representations*.

LLM-Core Xiaomi. 2025. Mimo-v2-flash technical report.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, and Chenxu Lv, et al. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.

Lei Yang, Renren Jin, Ling Shi, Jianxiang Peng, Yue Chen, and Deyi Xiong. 2025b. Probench: Benchmarking large language models in competitive programming. *arXiv preprint arXiv:2502.20868*.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. 2025c. Optibench meets resocratic: Measure and improve LLMs for optimization modeling. In *The Thirteenth International Conference on Learning Representations*.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. 2024. Reevo: Large language models as hyper-heuristics with reflective evolution. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Jihai Zhang, Wei Wang, Siyan Guo, Li Wang, Fangquan Lin, Cheng Yang, and Wotao Yin. 2024. Solving general natural-language-description optimization problems with large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pages 483–490.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations*.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. 2025. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. In *International Conference on Machine Learning*.

Ruiwen Zhou, Wenyue Hua, Liangming Pan, Sitao Cheng, Xiaobao Wu, En Yu, and William Yang Wang. 2025. RuleArena: A benchmark for rule-guided reasoning with LLMs in real-world scenarios. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 550–572.

Qin Zhu, Fei Huang, Runyu Peng, Keming Lu, Bowen Yu, Qinyuan Cheng, Xipeng Qiu, Xuanjing Huang, and Junyang Lin. 2025. Autologi: Automated generation of logic puzzles for evaluating reasoning abilities of large language models. *arXiv preprint arXiv:2502.16906*.

Yuxin Zuo, Shang Qu, Yifei Li, Zhang-Ren Chen, Xuekai Zhu, Ermo Hua, Kaiyan Zhang, Ning Ding, and Bowen Zhou. 2025. Medxpertqa: Benchmarking expert-level medical reasoning and understanding. In *International Conference on Machine Learning*.

A Extended Related Work

A.1 LLM for Optimization

LLMs have attracted growing interest in optimization and operations research (OR) due to their reasoning and code-generation abilities (Da Ros et al., 2025). Unlike symbolic or algebraic reasoning, CO requires understanding not only numerical relationships but also the logical structure and interplay of constraints, as well as trade-offs induced by objective functions. This makes CO more challenging than typical math problem solving (Satpute et al., 2024; Wei et al., 2025), where reasoning often reduces to deriving closed-form expressions or performing arithmetic transformations.

Existing work applying LLMs to optimization largely follows two paradigms: *code-synthesis* and *end-to-end* approaches. In the code-synthesis paradigm, LLMs generate or refine heuristic algorithms (Ye et al., 2024; Liu et al., 2024; Guo et al., 2025; Zheng et al., 2025) or construct model formulations and solver calls for classical optimization engines (Zhang et al., 2024; Ahmaditeshnizi et al., 2024; Xiao et al., 2024; Huang et al., 2025a; Jiang et al., 2025b). While often effective, this approach fails to isolate the models' intrinsic reasoning ability because it is tightly coupled to pretrained knowledge and dependent on external solvers.

The second paradigm is end-to-end solving, where LLMs directly interpret natural-language descriptions, devise feasible solutions, or iteratively refine them without solver assistance (Yang et al., 2024; Huang et al., 2025c; Abgaryan et al., 2025; Jiang et al., 2025a). Such an approach has the potential to minimize user expertise and lower the barrier for applying CO in practice, while more faithfully reflecting the models' own reasoning ability. However, existing work typically focuses on a small number of problem families or bespoke datasets, and there is still no systematic assessment of end-to-end reasoning across heterogeneous CO problems presented purely in natural language.

A.2 LLM Reasoning Benchmarking

Reasoning is widely regarded as a cornerstone of higher intelligence and is crucial for understanding the practical potential of LLMs (Ni et al., 2025). LLM reasoning benchmarking has evolved significantly over the past several years, transitioning from simple grade-school math evaluation (Cobbe et al., 2021; Uesato et al., 2022; Shi et al., 2023) to multifaceted assessment frameworks, such as Olympic competition problems (Huang et al., 2024), commonsense reasoning (Bhagavatula et al., 2020; Roewer-Després et al., 2025), logical reasoning (Liu et al., 2020; Zhu et al., 2025), and domain-specific reasoning (Reddy et al., 2024; Zuo et al., 2025; Yang et al., 2025b; Mostajabdaveh et al., 2025). These benchmarks have played an important role in charting the rapid progress of LLMs across different reasoning regimes and domains.

With the rapid development of reasoning techniques, emergent LLMs now attain strong performance on many widely used benchmarks. For example, the Qwen-3 model (Yang et al., 2025a) achieves a score of 94.39 on GSM8K (Cobbe et al., 2021), 98.0 on MATH-500 (Lightman et al., 2023), and 89.0 on AutoLogi (Zhu et al., 2025), suggesting that current benchmarks are increasingly saturated at the high end. Meanwhile, most of them focus on numerical puzzles, formal logic, or multiple-choice QA, and therefore weakly capture the challenges posed by CO: reasoning under constraints, combinatorial decision spaces, and explicit objective functions, which are instantiated in our benchmark.

A.3 Optimization Benchmarks

Traditional optimization benchmark datasets, such as TSPLIB (Reinelt, 1991), have long been used as standards for evaluating algorithmic performance. However, these datasets are not well-suited for benchmarking LLMs, as they lack natural-language context and often operate at scales that exceed the tractable reasoning capacity of current models.

More relevant to NLCO are language-based benchmarks that focus on optimization or graph reasoning tasks. As summarized in Table 1, early efforts largely evaluated the code-generation capabilities of LLMs, reflecting a period when model reasoning was comparatively weak and reliable problem solving required translating problems into executable programs (Ramamonjison et al., 2022; Xiao et al., 2024; Huang et al., 2025b; Yang et al., 2025c; Michailidis et al., 2025). When end-to-end problem solving was considered at all, benchmarks typically centered on relatively simple, polynomial-time tasks (Wang et al., 2023; Fan et al., 2024; Li et al., 2024; Tang et al., 2025), as the limited capacity of early LLMs made NP-hard CO prohibitively difficult to evaluate meaningfully.

By contrast, NLCO is built around canonical CO problems that are all framed in natural language, enabling a faithful assessment of LLM reasoning on constraint-driven decision-making problems.

#	CO Problem	Var. Sort	Family	Global pattern(s)	Canonical global constraint(s)	Objective
***Constraint-family group:* Graph**						
1	Traveling Salesman Problem (TSP)	Graph	Graph	Global$_1$	`circuit`	linear
2	Prize-Collecting TSP (PCTSP)	Graph	Graph	Global$_{1,2}$	`circuit`, `sum`	linear
3	Orienteering Problem (OP)	Graph	Graph	Global$_{1,2}$	`circuit`, `sum`	linear
4	Capacitated VRP (CVRP)	Graph	Graph	Global$_{1,2}$	`circuit`, `knapsack`	linear
5	TSP with Time Windows (TSPTW)	Graph	Graph	Global$_{1,7}$	`circuit`, `cumulative`	linear
6	Pickup-and-Delivery Problem (PDP)	Graph	Graph	Global$_{1,7}$	`circuit`, `precedence`, `cumulative`	linear
7	Minimum Latency Problem (MLP)	Graph	Graph	Global$_1$	`circuit`	linear
8	Quadratic Shortest Path Problem (QSPP)	Graph	Graph	Global$_1$	`path`	quadratic
9	Steiner Tree Problem (STP)	Graph	Graph	Global$_6$	`tree`	linear
10	Steiner Forest Problem (SFP)	Graph	Graph	Global$_6$	`tree`, `nTrees`	linear
11	k-Minimum Spanning Tree (KMST)	Graph	Graph	Global$_{6,2}$	`tree`, `cardinality`	linear
***Constraint-family group:* Pack**						
12	Bin Packing Problem (BPP)	Int	Pack	Global$_{2,5}$	`binPacking`	linear
13	Cutting Stock Problem (CSP)	Int	Pack	Global$_2$	`knapsack`, `sum`	linear
14	2D Strip Packing (2SP)	Int	Pack	Global$_7$	`noOverlap`, `maximum`	bottleneck
15	Job-Shop Scheduling Problem (JSP)	Int	Pack	Global$_7$	`noOverlap`, `precedence`	bottleneck
16	Flow-Shop Scheduling Problem (FSP)	Int	Pack	Global$_7$	`noOverlap`, `precedence`	bottleneck
17	Open-Shop Scheduling Problem (OSP)	Int	Pack	Global$_7$	`noOverlap`	bottleneck
18	RCPSP (makespan)	Int	Pack	Global$_{7,2}$	`cumulative`, `precedence`	bottleneck
19	Parallel Machines $P \parallel T_{\max}$ (PMS)	Int	Pack	Global$_7$	`noOverlap`	bottleneck
20	Single-Machine Total Weighted Tardiness	Int	Pack	Global$_7$	`noOverlap`	linear
***Constraint-family group:* Count**						
21	Minimum Dominating Set (MDS)	Set	Count	Global$_3$	`coverage`	linear
22	Set Cover Problem (SCP)	Set	Count	Global$_{3,2}$	`coverage`, `cardinality`	linear
23	Set Packing (SP)	Set	Count	Global$_{2,5}$	`disjoint`, `cardinality`	linear
24	Set Partitioning Problem (SPP)	Set	Count	Global$_5$	`partition`	linear
25	Hitting Set Problem (HSP)	Set	Count	Global$_3$	`coverage`	linear
26	Max k-Coverage (MkC)	Set	Count	Global$_2$	`coverage`, `cardinality`	linear
27	Generalized Assignment Problem (GAP)	Int	Count	Global$_{4,2}$	`sum`, `channel`	linear
28	Uncapacitated Facility Location (UFLP)	Int	Count	Global$_4$	`sum`, `channel`	linear
29	Capacitated Facility Location (CFLP)	Int	Count	Global$_{4,2}$	`sum`, `channel`	linear
30	p-Median (PMED)	Int	Count	Global$_{4,2}$	`cardinality`, `sum`	linear
31	p-Center (PCENTER)	Int	Count	Global$_{4,2}$	`cardinality`, `maximum`	bottleneck
32	Maximum Independent Set (MIS)	Int	Count	Global$_8$	`sum`†	linear
33	Minimum Vertex Cover (MVC)	Int	Count	Global$_8$	`sum`†	linear
34	Maximum Clique Problem (MCP)	Int	Count	Global$_8$	`sum`‡	linear
35	Knapsack Problem (KP)	Int	Count	Global$_2$	`knapsack`, `sum`	linear
36	Maximum Diversity Problem (MDP)	Int	Count	Global$_2$	`cardinality`	quadratic
37	Quadratic Knapsack Problem (QKP)	Int	Count	Global$_2$	`knapsack`	quadratic
38	Maximum Cut Problem (MAXCUT)	Int	Count	Global$_8$	`weightedSum`, `xor`	quadratic
***Constraint-family group:* Comp**						
39	Three-Index Assignment (AP3)	Int	Comp	Global$_4$	`allDifferent`, `channel`	linear
40	Quadratic Assignment Problem (QAP)	Int	Comp	Global$_4$	`allDifferent`	quadratic
41	Graph Coloring Problem (GCP) (min colors)	Int	Comp	Global$_{5,8}$	`neq`†, `nValues`	linear
42	Cutwidth Minimization Problem (CMP)	Int	Comp	Global$_1$	`allDifferent`, `maximum`	bottleneck
43	Linear Ordering Problem (LOP)	Int	Comp	Global$_1$	`allDifferent`	linear

†Applied over edges of the input graph (e.g., one of the incident endpoints must satisfy the corresponding constraint).
‡Applied over non-edges of the input graph (e.g., at most one endpoint of each non-edge may be selected).

Table 4: Taxonomy of NLCO problems. The column "Global pattern(s)" lists the set of global constraint patterns Global$_k$ associated with the canonical CP model, where Global$_1$–Global$_8$ denote permutation / tour, budgeted subset, coverage / hitting, assignment, partitioning, connectivity, temporal consistency, and local graph labeling, respectively. The column "Canonical global constraint(s)" records the canonical CP global constraints.

B NLCO Benchmark Taxonomy

Section 3 introduces the NLCO taxonomy $\tau(P) = (\text{VarSort}(P), \text{Family}(P), \text{Global}(P), \text{ObjClass}(P))$ that we use throughout the task selection and evaluation processes. This appendix provides additional details on how we connect this taxonomy to established standard and how we assign labels to each dimension in a deterministic way.

Our starting point is the XCSP^3 specification (Boussemart et al., 2016), which offers a compact language and catalogue of global constraints for representing constrained optimization and satisfaction problems, and thus applicable to CO problems. For each CO task in NLCO, we consider a canonical Constraint Programming (CP) model (either directly from the XCSP^3 catalogue or from standard CP/OR formulations) and attach the four labels in $\tau(P)$ to this underlying model rather than to any particular natural-language description.

Variable sort. As discussed in Section 3, we group the primary decision domains into INT (including BOOL), SET, and GRAPH. In practice, many CO problems admit multiple equivalent encodings. For example, Set Cover can be written either with set variables or with $0/1$ indicator variables, while TSP can be written either with graph-structured variables or with integer indices and auxiliary constraints. To make labels reproducible, we adopt a *native-sort-first* convention: whenever there exists a widely used formulation that naturally uses SET or GRAPH variables, we assign that sort as primary, even if there also exists a purely INT-based encoding. Concretely:

- Knapsack and related packing problems are labeled as INT; the canonical models rely on integer or Boolean decision variables only. Likewise, classic vertex-selection problems defined on aa graph (e.g., MIS/MVC/MC) are labeled as INT rather than GRAPH: the primary decisions are $0/1$ labels $x_v \in \{0, 1\}$ for vertices. In this taxonomy, GRAPH is reserved for problems whose decision objects are graph-structured (e.g., tours, paths) and are most naturally expressed with graph-scoped global constraints (e.g., `circuit` or `tree`).
- Set Cover and Set Partitioning are labeled as SET; their natural CP models use set variables with cardinality and inclusion constraints.
- Routing and network-design problems such as TSP and Steiner Tree are labeled as GRAPH, reflecting that their structure is most naturally expressed over graphs.

From the LLM's perspective, this convention explicitly distinguishes the primary mode of reasoning required for success: algebraic/logical (INT), set-based (SET), or relational/graph-based (GRAPH).

Constraint-family group. The Family(P) label identifies the dominant structural family controlling feasibility, using the top-level groupings of XCSP^3: COMP, COUNT, PACK, and GRAPH (see Section 3). Intuitively, each family corresponds to a prototypical reasoning pattern:

- COMP: satisfying precedence or inequality relations, e.g., assignment with order constraints.
- COUNT: respecting capacities, budgets, or coverage via sums of decision variables, e.g., knapsack or simple covering problems.
- PACK: arranging items in time or space without overlap, e.g., job-shop scheduling or bin packing in space or time.
- GRAPH: satisfying constraints scoped to graph structure, e.g., tours/cycles, matchings, or flows/cuts.

In ambiguous cases, the family label is tied to which constraints would typically drive propagation in a constraint satisfaction judgement (see "primary constraint rule" below).

Global constraint pattern(s). Within the chosen family, Global(P) abstracts away from classical global constraints and instead records the set of global patterns that a valid solution must satisfy. It provides a more fine-grained dimension for analyzing feasibility than the constraint-family group.

Concretely, we start from a canonical CP model and its associated global constraints (e.g., `circuit`, `binPacking`, `noOverlap`, `allDifferent`, `coverage`, `partition`, `networkFlow`) and map them into a small vocabulary of solution-structure templates, such as permutation, subset, and coverage. Intuitively, each pattern describes what a correct answer 'looks like' and what global relation over the decision variables must hold, as it would be stated in the text and checked by facility-checking scripts.

This design is motivated by two considerations. First, it keeps the taxonomy anchored in standard CP practice, as the patterns are derived from well-studied global constraints (Rossi et al., 2006; Régin, 2010), while providing a coarser, human-interpretable layer that better matches the structure of LLM-generated solutions. Second, it reduces the granularity of the global-constraint space to a handful of recurring reasoning motifs, since there are more than 400 global constraints recorded in existing catalogues (Beldiceanu et al., 2010). Our approach makes it possible to aggregate and compare performance across superficially different CO tasks that share the same underlying patterns.

Formally, we describe the eight global constraint patterns below:

Global$_1$ (permutation / tour). Solutions are permutations of a given set of items, which typically represent a path or a cycle (e.g., visiting all cities exactly once). Canonical CP constraints include `circuit`, `path`, `nPaths`, and `allDifferent` when used to enforce a permutation structure.

Global$_2$ (budgeted subset). Solutions select a subset of items subject to resource, capacity, or cardinality limits (e.g., weight or number of chosen items cannot exceed a bound). This pattern arises from `sum`, `weightedSum`, `knapsack`, `binPacking`, `cumulative`, and `cardinality`-type constraints when used as budget or capacity constraints.

Global$_3$ (coverage / hitting). Solutions ensure that a ground set is covered or dominated by the selected items or sets (e.g., every element is covered at least once, or every vertex is dominated by a chosen vertex). Canonical constraints include `coverage` and related hitting/dominating-set encodings.

Global$_4$ (assignment). Solutions assign items from one set to items in another (e.g., tasks to machines, facilities to customers), often under one-to-one or many-to-one constraints. This pattern is induced by `allDifferent`, `channel`, `element`, and similar linking constraints in assignment models.

Global$_5$ (partitioning). Solutions partition a ground set into disjoint groups or classes so that each element belongs to exactly one group (e.g., partitioning into sets or color classes). Canonical constraints include `partition`, `disjoint`, and `nValues` when used to encode partitions.

Global$_6$ (connectivity). Solutions form connected substructures such as paths, trees, or flows on a graph (e.g., connecting a set of terminals). Typical constraints are `path`, `tree`, `nTrees`, `networkFlow`, and `circuit` when used primarily to enforce connectivity.

Global$_7$ (temporal consistency). Solutions assign times or orders to activities under temporal, precedence, or non-overlap constraints (e.g., scheduling jobs on machines without conflict). Canonical constraints include `noOverlap`, `cumulative`, `precedence`, `sequence`, and `regular` in scheduling and rostering models.

Global$_8$ (local graph labeling). Solutions label vertices (or edges) of a graph so that local adjacency constraints are satisfied (e.g., adjacent vertices receive different colors, edges in a cut cross the partition). This pattern is induced by constraints such as `neq` on edges, `xor`, and `weightedSum` over incident variables in cut or independent-set formulations.

