TEACHING LLMS TO GENERATE CHALLENGING MILP INSTANCES VIA SOLVER FEEDBACK

Anonymous authors
Paper under double-blind review

ABSTRACT

Generating optimization instances that are both feasible and computationally challenging is crucial for benchmarking solvers and training learning-based optimization algorithms. Existing non-LLM generators rely on seed instances or parameter tuning, resulting in high test-time computational cost, while existing LLM generators lack explicit hardness measures. Recent reinforcement learning methods with verifier feedback evaluate only binary correctness, which is misaligned with generating challenging problems. We note that an optimization solver reports the cost of solving at several stages of its pipeline, and leverage this to design a reward that scores both the solvability and the hardness of generated problems, measured by branch-and-bound nodes and post-cut relaxation gaps. Our key idea is a challenger-solver asymmetric self-play approach, where an LLM challenger generates progressively harder instances and the solver verifies feasibility and hardness, so no seed or training MILP instances are required. We fine-tune Gemma-4-12B and Qwen3.5-4B with GRPO and a size curriculum into OptiScribe-12B and OptiScribe-4B, which generate feasible yet challenging MILP problems from natural language instructions. On capacitated facility location and max-cut, OptiScribe-12B raises median SCIP search nodes by 1.7–5$\times$ and post-cut gaps by 1.1–1.7$\times$ over its base model and improves the feasibility rate on facility location by 9–19 points, while OptiScribe-4B raises median nodes by up to 15.6$\times$. The problems cover a wider difficulty range than public benchmarks of the same size, follow instructions on density and difficulty, and can tune solver settings for families that public libraries lack. These results indicate that optimization-specific rewards, used in self-play mode, can teach LLMs to generate high-difficulty optimization benchmarks. We will release our code and models publicly on acceptance.

1 INTRODUCTION

Generating hard test instances for a given class of optimization problems has long been recognized as a problem in its own right (Hooker, 1995; Selman et al., 1996; Smith-Miles & Bowly, 2015). Hardness is usually known only after a solver has run, and instances produced by naive random generation are mostly easy (Cheeseman et al., 1991; Mitchell et al., 1992). For mixed-integer linear programs (MILPs), which model scheduling, logistics, network design and energy systems, such instances are in practical demand. Solver developers need large and diverse instance sets for benchmarking and stress tests (Gleixner et al., 2021), and learning-augmented solvers need training instances from the right distribution (Gasse et al., 2019; Bengio et al., 2021). Yet public collections cover only a narrow slice of problem types and sizes, and at a given size their instances tend to be either trivial or very hard, with little in between (Section 4.2). Solver effort, the usual proxy for hardness, can be increased artificially by making instances larger, loosening big-M constraints, or scaling coefficients badly, without making the problem structurally harder. A generator of hard MILP instances must therefore find where hardness lives without taking these shortcuts.

Although MILP instance generators have drawn growing attention (Bowly, 2019; Geng et al., 2023; Li et al., 2025), most do not learn what makes an instance hard. Their difficulty is set by hand, inherited from seed instances, or imposed by search after generation. Hand-written random generators, widely used in learning-based work, often yield small or easy instances (Prouvost et al., 2020; Huang et al., 2024). Seed-based methods edit given instances and aim to preserve their difficulty

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

(Geng et al., 2023; Guo et al., 2024; Liu et al., 2024), so their outputs stay close to the seeds and cannot be produced without them. LLM-based methods evolve or retrieve generator programs (Li et al., 2025; Yang et al., 2025), but the LLM itself is never trained on hardness. Their difficulty comes from a solver-in-the-loop parameter search that must be rerun for each new class and size. A new family or size therefore needs new seeds, a new search or both.

Our starting point is that a MILP solver does more than return an answer. It also records how hard it worked, in branch-and-bound nodes explored and relaxation gap closed by cutting planes. We use this record to train a language model to write hard instances, which then needs no solver call at generation. Given a short prompt naming a MILP family and a target size, the model acts as a challenger and emits an instance in a compact index-set template. Training uses no instances beyond one formatting exemplar, which is never edited. A frozen branch-and-cut solver acts as verifier. The loop resembles challenger-solver self-play (Zhao et al., 2025; Huang et al., 2026; Dong & Ma, 2025), where difficulty is defined by a co-trained solver whose skill shifts during training. Our verifier is fixed instead, so hardness is measured on one scale across checkpoints and base models.

We propose a reward designed to prevent the model from inflating solver effort without making the problem harder. A multiplicative validity gate requires a parseable, feasible instance with a finite optimum and rules out unbounded continuous variables, aggregated big-M links, and extreme coefficient ranges. Hardness is scored from branch-and-bound nodes and the post-cut root gap, both read after root cuts (Achterberg, 2009), so weaknesses that cutting planes repair earn little and difficulty confined to solvers with weaker cuts is largely not rewarded. A size term keeps instances near the requested size, so hardness cannot be bought with scale. Since hardness learned against one solver may still reflect its blind spots, we re-solve every instance at evaluation with HiGHS and the commercial solver Gurobi, neither of which enters training. We train with GRPO (Shao et al., 2024) and a curriculum that increases the variable count over time.

We train OPTISCRIBE-12B models from `Gemma-4-12B-it` (Gemma Team et al., 2026) and OPTISCRIBE-4B models from `Qwen3.5-4B` (Qwen Team, 2025). On capacitated facility location (CFL) and max-cut, both models generate harder instances than their base models. For OPTISCRIBE-12B, median SCIP branch-and-bound nodes rise by 1.7–5.0× on CFL and 1.9–4.5× on max-cut, median post-cut root gaps rise by 1.1–1.7×, and the feasible rate on CFL improves by 9–19 points. The ordering of models holds at sizes larger than those seen in training and under the two held-out solvers. Our generator also produces instances across the difficulty range between the trivial and very hard instances that dominate public benchmarks of similar size. Overall, our contributions are as follows.