For each problem P, Global(P) is a subset of $\{\text{Global}_1, \ldots, \text{Global}_8\}$ obtained by mapping the propagation-critical global constraints in a canonical CP model to these patterns. We report the resulting patterns as Global$_{k_1,k_2,\ldots}$ in Table 4.

For example:

- TSP and related routing problems are mapped to the *permutation / tour* pattern (i.e., Global$_1$): a solution is a permutation of cities forming a Hamiltonian tour (captured in CP by `circuit`); prize-collecting and capacitated variants additionally carry *budgeted subset* patterns (i.e., Global$_2$) via sum/knapsack-style constraints.
- Set Cover, Hitting Set, and Minimum Dominating Set are mapped to the *coverage / hitting* pattern (i.e., Global$_3$): a solution is a subset (or family of subsets) that covers a ground set (captured by `coverage`-style constraints); some variants also include a *budgeted subset* pattern when the size or cost of the chosen sets is bounded.

- Set Partitioning and closely related models fall under *partitioning* (i.e., Global_5): the ground set is split into disjoint groups, typically represented by `partition`, `disjoint`, or `nValues`.

- Job-Shop Scheduling, Flow-Shop, RCPSP, and Parallel-Machine scheduling are mapped to *temporal consistency* (i.e., Global_7): a solution assigns start times to tasks so that resources do not overlap and precedence relations are respected (captured by `noOverlap`, `cumulative`, and `precedence`); resource-constrained variants further add a *subset under budget or capacity* pattern on renewable resources.

- Graph Coloring, Max-Cut, and independent-set-type problems are mapped to *local graph labeling* (i.e., Global_8): each vertex receives a label or selection bit, and local adjacency constraints such as inequality or cut edges (`neq`, `xor`) must be satisfied.

We report $\text{Global}(P)$ as an ordered list and refer to the first element as the primary pattern. Anchoring evaluation to these (possibly multiple) patterns lets us interpret violations at the level of reasoning modes: a candidate solution can be reported as "violating the permutation/tour pattern" (not a valid tour) or "violating the coverage pattern" (uncovered elements remain), rather than simply "infeasible," while Table 4 still records the underlying global constraints for CP-oriented analysis.

Objective class. The $\text{ObjClass}(P)$ label groups objectives into three classes: linear, quadratic, and bottleneck (min–max / max–min). This is a coarse but useful abstraction:

- Linear objectives cover a large portion of classical optimization tasks (e.g., linear assignment, shortest paths, knapsack).

- Quadratic objectives arise in interaction-heavy problems such as Quadratic Assignment or Max-Cut, where costs depend on pairs of decisions.

- Bottleneck objectives appear in scheduling, where the goal is to minimize a worst-case quantity such as makespan.

Grouping tasks along this axis allows us to distinguish, for example, whether a model that performs well on linear assignment-type problems generalizes to quadratic interaction objectives.

Scope of the taxonomy. The taxonomy is not meant as an exhaustive classification of all CO problems. Instead, it is a principled coarsening of the XCSP^3 landscape tailored to the tasks in NLCO, chosen to balance granularity and interpretability when aggregating results across related tasks. Based on the framework above, Table 4 summarizes the taxonomic labels for all tasks in NLCO and can be used as a reference when interpreting the taxonomy-aware results.

Problem	**Data Source**	**Utilized Solver**	**Instance scale (S/M/L)**
Travelling Salesman Problem (TSP)	TSPLIB (Reinelt, 1991)	LKH3	5–10 / 11–15 / 16–25 nodes
Prize-Collecting TSP (PCTSP)	TSPLIB (Reinelt, 1991)	Gurobi	5–10 / 11–15 / 16–25 nodes
Orienteering Problem (OP)	Data in (Tsiligirides, 1984)	GA (Kobeaga et al., 2018), Gurobi	5–10 / 11–15 / 16–25 nodes
Capacitated Vehicle Routing (CVRP)	CVRPLIB Set-X (Uchoa et al., 2017)	HGS (Wouda et al., 2024)	5–10 / 11–15 / 16–25 nodes (incl. depot)
TSP with Time Windows (TSPTW)	TSPLIB (Reinelt, 1991)	LKH3	5–10 / 11–15 / 16–25 nodes
Pickup and Delivery Problem (PDP)	Li & Lim benchmark (Li and Lim, 2001)	Gurobi	3–5 / 6–8 / 9–14 requests
Minimum Latency Problem (MLP)	TSPLIB (Reinelt, 1991)	Gurobi	5–10 / 11–15 / 16–25 nodes
Quadratic Shortest Path Problem (QSPP)	Synthetic data following (Rostami et al., 2018)	Gurobi	5–10 / 11–15 / 16–25 nodes
Steiner Tree Problem (STP)	Data in (Leitner et al., 2014)	Gurobi	10–15 / 16–20 / 21–30 vertices
Steiner Forest Problem (SFP)	Data in (Leitner et al., 2014)	Gurobi	10–15 / 16–20 / 21–30 vertices
k-Minimum Spanning Tree (KMST)	Data in (Leitner et al., 2014)	Gurobi	10–15 / 16–20 / 21–30 vertices
Bin Packing Problem (BPP)	Data in (Falkenauer, 1996)	Gurobi	5–10 / 11–15 / 16–25 items
Cutting Stock Problem (CSP)	BPPLIB (Delorme et al., 2018)	Gurobi	5–10 / 11–15 / 16–25 items
2D Strip Packing (2SP)	2DPackLib (Iori et al., 2022)	Gurobi	5–10 / 11–15 / 16–25 items
Job-Shop Scheduling Problem (JSP)	Taillard Generator (Taillard, 1993)	CP-SAT	2–5 / 6–8 / 9–12 jobs 2–3 / 3–5 / 5–6 machines
Flow-Shop Scheduling Problem (FSP)	Taillard Generator (Taillard, 1993)	CP-SAT	2–5 / 6–8 / 9–12 jobs 2–3 / 3–5 / 5–6 machines
Open-Shop Scheduling Problem (OSP)	Taillard Generator (Taillard, 1993)	CP-SAT	2–5 / 6–8 / 9–12 jobs 2–3 / 3–5 / 5–6 machines
Resource-Constrained Project Scheduling Problem (RCPSP)	Constraint benchmarking tools suite (Pelleau et al., 2013)	CP-SAT	4–10 / 11–15 / 16–21 tasks
Parallel Machines Scheduling (PMS)	Tanaka and Araki benchmark (Tanaka and Araki, 2008)	CP-SAT	5–10 / 11–15 / 16–25 jobs 2–2 / 2–3 / 2–5 machines
Single-Machine Total Weighted Tardiness (SMTWT)	OR-library (Beasley, 1990)	CP-SAT	5–10 / 11–20 / 21–30 jobs
Minimum Dominating Set (MDS)	DIMACS10 Street Network (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 vertices
Set Covering Problem (SCP)	DIMACS10 Road Network (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 elements
Set Packing Problem (SP)	DIMACS10 Road Networks (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 elements
Set Partitioning Problem (SPP)	Synthetic Benchmark Instances (following (Kadioglu et al., 2012))	Gurobi	8–12 / 13–18 / 19–25 elements
Hitting Set Problem (HSP)	DIMACS10 Road Networks (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 elements
Max k-Coverage (MkC)	DIMACS10 Road Networks (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 elements
Generalized Assignment Problem (GAP)	OR-Library (Chu and Beasley, 1997)	Gurobi	5–10 / 11–15 / 16–25 items
Uncapacitated Facility Location (UFLP)	UflLIB (Hoefer, 2006)	Gurobi	5–8 / 9–14 / 15–20 customers
Capacitated Facility Location (CFLP)	Data in (Holmberg et al., 1999)	Gurobi	5–8 / 9–14 / 15–20 customers
p-Median Facility Location (PMED)	OR-Library (Beasley, 1985)	Gurobi	5–8 / 9–14 / 15–20 customers
p-Center Facility Location (PCENTER)	OR-Library (Beasley, 1985)	Gurobi	5–8 / 9–14 / 15–20 customers
Maximum Independent Set (MIS)	DIMACS10 Redistricting Graph (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 vertices
Minimum Vertex Cover (MVC)	DIMACS10 Redistricting Graph (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 vertices
Maximum Clique Problem (MCP)	DIMACS10 Citation / Co-author Networks (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 vertices
Knapsack Problem (KP)	Synthetic (uncorrelated) data following (Pisinger, 2005)	Gurobi	5–10 / 11–20 / 21–30 items
Maximum Diversity Problem (MDP)	MDPLIB (Martí et al., 2021)	Gurobi	8–12 / 13–18 / 19–25 items

Continued on next page

Table 5 – *Continued from previous page*

Problem	**Data Source**	**Utilized Solver**	**Instance scale (S/M/L)**
Quadratic Knapsack Problem (QKP)	QKPLIB (Jovanovic, 2023)	Gurobi	5–10 / 11–15 / 16–25 items
Maximum Cut Problem (MAXCUT)	DIMACS10 Redistricting Graph (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 vertices
Three-Index Assignment Problem (AP3)	Data in (Balas and Saltzman, 1991)	Gurobi	3–5 / 6–8 / 9–12 items
Quadratic Assignment Problem (QAP)	QAPLIB (Burkard et al., 1997)	Gurobi	3–5 / 6–8 / 9–12 items
Graph Coloring Problem (GCP)	DIMACS10 Citation / Co-author Networks (Bader et al., 2014)	Gurobi	8–12 / 13–18 / 19–25 vertices
Cutwidth Minimization Problem (CMP)	CMPLIB (Martí et al., 2013)	Gurobi	8–12 / 13–18 / 19–25 vertices
Linear Ordering Problem (LOP)	LOLIB (Martí et al., 2012)	Gurobi	5–10 / 11–15 / 16–25 items

Table 5: Summary of CO problem data sources, utilized solvers, and instance scales. The last column reports the small (S), medium (M), and large (L) instance size ranges used in NLCO, expressed in terms of the main size parameter for the corresponding problem (e.g., number of nodes, customers, items, vertices, or elements).

C Problem Details

This section introduces all the NLCO tasks, along with their data generation processes. Table 5 provides an overview on the data source, utilized solver, and instance scale of each task, while a more detailed statistic for set problems (i.e., SCP, MkC, SP, HSP, and SPP) and graph problems (i.e., MIS, MVC, MCP, GCP, MDS, and MAXCUT) are summarized in Table 9 and Table 10, respectively.

For all problems, the utilized solver (e.g., Gurobi) automatically determines whether a feasible solution exists for the sampled instance, thereby ensuring the feasibility of the generated data.

C.1 Constraint-family Group: GRAPH

C.1.1 Traveling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) seeks a minimum-length tour that visits each of the n nodes exactly once and returns to the starting point.

Instance Creation. For each TSP instance, node coordinates are extracted by sampling n points from a randomly selected TSP instance from TSPLIB (Reinelt, 1991). The sampled coordinates are then normalized to an integer grid within $[0, 100]^2$. Each resulting instance is solved by LKH-3 for the reference solution.

C.1.2 Prize-Collecting TSP (PCTSP)

Instance Creation. PCTSP instances are generated using the same TSPLIB-based coordinate sampling procedure as standard TSP instances. The coordinates are then normalized to the integer grid $[0, 100]^2$, with node 1 designated as the depot.

Node prizes are sampled i.i.d. from the discrete uniform distribution $\{0, \ldots, 100\}$, with the depot assigned a prize of zero. The minimum required collected prize is set to $25n$, corresponding to half of the expected total prize under our discrete $\{0, \ldots, 100\}$ prize distribution.

Penalties are scaled using the size–dependent tour-length estimates L_n (we sample 1,000 instances for each setting and calculate tour length for the corrsponding TSP), i.e., $L_n \in \{334,\ 388,\ 451\}$ for instance ranges $n \in [5, 10]$, $n \in [11, 20]$, and $n > 21$, respectively. Each penalty is sampled independently as $\beta_i \sim \mathcal{U}\big(0,\ \frac{3L_n}{2n}\big)$, following the settings in Kool et al. (2018).

C.1.3 Orienteering Problem (OP)

The Orienteering Problem (OP) is a classical NP-hard routing problem, which chooses a subset of n nodes to visit within a limited travel-length budget T_n in order to maximize the total collected prize.

Instance Creation. For each OP instance, node coordinates are sampled from the instances used by Tsiligirides (1984) and normalized to an integer grid within $[0, 100]^2$. The instance is modeled as a closed tour that starts and ends at the origin (the first point), which is with zero prize. A travel-length budget T_n is then sampled uniformly at random from $\{5, ..., 30\}$. When generating an instance of size n, the depot is always retained and the remaining $n - 1$ nodes are uniformly sub-sampled without replacement from the non-depot nodes.

For instances with $n \leq 20$, an exact Gurobi Mixed Integer Linear Programming (MILP) formulation is used; otherwise, a Genetic Algorithm (GA)-based solver (Kobeaga et al., 2018) is applied. The recorded objective value is the collected prize of the returned tour.

C.1.4 Capacitated Vehicle Routing Problem (CVRP)

The Capacitated Vehicle Routing Problem (CVRP) seeks a set of routes that serve all customers exactly once using a fleet of identical vehicles with limited capacity. Each route starts and ends at a single depot, and the sum of customer demands on any route must not exceed the vehicle capacity Q. Under Euclidean distances, the objective is to minimize the total travel distance across all routes.

Instance Creation. For each CVRP instance, a base problem is sampled uniformly at random from a CVRPLIB instance (Uchoa et al., 2017), where each entry provides depot and customer coordinates, node demands, and a vehicle capacity. Given a target size n (counting the depot), the extractor always retains the depot (node 0) and sub-samples $n - 1$ customers uniformly without replacement from the remaining nodes. The resulting coordinates are then min–max normalized to $[0, 100]^2$ (independently per coordinate dimension), while demands and capacity are kept from the original instance. The depot index is fixed to 0.

Each instance is solved using a Hybrid Genetic Search (HGS) solver under the original vehicle capacity Q. The solver returns a set of vehicle routes and the corresponding total travel distance. The recorded objective value is this total distance.

C.1.5 Traveling Salesman Problem with Time Windows (TSPTW)

The Traveling Salesman Problem with Time Windows (TSPTW) extends the classical TSP by adding temporal constraints. Each customer node i is associated with a time window $[l_i,\ u_i] \in \mathbb{Z}^+$ and $l_i < u_i$) and the vehicle must arrive within this interval; early arrivals must wait until l_i, while late arrivals are infeasible. The objective is to construct a TSP-like minimum-length tour while respecting all time-window constraints.

Instance Creation. For instance creation, each instance is created by sampling n nodes from instances in TSPLIB and normalizing all coordinates to an integer grid in $[0, 100]^2$. To set a meaningful time scale for sampling time windows, we follow the generation scheme in recent works (Bi et al., 2024; Chen et al., 2024), which requires a size–dependent estimate of the expected TSP tour length L_n. Because the spatial distribution is not uniform, we calibrate these values offline by generating 1000 additional TSP instances per size range, solving them using LKH, and computing the average tour length L_n[3] for instance ranges $n \in [5, 10]$, $n \in [11, 20]$, and $n > 20$, respectively.

For each customer node i, the lower bound of its time window is sampled uniformly from $l_i \sim \mathrm{Unif}[0,\ L_n]$, and the window width is drawn from $W_i = L_n \cdot \mathrm{Unif}[\alpha, \beta]$, with $\alpha = 0.5$ and $\beta = 0.75$, giving the upper time window bound $u_i = l_i + W_i$. The depot is assigned a wide and always–feasible window $[0,\ (2 + \beta)L_n]$ to allow free departure and return. Each instance is then solved with LKH-3 to obtain the (near-)optimal total travel distance.

C.1.6 Pickup and Delivery Problem (PDP)

The Pickup and Delivery Problem (PDP) requires constructing a minimum-distance route that starts and ends at a depot while serving a set of pickup–delivery request pairs. Each pickup must be visited before its corresponding delivery. In this implementation, travel time between any pair of nodes is propotinal to their Euclidean distance, and time windows are included for each node.

Instance Creation. We sample instances from the Li & Lim benchmark, which provides node coordinates, demands, time windows, and pickup–delivery pairs. To create a sub-instance with a target number of requests r, we first sample r pickup–delivery pairs. The depot (node 0) is always included, forming a complete instance of $n = 1 + 2r$ nodes. Finally, we normalize the node coordinates and consistently re-index the retained time windows.

Each generated instance is solved using a Gurobi MILP implementation that minimizes total travel distance. The model enforces a single route from a start depot to an end depot (implemented by duplicating the depot as the end node), eliminates subtours via MTZ-style ordering variables, enforces time-window feasibility via continuous arrival-time variables with big-M propagation, and enforces precedence constraints for every pickup–delivery pair. The targeted objective is to minimize the total travel distance.

C.1.7 Minimum Latency Problem (MLP)

The Minimum Latency Problem (MLP) seeks a tour that visits each of the n nodes exactly once while minimizing the sum of arrival times (total latency) rather than the total tour length. Under Euclidean distances, the latency of a node is the cumulative travel distance from the depot along the tour until the node is first reached, and the MLP objective is the sum of these latencies over all visited nodes.

Instance Creation. For each MLP instance, node coordinates are obtained from the same data sources as TSP, and the data generation also follows the process of TSP.

The exact optimal solution to each resulting instance is solved using a Gurobi-based formulation for MLP (as LKH is tailored to tour-length objectives). The reference tour is computed by considering node with index 0 as the depot, and the recorded objective value is the total latency, i.e., the sum of arrival times along the returned tour.

C.1.8 Quadratic Shortest Path Problem (QSPP)

The Quadratic Shortest Path Problem (QSPP) is defined on a directed graph $G = (V, A)$ with a designated source node s and target node t. Each arc $a \in A$ is associated with a binary decision variable $x_a \in \{0, 1\}$ indicating whether the arc is selected. The objective combines linear and quadratic interaction costs $\min\ x^\top Q x + g^\top x + c$, where $g \in \mathbb{R}^{|A|}$ are linear arc costs, $Q \in \mathbb{R}^{|A|\times|A|}$ is a (possibly dense) quadratic cost matrix, and c is a constant. Feasibility is enforced by one-unit $s \to t$ flow con-

---

[3] Using 1000 sampled instances per size range, the empirical tour lengths are $L_{5\text{–}10} = 334.20$, $L_{11\text{–}20} = 388.07$, $L_{21\text{–}30} = 451.38$, so we set $L_n \in \{334, 388, 451\}$

straints:

$$\sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a = \begin{cases} 1 & v = s, \\ -1 & v = t, \\ 0 & \text{otherwise}, \end{cases}$$

which selects a directed s–t path (potentially with extra cycles in general, though the path is extracted from the selected subgraph).