- **Learning hardness from a fixed verifier:** a seed-free self-learning loop in which an LLM challenger learns hardness against a frozen solver (Section 3).
- **A reward that resists false hardness:** a validity-gated hardness score read after presolve and root cuts, with size and structural-diversity terms (Section 3.2).
- **Language-controlled generation:** The trained model generates instances directly from text, keeps the base model's ability to follow instructions on domain and difficulty, and shifts each instruction toward harder instances. Such instances can be used to tune a solver for a problem type that public libraries lack (Section 4.5).
- **Released generators and evaluation:** OPTISCRIBE-12B, OPTISCRIBE-4B, and an evaluation protocol that checks hardness ordering across the used solvers.

2 BACKGROUND AND RELATED WORK

2.1 VERIFIER-IN-THE-LOOP RL

Recent work lets language models write their own training problems and score them with a verifier. Absolute Zero (Zhao et al., 2025) trains one model to propose and solve code-reasoning tasks, rewarding tasks it solves only some of the time. R-Zero (Huang et al., 2026) alternates a Challenger and a Solver from one base model and rewards questions on which the Solver matches its own majority vote about half the time, a label that weakens as questions get harder. STP (Dong & Ma, 2025) rewards conjectures the current prover proves only occasionally, with every proof checked by a formal verifier. In all three, difficulty is relative to a co-trained model, so the target moves during training and checkpoints share no fixed scale. Their verifiers also return only pass or fail, whereas a MILP solver reports how much effort a solve took. SIRL (Chen et al., 2025) and ORLM

Table 1: Comparison with verifier-in-the-loop RL and MILP instance generators.

Method	Artifact	Difficulty Signal	Difficulty Scale	Learns Hardness	Input at Generation	Solver at Inference	Text control
Verifier-in-the-loop RL							
Absolute Zero	Code task	Solver success rate	Relative	✓	Past tasks	✓	✗
R-Zero	Question	Majority agreement	Relative	✓	None	✗	✗
STP	Conjecture	Prover pass rate	Relative	✓	Seed theorems	✗	✗
MILP instance generators							
Random Gens.	Instance	None	Hand-set	✗	Parameters	✗	✗
G2MILP	Instance	None	Preserved	✗	Seed instances	✗	✗
ACM-MILP	Instance	None	Preserved	✗	Seed instances	✗	✗
MILP-StuDio	Instance	Solve time[a]	Preserved	✗	Seed instances	✗[a]	✗
MILP-Evolve	Code	Time, nodes, gap, size	Absolute	✗	Seed classes	✓	✗
MILP-Retrieval	Code	Similarity, time	Absolute	✗	Target instance	✓	✗
OPTISCRIBE (Ours)	Instance	Validity, nodes, gap	Absolute	✓	Exemplar	✗	✓

Relative: difficulty is set against co-trained model (drifts during training). *Preserved*: reproduces supplied-seed difficulty. *Hand-set*: fixed via manual parameters. *Learns hardness*: generator trained on a difficulty signal. *Text control*: hardness specified in natural language. [a]MILP-StuDio generates instances without a solver; only its hard-instance study solves edits and keeps the slowest.

(Huang et al., 2025) use a solver to check formulations of user-provided problems, so they reward correctness rather than effort and create no new problems. We instead reward solver effort under a fixed verifier, which gives difficulty an absolute scale (Table 1).

2.2 MILP INSTANCE GENERATORS

Most MILP instance generators do not learn what makes an instance hard. Hand-written generators for set cover, combinatorial auctions and facility location (Balas & Ho, 1980; Leyton-Brown et al., 2000; Cornuéjols et al., 1991) set difficulty through size and density and supply most training data in learning-based work (Gasse et al., 2019; Prouvost et al., 2020), yet at moderate sizes solvers often close their instances quickly (Huang et al., 2024). Instance-space methods evolve instances toward target regions of a feature space (Smith-Miles & Bowly, 2015; Bowly, 2019). Seed-based methods edit existing instances by masked graph reconstruction (Geng et al., 2023), grouped constraint modification (Guo et al., 2024) or block manipulation (Liu et al., 2024). They aim to preserve the seed's difficulty and need seeds of the target family, and MILP-StuDio obtains hard instances only by keeping the slowest of repeated edits. LLM-based methods write generator programs instead. MILP-Evolve (Li et al., 2025) evolves generator code with a frozen LLM and keeps parameter settings whose instances hit solver-measured targets (size, solve time, node count, integrality gap). MILP-Retrieval (Yang et al., 2025) retrieves the library program closest to a target instance and tunes it by Bayesian optimization on solve time. In both, difficulty comes from searched parameters, the search reruns for each class and size, and solve-time targets vary with hardware and random seeds (Lodi & Tramontani, 2013). None accepts a natural-language specification, so a user cannot simply ask for a denser graph or a harder instance. Our generator is trained directly against solver effort, needs no seed instance or solver call at generation, and takes its specification as text (Table 1).