Instance Creation. To reliably control the instance size, we generate instances directly using the grid families proposed by Rostami et al. (2018). Given a target number of nodes $n = |V|$, we construct one of the following families:

- **Grid1:** a $k \times k$ directed grid with $n = k^2$ nodes. Arcs connect each node to its *right* and *up* neighbors when they exist. The source is the lower-left corner and the target is the upper-right corner.
- **Grid2:** an $n_r \times n_c$ transshipment grid plus explicit source and target, so $n = n_r n_c + 2$. The source connects to all nodes in the first column, all nodes in the last column connect to the target, and internal arcs connect each transshipment node to its *right* and *down* neighbors when they exist. To match a requested n, we set $N = n - 2$ and choose (n_r, n_c) either as a factor pair closest to $\sqrt{N}$ (square) or using a $16 \times (N/16)$ heuristic when divisible (long/wide); otherwise we fall back to $1 \times N$ or $N \times 1$.

For each generated graph, arc indices are assigned in the construction order (`var_index = 0, . . . ,|A|-1`). Linear costs are sampled i.i.d. as integers $g_a \sim \text{Unif}\{1, \ldots, 10\}$. Quadratic costs are sampled as a dense symmetric matrix: for $i \leq j$, $Q_{ij} \sim \text{Unif}\{1, \ldots, 10\}$, and we set $Q_{ji} = Q_{ij}$. The constant term is set to $c = 0$.

Each instance is solved with Gurobi as a (generally nonconvex) binary quadratic program. We enforce the unit-flow constraints above and minimize $x^\top Q x + g^\top x + c$. The recorded solution is the extracted s–t path (found by running BFS in the subgraph induced by arcs with $x_a = 1$), and the recorded objective value is the solver's optimal objective $x^\top Q x + g^\top x + c$.

C.1.9 Steiner Tree Problem (STP)

The Steiner Tree Problem (STP) seeks a minimum-cost tree in an undirected weighted graph that connects a given set of terminal nodes. To reduce the overall cost, the solution may include additional non-terminal vertices, referred to as Steiner nodes.

Instance Creation. We construct STP instances based on real-world benchmark graphs from Leitner et al. (2014). For each instance, we randomly select a source graph and parse its undirected weighted edges.

Given a target size n, we sample a structurally coherent node subset V' using a random walk with restart (RWR), following the procedure in GraphArena (Tang et al., 2025). Starting from a randomly chosen seed node (sampled from available terminals when provided, otherwise from all vertices), the walk is executed for $L = 5000$ steps with restart probability $\alpha = 0.15$. Vertices are ranked by visitation frequency, and the top-n vertices are retained to form an induced subgraph $G' = G[V']$. Disconnected samples are rejected. We then generate the terminal set $T' \subseteq V'$ by sampling a terminal ratio $r \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ and selecting $|T'| = \max(2, \lfloor r|V'| \rfloor)$ terminals uniformly at random from V'. Finally, vertices in G' are relabeled to contiguous indices $\{1, \ldots, |V'|\}$, while original edge costs are preserved.

We categorize instances into three difficulty tiers based on subgraph size:

$$\begin{aligned} &\text{Set-S:} \quad |V'| \in [10, 15], \\ &\text{Set-M:} \quad |V'| \in [16, 20], \\ &\text{Set-L:} \quad |V'| \in [21, 30]. \end{aligned}$$

Table 6 summarizes the statistics of the generated STP instances across the three size levels. Ground truth solutions are computed using Gurobi.

Tier	$\overline{D}$	$D_{\min}$	$D_{\max}$	$\overline{\lvert T\rvert}$	$\lvert T\rvert_{\min}$	$\lvert T\rvert_{\max}$	$\overline{\lvert T\rvert/\lvert V\rvert}$
Set-S	0.2064	0.1429	0.2727	3.68	2	7	0.3051
Set-M	0.1447	0.1158	0.1833	5.78	2	10	0.3233
Set-L	0.1037	0.0805	0.1381	6.92	2	14	0.2768

Table 6: Statistics of generated Steiner Tree (STP) instances across three levels. D denotes graph density, $|T|$ denotes the number of terminals, and $|V|$ denotes the number of vertices in the sampled subgraph.

C.1.10 Steiner Forest Problem (SFP)

The Steiner Forest Problem (SFP) generalizes the Steiner Tree Problem by requiring connectivity only within multiple disjoint terminal groups. The objective is to find a minimum-cost forest such that terminals belonging to the same group are connected, while terminals from different groups need not be connected.

Instance Creation. We generate SFP instances using the same subgraph extraction pipeline as in STP. Given the sampled subgraph, we generate multiple terminal groups. We first sample a terminal ratio $r \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ and select $|T|$ terminals accordingly. The selected terminals are then randomly partitioned into g groups, where $g \in [2, 4]$ is sampled subject to feasibility constraints, and each group contains at least two terminals. All vertices are relabeled to contiguous indices, and original edge costs are preserved.

We categorize SFP instances into the same three difficulty tiers based on subgraph size as in STP.

We solve Steiner forest instances using Gurobi. Table 7 reports summary statistics of the generated SFP instances across the three size levels.

Tier	$\overline{D}$	$D_{\min}$	$D_{\max}$	$\overline{\|T\|}$	$\overline{\|T\|/\|V\|}$	$\overline{\#\text{Groups}}$
Set-S	0.2063	0.1333	0.2667	4.48	0.3743	2.14
Set-M	0.1447	0.1158	0.1833	5.64	0.3175	2.24
Set-L	0.1036	0.0813	0.1333	7.28	0.2900	2.42

Table 7: Statistics of generated Steiner Forest (SFP) instances across three levels. D denotes graph density, $|T|$ the total number of terminals, $|V|$ the number of vertices in the sampled subgraph, and $\#$Groups the number of terminal groups

C.1.11 k-Minimum Spanning Tree (KMST)

The k-Minimum Spanning Tree (KMST) problem seeks a minimum-cost tree that spans exactly k vertices of an undirected weighted graph.

Instance Creation. We generate KMST instances using the same subgraph extraction pipeline as in STP. Given the sampled subgraph, we determine the target size k by sampling a ratio $r \in \{0.3, 0.4, 0.5\}$ relative to the subgraph size and setting $k = \max(2, \lfloor r|V'|\rfloor)$. All vertices are relabeled to contiguous indices, and original edge costs are preserved. We categorize k-MST instances into the same three difficulty tiers based on subgraph size as in STP. We solve all KMST instances to optimality using Gurobi. Table 8 reports summary statistics of the generated KMST instances.

Tier	$\overline{k}$	$\overline{k/\|V\|}$	$\overline{D}$	$D_{\min}$	$D_{\max}$
Set-S	4.74	0.3906	0.2127	0.1429	0.2667
Set-M	7.32	0.4120	0.1451	0.1170	0.1750
Set-L	10.76	0.4287	0.1045	0.0805	0.1333

Table 8: Statistics of generated KMST instances across three levels. k denotes the number of selected vertices, $|V|$ the number of vertices in the sampled subgraph, and D the graph density.

C.2 Constraint-family Group: PACK

C.2.1 Bin Packing Problem (BPP)

The Bin Packing Problem (BPP) asks to pack n items of given sizes into identical bins of fixed capacity while using as few bins as possible. Each item must be assigned to exactly one bin, and the total size of items cannot exceed the bin capacity.

Instance Creation. For all BPP instances, item sizes and bin capacities are drawn from the OR-Library datasets (Falkenauer, 1996). For each instance, a dataset file is selected at random and n items are uniformly sampled, while the original bin capacity is retained. Each sampled instance is solved using Gurobi.

C.2.2 Cutting Stock Problem (CSP)

The Cutting Stock Problem (CSP) seeks to determine how to cut large stock materials (e.g., rolls or bars) into smaller requested piece sizes while meeting all demands and minimizing waste (or, equivalently, minimizing the number of stock pieces used). Each stock piece has a fixed length/capacity, each item type has a required size and demand, and a solution specifies cutting patterns that exactly cover the demands without exceeding the stock capacity.

Instance Creation. CSP instances are generated by sub-sampling from BPPLIB (Delorme et al., 2018). For each generated instance, we specify the number of distinct item types, m. We first select a source problem from the dataset and inherit its bin capacity C. We then randomly sample m item types from the source data, extracting their associated weights w_i and demands d_i. Ground truth optimal solutions are computed using a pattern-based integer linear programming formulation solved by Gurobi, where the decision variables represent the usage frequency of feasible cutting patterns.

C.2.3 2D Strip Packing (2SP)

The goal of 2D Strip Packing (2SP) is to place a set of axis-aligned rectangles, without overlap, into a strip of fixed width and unbounded height so as to minimize the used height. Each rectangle must be packed entirely within the strip, and the objective is to produce a compact layout with the smallest possible makespan height.

Instance Creation. 2SP instances are generated by sub-sampling item types from the 2DPackLib benchmark (Iori et al., 2022). Each item type i is characterized by width w_i, height h_i, and demand d_i, yielding a total of $\sum_i d_i$ rectangles.

To avoid trivial instances where the optimal solution height equals the tallest item, we apply several techniques. The strip width is set to $W = 0.8 \cdot W_{\text{o}}$ to increase packing density (W_{o} is the original bin width of the benchmark data). We scale the height limit to preserve the parent instance tightness while ensuring non-triviality: letting $A = \sum_i w_i h_i d_i$ and $LB = \lceil A/W \rceil$, we compute the parent tightness ratio $\rho = H_{\text{parent}}/LB_{\text{parent}}$ and set the sub-instance height to $H_{\text{sub}} = \max(\lceil 1.3 \cdot \max_i h_i \rceil, \lceil \rho \cdot LB_{\text{sub}} \rceil)$. During sampling, we reject item subsets that lack height diversity (requiring at least two items with height $\geq 0.7 \cdot \max_i h_i$), have insufficient material (total stacked height $\sum_i h_i d_i < 1.5 \cdot H_{\text{sub}}$), can fit in a single row (total width demand $\sum_i w_i d_i < 2W$), or have area utilization outside $[0.60, 0.95]$. After solving with Gurobi, instances with optimal height $H^* < 1.1 \cdot \max_i h_i$ are resampled (up to 30 retries) to ensure non-trivial lower bounds. Optimal solutions are obtained via Gurobi with a position-based MIP formulation, where non-overlap constraints are enforced via binary indicator variables.

C.2.4 Job-Shop Scheduling Problem (JSP)

Job-shop Scheduling Problem (JSP) aims at minimizing the makespan by finding an optimal schedule for a set of jobs, each consisting of a sequence of operations that must be processed on specific machines in a fixed order.

Instance Creation. Instances are generated following the well-established standard of Taillard (1993). Processing times are independently drawn from a uniform integer distribution in $[1, 99]$ using Taillard's linear congruential random number generator to ensure reproducibility. Each job consists of exactly m operations, where m is the number of machines, and visits every machine exactly once. For each job, the machine order is given by a uniformly random permutation. The reference solutions are obtained using OR-Tools CP-SAT solver.

C.2.5 Flow-Shop Scheduling Problem (FSP)

The Flow-shop Scheduling Problem (FSP) considers a set of jobs that must all be processed on the same set of machines in an identical order. Each job consists of exactly one operation per machine. The objective is typically to minimize the makespan.

Instance Creation. Instances are generated following Taillard (1993). Processing times of generated instances are drawn uniformly from $[1, 99]$ following Taillard's benchmark methodology, and all jobs follow the fixed machine sequence $(1, 2, \ldots, m)$. The reference solutions are also obtained using OR-Tools CP-SAT solver.

C.2.6 Open-Shop Scheduling Problem (OSP)

The Open-shop Scheduling Problem (OSP) generalizes both JSP and FSP settings by removing predefined operation orders within jobs. Each job requires processing on every machine exactly once, but the order in which its operations are executed is not fixed and must be decided by the scheduler.

Instance Creation. As in the flow-shop case, processing times are generated uniformly in $[1, 99]$ following Taillard (1993). The reference solutions are calculated using OR-Tools CP-SAT solver.

C.2.7 Resource-Constrained Project Scheduling Problem (RCPSP)

The Resource-Constrained Project Scheduling Problem (RCPSP) consists of a set of tasks subject to precedence constraints and limited renewable resources. Each task is characterized by a fixed processing duration and a vector of resource demands, and each resource has a fixed capacity. A task can only start once all its predecessors have finished and sufficient resource capacity is available. The objective considered in this work is the minimization of the makespan, i.e., the completion time of the last task.

Instance Creation. Generated instances are re-sampled from the RCPSP benchmark (Pelleau et al., 2013) by applying random modifications. We specify the number of tasks and resources, resource capacities, task durations, resource demands, and precedence relations. Task durations, resource demands, and resource capacities are scaled by independent random factors drawn from predefined ranges, while preserving feasibility by ensuring that each resource capacity is at least as large as the maximum demand of any task. Additionally, instances can be extended or reduced by adding or removing tasks and precedence relations, with care taken to avoid cycles in the precedence graph. We use CP-SAT solver for reference solutions.

C.2.8 Parallel Machines Scheduling (PMS)

Parallel Machines Scheduling (PMS) considers a set of independent jobs that must be processed on a set of identical parallel machines. Each job is characterized by a processing time, a release date, and a deadline, and can be assigned to any of the available machines. At most one job can be processed on a machine at any time, and a job may only start after its release date. It minimizes tardiness.

Instance Creation. Each generated instance specifies the number of jobs and machines, followed by job-specific processing times, release dates, and deadlines of existing instances (Tanaka and Araki, 2008). To increase instance diversity and allow fine-grained control over problem size and difficulty, we resample the original instances by randomly scaling processing times, release dates, and deadlines within predefined ranges, while preserving feasibility by ensuring that each job's deadline remains no earlier than its release date plus processing time. In addition, instances can be extended or reduced by adding or removing jobs, with new jobs generated to match the statistical properties of the original data.

C.2.9 Single-Machine Total Weighted Tardiness (SMTWT)

The Single Machine Total Weighted Tardiness (SMTWT) problem involves scheduling a set of jobs on one machine. Each job needs to be completed without stopping, and the machine can only work on one job at a time.

Instance Creation. The instances are randomly generated following Beasley (1990). For each job, the processing time $p(j)$ is drawn uniformly from $[1, 100]$ and the weight $w(j)$ from $[1, 10]$. Due dates are generated to control instance difficulty using two parameters: the relative range of due dates (RDD) and the tardiness factor (TF), with values in $\{0.2, 0.4, 0.6, 0.8, 1.0\}$. Let $P = \sum_{j=1}^{n} p(j)$. For a given pair (RDD, TF), each due date $d(j)$ is drawn uniformly from the interval $\big[P(1 - \text{TF} - \text{RDD}/2),\ P(1 - \text{TF} + \text{RDD}/2)\big]$.

C.3 Constraint-family Group: COUNT

C.3.1 Minimum Dominating Set (MDS)

The Minimum Dominating Set Problem (MDS) finds the smallest subset of vertices such that every vertex in the graph is either selected or adjacent to a selected vertex.

Instance Creation. MDS instances are generated using the similar RWR-based subgraph extraction procedure as in STP. We sample subgraphs from the DIMACS10 Street Networks (Bader et al., 2014). Specifically, we use the following graphs: *Belgium*, *Great Britain*, *Italy*, *Luxembourg*, and *Netherlands*. For each MDS instance, we randomly select one of these networks and RWR procedure to obtain a structurally meaningful local subgraph. Starting from a randomly chosen seed node, the random walk is executed for 10,000 steps with a restart probability of $\alpha = 0.15$. The nodes with the highest visitation frequency are retained to form the subgraph. All sampled graphs are treated as unweighted, undirected, and required to be connected.

C.3.2 Set Covering Problem (SCP)

The Set Covering Problem (SCP) aims to select the minimum number of sets such that their union covers all elements in the universe.

Instance Creation. We generate SCP instances by mapping small graph neighborhoods into set systems, following the approach of (Khalil et al., 2017). Specifically, we begin with the DIMACS10 *road_central* network (Bader et al., 2014) and sample connected subgraphs using a random walk with restart (RWR) procedure, as employed in the construction of MDS instances.

Given a sampled subgraph $G = (V, E)$, we define the SCP ground set by treating each vertex $v \in V$ as an element. For each vertex $u \in V$, we form a candidate set by taking the k-hop neighborhood of u in G (here $k = 2$), so that each set corresponds to the elements (vertices) within distance at most k of its center. To induce stochastic sparsity, each vertex in this neighborhood is independently retained with probability 0.7. If the thinning process removes all vertices, the set is replaced with $\{u\}$, ensuring that every element contributes at least one valid candidate set. Finally, we discard sets that are too small, remove duplicate sets with identical coverage. Instances in which some elements remain uncovered after set construction are discarded to guarantee feasibility. We finally

Problem	**Tier**	$\overline{M}$	M_{min}	M_{max}	$\overline{N}$	N_{min}	N_{max}	$\overline{D}$	D_{min}	D_{max}	$\overline{k}$	k_{min}	k_{max}
Set Covering (SCP)	S	9.92	8	12	9.20	6	12	0.416	0.278	0.661	–	–	–
	M	15.36	13	18	14.40	11	18	0.282	0.190	0.420	–	–	–
	L	21.44	19	25	20.44	15	25	0.208	0.150	0.275	–	–	–
Set Packing (SP)	S	9.78	8	12	9.02	6	12	0.400	0.265	0.625	–	–	–
	M	15.26	13	18	14.26	11	18	0.281	0.219	0.382	–	–	–
	L	21.42	19	25	20.28	17	25	0.207	0.150	0.318	–	–	–
Set Partitioning Problem (SPP)	S	9.88	8	12	30.18	19	46	0.168	0.146	0.188	–	–	–
	M	15.16	13	18	64.44	43	91	0.168	0.152	0.182	–	–	–
	L	22.20	19	25	123.98	85	163	0.172	0.159	0.184	–	–	–
Hitting Set Problem (HSP)	S	9.84	8	12	8.90	6	12	0.410	0.303	0.554	–	–	–
	M	15.12	13	18	14.20	10	18	0.285	0.209	0.426	–	–	–
	L	21.86	19	25	20.68	17	25	0.211	0.158	0.313	–	–	–
Max k-Coverage (MkC)	S	9.78	8	12	9.20	7	12	0.416	0.285	0.571	1.34	1	2
	M	15.28	13	18	14.16	10	18	0.279	0.183	0.354	2.44	2	3
	L	21.54	19	25	20.54	17	25	0.211	0.150	0.316	3.64	3	5

Table 9: Summary statistics of graph-based Set Covering Problem (SCP), Max k-Coverage (MkC), Set Packing (SP), Hitting Set Problem (HSP), and Set Partitioning (SPP) instances constructed from ROAD networks across three configurations (S, M, L). M denotes the number of elements, N the number of sets, D the density $nnz/(M \cdot N)$. For MkC, k denotes the budget (maximum number of selected sets).

compute SCP optimal solution by Gurobi.

C.3.3 Set Packing (SP)

Set Packing (SP) seeks to choose the maximum number of mutually disjoint sets, i.e., no two selected sets share an element.

Instance Creation. We generate SP instances using exactly the same graph-to-set construction pipeline as described in the SCP instance creation, including subgraph sampling, k-hop neighborhood formation, stochastic thinning, and duplicate removal. The only difference lies in the optimization objective: we solve the Set Packing problem using Gurobi to maximize the number of selected sets subject to element-disjointness constraints.

C.3.4 Set Partitioning Problem (SPP)

The Set Partitioning Problem (SPP) requires selecting a collection of sets such that each element in the universe is covered by exactly one selected set.

Instance Creation. We generate SPP instances following a randomized construction procedure inspired by (Kadioglu et al., 2012). Each instance is defined over a universe of m elements and a collection of candidate sets.

We first determine the number of non-singleton sets as $n = \lfloor f \cdot m \rceil$, where the scaling factor f is sampled uniformly from $[2, 4]$ for small (S), $[3, 5]$ for medium (M), and $[4, 6]$ for large (L) instances. For each set S_j ($j = 1, \ldots, n$), we sample a per-set density $d_j \sim \text{Uniform}(0.1, 0.3)$ and fix its cardinality as $|S_j| = \lfloor d_j \cdot m \rceil$.

Elements in each set are selected using a clustered sampling strategy: a random element is chosen as a center, and additional elements are sampled *without replacement* with probabilities that decay with their index distance to the center according to a Gaussian weighting, inducing local overlap structure among sets. Duplicate sets with identical element coverage are removed.

Set costs follow the *tails* cost model introduced in (Kadioglu et al., 2012). Specifically, each set S_j is assigned a cost $c_j = u_j \cdot |S_j|$, where $u_j \sim \text{Uniform}(1, 100)$.

Since the randomly generated sets may not form an exact partition, we additionally include a singleton set $\{i\}$ for each element i, each assigned a large fixed cost of 10^4. This guarantees that every instance admits at least one feasible exact partition.

C.3.5 Hitting Set Problem (HSP)

The Hitting Set Problem aims to choose the smallest number of elements such that each set in the collection contains at least one selected element.

Instance Creation. We generate HSP instances using the same graph-based set creation pipeline as in the SCP instance creation. We solve the Hitting Set Problem with Gurobi.

C.3.6 Max k-Coverage (MkC)

Max k-Coverage (MkC) is a constrained version of Maximum Coverage: select at most k sets to maximize the number of covered elements.

Instance Creation. MkC instances are built on the same graph-based set creation pipeline as in the SCP instance creation; however, the optimization objective differs. Instead of covering all elements with as few sets as possible, MkC restricts the solution to a fixed number of sets and maximizes the number of elements they collectively cover.

After sets are constructed from an RWR-sampled subgraph (the same as SCP), we set a budget k that limits how many sets may be selected. This budget is determined automatically as a fixed ratio (i.e., 20%) of the number of available sets. We get the reference solution using Gurobi.

C.3.7 Generalized Assignment Problem (GAP)

The Generalized Assignment Problem (GAP) assigns m tasks to n agents such that each task is assigned to exactly one agent, agent capacities are respected, and the total assignment cost is minimized. Each assignment (i, j) incurs a cost c_{ij} and consumes a_{ij} units of the limited resource of agent i, which has capacity b_i. The objective is to minimize $\sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$ subject to $\sum_{i=1}^{n} x_{ij} = 1$ for all tasks j, $\sum_{j=1}^{m} a_{ij} x_{ij} \leq b_i$ for all agents i, and $x_{ij} \in \{0, 1\}$.

Instance Creation. We generate GAP instances by sub-sampling from the OR-Library GAP dataset (Chu and Beasley, 1997). For a target size of m tasks, the number of agents n is determined proportionally to maintain a challenging agent-to-task ratio. Specifically, n is set as $\max(2, \text{round}(m/\alpha))$, where the ratio α is randomly sampled from a uniform distribution $\mathcal{U}(2.5, 3.5)$. We then randomly select m tasks and n agents from a larger source instance and extract the corresponding $n \times m$ sub-matrices for assignment costs and resource consumptions, along with the capacity sub-vector for the n selected agents. Ground truth optimal solutions are obtained by solving the MILP formulation for the assignment problem with capacity constraints, using the Gurobi optimizer.

C.3.8 Uncapacitated Facility Location (UFLP)

The Uncapacitated Facility Location Problem (UFLP) chooses which facilities to open and assigns each customer to an open facility to minimize opening plus service costs.

Instance Creation. UFLP instances are constructed by sub-sampling from the Bilde–Krarup benchmark (Bilde and Krarup, 1977) to preserve its original cost structures. For each generated instance, we first define the number of customers, n_c, and determine the number of candidate facilities, n_f, based on a facility-to-customer density ratio α. Specifically, we sample α uniformly from the interval $[2.0, 2.67]$ (2.0 and 2.67 are the ratio in the original dataset) and set $n_f = \text{round}(n_c/\alpha)$. We then randomly sample n_c customers and n_f facilities from the source data, extracting the associated opening and connection cost sub-matrices. Ground truth optimal solutions are obtained using the Gurobi mixed-integer programming solver.

C.3.9 Capacitated Facility Location (CFLP)

The Capacitated Facility Location Problem (CFLP) selects a subset of facilities to open and assigns customers to open facilities to minimize the sum of facility opening costs and assignment costs, while ensuring that the total demand assigned to each facility does not exceed its capacity.

Instance Creation. Similar to UFLP, CFLP instances are generated by sub-sampling from the existing benchmark (Holmberg et al., 1999). The sampling of customers n_c and facilities n_f based on the density ratio α follows the same protocol described above. In addition to opening and connection costs, we extract the corresponding facility capacities and customer demands. Ground truth optimal solutions are also obtained using Gurobi.

C.3.10 p-Median Facility Location (PMED)

The p-median problem selects exactly p facility locations and assigns each customer to its chosen facility to minimize the total assignment cost.

Instance Creation. Similar to UFLP, p-median instances are constructed by sub-sampling from a specific dataset used in the classical OR literature (Beasley, 1985). For each generated instance with n vertices, we determine the number of medians, p, by scaling the original parameter proportionally to the new problem size (maintaining the original ratio p/N). We randomly sample n vertices and extract the corresponding pairwise shortest path distances, computed on the original graph, to form the cost matrix. Ground truth optimal solutions are solved by Gurobi.

C.3.11 p-Center Facility Location (PCENTER)

The p-center problem selects exactly p facility locations and assigns each customer to a chosen facility to minimize the maximum assignment distance.

Instance Creation. The generation of p-center instances is the same as the p-median facility location problem, but with a different Gurobi model for modeling the different objective.

C.3.12 Maximum Independent Set (MIS)

The Maximum Independent Set problem (MIS) seeks the largest subset of vertices in which no two selected vertices share an edge.

Instance Creation. We generate MIS instances by extracting subgraphs from the DIMACS10 Redistricting Graph collection (Bader et al., 2014), which provides high-quality real-world regional adjacency graphs used in political redistricting. In our experiments, we use the five state-level graphs (*vt2010*, *wa2010*, *wi2010*, *wv2010*, *wy2010*) as the sources for random-walk subgraph sampling. For each MIS instance, we randomly select one of these networks and apply the same RWR procedure as in MDS to obtain a structurally meaningful local subgraph. To avoid trivial structures that are either nearly empty or overly dense, we set density constraints specific to each difficulty tier:

$$\begin{aligned} &\text{Set-S:} && 0.15 \le \rho \le 0.55, \\ &\text{Set-M:} && 0.12 \le \rho \le 0.50, \\ &\text{Set-L:} && 0.08 \le \rho \le 0.45. \end{aligned}$$

Ground truth optimal solutions are obtained using Gurobi.

C.3.13 Minimum Vertex Cover (MVC)

The Minimum Vertex Cover Problem (MVC) aims to find the smallest set of vertices such that every edge in the graph has at least one endpoint selected.

Instance Creation. MVC instances are generated using the same subgraph extraction pipeline as MIS. We sample subgraphs from the DIMACS10 Redistricting Graph collection (Bader et al., 2014), using the same random walk with restart (RWR) procedure as in MDS. We apply the same density constraints as MIS.

C.3.14 Maximum Clique Problem (MCP)

The Maximum Clique Problem (MCP) searches for the largest subset of vertices that are all pairwise adjacent on a graph.

Instance Creation. We again employ the same RWR-based sampling strategy used for MDS. For MCP, we use citation graphs from the DIMACS10 Co-author and Citation Networks collection (Bader et al., 2014). We restrict MCP instances to the following density ranges:

$$\begin{aligned} &\text{Set-S:} && 0.30 \le \rho \le 0.70, \\ &\text{Set-M:} && 0.25 \le \rho \le 0.65, \\ &\text{Set-L:} && 0.20 \le \rho \le 0.60. \end{aligned}$$

C.3.15 Knapsack Problem (KP)

The Knapsack Problem (KP) seeks a subset of n items to maximize total value subject to a capacity constraint. Each item i has a value v_i and a weight w_i, and the decision variable $x_i \in \{0, 1\}$ indicates whether the item is selected. The objective is to maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \le C$, where C is the knapsack capacity.

Instance Creation. For all KP instances, the creation follows the uncorrelated instance class defined in Pisinger (2005). Item weights and profits are generated i.i.d. from the discrete uniform distribution on $\{1, \dots, 100\}$. The knapsack capacity is set to $C = \alpha \sum_{i=1}^{n} w_i$, with $\alpha = 0.5$. The reference optimal solution is obtained through Gurobi.

C.3.16 Maximum Diversity Problem (MDP)

The MDP aims to select a subset of m elements from a ground set of n elements such that the selected elements are as mutually dissimilar as possible. Given a pairwise distance matrix $D = (d_{ij})$, the objective is to maximize the total pairwise distance among the selected elements, i.e., $\sum_{i=1}^{n} \sum_{j=i+1}^{n} d_{ij} x_i x_j$, subject to selecting exactly m elements: $\sum_{i=1}^{n} x_i = m$ with $x_i \in \{0, 1\}$.

Instance Creation. Maximum Diversity Problem instances are generated by sub-sampling from GKD and SOM sets in MDPLIB (Martí et al., 2021). For each generated instance, we specify the number of vertices, n, and randomly select a source problem from the dataset. We then sample n vertices from the full graph and extract the corresponding pairwise distance sub-matrix. The diversity parameter m (number of elements to select) is scaled proportionally to maintain the original selection ratio: $m = \max(2, \text{round}(m_{\text{original}} \times n/N))$, where N is the size of the source instance. Distance matrices are represented as integers. Ground truth optimal solutions are obtained using Gurobi mixed-integer programming with a linearized formulation of the quadratic diversity objective.

Problem	**Tier**	$\overline{\lvert V\rvert}$	$\lvert V\rvert_{\min}$	$\lvert V\rvert_{\max}$	$\overline{\lvert E\rvert}$	$\lvert E\rvert_{\min}$	$\lvert E\rvert_{\max}$	$\overline{D}$	$D_{\min}$	$D_{\max}$
Maximum Independent Set (MIS)	S	9.74	8	12	16.38	10	24	0.391	0.273	0.536
	M	15.24	13	18	28.70	19	40	0.267	0.203	0.333
	L	21.70	19	25	43.24	30	54	0.195	0.156	0.247
Minimum Vertex Cover (MVC)	S	10.04	8	12	16.70	10	27	0.372	0.258	0.500
	M	15.24	13	18	27.94	18	37	0.260	0.206	0.333
	L	21.64	19	25	42.62	32	56	0.192	0.147	0.240
Max Clique (MCP)	S	10.00	8	12	19.72	9	42	0.427	0.303	0.691
	M	15.44	13	18	42.96	22	98	0.384	0.250	0.647
	L	22.18	19	25	73.04	39	129	0.314	0.203	0.579
Graph Coloring (GCP)	S	10.16	8	12	21.22	10	39	0.448	0.306	0.679
	M	15.54	13	18	43.10	21	92	0.378	0.250	0.618
	L	22.20	19	25	79.06	38	174	0.333	0.205	0.580
Minimum Dominating Set (MDS)	S	9.92	8	12	9.30	7	13	0.214	0.167	0.357
	M	15.48	13	18	14.94	12	19	0.135	0.111	0.179
	L	22.04	19	25	21.78	18	26	0.095	0.080	0.126
Max Cut (MAXCUT)	S	10.34	8	12	17.26	8	24	0.358	0.242	0.500
	M	15.86	13	18	29.18	18	41	0.250	0.158	0.333
	L	22.72	19	25	44.40	24	62	0.180	0.133	0.240

Table 10: Summary statistics of the generated graph for the Maximum Independent Set (MIS), Minimum Vertex Cover (MVC), Maximum Clique Problem (MCP), Graph Coloring Problem (GCP), Minimum Dominating Set (MDS), and Maximum Cut (MAXCUT) across the three difficult tiers (S, M, L). For each tier, we report the average, minimum, and maximum values of the number of vertices ($|V|$), number of edges ($|E|$), and graph density (D).

C.3.17 Quadratic Knapsack Problem (QKP)

QKP extends the 0–1 knapsack by allowing the profit of a selected set of items to include pairwise interaction gains (or penalties). Each item i has a weight a_i, and the selection must satisfy a single capacity constraint C. The objective is to maximize the total profit consisting of linear terms c_i and quadratic interaction terms c_{ij}, i.e., $\sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n}\sum_{j=i+1}^{n} c_{ij} x_i x_j$, subject to $\sum_{i=1}^{n} a_i x_i \le C$ and $x_i \in \{0,1\}$.

Instance Creation. QKP instances are constructed by sub-sampling from the QAPLIB benchmark datasets to preserve their original coefficient structures. For each generated instance, we randomly select a subset S of n variables from a source problem of size N, and extract the corresponding linear coefficients $\{c_i\}_{i\in S}$, quadratic coefficients $\{c_{ij}\}_{i,j\in S,\, i<j}$ (upper triangular), and item weights $\{a_i\}_{i\in S}$. To maintain the constraint tightness, we scale the capacity proportionally as:

$$C_{\text{new}} = C_{\text{original}} \times \frac{\sum_{i\in S} a_i}{\sum_{i=1}^{N} a_i},$$

so that the capacity-to-weight ratio remains consistent with the source instance. The resulting QKP maximizes $\sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n}\sum_{j=i+1}^{n} c_{ij} x_i x_j$ subject to $\sum_{i=1}^{n} a_i x_i \le C_{\text{new}}$ and $x_i \in \{0,1\}$. Ground truth optimal solutions are obtained using the Gurobi mixed-integer programming solver with Kaufman–Broeckx linearization.

C.3.18 Maximum Cut Problem (MAXCUT)

The Maximum Cut Problem (MAXCUT) partitions the vertices into two subsets to maximize the number of edges crossing between them.

Instance Creation. We sample subgraphs from the DIMACS10 Redistricting Graph collection (Bader et al., 2014), using RWR as in MDS, enforcing connectivity, and adopting the same density constraints as those in MIS.

C.4 Constraint-family Group: COMP

C.4.1 3-Index Assignment Problem (AP3)

The 3-Index Assignment Problem (AP3) is a higher-dimensional generalization of the classic linear assignment. It seeks to select n triples (i,j,k) from three disjoint sets of size n such that every element in each set is selected exactly once, minimizing the total cost. Given a cost tensor $C = (c_{ijk})$ where $i,j,k \in \{1,\ldots,n\}$, the objective is to find binary variables x_{ijk} that minimize the total cost:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} c_{ijk}x_{ijk}$$

subject to the constraints:

$$\sum_{j=1}^{n}\sum_{k=1}^{n} x_{ijk} = 1 \quad \forall i,$$
$$\sum_{i=1}^{n}\sum_{k=1}^{n} x_{ijk} = 1 \quad \forall j,$$
$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ijk} = 1 \quad \forall k$$

Instance Creation. We generate AP3 instances by sub-sampling from Balas benchmark (Balas and Saltzman, 1991), which provides dense integer cost tensors. For a target dimension n, we randomly select n indices from the source instance's dimension N (where $N \geq n$) and extract the corresponding $n \times n \times n$ sub-tensor. This approach maintains the structural correlations present in the original data. Ground truth optimal solutions are computed using a direct MILP formulation solved via Gurobi.

C.4.2 Quadratic Assignment Problem (QAP)

The Quadratic Assignment Problem (QAP) seeks an assignment of n facilities to n locations that minimizes the total interaction cost induced by facility-to-facility flows and inter-location distances. Given a flow matrix $F = (f_{ij})$ and a distance matrix $D = (d_{kl})$, the objective is to find a permutation π that minimizes $\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij}\, d_{\pi(i)\pi(j)}$.

Instance Creation. We generate QAP instances by sub-sampling from the QAPLIB dataset (Burkard et al., 1997) to preserve realistic flow and distance structures. For a target problem size n, we randomly select n locations from a source instance and extract the corresponding $n \times n$ flow and distance sub-matrices. Ground truth optimal solutions are obtained by solving an MILP formulation, specifically utilizing the Kaufman–Broeckx linearization, via the Gurobi optimizer.

C.4.3 Graph Coloring Problem (GCP)

The Graph Coloring Problem (GCP) assigns the minimum number of colors to vertices so that no adjacent vertices share the same color.

Instance Creation. GCP uses the same RWR extraction method and size ranges as MDS. Instances are generated from the same DIMACS10 Co-author and Citation Networks as MCP (Bader et al., 2014). We adopt the same density constraints as MCP.