2.3 MILPS AND SOLVER EFFORT SIGNALS

We consider MILPs of the form

$$\mathcal{I}: \quad z^\star = \min_{x_C,\, x_D} \; c^\top [x_C; x_D] \qquad \text{s.t.} \quad A\,[x_C; x_D] \le b, \qquad \ell \le [x_C; x_D] \le u, \\ x_C \in \mathbb{R}^d,\; x_D \in \mathbb{Z}^{n-d},\; c \in \mathbb{R}^n,\; A \in \mathbb{R}^{m\times n},\; b \in \mathbb{R}^m, \tag{1}$$

where n is the number of variables, m the number of constraints, and d the number of continuous variables. A discrete variable with bounds 0 and 1 is binary. Bounds may be infinite, and we set $z^\star = +\infty$ if $\mathcal{I}$ is infeasible and $z^\star = -\infty$ if it is unbounded.

A branch-and-cut solver works in four phases, and each reports a quantity that could serve as a reward. *Presolve* shrinks $\mathcal{I}$ to an equivalent instance by propagating bounds, fixing forced variables,

and tightening coefficients. A large reduction signals redundancy rather than difficulty. The *root relaxation* drops integrality and gives a bound $z_{\mathrm{LP}} \le z^\star$, and the difference $z^\star - z_{\mathrm{LP}}$ is the integrality gap of the formulation. The *root cutting loop* adds valid inequalities that tighten this bound to z_{cut}, with $z_{\mathrm{LP}} \le z_{\mathrm{cut}} \le z^\star$. *Search* then closes the remaining gap $z^\star - z_{\mathrm{cut}}$ using $N(\mathcal{I})$ branch-and-bound nodes. In Section 3.2, we design a reward function using these quantities.

3 Proposed Seed-Free Asymmetric Self-Play Approach

Our goal is an LLM that, given a MILP family and a target size, writes a valid instance that is hard to solve. With no library of hard instances to learn from, the model must discover hardness itself. It proposes instances, a solver attempts them, and the model is rewarded by the solver's effort, learning over training which choices make the solver work harder.

3.1 Challenger and verifier

We build on challenger-solver self-play, but unlike prior work we do not train the solver. The LLM policy π_θ acts as the challenger and proposes instances, while a branch-and-cut solver $\mathcal{S}$ with a fixed configuration acts as the verifier, checking feasibility, proving optimality, and reporting the search effort required. Because only the challenger learns, the game is asymmetric. Training is also seed-free. Prompts only mention problem family, target size, output format, and one worked example at a different size. We never use example to target hardness, or instance corpus, or solver call at inference time. Appendix A gives the training prompt.

3.2 A solver-effort reward

We build the reward from the quantities the solver reports (Section 2.3). As discussed earlier, it must separate an instance that the solver closes at the root from one that forces a long search, without being inflated by formulations that only look hard.

Where to read the reward: Rewarding the raw gap $|z^\star - z_{\mathrm{LP}}|$ would also reward weak but equivalent formulations. For example, $\sum_j x_j \le Mz$ with a large M lets the relaxation take a fractional z and inflates the gap, although the decision problem is unchanged. Disaggregated links $x_j \le u_j z$ largely remove this, and presolve and cuts often do so anyway. We therefore read the reward after presolve and the root cutting loop, so that it measures the difficulty that remains after the solver's own repairs. Section 4.4 gives supporting evidence that difficulty read before cuts can be specific to one solver. We define reward for each completion o_i, which parsed into an instance $\mathcal{I}_i$ as:

$$R_i \;=\; V(\mathcal{I}_i) \cdot \Big[\, w_H\, H(\mathcal{I}_i) + w_v\, r_{\mathrm{var}}(\mathcal{I}_i) + w_d\, r_{\mathrm{div}}(\mathcal{I}_i) \,\Big], \tag{2}$$

where V is a validity gate, H the hardness, r_{var} a size term, and r_{div} a diversity term, with weights (w_H, w_v, w_d). The gate $V(\mathcal{I}_i) \in \{0, 1\}$ equals 1 only if the instance parses, matches the requested family, and the solver finds a feasible solution with a finite optimum. It also requires bounded continuous variables, a bounded coefficient range, and no big-M constraint that ties many continuous variables to one binary, which rules out apparent hardness that vanishes under presolve or comes from ill-conditioning (Appendix B.1). For a valid instance, hardness combines the post-cut gap with the size of the search tree,

$$H(\mathcal{I}_i) = 0.25\, r_{\mathrm{cut}}(\mathcal{I}_i) \;+\; 0.75\, r_{\mathrm{node}}(\mathcal{I}_i), \tag{3}$$

$$g_{\mathrm{cut}}(\mathcal{I}_i) = \frac{|z_i^\star - z_{\mathrm{cut},i}|}{\tau\,|z_i^\star|},\;\; r_{\mathrm{cut}}(\mathcal{I}_i) = \mathrm{clip}\big(g_{\mathrm{cut}}(\mathcal{I}_i), 0, 1\big),\;\; r_{\mathrm{node}}(\mathcal{I}_i) = \mathrm{clip}\left(\frac{\log N(\mathcal{I}_i)}{\log N_{\mathrm{ref}}}, 0, 1\right). \tag{4}$$

With $\tau = 0.10$, relative gaps are resolved up to 10% and larger ones are capped, and the absolute value makes r_{cut} independent of the optimization sense. Node counts are heavy-tailed, so r_{node} is log-scaled with $N_{\mathrm{ref}} = 5{,}000$ and is zero for an instance solved at the root ($N = 1$). Because a large post-cut gap does not guarantee a long search, the node term gets the larger weight and the gap term serves for shaping. Since the easiest way to raise r_{node} is to write larger instances, the size term rewards closeness to the requested size,

$$r_{\mathrm{var}}(\mathcal{I}_i) \;=\; \exp\!\Big(-\alpha\, \frac{|n_i - n^\star|}{n^\star}\Big), \tag{5}$$