C.4.4 Cutwidth Minimization Problem (CMP)

The Cutwidth Minimization Problem (CMP) seeks a linear layout of a graph's vertices such that the maximum number of edges crossing any cut between consecutive vertices in the layout is minimized. Given a graph $G = (V, E)$ with n vertices, the goal is to find a bijection $\pi : V \to \{1, \ldots, n\}$ that minimizes the cutwidth:

$$\max_{1 \leq i < n} |\{(u, v) \in E : \pi(u) \leq i < \pi(v)\}|$$

Instance Creation. We generate CMP instances by extracting induced subgraphs from CMPLIB graphs (Martí et al., 2012). For a target size n, we randomly select n vertices and include all edges connecting these vertices (induced subgraph). To ensure non-trivial instances, we initialize the selection with the endpoints of a randomly chosen edge before sampling the remaining vertices. Ground truth optimal solutions are computed using an MILP formulation solved via Gurobi.

C.4.5 Linear Ordering Problem (LOP)

The Linear Ordering Problem (LOP) asks for a permutation (ordering) of a set of n nodes that maximizes the total weight of arcs that are consistent with the order. Given a weighted directed complete graph represented by a cost matrix $C \in \mathbb{R}^{n\times n}$, the objective is to find a permutation π over $\{1, \ldots, n\}$ that maximizes $\sum_{i<j} C_{\pi_i,\pi_j}$, i.e., the sum of weights from nodes placed earlier in the order to nodes placed later.

Instance Creation. For each LOP instance, a base problem is sampled uniformly at random from the LOPLIB dataset (Martí et al., 2012), where each entry specifies a single square cost matrix C. Given a target size n, the extractor sub-samples n nodes uniformly without replacement from the original matrix indices, and induces the corresponding principal submatrix $C' = C[\mathcal{I}, \mathcal{I}]$, where $\mathcal{I}$ is the selected index set. No additional normalization or transformation is applied; the submatrix values are inherited directly from the source instance.

Each sampled instance is solved using a mixed-integer programming formulation with Gurobi.

D Details of Natural-Language Contextualization

This section provides additional details of the natural-language contextualization pipeline used to construct textual instances from numerical CO instances. The pipeline transforms a numerical instance into a scenario-grounded textual input. At a high level, the pipeline proceeds by 1) generating and verifying diverse scenario descriptions for a target problem, 2) constructing reusable scenario-level instruction templates, 3) deriving scenario-specific field specifications and natural-language input templates, and 4) instantiating numerical instances under sampled surface input formats. Across all steps in Appendix D.1–D.4, we use GPT-5-mini for LLM generation and checking.

D.1 Scenario Pool Construction (Generate–Filter–Verify)

What this step produces. For each CO problem P, we build a scenario pool $\mathcal{S}_P = \{s_{P,1}, s_{P,2}, \ldots, s_{P,N_P}\}$ where each $s_{P,i}$ is a short, everyday story that describes the same task as P. Across the pool, scenarios should look different on the surface (domain, wording), but they keep the same decision variables, constraints, and objective.

Generate candidates. We iteratively expand $\mathcal{S}_P$. In each round, a generator LLM is prompted to produce $K{=}20$ candidate scenarios for P using the prompt in Prompt 1. The candidates are encouraged to be brief, and to only change narrative details.

Filter near-duplicates. To ensure the diversity of the scenarios, we remove candidates that are too similar to existing ones in $\mathcal{S}_P$. We embed each text using SentenceTransformer `all-MiniLM-L6-v2` and compute cosine similarity. A candidate c is discarded if $\max_{s \in \mathcal{S}_P} \cos\big(e(c), e(s)\big) > 0.7$.

Verify task consistency. For the remaining candidates, we run an LLM as a semantic verifier. Given a candidate scenario s and the formal CO task description $\mathcal{D}_P$, the verifier checks whether s still describes the same CO task: it must preserve the original problem structure, so we reject scenarios that add extra rules, change the objective or constraints. The verifier prompt is shown in Prompt 2.

Stopping rule. Only candidates that pass verification are added to $\mathcal{S}_P$. We repeat the generate–filter–verify loop until $|\mathcal{S}_P| \geq N_P$, and we set $N_P{=}50$ in our construction.

Example. Taking CVRP as an example, below are two accepted scenarios that differ in narrative but encode the same problem structure:

- *I need the postmaster to assign parcel delivery runs from the distribution center so that every address is served exactly once, each run starts and ends at the center, no van is loaded beyond its capacity, and the overall route length is minimized.*
- *The school district planner should design bus routes from the depot that pick up each stop exactly once, begin and finish at the depot without exceeding bus seating limits, and keep the total driving distance as low as possible.*

This scenario pool $\mathcal{S}_P$ is used in the next step to build reusable instruction templates.

D.2 Instruction Template Construction

What this step produces. For each verified scenario $s \in \mathcal{S}_P$, we build a reusable instruction template $\mathcal{T}_s$. The key idea is to separate the *scenario wording* from the *instance data*, so we do not need to regenerate a full instruction for every instance.

Two-stage prompting: intro then continuation. Template construction uses two prompts.

Stage 1: scenario intro variants. Given the scenario text s, the problem P, and the formal description $\mathcal{D}_P$, the LLM rewrites s into multiple short introductions that keep the same task but vary in tone and framing $(s,\, P,\, \mathcal{D}_P) \;\longrightarrow\; \{\, \iota_s^{(1)}, \ldots, \iota_s^{(8)} \,\}$. We then randomly select one introduction ι_s^* and denote it as the template prefix $\mathcal{I}_s^{\text{intro}} = \iota_s^*$. The prompt used in this stage is shown in Prompt 3.

Stage 2: instruction continuation with output requirements. Next, conditioned on the chosen intro $\mathcal{I}_s^{\text{intro}}$, the LLM generates a continuation that 1) explains what to do with instance data and 2) states the required output format $\mathcal{O}_P$: $(\mathcal{I}_s^{\text{intro}},\, P,\, \mathcal{D}_P,\, \mathcal{O}_P) \;\longrightarrow\; \kappa_s$. The prompt used in this stage is shown in Prompt 4.

Final template with a single instance placeholder. We form the final instruction template by inserting exactly one placeholder for instance input:

$$\mathcal{T}_s \;=\; \mathcal{I}_s^{\text{intro}} \,\|\, \langle \texttt{INSTANCE\_INPUT} \rangle \,\|\, \kappa_s.$$

At dataset construction time, we replace `INSTANCE_INPUT` with instance-specific content (rendered in one of the surface formats described in the next subsection).

D.3 Field Specifications and Surface Input Formats

What this step produces. This step produces scenario-specific rendering assets that control how instance data is shown: 1) a scenario-specific field specification $\mathcal{H}_s$ for structured formats (JSON/CSV/Markdown); 2) a natural-language rendering template $\mathcal{N}_s$ for the free-text (NL) format. Both assets change only how the input is presented to mimic realistic user inputs, not the underlying numbers, constraints, or feasible solution space.

Scenario-specific field specification for structured formats. We begin with a generic structured field definition $\mathcal{H}$, shared across scenarios for the same problem family. Concretely, $\mathcal{H}$ specifies 1) which input fields exist, 2) short explanations of what each field means, and 3) how fields are grouped. For example, `global_fields` that describe the whole instance (e.g., total capacity) versus `item_fields` that repeat for each record (e.g., each location). Given a verified scenario $s \in \mathcal{S}_P$ and its chosen intro $\mathcal{I}_s^{\text{intro}}$, we ask the LLM to rewrite $\mathcal{H}$ into a scenario-aligned specification $(\mathcal{H},\, s,\, \mathcal{I}_s^{\text{intro}}) \longrightarrow \mathcal{H}_s$.

$\mathcal{H}_s$ preserves the same structure as $\mathcal{H}$ (same groups and same fields), but replaces technical names and descriptions with story-consistent ones. For example, in a CVRP-style delivery narrative, a generic field like `depot` may be renamed to `hub_node_id` and described as "the hub where routes start and end"; `capacity` may become `truck_capacity_units` with a description such as "maximum load each truck can carry"; and an item-level field like `demand` may become `store_demand_units` to match a "store replenishment" story. Similarly, coordinate fields `x,y` can be renamed to `coord_x, coord_y` while keeping the same meaning (used to compute travel distances). These changes are purely semantic *relabeling*: they make structured inputs easier to read under the scenario, without changing any data content.

The prompt used to generate field specifications is shown in Prompt 5.

Rendering into JSON/CSV/Markdown. Using $\mathcal{H}_s$, we render a numerical instance $I_{P,j}$ into different structured surface formats $(\mathcal{H}_s,\, I_{P,j},\, f) \longrightarrow \text{input}_{P,j}^{(f)}, f \in \{\text{JSON, CSV, MD}\}$. This step only changes labels and presentation; it does not change any values, constraints, or identifier relationships.

Natural-language rendering for the NL format. For the natural-language surface format, we also build an item-level template $\mathcal{N}_s$ that turns structured instance data into readable sentences. Concretely, $\mathcal{N}_s$ specifies: 1) a short header for the overall instance information (e.g., Total locations: 12; hub node: 0; truck capacity: 30 units.), 2) a per-record sentence pattern (e.g., Location B at coordinates (4, 7) has demand 10 units.), and 3) an optional footer for constraint specification (e.g., Every route must start and end at the hub and never exceed the truck capacity capacity.). We then render $I_{P,j}$ into free text by applying $\mathcal{N}_s$ record-by-record, so the input resembles what a user might type while remaining fully consistent with the structured data.

D.4 Instance-Level Contextualization and Representation Diversification

What this step produces. This step produces the final dataset instances: each numerical instance is paired with a natural-language instruction template and a chosen input surface format.

Notation for numerical instances. Let $\mathcal{I}_P = \{I_{P,1}, \ldots, I_{P,M_P}\}$ be the set of numerical instances for P. (We use $\mathcal{D}_P$ for the formal task description, and $\mathcal{O}_P$ for the output format.)

Sampling a scenario template. For each numerical instance $I_{P,j}$, we uniformly sample one scenario index $u \in \{1, \ldots, N_P\}$ and take the corresponding template $\mathcal{T}_{s_{P,u}}$. This chooses the narrative framing and output requirements, while $I_{P,j}$ supplies the instance data.

Choosing an input surface format. To reflect variation in user input, we randomly sample a surface format $f \in \{\text{NL, JSON, CSV, MD}\}$, and render $I_{P,j}$ using the natural-language rendering template $\mathcal{N}_{s_{P,u}}$ if $f = \text{NL}$; otherwise, we use the structured specification $\mathcal{H}_{s_{P,u}}$ to obtain $\text{input}_{P,j}^{(f)}$.

Identifier variations. We also vary naming schemes for identifiers (e.g., numeric indices vs. alphabetical labels) to increase surface diversity. We enforce strict consistency: identifiers used in the input always match those expected in the output. These changes affect only presentation and do not modify the underlying optimization problem.

Final assembly. We then fill the template placeholder `INSTANCE_INPUT` with the rendered input, producing the final contextualized instance $d_{P,j} = \big(\mathcal{T}_{s_{P,u}},\, \text{input}_{P,j}^{(f)}\big)$ for LLM evaluation. An example of contextualized instance for CVRP (with CSV format) is presented by Table 11.

Scenario Generation Prompt

You are an expert in creating diverse and realistic real-world scenarios that correspond to optimization problems. You will be given:

- **$problem_type** (internal only; don't mention it)
- **$task_description** (internal only)

Your reasoning

Before writing anything, silently determine:

1. **What the decision-maker chooses** (e.g., selecting, ordering, grouping, assigning, matching, scheduling, placing)
2. **The single optimization objective** (its direction, how it is aggregated, and its meaning)
3. **All feasibility constraints** (no more, no fewer than described in the description)

Once inferred, **these elements must remain completely fixed**:

- Do **not** add, remove, change, relax, or tighten any constraint.
- Do **not** alter the type, direction, or structure of the objective.
- Do **not** introduce any additional rules (time windows, capacities, precedence, fairness, deadlines, multi-resources, spatial layout, etc.) unless explicitly present in the canonical description.
- The feasible set and the difficulty of the problem must stay exactly the same.
- Only the *surface narrative* may change—not the underlying mathematical meaning.

Scenario transformation rules

Semantic preservation

You may change the real-world context and nouns, but:

- Each item must correspond to a concrete entity in the story.
- The meaning of the decision remains identical.
- The objective must preserve its mathematical form.
- All constraints must preserve their exact structure.

Objective wording

You may phrase the objective in natural language (e.g., "keep the overall effort as small as possible"), but the mathematical nature of the objective must stay the same.

What to generate

Produce **exactly $k** natural-language instructions. Each instruction must:

- Be one or two sentences.
- Be a realistic, everyday request.
- Encode the **same** decision structure, objective, and constraints as the canonical problem.
- Omit all additional constraints; include all canonical ones.
- Contain **no digits or ordinal words** in the prose.
- Be independently interpretable and solvable.

Strict output format

```
1. ...
2. ...
```

No additional text.

Prompt 1: Prompt used to generate diverse natural-language decision-making scenarios.

Problem-Consistency Verification Prompt

You are a task type verifier for optimization benchmark generation.

Target Problem

The expected problem type is: **expected_problem**

Problem description: “problem_description”

Verification Task

Determine whether the candidate instruction fundamentally describes a standard expected_problem problem.

Rules

YES if:

- Core decision structure is equivalent to the described formulation.
- Exactly one constraint and one objective.
- Real-world setting may vary but math structure is unchanged.
- Objective/constraint synonyms are acceptable (e.g., budget ≈ cost ≈ capacity).

NO if:

- Extra structural complexity (multiple constraints, dependencies, time windows, multi-objective).
- Deviation from the standard structure.

Do not treat objective wording as an extra constraint.

Candidate Instruction

```
text
```

Required Output

```
Answer: YES or NO
Reason: brief justification
```

Prompt 2: Verifier prompt used to check whether a generated scenario preserves the target optimization problem.

Casual Scenario Introduction Rewriting Prompt

You are helping to rewrite a task description so it sounds casual, human, and conversational.

You will be given:

- **problem type** (internal only; not to be mentioned)
- **task description** (internal only)
- **base text**: a stiff scenario that should be rewritten in a more natural style

Your task is to produce **eight different rewritten introductions**, each satisfying the following requirements.

Casual scenario intro

For each version:

- Rewrite the meaning of the base text in natural, everyday language.
- Implicitly reflect the core task described in the task description, but do so in an informal, story-like way.
- The rewritten version must clearly convey:
  - what situation is taking place,
  - what decision needs to be made,

  - what makes one decision better than another,
  - how that notion of "better" is determined,
  - and what practical requirements must be respected (e.g., nothing can be left out or duplicated).
- All of this should be communicated naturally through the scenario itself, without explicitly mentioning objectives, constraints, optimization, formulas, or technical terms.
- Do not mention the problem type, internal terminology, or any formal modeling concepts.
- Each version must start differently (e.g., I…, We…, There is…, Someone…, Recently…, Many people…).
- Do not address the reader directly or imply that the reader is a character in the scenario.
- Keep the tone human, conversational, and slightly varied across versions.

Reference to upcoming instance details

Each version must casually mention that the concrete details of the situation will be shown below.

Diversity requirements

- Number the versions using lettered markers (A), (B), (C), etc.
- Each version must sound noticeably different in opening style, tone, and framing.
- Avoid reusing similar sentence structures or phrasing patterns.
- All versions must remain faithful to the meaning of the base text.

Output rules

- Output exactly eight versions.
- Do not use JSON or structured formats.
- Do not include placeholders.
- Do not use quotation marks or code formatting.

Prompt 3: Prompt used to rewrite formal task descriptions into multiple casual, human-sounding scenario introductions while preserving the underlying task semantics.

Instruction Continuation with Structured Output Prompt

You are continuing a casual, human-sounding task description.

You will be given:

- **problem type** (internal only; not to be mentioned)
- **task description** (internal only)
- **output format**: the exact JSON structure the final answer must follow

Below is a draft that already covers the first part of the description, including a casual scenario introduction and a reference to the instance details.

Existing introduction

```
[Partial introduction text omitted here]
```

Do not change or repeat anything from the existing introduction. Your task is to add the remaining part that follows naturally from it.

Bringing up the JSON format casually

When writing the continuation:

- Work the mention of the JSON format into the flow in a relaxed, conversational way, as if casually noting how the response should be written.

- After that casual mention, include a JSON block that follows exactly the same structure as the given output format.

For the JSON block:

- The top-level JSON keys must remain exactly identical to those in the output format.
- For nested placeholder keys and values:
    - If a placeholder looks like an identifier-style label (e.g., depot identifiers, node identifiers, item identifiers), rename only the value prefix to better match the story context.
    - Keep all JSON keys unchanged.
    - Keep the overall JSON structure unchanged.
    - Keep the placeholder format unchanged (it must remain a placeholder, not a concrete value).
- If the placeholder values are not identifier-style (e.g., vectors, coordinates, counts, costs, times, or booleans), do not rename or modify them.

After presenting the JSON block:

- Give a short, easygoing explanation of what each part of the JSON represents in the context of the story.
- Keep the explanation informal and non-technical, more like describing a form than a data schema.
- Make clear that the JSON is only a sketch of the expected structure, not the actual answer.

Finally, gently remind that all identifiers must be used exactly as they appear in the instance input, with no renaming and no new labels. Valid identifiers include:

- plain numbers such as “1” or “23”,
- single capital letters such as “A” or “B”,
- a capital letter followed by digits such as “A1” or “X7”.

Critical constraints

- Do not repeat or alter the existing introduction.
- Do not mention the problem type or any technical optimization terminology.
- Keep the style consistent with the introduction: casual, conversational, and human.
- Do not number any part of the continuation.
- Do not write lines starting with digits and a period.

Prompt 4: Prompt used to continue a casual task description with a structured JSON output while preserving tone, narrative flow, and schema constraints.

Scenario-Specific Field Interpretation Prompt

You are an expert at interpreting generic structured-field definitions within a specific natural-language scenario.

You will receive:

- Task name
- Task description
- Scenario text describing the concrete instance
- A precomputed instruction introduction that sets the tone, role, and framing
- A generic hint JSON defining the structured fields

Your task

Rewrite the hint JSON so that every field has a scenario-specific meaning.

You must interpret the fields *as they would be understood under the instruction introduction style* provided above. In particular:

- The implied role, perspective, and constraints from the introduction must be respected.
- Field meanings should align with how a solver would understand the task after reading that introduction.