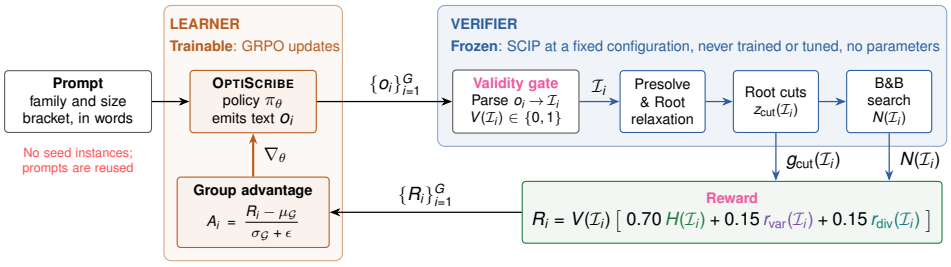

Figure 1: **Seed-free asymmetric self-play with trainable challenger and frozen verifier.** From a prompt naming a family and a size bracket, the policy generates G=64 instances $\mathcal{I}_i$, which SCIP solves with a fixed configuration. The verifier returns the validity gate V, post-cut gap g_{cut} and node count N, to compute hardness H. H is then combined with size r_{var} and diversity r_{div} rewards for GRPO-based Challender update. The untrained verifier keeps difficulty on a fixed scale.

where n_i is the realized variable count, $n^\star$ the midpoint of the requested bracket, and $\alpha = 3$. The term is soft, so an instance outside the bracket loses at most w_v. The diversity term r_{div} ranks each valid instance by how rare its structural fingerprint is within its group, which discourages the policy from collapsing onto one structure. It averages exactly 0.5 in every group and therefore only redistributes advantage among valid instances (Appendix B.2). Appendix B.3 gives a worked example of the complete reward. We also train OPTISCRIBE-4B-xH, that drops the diversity term and moves its weight to hardness.

3.3 INSTANCE REPRESENTATION

Each instance is written in a compact index-set template rather than as an explicit coefficient matrix. The template has a header (family and optimization sense), a structure block (index sets, variable types and bounds, an objective, and constraint families quantified over the sets), and a data block of numeric arrays. A deterministic parser expands this text into an explicit MILP. Because the structure block does not grow with the instance, token cost grows with the number of parameters rather than the number of constraints (Appendix C). Token caps are set per size bracket from the length distribution of complete instances, and a completion that exceeds its cap fails to parse and receives zero reward.

3.4 TRAINING LOOP

Figure 1 illustrates a single training iteration. Given a prompt p, the policy samples a group of G completions $\{o_i\}_{i=1}^G$. We parse each completion into an instance $\mathcal{I}_i$ (Section 3.3), solve it with $\mathcal{S}$, and compute its reward R_i (2). We fine-tune the policy with LoRA (Hu et al., 2022) using group relative policy optimization (GRPO) (Shao et al., 2024), which takes the within-group standardized reward as the advantage, $A_i = (R_i - \text{mean}_j\, R_j)/(\text{std}_j\, R_j + \epsilon)$. A completion is therefore reinforced only when it outperforms its peers for the same prompt, for example by yielding a harder instance, or a valid one where its peers fail. We apply the clipped GRPO update with a KL penalty that keeps the policy close to the base model (Appendix D). Each group is drawn from a single problem family and size bracket, and families alternate across steps, so instances are never compared across families.

4 EMPIRICAL RESULTS AND DISCUSSION

We train two base LLMs, Gemma-4-12B and Qwen3.5-4B, on three standard MILP families that differ in structure: capacitated facility location (CFL) has linking constraints, linearized Max-Cut is a graph problem, and multiple knapsack is a packing problem. We evaluate the generated instances along two axes, their gain in hardness over the untrained base model and their standing against public benchmarks of similar size, and then examine robustness, design choices and uses. Our experiments address six questions. **1)** Do trained models generate harder instances than their base models (Section 4.1)? **2)** How do these instances compare with public benchmarks of similar size (Section 4.2)? **3)** Does the added difficulty hold under solvers not used in training (Section 4.3)? **4)** How do reward design choices, in particular where the reward is read, affect the results (Section 4.4)?

5) Can the trained model still be steered with language (Section 4.5)? **6)** Can generated instances be used to tune solver settings (Section 4.6)?

Experimental setup: We train each base model in two ways. The *curriculum* models, OPTISCRIBE-12B and OPTISCRIBE-4B, train on three size brackets in succession, 76–110 (B0), 111–170 (B1) and 171–225 (B2) variables, with each stage initialized from the previous one. The *direct* models, OPTISCRIBE-12B-D and OPTISCRIBE-4B-D, train on B2 alone with the same reward and hyperparameters. We evaluate on B1, B2, B3 (226–350) and B4 (351–500), so B3 and B4 test size extrapolation. Gemma-4-12B is our primary model, and the smaller Qwen3.5-4B replicates the main results and carries the node-reference and reward-weight ablations. Unless stated otherwise, the reward weights are $(w_H, w_v, w_d) = (0.70, 0.15, 0.15)$, and SCIP runs on one thread with a 50,000-node cap (Appendix D.2). All models receive identical prompts (Appendix A) with matched sampling seeds, so comparisons are paired. We report parse rate, feasible rate, branch-and-bound nodes and the post-root-cut gap $|z^\star - z_{\text{cut}}|/|z^\star|$, where more nodes and larger gaps mean more solver effort. We avoid solve time, which depends on the machine, except in Section 4.6. Appendices D and E detail the curriculum, geometry choice, hyperparameters and measurement conventions.