The **top-level structure must remain identical** to the input hint:

- Keep all existing top-level keys exactly as they appear.
- Preserve all field groups (e.g., global fields, item fields, facility item fields).
- If the input is flat, keep it flat.
- Do not add, remove, or rename any top-level keys.

How to rewrite each field

Each field object in the input has the form:

```
{
  "name": "field_name",
  "description": "generic meaning"
}
```

For every field object in every group, output an object with exactly the following structure:

```
{
  "name": "field_name",
  "new_name": "scenario_specific_snake_case",
  "scenario_description": "scenario-specific interpretation"
}
```

The following rules must be followed:

- The `name` field must be copied exactly from the input.
- The `new_name` must be a concise snake_case label that is meaningful in the scenario and unique within its group.
- The `scenario_description` must clearly explain how the field should be understood *within the scenario text*.
- Do not include the original generic description or any extra keys.

Allowed placeholders

Copy the `allowed_placeholders` field from the input *exactly*, without any modification.

Output (strict)

Output exactly one JSON object:

- The top-level keys and their ordering must match the input.
- All field groups must be rewritten as specified above.
- Do not include any text outside the JSON object.

Prompt 5: Prompt used to reinterpret generic structured-field hints into scenario-specific semantics while preserving the original schema.

Item-Level Natural Language Template Generation Prompt

You will create reusable natural-language templates for verbalizing item-level input in an optimization problem.

All generated text must follow the tone, perspective, and style implied by:

- the task type,
- the instruction template,
- the field-shape hint describing the structured input.

Output mode

If the hint contains exactly one item-field group named `item_fields` and no other item-field groups, use the *single-group mode*:

```
{"line_template": "...",
  "header": "...",
  "footer": "..."}
```

If the hint contains multiple item-field groups, use the *multi-group mode*:

```
{"<group1>_line_template": "...",
  "<group2>_line_template": "...",
  "...": "...",
  "header": "...",
  "footer": "..."}
```

Each group name must match the corresponding item-field group name in the hint exactly.

Template rules (strict)

Allowed placeholders

Only use placeholders that appear in:

- the list of allowed placeholders in the hint,
- global field names (for the header and footer),
- item-field names of the relevant group (for its line template).

Use placeholders only in the form `{field_name}`. Do not use scenario-specific names, aliases, or rewritten field names as placeholders. Do not invent any additional fields or placeholder names.

Line templates

For each item-field group:

- Include *all* placeholders from that group at least once.
- Do not include placeholders from other item groups or from global fields.
- Write a short, natural sentence describing exactly one item.
- Match the tone, perspective, and style of the instruction template.
- Do not introduce new meanings or constraints.

Header

If global fields exist:

- Include each global placeholder at least once.
- Exclude all item-level placeholders.
- Write the header as a direct continuation of the instruction template, as if it were the next line in the same explanation.
- Preserve the exact narrative perspective and tone of the instruction template.
- Keep the header concise and scenario-consistent.

If no global fields exist, the header must be an empty string.

Footer

The footer is optional and may be an empty string. If present:

- Do not include any item-level placeholders.
- Use global placeholders only if they also appear in the header.
- Maintain the same tone and narrative perspective as the instruction template.
- Provide a brief, natural closing remark.

Prompt 6: Prompt used to generate reusable item-level natural language templates that verbalize structured inputs while strictly preserving schema and narrative consistency.

Illustrative Contextualized Instance Example (CVRP)

We’re coordinating the day’s routes so a handful of carriers can hand over every piece of mail one time and then return to the post office. The choice is which streets and houses each carrier covers, keeping each route’s total mail within the capacity of that carrier’s bag. The goal is to keep the overall driving low: take each carrier’s route length, add them up, and the plan with the lowest total miles is best. The concrete stop-by-stop details appear below.

```
# total_stops_including_post_office=9
# post_office_stop_id=A
# carrier_mailbag_capacity=100
stop_id,map_x_coordinate,map_y_coordinate,mail_items_to_deliver
A,83,76,0
B,50,5,20
C,100,14,26
D,21,0,17
...
... (remaining stops omitted)
```

Also, when you send the planned routes back, please use this simple JSON shape so it's easy to read and parse:

```
{
  "solution": [[post_office_id, house_id, ... , post_office_id], [post_office_id, house_id, ...
  , post_office_id], ...]
}
```

This shows the general layout: "solution" is a list of routes, each inner list is the sequence of stops a single carrier will follow (starting and ending at the post office). It's just a sketch of the expected shape --- not the actual answer.

Please make sure to use the exact identifiers from the instance input --- do not rename them or invent new labels.
- for example: "Valid identifiers look like plain numbers such as “1” or “23”, single capital letters like “A” or “B”, or a capital letter followed by digits like “A1” or “X7”."

Table 11: Contextualized instance example under a postal-delivery narrative.

E Experiment Configurations

Parameter Settings. To ensure a fair evaluation across both reasoning and non-reasoning models, as well as reasoning models operating in non-reasoning modes, we describe our parameter settings below, which aim to allow each model to perform at its full capability.

For standard chat LLMs, we set the decoding temperature to 0. For reasoning-oriented LLMs, we do not specify a temperature. The maximum output token number is a crucial parameter for evaluation, as solving CO problems typically requires a large number of tokens for reasoning. Specifically, for DeepSeek and all proprietary LLMs, we evaluate them by calling their official APIs while setting the maximum output token to the maximum allowed value by their API service, i.e., 8,192 for DeepSeek-V3.2 (standard) and 64K for Claude-Sonnet-4.5 (with a thinking budget of 60k). For DeepSeek-V3.2 (reasoning), o4-mini (reasoning), GPT-5.1 (reasoning), and Gemini-3-Flash (reasoning), we do not specify the maximum output token, allowing the API to utilize its full supported output range. For Llama-4-Maverick-Instruct and Qwen3-235B-Instruct, we utilize the Google Vertex AI platform for LLM calling and set the maximum output token count to 8,192 and 16,384, respectively, i.e., the maximum allowed value. For other models, we use the API service of the OpenRouter Platform for LLM calling, and we set the maximum output token to the highest value allowed by the platform for each corresponding model: 16384 for Qwen3-14B, 65536 for Ministral3-14B, 65536 for Nemotron3-Nano-30B-A3B, 65536 for MiMo-V2-Flash, and 30K for Grok-4.1-Fast (reasoning).

For models that support explicit reasoning mode, we configure the reasoning-related parameters. Specifically, for Gemini-3-Flash, we set the thinking level to high. For OpenAI models, we enable reasoning and set the reasoning effort to high for o4-mini, and to medium for GPT-5.1.

Prompting Strategies. For all LLMs with explicit reasoning mode, we directly invoke the model with NLCO instances, as they will autonomously generate the thinking processes. For the standard chat LLMs, we append the prompt "Please think step by step, output your reasoning trace, and then output the JSON in the required format" to the textual instances, guiding the LLMs to generate the chain of thought for solving CO problems.

F Additional Results

Performance profiles. To complement aggregate statistics (e.g., ALOG) on a benchmark with various tasks, we also use **performance profiles** (Dolan and Moré, 2002), a standard tool in optimization for comparing methods across heterogeneous tasks without allowing a small number of difficult cases to dominate the overall picture. Let $f_{i,m}$ denote the objective returned by LLM model m on instance i (lower is better; infeasible outputs are set to $f_{i,m} = +\infty$), and let $f_i^{\min} = \min_m f_{i,m}$ be the best value achieved on instance i across all LLMs. Assuming objectives are nonnegative, we define the per-instance performance ratio $r_{i,m} = \frac{f_{i,m}+\varepsilon}{f_i^{\min}+\varepsilon}$ with a small $\varepsilon > 0$ to avoid division by zero, and plot for each model the cumulative fraction of instances where it is within a factor γ of the best (with N denotes the number of instances):

$$\rho_m(\gamma) = \frac{1}{N} \left|\{i : r_{i,m} \leq \gamma\}\right|, \qquad \gamma \geq 1.$$

A curve that is higher, especially near $\gamma = 1$, indicates a model that more consistently matches the strongest observed performance across instances.

With the performance profiles shown in Figure 5, which report the fraction of instances on which each LLM achieves an objective within a factor γ of the best observed value (and $\gamma = 1$ corresponds to best/tied-best), we find that instance size is the main separator: from Set-S (a) to Set-L (c), curves systematically shift downward and the gaps between models widen. This indicate that many approaches that look "competitive" on small instances stop being reliably near-best once the problem size grows. Two patterns consistently stand out:

- Heavy-tailed behavior of some LLMs: top-tier models (e.g., OpenAI GPT-5.1, and Gemini-3-Flash) have curves that rise steeply and plateau high even on Set-L, indicating high consistency. In contrast, several baselines (e.g., Llama-4-Maverick-Instruct, MiMo-V2-Flash, Ministral-3-14B) rise more gradually and plateau much lower, especially in (c), which is characteristic of heavy-tailed behavior: they sometimes find decent solutions but frequently fall far from the best.

- Reasoning helps more as scale increases: the benefit of reasoning-enabled variants is modest on Set-S (many models already reach

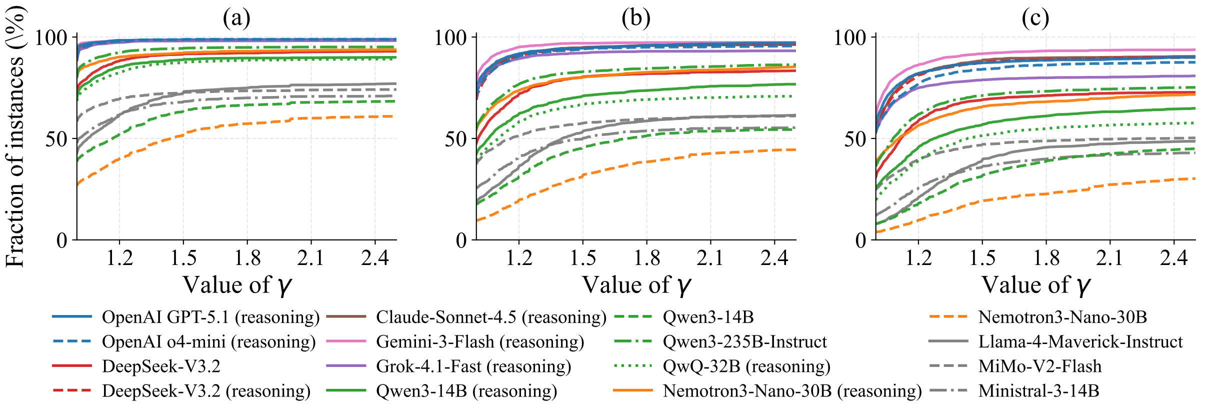


Figure 5: LLM performance profiles on different NLCO difficulty tiers. (a) Set-S; (b) Set-M; (c) Set-L.

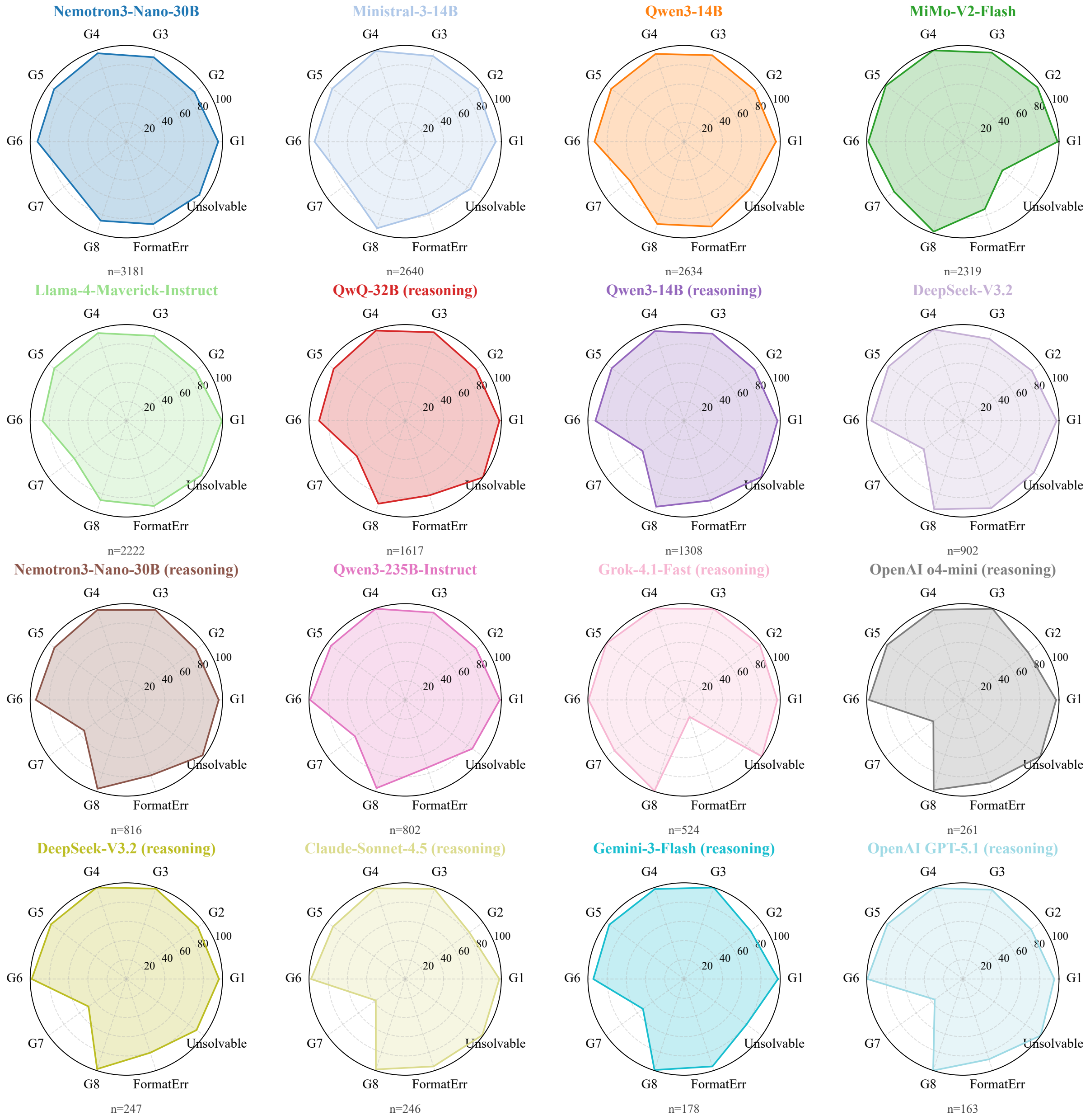


Figure 6: Per-model radar charts showing infeasibility mode distribution across LLMs; larger filled area indicates lower frequency of that mode, with n denoting sample count per model.

Problem	AFR			Acc.			tok.		
	Set-S	Set-M	Set-L	Set-S	Set-M	Set-L	Set-S	Set-M	Set-L
Traveling Salesman Problem	0.97	0.93	0.91	0.21	0.16	0.05	12.8K	15.3K	14.0K
Prize-Collecting TSP	0.93	0.96	0.93	0.47	0.15	0.03	13.8K	15.9K	16.6K
Orienteering Problem	0.89	0.85	0.83	0.76	0.52	0.40	10.6K	13.5K	15.1K
Capacitated VRP	0.96	0.94	0.85	0.42	0.09	0.02	11.2K	12.1K	11.9K
TSP with Time Windows	0.72	0.46	0.19	0.42	0.10	0.01	15.6K	19.1K	19.3K
Pickup-and-Delivery Problem	0.69	0.49	0.34	0.40	0.07	0.00	14.7K	16.8K	15.7K
Minimum Latency Problem	0.91	0.91	0.87	0.13	0.03	0.05	11.5K	13.5K	13.3K
Quadratic Shortest Path Problem	0.98	0.96	0.92	0.94	0.86	0.65	5.13K	7.28K	12.9K
Steiner Tree Problem	0.82	0.68	0.61	0.67	0.48	0.41	9.02K	15.0K	17.1K
Steiner Forest Problem	0.84	0.68	0.57	0.69	0.47	0.33	10.6K	16.8K	20.3K
k-Minimum Spanning Tree	0.88	0.74	0.62	0.76	0.60	0.41	10.1K	16.8K	19.5K
Bin Packing Problem	0.98	0.92	0.83	0.94	0.81	0.64	3.04K	6.71K	10.2K
Cutting Stock Problem	0.72	0.58	0.48	0.65	0.48	0.34	9.26K	15.5K	18.6K
2D Strip Packing	0.53	0.33	0.15	0.38	0.11	0.04	15.8K	20.6K	22.3K
Job-Shop Scheduling Problem	0.71	0.48	0.44	0.59	0.12	0.00	8.54K	15.2K	12.9K
Flow-Shop Scheduling Problem	0.80	0.58	0.49	0.74	0.20	0.01	6.37K	16.1K	15.6K
Open-Shop Scheduling Problem	0.69	0.53	0.45	0.28	0.06	0.01	8.35K	14.4K	13.9K
RCPSP (makespan)	0.60	0.38	0.31	0.56	0.27	0.13	6.98K	14.4K	17.5K
Parallel Machines $P \parallel T_{\max}$	0.96	0.89	0.81	0.81	0.46	0.19	6.70K	11.8K	14.4K
Single-Machine Total Weighted Tardiness	0.98	0.94	0.88	0.63	0.32	0.20	12.1K	14.6K	15.1K
Minimum Dominating Set	0.84	0.72	0.63	0.78	0.64	0.54	4.91K	8.86K	13.2K
Set Cover Problem	0.96	0.90	0.83	0.92	0.84	0.73	3.25K	7.06K	10.8K
Set Packing	0.88	0.83	0.83	0.86	0.79	0.75	10.2K	15.2K	17.9K
Set Partitioning Problem	0.80	0.65	0.44	0.68	0.41	0.12	11.2K	19.7K	25.4K
Hitting Set Problem	0.96	0.92	0.77	0.87	0.79	0.60	3.37K	6.41K	11.0K
Max k-Coverage	0.99	0.98	0.96	0.96	0.85	0.66	2.17K	7.87K	13.2K
Generalized Assignment Problem	0.96	0.92	0.77	0.84	0.66	0.37	4.35K	6.61K	10.5K
Uncapacitated Facility Location	0.99	0.96	0.91	0.93	0.78	0.56	2.88K	9.45K	15.6K
Capacitated Facility Location	0.99	0.97	0.90	0.90	0.70	0.47	3.02K	8.91K	13.9K
p-Median	0.97	0.95	0.87	0.90	0.70	0.44	2.79K	9.72K	15.3K
p-Center	0.96	0.95	0.88	0.92	0.73	0.54	2.85K	7.48K	12.6K
Maximum Independent Set	0.88	0.86	0.79	0.83	0.73	0.63	8.59K	12.2K	15.0K
Minimum Vertex Cover	0.86	0.77	0.66	0.80	0.72	0.54	9.15K	13.2K	17.7K
Maximum Clique Problem	0.93	0.89	0.85	0.90	0.83	0.78	4.40K	6.28K	7.65K
Knapsack Problem	0.96	0.92	0.83	0.85	0.68	0.52	6.57K	11.4K	14.5K
Maximum Diversity Problem	0.97	0.95	0.95	0.88	0.72	0.56	6.26K	9.81K	11.6K
Quadratic Knapsack Problem	0.94	0.90	0.85	0.75	0.48	0.29	8.65K	14.7K	16.1K
Maximum Cut Problem	0.97	0.95	0.94	0.60	0.37	0.16	11.7K	16.4K	16.0K
Three-Index Assignment	0.93	0.85	0.75	0.61	0.18	0.01	12.5K	18.7K	18.7K
Quadratic Assignment Problem	0.98	0.95	0.91	0.85	0.44	0.22	6.97K	14.6K	15.6K
Graph Coloring Problem	0.81	0.71	0.61	0.76	0.67	0.55	5.30K	8.69K	12.2K
Cutwidth Minimization Problem	0.98	0.96	0.92	0.84	0.69	0.47	6.47K	9.45K	13.2K
Linear Ordering Problem	0.96	0.94	0.89	0.51	0.16	0.02	13.4K	18.9K	20.5K

Table 12: Average feasibility, accuracy, and token usage across LLMs on 43 NLCO tasks.

high fractions under relaxed γ), but becomes pronounced on Set-M/L. For example, Nemotron3-Nano-30B (reasoning) is consistently above its non-reasoning counterpart across panels, and the gap is largest on Set-L, suggesting that additional structured computation is particularly valuable for large instances. A similar pattern appears within the Qwen family (e.g., Qwen3-14B (reasoning) vs. Qwen3-14B), where the reasoning variant retains a noticeably higher fraction of near-best solutions as γ tightens.

Per-model Infeasibility mode Distribution. According to Figure 6, most LLMs' infeasibility is concentrated in a small number of modes, rather than evenly spread. In particular, Global$_7$ is the most common sink (the most pronounced dip in several panels), indicating a persistent structural bottleneck. FormatError also varies substantially by model: some models have a noticeable dip (e.g., Grok) on the FormatErr axis (format compliance is a major source of infeasibility), while others show a much larger radius there, suggesting far fewer parsing/format failures. Finally, unsolvable appears model-dependent: for some LLMs it is relatively rare (large radius), whereas for others (e.g., Mimo-V2-Flash) it forms a clearer dip, consistent with token-budget exhaustion being a non-negligible cause of infeasibility in those systems.

Per-task Performance. We report the average performance of all LLMs on each NLCO task in Table 12, revealing substantial heterogeneity across problems. It can be found that:

1) **Feasibility can remain high while optimality collapses on routing tasks.** For classical routing problems, models often find feasible solutions but struggle to reach optimality, especially as size increases. For instance, TSP maintains high AFR (0.97/0.93/0.91 from Set-S/M/L) but exhibits very low Acc., suggesting that producing a valid tour is far easier than globally optimizing it. A similar pattern also holds for CVRP and MLP.

2) **Some tasks are "LLM-friendly" even at larger sizes.** Several problems retain both high feasibility and relatively strong optimality, suggesting that their structure admits more direct constructive reasoning or local verification. Examples include Quadratic Shortest Path (AFR 0.98/0.96/0.92; Acc. 0.94/0.86/0.65), Bin Packing (AFR 0.98/0.92/0.83; Acc. 0.94/0.81/0.64), and graph selection tasks such as Maximum Clique (AFR 0.93/0.89/0.85; Acc. 0.90/0.83/0.78), which remain relatively stable compared to routing/scheduling.

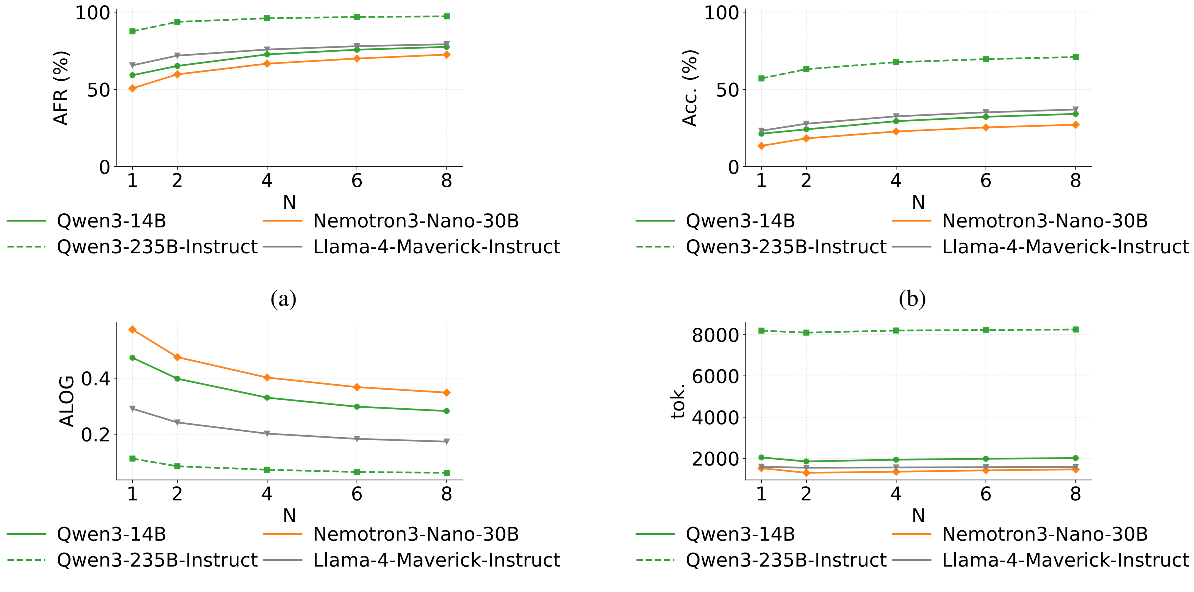
3) **Token usage broadly tracks output complexity and grows with size.** Across most tasks, tok. increases from Set-S to Set-L (e.g., Set Partitioning: 11.2K → 25.4K; Linear Ordering: 13.4K → 20.5K), reflecting longer solution representations and/or more extensive intermediate reasoning. Notably, high token usage does not guarantee high accuracy: several tasks remain expensive yet low-Acc. (e.g., TSPTW, 2SP), indicating that larger combinatorial spaces and tighter constraints can dominate additional inference-time computation.

Set	1-global	2-global	Diff.
Set-S	88.8%	88.1%	-0.7%
Set-M	80.9%	79.8%	-1.1%
Set-L	73.6%	70.7%	-2.9%

Table 13: AFR on NLCO tasks with different number of global constraint patterns. 1-global: tasks with 1 global patterns; 2-global: tasks with 2 global patterns. **Diff.** indicates the difference between two values (2-1)

The Impact of Global Pattern Numbers. From the NLCO taxonomy presented in Table 4, we can find that some CO problems are featured by one global pattern, while some are with two, which means that an LLM needs to make sure that the produced solutions for these problems are supposed to simultaneously satisfy two distinct types of global constraints. We aggregate the performance of all LLMs and report the average feasibility performance in both scenarios in Table 13. It can be observed that, on average, the LLMs achieve lower feasibility on CO tasks with 2 global patterns. Meanwhile, with the increase of problem scale (from Set-S to Set-L), the gap in AFR becomes more significant. This observation is intuitive: LLM reasoning is more likely to fail in a more constrained and combinatorial space.

Example Chain-of-thought (CoT) reasoning traces. Taking the instance shown in Table 14 as an example, we present the CoT processes of DeepSeek-V3.2 in Table 15, Table 16, and Table 17 for different tiers. Across the 3 difficulty tiers, the reasoning evolves from distance-explicit, savings-inspired manual optimization (Set-S) to capacity-feasible clustering with light cross-route refinement (Set-M), and finally to scalable geographic decomposition with pragmatic route splitting (Set-L). In



(a) (b)

(c) (d)

Figure 7: Best-of-N performance scaling across four models. We report (a) AFR (%), (b) Acc. (%), (c) ALOG, and (d) tok. (of the selected candidate solution) as functions of N.

Set-S, the model computes many explicit Euclidean distances, identifies geographic clusters, and uses Clarke–Wright–style "savings" intuition to justify linking nearby customers far from the depot; it then iteratively improves by reordering stops within a route and explicitly comparing the two-route solution against a three-route alternative, making the reasoning strongly numerical and improvement-driven. In Set-M, the model prioritizes feasibility first by constructing two capacity-balanced groups, then performs a small cross-route move (swapping a customer between vans) guided by approximate route-length calculations; the reasoning is less exhaustive than Set-S (fewer pairwise checks) and reads like a construct-then-refine heuristic rather than a detailed savings narrative. In Set-L, the model shifts to a more scalable strategy: it starts with coarse geographic clustering and distance-to-depot cues, assigns customers to routes under a larger capacity, and then introduces an additional route by splitting a spatially dispersed cluster to avoid extreme detours; the emphasis is on managing spatial spread and tour complexity rather than detailed distance arithmetic, reflecting a transition from local numerical optimization to higher-level decomposition and route-structure control.

Scaling Inference-time Computation. A natural question is whether increasing inference-time computation can improve reasoning outcome (i.e., solution feasibility and optimality), without changing the model or training. We therefore evaluate a simple compute-scaling strategy: Best-of-N sampling, where we generate N independent candidate solutions per instance and select the best one (feasibility first, then ALOG metric among feasible solutions). This isolates the effect of allocating more test-time compute to search over candidates.

Figure 7 shows consistent benefits as N increases. First, feasibility improves monotonically: even moderate sampling (e.g., $N{=}2$) yields clear gains, indicating that additional rollouts can reduce the chance of producing an invalid solution. Second, solution quality also improves, suggesting that compute scaling helps not only "get a feasible answer" but also "get a better one." Meanwhile, the gains exhibit diminishing returns: most improvements occur at small N, while the curves flatten as N grows, implying a practical knee point for trading off compute and performance. Finally, Figure 7 (d) shows that the token usage of the selected solution is largely stable across N, so the improvements are primarily driven by exploring more CO solutions rather than producing longer single outputs; however, the total inference cost still scales roughly linearly with N due to multiple generations.

Overall, best-of-N provides a simple and effective way to scale inference-time computation, improving both feasibility and optimality, but it does not eliminate the performance gap between LLMs: stronger models remain ahead across all N.

Problem Overview

Task: Capacitated Vehicle Routing Problem (CVRP)
Instance tiers: Set-S / Set-M / Set-L
Instance format: Someone from the store is lining up the drivers and thinking about how to split the work: every driver should start at the shop, cover a set of houses with their requested groceries exactly once (nothing can be skipped or visited twice), and return to the shop without exceeding the van's load limit. To pick the best way, add up the distance each van travels (using Euclidean distance between coordinates) and choose the arrangement with the smallest total distance. The concrete list of locations, coordinates, grocery demands, and the van capacity follows below. There are N locations in total; the shop is node `A`, and each van can carry up to C units.
`<INSTANCE_DATA>`
Assign routes so that: (i) each route starts and ends at the shop `A`; (ii) every house `B-J` is visited exactly once across all routes; (iii) the total demand on each route never exceeds C; and (iv) the sum of travel distances across all routes is minimized.
Also, to keep things tidy, when you send back the routing plan just stick to a simple JSON shape like this so I can read it straight away:

```
{ "solution": [[shop_id, house_id, ... , shop_id],
               [shop_id, house_id, ... , shop_id],
               ...] }
```

Pretty much: `"solution"` is the outer list of routes; each inner list is one route that *starts and ends* with the `shop_id` (here, `A`) and lists the visited `house_id` values in travel order. It's just an example of the shape I expect—don't treat these placeholders as real answers. Please use the exact identifiers from the instance input—no renaming and no made-up labels. For example: "Valid identifiers look like plain numbers such as `1` or `23`, single capital letters like `A` or `B`, or a capital letter followed by digits like `A1` or `X7`." Please think step by step, output your reasoning trace, and then output the JSON in the required format.

Table 14: CVRP instance example used for LLM evaluation.

CoT (Set-S / CVRP example)

```
Alright, let's go step by step.

---

1. Understanding the problem

We have:
- Shop: node A (0, 71), demand 0.
- Houses: B, C, D, E, F, G, H, I, J, each with given coordinates and demand.
- Van capacity: 100 units.
- Goal: Partition houses into routes (each starting and ending at A) so that total demand per
    route <= 100, and total Euclidean travel distance is minimized.
- Output: JSON list of routes, each route like ["A", ... ,"A"].

---

2. Data table

From the table:

| Node | x | y | demand |
|------|---|---|--------|
| A | 0 | 71 | 0 |
| B | 62 | 6 | 12 |
```

C	71	8	24
D	62	90	3
E	76	9	14
F	87	59	19
G	70	0	4
H	100	31	24
I	75	100	19
J	68	98	2

Total demand (excluding A):
12 + 24 + 3 + 14 + 19 + 4 + 24 + 19 + 2 = 121 units.

Since capacity = 100, we need at least 2 vans.
Possible to do in 2 vans because 121 <= 200.

---

3. Euclidean distances

Let's compute distances between all points (rounded to 2 decimals for reasoning, but final route choice will be based on minimizing total).

From A (0,71):
- A->B: sqrt((62-0)^2 + (6-71)^2) = sqrt(3844 + 4225) = sqrt(8069) ~= 89.83
- A->C: sqrt(71^2 + (8-71)^2) = sqrt(5041 + 3969) = sqrt(9010) ~= 94.92
- A->D: sqrt(62^2 + (90-71)^2) = sqrt(3844 + 361) = sqrt(4205) ~= 64.85
- A->E: sqrt(76^2 + (9-71)^2) = sqrt(5776 + 3844) = sqrt(9620) ~= 98.08
- A->F: sqrt(87^2 + (59-71)^2) = sqrt(7569 + 144) = sqrt(7713) ~= 87.82
- A->G: sqrt(70^2 + (0-71)^2) = sqrt(4900 + 5041) = sqrt(9941) ~= 99.70
- A->H: sqrt(100^2 + (31-71)^2) = sqrt(10000 + 1600) = sqrt(11600) ~= 107.70
- A->I: sqrt(75^2 + (100-71)^2) = sqrt(5625 + 841) = sqrt(6466) ~= 80.41
- A->J: sqrt(68^2 + (98-71)^2) = sqrt(4624 + 729) = sqrt(5353) ~= 73.16

Between houses (just note clusters):
- Lower-left cluster: B(62,6), C(71,8), E(76,9), G(70,0) -- all near bottom-left of grid, far from A horizontally but far in y from A.
- Upper cluster: D(62,90), I(75,100), J(68,98) -- near top. Actually A is at y=71, so D is at y =90 (19 vertical), I at y=100 (29 vertical), J at y=98 (27 vertical). D is closest to A among these.
- Middle-right: F(87,59), H(100,31).

---

4. Demand grouping for capacity

We want to minimize total travel, so likely group geographically.

Cluster 1 (Lower-left): B(12), C(24), E(14), G(4) -> total = 54 units. Could add more if nearby.
Cluster 2 (Upper): D(3), I(19), J(2) -> total = 24 units. Could add F or H if they fit geographically.
Cluster 3 (Middle-right): F(19), H(24) -> total = 43 units. But we must assign all to vans, possibly 2 vans only.

Check if 2 vans possible:
Van1: Cluster1 (54) + maybe D (3) = 57, still room.
Van2: Cluster3 (43) + Cluster2 without D (I,J = 21) = 64, total 64.

That leaves D unassigned in that scheme, but D is close to A and I,J, so maybe:
Better: Van1: B,C,E,G (54) + F (19) = 73 (F is not too far from lower-left? Actually F is 87,59 -- far from B,C,E,G in x but y=59 is between top and bottom). Might not be good geographically.

Let's think of savings heuristic (Clarke-Wright).

---

5. Clarke-Wright savings intuition

Savings s(i,j) = d(A,i) + d(A,j) - d(i,j).
We want to merge routes if savings high and capacity allows.

Let's compute a few key savings (approx):

B and C:
d(A,B)~=89.83, d(A,C)~=94.92,
d(B,C)=sqrt((71-62)^2+(8-6)^2)=sqrt(81+4)=sqrt(85)~=9.22
Savings ~= 89.83+94.92-9.22 ~= 175.53 -> huge because B and C are very close.

B and G:
d(B,G)=sqrt((70-62)^2+(0-6)^2)=sqrt(64+36)=sqrt(100)=10
Savings ~= 89.83+99.70-10 ~= 179.53 -> even bigger.

C and E:
d(C,E)=sqrt((76-71)^2+(9-8)^2)=sqrt(25+1)=sqrt(26)~=5.10
Savings ~= 94.92+98.08-5.10 ~= 187.90 -> huge.

So B,C,E,G all close to each other, far from A -> big savings if linked.

Upper cluster:
D and J:
d(D,J)=sqrt((68-62)^2+(98-90)^2)=sqrt(36+64)=sqrt(100)=10
d(A,D)~=64.85, d(A,J)~=73.16
Savings ~= 64.85+73.16-10 ~= 128.01

D and I:
d(D,I)=sqrt((75-62)^2+(100-90)^2)=sqrt(169+100)=sqrt(269)~=16.40
Savings ~= 64.85+80.41-16.40 ~= 128.86

I and J:
d(I,J)=sqrt((68-75)^2+(98-100)^2)=sqrt(49+4)=sqrt(53)~=7.28
Savings ~= 80.41+73.16-7.28 ~= 146.29

So upper cluster D,I,J also good to link.

F and H:
d(F,H)=sqrt((100-87)^2+(31-59)^2)=sqrt(169+784)=sqrt(953)~=30.87
d(A,F)~=87.82, d(A,H)~=107.70
Savings ~= 87.82+107.70-30.87 ~= 164.65 -> big, so F and H should be together.

---

6. Try feasible 2-route partition

We need each route <= 100 demand.