4.1 TRAINED MODELS GENERATE HARDER INSTANCES THAN THEIR BASE MODELS

Table 2 reports paired comparisons for Gemma. In every bracket, OPTISCRIBE-12B generates the hardest CFL and Max-Cut instances, raising median nodes over Gemma-4-12B by 1.7–5.0× (CFL) and 1.9–4.5× (Max-Cut) and median post-cut gaps by 1.4–1.7× and 1.1–1.3×. The gains persist at the unseen sizes B3–B4 (2.4× nodes on CFL, 2.4×/1.9× on Max-Cut), and the central 95% range of node counts is 1.7–5.2× wider, covering a broader span of difficulty. On CFL, parse and feasible rates also rise by 9–15 and 9–19 points. On Max-Cut, where Gemma-4-12B already parses nearly everything, the only notable drop (76.6% vs. 98.8% parsed at B4) comes mostly from unfinished completions, as every parsed OPTISCRIBE-12B instance is feasible.

Direct training at the target size fails for Gemma. OPTISCRIBE-12B-D became unstable as more completions hit the token cap and scored zero, and we stopped it after 47 of 61 updates (Appendix Figure 5), with nodes and gaps near or below those of Gemma-4-12B. Multiple knapsack stays easy for all models, with median gaps of about 0.5% (Appendix G.2).

A second base model: To test transfer across LLMs, we repeat the analysis on Qwen3.5-4B (Table 3). OPTISCRIBE-4B matches or improves parsing (by up to 17 points on CFL), raises median nodes by 2.9–11.6× on Max-Cut and 1.6–15.6× on CFL, and raises median gaps by 9.7–11.1 and 0.8–2.1 points, respectively. Unlike Gemma, Qwen trains stably at the target bracket (OPTISCRIBE-4B-D improves on Qwen3.5-4B in 21 of 24 cells), yet within the training brackets OPTISCRIBE-4B still beats OPTISCRIBE-4B-D in 10 of 12 comparisons, with one tie. The curriculum is thus necessary for Gemma and beneficial for Qwen. Qwen models also tend to generate harder instances than Gemma (Appendix Figures 9 and 10), partly because Qwen3.5-4B is already stronger on Max-Cut, so the difference reflects the base models rather than the framework.

4.2 GENERATED INSTANCES FILL GAPS IN PUBLIC BENCHMARKS

We pool public instances with 111–500 variables from eight sources, including MIPLIB 2010/2017, MILP-Evolve, and D-MIPLIB (full list and citations in Appendix G.4). We compare public mixed-integer instances against our CFL pools, and public pure-integer instances against our Max-Cut pools, using OPTISCRIBE-12B, two OPTISCRIBE-4B runs ($N_{\text{ref}} = 5{,}000$ and 50,000, Section 4.4), and OPTISCRIBE-4B-xH (Figure 2). Public coverage in this range is sparse and largely bimodal. Among the 60 mixed-integer instances, most are either trivial (≈ 10 nodes) or nearly hit the 50,000-node cap, while the pure-integer set adds only a narrow band around 1,000–10,000 nodes. Our generated pools span the full difficulty spectrum, with several instances reaching the cap, and OPTISCRIBE-4B-xH reaches the cap about twice as often as the other pools. Public mixed-integer instances solved to optimality typically end with gaps below $\approx 6\%$, similar to our CFL pools, whereas our Max-Cut pools retain much larger post-cut gaps (41.9–49.9% vs. 11.9%). Standard synthetic generators are easier still at this scale. Their pooled median instance is solved at the root and none needs more than 189 nodes, against median node counts of 347 on CFL and 634 on Max-Cut for OPTISCRIBE-12B (Appendix G.3). Since our generator produces instances of a requested family and size on demand, we use it for tuning in Section 4.6.

Table 2: **Main results for CFL and Max-Cut under SCIP.** 256 instances per arm and bracket. Nodes is the median node count and Gap the median post-root-cut gap ($\times 10^{-3}$), both over feasible instances, with the 2.5th and 97.5th percentiles in brackets. Bold marks the best value.

		Capacitated Facility Location (CFL)				Max-Cut			
Bracket	Arm	Parse ↑	Feas ↑	Nodes ↑	Gap ↑	Parse ↑	Feas ↑	Nodes ↑	Gap ↑
B1	Gemma-4-12B	78.9	72.7	22 [1, 456]	18 [1, 63]	**100.0**	85.9	41 [15, 91]	258 [182, 314]
	OPTISCRIBE-12B-D	84.8	82.4	21 [1, 388]	18 [1, 55]	99.6	**89.1**	41 [15, 90]	260 [190, 315]
	OPTISCRIBE-12B	**94.1**	**91.8**	**109** [2, 755]	**30** [2, 74]	**100.0**	**89.1**	**142** [29, 297]	**338** [237, 392]
B2	Gemma-4-12B	88.3	86.7	224 [1, 1873]	22 [4, 63]	99.6	91.8	81 [29, 182]	298 [237, 354]
	OPTISCRIBE-12B-D	85.5	85.2	170 [1, 1587]	24 [2, 65]	99.6	**96.9**	85 [31, 193]	300 [238, 355]
	OPTISCRIBE-12B	**99.6**	**96.5**	**391** [1, 3539]	**35** [3, 82]	99.6	89.8	**364** [103, 821]	**396** [316, 456]
B3	Gemma-4-12B	86.3	82.0	293 [1, 6272]	20 [2, 52]	**100.0**	95.3	423 [201, 744]	395 [332, 446]
	OPTISCRIBE-12B-D	87.5	86.7	211 [1, 5591]	20 [2, 62]	98.8	95.7	369 [148, 676]	377 [317, 436]
	OPTISCRIBE-12B	**98.0**	**94.9**	**710** [2, 18296]	**27** [2, 70]	99.6	**96.1**	**1002** [510, 1770]	**465** [405, 529]
B4	Gemma-4-12B	76.6	75.8	420 [1, 20671]	16 [1, 64]	**98.8**	**97.3**	1350 [823, 2180]	485 [426, 537]
	OPTISCRIBE-12B-D	69.1	68.4	194 [0, 10658]	14 [1, 40]	82.4	80.5	1096 [477, 1801]	456 [379, 515]
	OPTISCRIBE-12B	**85.5**	**84.4**	**1014** [2, 34277]	**25** [2, 58]	76.6	76.6	**2578** [1011, 8122]	**546** [467, 623]