Option 1:
Route1: B(12), C(24), E(14), G(4) = 54, add F(19) = 73, add H(24) = 97 (within 100).
That leaves D,I,J for other route: D(3)+I(19)+J(2)=24, well under 100.

Check distances:
Route1: A -> B -> C -> E -> G -> F -> H -> A? Or better order to minimize distance.
Route2: A -> D -> I -> J -> A.

Option 2:
Route1: B,C,E,G,F = 73
Route2: D,I,J,H = 48 (3+19+2+24=48)
But H is far from D,I,J, so maybe not good.
Better: Route1: B,C,E,G,H = 78 (H linked to G? far)
Route2: D,I,J,F = 43 (F far from D,I,J) -> seems worse.

So Option 1 seems better: Route1: Lower-left + F + H; Route2: Upper cluster (D,I,J).

---

7. Optimize route order within each

Route1: A -> ... -> A with B,C,E,G,F,H.

Start from A: A to nearest among them?
A to B=89.83, A to G=99.70, A to F=87.82, A to H=107.70, A to C=94.92, A to E=98.08.
Actually F is closest to A in this group (87.82), but F is in middle-right, not close to B,C,E,G.

Better to start with cluster B,C,E,G then go to F then H then back to A.

Try sequence: A -> B -> C -> E -> G -> F -> H -> A.

Compute total:
A->B: 89.83
B->C: 9.22
C->E: 5.10
E->G: sqrt((70-76)^2+(0-9)^2)=sqrt(36+81)=sqrt(117)~=10.82
G->F: 61.40
F->H: 30.87
H->A: 107.70
Sum ~= 314.94

Route2: A -> D -> I -> J -> A.
A->D: 64.85
D->I: 16.40
I->J: 7.28
J->A: 73.16
Sum ~= 161.69

Total both routes ~= 314.94+161.69=476.63

---

8. Try to improve Route1 order

Maybe A -> G -> B -> C -> E -> F -> H -> A.

A->G: 99.70
G->B: 10.00
B->C: 9.22
C->E: 5.10
E->F: sqrt((87-76)^2+(59-9)^2)=sqrt(121+2500)=sqrt(2621)~=51.20
F->H: 30.87
H->A: 107.70
Sum ~= 313.79 (slightly better than 314.94)

Total with Route2 (161.69) ~= 475.48

Try A -> G -> B -> C -> E -> H -> F -> A:

A->G: 99.70
G->B: 10.00
B->C: 9.22
C->E: 5.10
E->H: sqrt((100-76)^2+(31-9)^2)=sqrt(576+484)=sqrt(1060)~=32.56
H->F: 30.87
F->A: 87.82
Sum ~= 275.27 (much better!)

Check capacity still OK:
B,C,E,G,H,F = 12+24+14+4+24+19=97.

Route2 same: D,I,J = 24 units.

Total ~= 275.27 + 161.69 = 436.96 (big improvement).

---

9. Check if 3 routes could be shorter

If we split into 3 routes, each shorter distances but more A-trips.

Example:
R1: B,C,E,G = 54
A->B->C->E->G->A:
A->B: 89.83
B->C: 9.22
C->E: 5.10
E->G: 10.82
G->A: 99.70
Total ~= 214.67

R2: F,H = 43
A->F->H->A:
A->F: 87.82
F->H: 30.87
H->A: 107.70
Total ~= 226.39

R3: D,I,J = 24
A->D->I->J->A:
Total ~= 161.69

Total ~= 214.67+226.39+161.69=602.75 -> worse than 436.96.
So 2 routes better.

---

10. Final best found

Route1: A -> G -> B -> C -> E -> H -> F -> A (demand 97)
Route2: A -> D -> I -> J -> A (demand 24)
Total distance ~= 436.96.

Final JSON:
```
{
  "solution": [
    ["A","G","B","C","E","H","F","A"],
    ["A","D","I","J","A"]
  ]
}
```

Table 15: DeepSeek CoT trace on a Set-S CVRP instance.

CoT (Set-M / CVRP example)

1. Understanding the problem

We have:

- Shop: node A (demand 0)
- Houses: B, C, D, E, F, G, H, I, J, K (10 houses total)
- Each has a demand in units (given in table).
- Van capacity: 100 units.
- Euclidean distances between coordinates (since coordinates given).
- Need to partition houses into routes starting and ending at A, total demand per route <= 100, minimize total travel distance (sum of lengths of all routes).
- Output: list of routes, each route as [A, ... houses ..., A].

---

2. Data extraction

Coordinates (x, y):

A: (87, 24)
B: (8, 13)
C: (84, 43)
D: (0, 52)
E: (55, 100)
F: (88, 43)
G: (61, 39)
H: (0, 98)
I: (97, 31)
J: (100, 6)
K: (84, 0)

Demands:
B: 23
C: 12
D: 19
E: 19
F: 17
G: 8
H: 12
I: 18
J: 18
K: 21

Total demand = 23+12+19+19+17+8+12+18+18+21 = 167 units.

Since van capacity = 100, minimum number of vans = ceil(167/100) = 2 vans.
But because demands are not huge, maybe 2 vans possible if split well.

---

3. Check if 2 vans possible

We need to partition 10 houses into 2 groups, each <= 100 units.

Total 167 means if one van carries X, the other carries 167-X <= 100 -> X >= 67.
So each van must carry between 67 and 100 units.

Largest demands: B(23), D(19), E(19), F(17), I(18), J(18), K(21), C(12), G(8), H(12).

Try to make two groups ~83.5 each.

Group 1: B(23) + D(19) + E(19) + H(12) = 73, add C(12) -> 85, add G(8) -> 93.
That's: B, D, E, H, C, G = 93 units.

Remaining: F(17), I(18), J(18), K(21) = 74 units.

So yes, 2 vans possible:
Van 1: 93 units (B, D, E, H, C, G)
Van 2: 74 units (F, I, J, K)

---

4. Routing for minimal distance

We need to sequence each route starting and ending at A to minimize total distance.

Van 1: B, D, E, H, C, G (93 units)

Coordinates:
A(87,24), B(8,13), D(0,52), E(55,100), H(0,98), C(84,43), G(61,39).

Try a reasonable path:
A -> B -> D -> H -> E -> G -> C -> A.

Compute distances roughly (Euclidean):

d(A,B) = sqrt((87-8)^2 + (24-13)^2) = sqrt(79^2 + 11^2) = sqrt(6241+121)= sqrt(6362) ~= 79.76
d(B,D) = sqrt((8-0)^2 + (13-52)^2) = sqrt(64 + 1521) = sqrt(1585) ~= 39.81
d(D,H) = sqrt((0-0)^2 + (52-98)^2) = 46
d(H,E) = sqrt((55-0)^2 + (100-98)^2) = sqrt(3025+4)= sqrt(3029) ~= 55.04
d(E,G) = sqrt((61-55)^2 + (39-100)^2) = sqrt(36+3721)= sqrt(3757) ~= 61.30
d(G,C) = sqrt((84-61)^2 + (43-39)^2) = sqrt(529+16)= sqrt(545) ~= 23.35
d(C,A) = sqrt((87-84)^2 + (24-43)^2) = sqrt(9+361)= sqrt(370) ~= 19.24

Total ~= 79.76+39.81+46+55.04+61.30+23.35+19.24 = 324.5

Van 2: F, I, J, K (74 units)

Coordinates:
F(88,43), I(97,31), J(100,6), K(84,0), A(87,24).

Good order: A -> K -> J -> I -> F -> A.

d(A,K) = sqrt((87-84)^2 + (24-0)^2) = sqrt(9+576)= sqrt(585) ~= 24.19
d(K,J) = sqrt((100-84)^2 + (6-0)^2) = sqrt(256+36)= sqrt(292) ~= 17.09
d(J,I) = sqrt((97-100)^2 + (31-6)^2) = sqrt(9+625)= sqrt(634) ~= 25.18
d(I,F) = sqrt((88-97)^2 + (43-31)^2) = sqrt(81+144)= sqrt(225)=15
d(F,A) = sqrt((87-88)^2 + (24-43)^2) = sqrt(1+361)= sqrt(362) ~= 19.03

Total ~= 24.19+17.09+25.18+15+19.03 = 100.49

---

5. Check if swapping houses between vans reduces total distance

Van 1 length ~= 324.5, van 2 ~= 100.5, total ~= 425.

Move C from van 1 to van 2:
Van 1: B, D, E, H, G (23+19+19+12+8 = 81)
Van 2: F, I, J, K, C (17+18+18+21+12 = 86)

Van 2 route: A -> C -> F -> I -> J -> K -> A:
d(A,C)=19.24
d(C,F)= sqrt((88-84)^2+(43-43)^2)=4
d(F,I)=15
d(I,J)=25.18
d(J,K)=17.09
d(K,A)=24.19
Sum ~= 19.24+4+15+25.18+17.09+24.19 = 104.7

Van 1 without C: A -> B -> D -> H -> E -> G -> A:
d(G,A)= sqrt((87-61)^2+(24-39)^2)= sqrt(676+225)= sqrt(901) ~= 30.02
Sum ~= 79.76+39.81+46+55.04+61.30+30.02 = 311.93

Total ~= 311.93+104.7 = 416.63 (better than 425).

---

6. Final routes after improvement

Van 1: B, D, E, H, G (81 units)
Route: A -> B -> D -> H -> E -> G -> A
Length ~= 311.93

Van 2: F, I, J, K, C (86 units)
Route: A -> C -> F -> I -> J -> K -> A
Length ~= 104.7

Total ~= 416.63

---

7. Final answer in required JSON

```
{
  "solution": [
    ["A","B","D","H","E","G","A"],
    ["A","C","F","I","J","K","A"]
  ]
}
```

Table 16: DeepSeek CoT trace on a Set-M CVRP instance.

CoT (Set-L / CVRP example)

Alright, let's go step by step.

Step 1 – Understanding the problem

We have:

- Shop node ID: `A` (coordinates 43, 54), demand 0.
- Total locations: 19 (shop + 18 houses).
- Van capacity: 150 units.
- Each house has a demand (in units).
- We need to partition the 18 houses into routes, each starting and ending at `A`, total demand per route <= 150.
- Objective: minimize total Euclidean distance traveled by all vans.
- Output: list of routes in JSON format.

Step 2 – Data summary

From the table:

ID	x	y	demand
A	43	54	0
B	76	68	30
C	26	32	9
D	60	38	21
E	28	65	15
F	21	90	19
G	81	43	11
H	79	24	5
I	53	16	10
J	14	11	9
K	5	73	10
L	3	51	28
M	38	98	7
N	43	67	15
O	55	100	11
P	97	43	26
Q	100	100	6
R	0	0	7
S	0	92	11

Total demand (excluding A):
30+9+21+15+19+11+5+10+9+10+28+7+15+11+26+6+7+11 = 250 units.

Minimum number of vans (if perfectly packed): ceil(250/150) = 2 vans, but likely more because of spatial spread.

Step 3 – Initial clustering by geography

Let's group houses by rough region relative to shop `A` (43,54):

1. North-East (higher x, higher y): B(76,68), O(55,100), Q(100,100), M(38,98), N(43,67), maybe F (21,90) is more north-west.
2. East (higher x, near y=54): G(81,43), P(97,43), H(79,24), D(60,38), I(53,16).
3. West (low x): C(26,32), J(14,11), R(0,0), L(3,51), K(5,73), S(0,92), F(21,90), E(28,65), M (38,98) -- but M is north of A.

Actually, let's just compute distances from A to see natural clusters.

Step 4 – Distances from A (approx Euclidean)

Let's compute roughly:

- A to B: dx=33, dy=14 -> dist ~= 35.8
- A to C: dx=-17, dy=-22 -> dist ~= 27.8
- A to D: dx=17, dy=-16 -> dist ~= 23.3
- A to E: dx=-15, dy=11 -> dist ~= 18.6
- A to F: dx=-22, dy=36 -> dist ~= 42.0
- A to G: dx=38, dy=-11 -> dist ~= 39.6
- A to H: dx=36, dy=-30 -> dist ~= 46.9
- A to I: dx=10, dy=-38 -> dist ~= 39.4
- A to J: dx=-29, dy=-43 -> dist ~= 51.5
- A to K: dx=-38, dy=19 -> dist ~= 42.5
- A to L: dx=-40, dy=-3 -> dist ~= 40.1
- A to M: dx=-5, dy=44 -> dist ~= 44.3
- A to N: dx=0, dy=13 -> dist ~= 13.0
- A to O: dx=12, dy=46 -> dist ~= 47.6
- A to P: dx=54, dy=-11 -> dist ~= 55.1
- A to Q: dx=57, dy=46 -> dist ~= 73.4
- A to R: dx=-43, dy=-54 -> dist ~= 68.9
- A to S: dx=-43, dy=38 -> dist ~= 57.4

Step 5 – Grouping for TSP-like routes

We can try to form clusters by proximity and capacity.

Cluster 1 (North-East): B(30), N(15), O(11), M(7), Q(6) → total = 69. Add F(19) → 88. Add E(15) → 103. Add S(11) → 114. Add K(10) → 124. Still under 150.
But S, K are far west, so maybe not good for distance.

Better:
Route 1 (North of A): N(15), E(15), M(7), O(11), B(30) → total 78. Could add F(19) → 97. Could add S(11) → 108. But S is far from B/O.

Actually, let’s try geographic clusters:

West group: R(7), J(9), C(9), L(28), K(10), S(11), F(19), E(15), M(7) → sum = 115. That’s one route under 150. But spatially huge spread --- maybe split.

Better:
West-South: R, J, C, L, K → sum 7+9+9+28+10 = 63. Add S(11) → 74. Add F(19) → 93. Add E(15) → 108. Add M(7) → 115. Still okay. But visiting all in one route is long in distance.

Given time, I’ll try a sensible manual split minimizing distance:

Route 1 (Close North/East): A → N(15) → E(15) → M(7) → O(11) → B(30) → A.
Demand = 78. Could add more but B is far from O? Actually O to B: $\Delta x = 21,\ \Delta y = -32 \Rightarrow \text{dist} \sim 38$. Not too bad.

Route 2 (East): A → D(21) → G(11) → P(26) → H(5) → I(10) → A.
Demand = 73. Could add Q(6) → 79, but Q is far from I/H. Q is near O/B but already in Route 1? No, Q not in Route 1 yet. Q is far from A though.

Route 3 (West): A → C(9) → L(28) → K(10) → S(11) → F(19) → A.
Demand = 77. Add E(15)? E is in Route 1 already. Add M? M in Route 1. Add J(9) → 86. Add R(7) → 93.

Actually, let's systematically assign:

Step 6 – Assign with capacity constraint

We need all 18 houses covered exactly once.

Try clustering:

Cluster A (North): N, E, M, O, B, F, S, K
Sum = 15+15+7+11+30+19+11+10 = 118.
This is compact? Not really --- S,K are far west from B,O. Might be inefficient.

Better:
Cluster 1 (North-East): N(15), E(15), M(7), O(11), B(30), Q(6) → total 84.
Cluster 2 (East): D(21), G(11), P(26), H(5), I(10) → total 73.
Cluster 3 (West): C(9), L(28), K(10), S(11), F(19), J(9), R(7) → total 93.

Check all houses:
We have N,E,M,O,B,Q,D,G,P,H,I,C,L,K,S,F,J,R → 18 houses. Yes.

Step 7 – Route ordering within clusters

We want to sequence to minimize distance.

Route 1 (North-East): Start A → N(43,67) → E(28,65) → M(38,98) → O(55,100) → Q(100,100) → B(76,68) → A.
Check: N→E is close, E→M is north, M→O is east, O→Q is far east, Q→B is southwest, B→A is southwest. Might be improved: maybe A → N → E → B → O → M → Q → A? Let's keep simple for now.

Route 2 (East): A → D(60,38) → G(81,43) → P(97,43) → H(79,24) → I(53,16) → A.
Seems fine.

Route 3 (West): A → C(26,32) → L(3,51) → K(5,73) → S(0,92) → F(21,90) → J(14,11) → R(0,0) → A.
This is very spread out --- maybe split into 2 routes for shorter travel.

Step 8 – Split West cluster into 2 routes for lower distance

West cluster demand 93, but visiting R(0,0) after F(21,90) is a huge detour.

Better:
Route 3a (West-North): A → C(26,32) → L(3,51) → K(5,73) → S(0,92) → F(21,90) → A.
Demand = 9+28+10+11+19 = 77.
Route 3b (West-South): A → J(14,11) → R(0,0) → A.
Demand = 9+7=16. Very short route but maybe merge with another? Could merge J,R into Route 2? But Route 2 is east, J,R are far west --- increases distance a lot.

Maybe merge J,R into Route 3a? Then Route 3a: C,L,K,S,F,J,R → total 93. That's fine for capacity but bad for TSP length.

Given the trade-off, maybe keep 4 routes total:

Route 1: N, E, M, O, B, Q (84)

Route 2: D, G, P, H, I (73)
Route 3: C, L, K, S, F (77)
Route 4: J, R (16)

That covers all: N,E,M,O,B,Q,D,G,P,H,I,C,L,K,S,F,J,R → 18.

Step 9 – Optimize sequence within each route

Route 1: A → N(43,67) → E(28,65) → M(38,98) → O(55,100) → Q(100,100) → B(76,68) → A.
Maybe better: A → N → E → B → O → M → Q → A? Let's pick a reasonable order:
A → N → E → B → O → M → Q → A.
Check: N→E close, E→B far? E(28,65) to B(76,68): $\Delta x = 48,\ \Delta y = 3,$ dist $\sim 48.1.$
E→M is closer? E to M: $\Delta x = 10,\ \Delta y = 33,$ dist $\sim 34.5.$
So maybe A → N → E → M → O → Q → B → A.

Let's finalize:
A → N → E → M → O → Q → B → A.

Route 2: A → D → G → P → H → I → A.

Route 3: A → C → L → K → S → F → A.

Route 4: A → J → R → A.

Step 10 – Final check

All houses assigned exactly once, each route ≤ 150 capacity.

Final routes:

1. ["A","N","E","M","O","Q","B","A"]
2. ["A","D","G","P","H","I","A"]
3. ["A","C","L","K","S","F","A"]
4. ["A","J","R","A"]

```
{
  "solution": [
    ["A", "N", "E", "M", "O", "Q", "B", "A"],
    ["A", "D", "G", "P", "H", "I", "A"],
    ["A", "C", "L", "K", "S", "F", "A"],
    ["A", "J", "R", "A"]
  ]
}
```

Table 17: DeepSeek CoT trace on a Set-L CVRP instance.