Table 3: Qwen arms against untrained Qwen3.5-4B under SCIP. Parsing and median post-root-cut gap are differences in percentage points (pp). Nodes is the ratio of median node counts. 128 completions per model and cell, feasible instances only. OPTISCRIBE-4B-xH raises the hardness weight and drops the diversity term. Bold marks the largest value.

		Max-Cut			Capacitated facility location		
Bracket	Arm	Parsing (pp)	Nodes	Gap (pp)	Parsing (pp)	Nodes	Gap (pp)
111–170	OPTISCRIBE-4B-D	+4.69	4.96×	+10.61	+14.84	2.23×	−0.27
	OPTISCRIBE-4B	+2.34	5.08×	+10.74	**+17.19**	12.23×	+1.14
	OPTISCRIBE-4B-xH	**+9.38**	**7.22×**	**+13.40**	**+17.19**	**26.68×**	**+2.75**
171–225	OPTISCRIBE-4B-D	−0.78	4.74×	+10.40	**+1.56**	6.93×	+1.13
	OPTISCRIBE-4B	**+0.78**	5.26×	+11.02	**+1.56**	15.63×	+1.45
	OPTISCRIBE-4B-xH	**+0.78**	**6.56×**	**+14.14**	**+1.56**	**26.51×**	**+1.81**
226–350	OPTISCRIBE-4B-D	−1.56	3.38×	+12.37	+8.59	12.69×	+0.54
	OPTISCRIBE-4B	0.00	2.85×	+9.66	**+9.38**	7.64×	**+2.08**
	OPTISCRIBE-4B-xH	**+5.47**	**3.82×**	**+13.38**	**+9.38**	**31.51×**	+0.53
351–500	OPTISCRIBE-4B-D	+6.25	16.63×	+12.62	+3.12	2.76×	+0.19
	OPTISCRIBE-4B	**+7.81**	11.58×	+11.11	+4.69	1.61×	**+0.83**
	OPTISCRIBE-4B-xH	**+7.81**	**28.83×**	**+14.35**	**+5.47**	**11.14×**	+0.37

4.3 THE HARDNESS ORDERING HOLDS UNDER HELD-OUT SOLVERS

Since the reward comes from SCIP, the model could learn instances that are hard only for SCIP. We re-solve all instances with HiGHS and Gurobi 13.0.3, neither used in training (Figure 3a,b). For OPTISCRIBE-12B, median nodes grow with size under every solver. HiGHS follows SCIP's trend, while Gurobi needs fewer nodes on CFL and more on Max-Cut. The model ranking is unchanged: OPTISCRIBE-12B has the highest median in every bracket except CFL B1 under Gurobi, where all medians are root-solved (Appendix G.5). Whenever both prove optimality, SCIP and Gurobi agree on the optimal value within tolerance, so the difficulty is not an artifact of numerical error.

4.4 REWARD DESIGN: READOUT POINT, NODE REFERENCE AND WEIGHTS

We ablate three reward choices. *Readout point.* Section 3.2 discussed that a reward read before root cuts can be inflated by weak formulations. A model trained with a pre-cut reward and HiGHS in

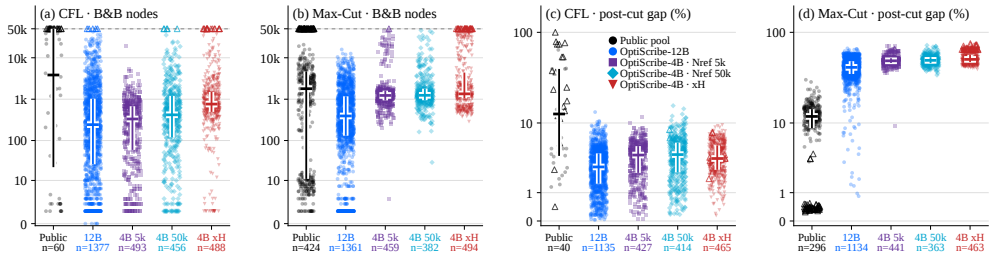

Figure 2: **Generated instances against public benchmarks,** 111–500 **variables.** Public mixed-integer instances against generated CFL (a, c) and public pure-integer instances against generated Max-Cut (b, d). (a, b) SCIP nodes, (c, d) post-root-cut gap in percent. Points are single instances, all weighted equally. Crossbars are medians and vertical bars the interquartile range. Triangles mark instances that hit the 50,000-node cap (dashed line), and their gaps are measured against the best incumbent.

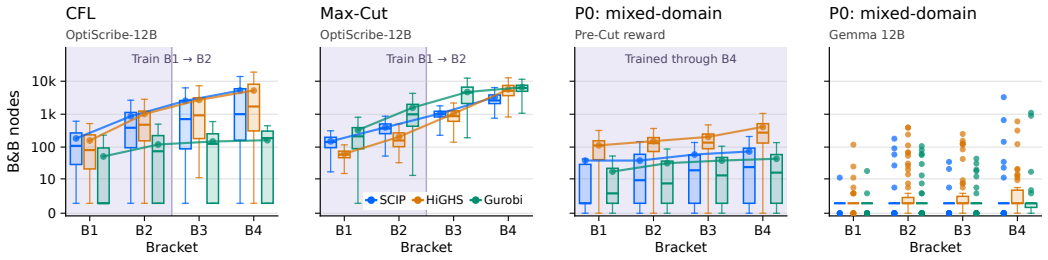

Figure 3: **Node counts under three solvers.** (a, b) OPTISCRIBE-12B CFL and Max-Cut instances under SCIP (training solver), HiGHS and Gurobi, with training brackets shaded. (c) A model trained on prompt P0 with a pre-cut reward and HiGHS in the loop, through all four brackets. (d) Untrained Gemma-4-12B on P0. Boxes show median and interquartile range, dots are outliers, lines join per-bracket medians. Log vertical axis.

the loop, on the mixed-domain prompt P0 (Appendix F), illustrates this. Its instances tie binaries to continuous sums through single loose big-M rows and leave 67–95% of variables without an upper bound, so the LP relaxation opens facilities almost for free. Tightening M to a valid value left HiGHS node counts unchanged on all 1,715 instances. SCIP's root cuts, by contrast, repair the relaxation, and disabling them raises SCIP's node count by a median factor of 8–10. SCIP and Gurobi therefore solve one-third to over half of these instances at the root, and HiGHS at most 6% (Figure 3 c,d; Appendix Figure 6), which is why we read the reward after root cuts. *Raising the node reference* N_{ref}, where the node term in equation 4 saturates, to 50,000 for Qwen changes little inside the training brackets (Appendix Figure 7). Outside them it raises CFL node gains at B3/B4 (24.5×/5.7× vs. 7.6×/1.6×) but lowers Max-Cut at B4 (6.6× vs. 11.6×) and costs over 10 points of parse rate. *The hardness weight* acts as a dial. Raising it and dropping the diversity term (OPTISCRIBE-4B-xH) gives the highest node count in every cell of Table 3. That is 3.8–28.8× (Max-Cut) and 11.1–31.5× (CFL) over Qwen3.5-4B, or 1.2–2.5× and 1.7–6.9× over OPTISCRIBE-4B, with no parse-rate loss but tighter clustering in Figure 2, i.e. less diversity.

4.5 LANGUAGE CONTROL IS PRESERVED ON MAX-CUT

An LLM generator should do what the prompt says. For Max-Cut at 171–225 variables, we append one unseen sentence to a fixed prompt, requesting a harder instance, a sparser graph at target edge density κ, or both (Appendix Table 14), and sample 48 completions per sentence from Gemma-4-12B and OPTISCRIBE-12B (Figure 4). *Both models follow the density instructions.* For requested densities 0.8/0.5/0.2, median realized densities are 0.96/0.56/0.18 for Gemma-4-12B and 0.93/0.35/0.10 for OPTISCRIBE-12B, which overshoots but keeps the order. "Harder" raises node counts from 92 to 135 for Gemma-4-12B and from 344 to 400 for OPTISCRIBE-12B, so training raises the level at which a sentence lands (3.7× and 3.0× over Gemma-4-12B) without amplifying its marginal effect. On CFL, OPTISCRIBE-12B does not track the analogous density request, though its instances still become easier (Appendix Figure 8). *The effect is bounded by problem*

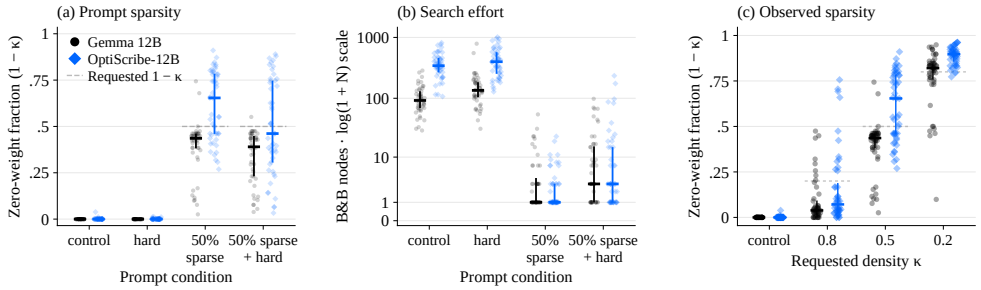

Figure 4: **Language control over Max-Cut.** Bracket 171–225, 48 instances per model/prompt, with SCIP. (a) Share of zero-weight vertex pairs, $1 - \kappa$, under each prompt. Dashed lines are requested values. (b) Branch-and-bound nodes. (c) Realized zero-weight share vs. requested density. Points are single completions, horizontal bars are medians and vertical bars span the interquartile range.

structure. Our Max-Cut LP bound equals the total edge weight (Appendix C.2), so sparser graphs raise the maximum cut's share of it. At densities 0.5 and 0.2 the median share rises from about 0.6 to 0.86–1.00, the bound becomes nearly tight and SCIP often closes at the root, and even "harder" with a sparse request reaches a median of only 3 nodes. New requirements can thus be stated as unseen sentences, and the trained model follows them within what the family's mathematics allows.

4.6 Generated instances support solver tuning

Solver tuning works best on instances that match the target workload, but public collections often miss the right family or size. We test whether OptiScribe-12B can fill that gap by tuning SCIP on 40 generated instances with 226–350 variables and evaluating the chosen setting on held-out instances. *In-distribution utility:* MILP-Evolve has no CFL instances at this size, so we generated them with OptiScribe-12B and tested on 24 held-out generated instances. The best setting caps aggregation-based cutting planes at five rounds at the root (instead of no round limit), reducing mean solve time by 16.0% (95% bootstrap CI 6.3–25.5%) with identical optimal values. *Out-of-distribution transfer:* We selected among 10 SCIP configurations on generated Max-Cut instances and applied the winning configuration unchanged to 40 same-size MILP-Evolve combinatorial-auction instances. The wining configuration disables cutting planes and solves 35/40 instances within 10 s, compared to 14/40 with default settings. It also cuts mean PAR2[1] from 15.15 to 7.50 s (paired difference −7.64 s, 95% bootstrap CI −9.43 to −5.88 s). Tuning directly on MILP-Evolve selects the same setting. Generated instances thus recover the setting chosen on the target library, and they allow tuning for a family and size that the library lacks.

5 Conclusions and Limitations

We showed that solver effort after root cuts is a usable reward for teaching LLMs to write hard MILP instances. With a frozen verifier and no seed instances or instance corpus, OptiScribe-12B and OptiScribe-4B generate harder CFL and Max-Cut instances than their base models at trained and larger sizes, and the ranking holds for two held-out solvers. The instances cover difficulty levels that public benchmarks of similar size leave sparse, and stay controllable through natural-language requests such as density and difficulty. They are also useful in practice: settings tuned on them match those tuned on MILP-Evolve, and they allow tuning for a family and size that public libraries lack.

Our study has limitations. It covers three families, and multiple knapsack stays easy for every model. Generated instances currently have at most 500 variables. We train each arm once and tune on a small set of instances. Broader families, larger problems, and wider evaluations are left for future work.

[1]Penalized average runtime, the mean solve time with each unsolved instance counted at twice the time limit (Froleyks et al., 2021).

AI USE STATEMENT

In this work, we used generative AI tools for two purposes. First, they assisted with writing and debugging code for experiment scripts, data processing and figure generation. Second, they helped with language editing of the manuscript, including grammar, wording and sentence flow. We did not use generative AI tools to generate research ideas, design the method or experiments, produce or alter experimental results, or write the related-work analysis. All AI-assisted code was reviewed, tested and verified for correctness by the authors. All AI-edited text was reviewed by the authors for accuracy and consistency with our results. We take full responsibility for the final content of this work, including text, claims and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects, personal data or crowdsourced annotation. All training and evaluation instances are synthetic MILP problems generated by our models, and the public benchmark instances used for comparison are drawn from openly available libraries under their respective licenses. The base language models, Gemma-4-12B and Qwen3.5-4B, are used under their published licenses. The authors declare no conflicts of interest, and this work received no specific funding or sponsorship.

REPRODUCIBILITY STATEMENT

We will release the full source code, including the training pipeline, reward implementation, instance parser, evaluation scripts and trained model checkpoints. An anonymized version is provided as supplementary material for review. The exact training prompt, per-family mandates and in-context exemplars are reproduced verbatim in Appendix A, and the index-set template and its grammar are given in Appendix C. The size curriculum, rung geometries, training hyperparameters and measurement conventions are listed in Appendices D and E (Table 7). All evaluations use fixed prompts, matched seeds and stated solver versions, node caps and time limits, and all reported metrics are computed from branch-and-bound node counts and gaps rather than machine-dependent solve times, except in the tuning study (Section 4.6), which reports its protocol and confidence intervals. The public benchmark libraries used for comparison are openly available and cited in Section 4.2.

REFERENCES

Tobias Achterberg. Scip: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009. doi: 10.1007/s12532-008-0001-1.

Egon Balas and Andrew Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In *Combinatorial Optimization*, volume 12 of *Mathematical Programming Studies*, pp. 37–60. Springer, 1980.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2): 405–421, 2021. doi: 10.1016/j.ejor.2020.07.063.

Suresh Bolusani, Mathieu Besançon, Ambros Gleixner, Timo Berthold, Claudia D'Ambrosio, Gonzalo Muñoz, Joseph Paat, and Dimitri Thomopulos. The MIP workshop 2023 computational competition on reoptimization. *Mathematical Programming Computation*, 16:255–266, 2024. doi: 10.1007/s12532-024-00256-w.

Simon Bowly. *Stress Testing Mixed Integer Programming Solvers through New Test Instance Generation Methods*. PhD thesis, University of Melbourne, 2019.

Peter C. Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *IJCAI*, 1991.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed RL: Grounding large language models for authentic optimization modeling. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Gérard Cornuéjols, Ramaswamy Sridharan, and Jean-Michel Thizy. A comparison of heuristics and relaxations for the capacitated plant location problem. *European Journal of Operational Research*, 50(3):280–297, 1991.

Kefan Dong and Tengyu Ma. STP: Self-play LLM theorem provers with iterative conjecturing and proving. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 14114–14136. PMLR, 2025.

Nils Froleyks, Marijn Heule, Ashlin Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021. doi: 10.1016/j.artint.2021.103572.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 32, pp. 15587–15596, 2019. URL https://proceedings.neurips.cc/paper/2019/file/d14c2267d848abeb81fd590f371d39bd-Paper.pdf.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, et al. Gemma 4 Technical Report. *arXiv preprint arXiv:2607.02770*, 2026. URL https://arxiv.org/abs/2607.02770.

Zijie Geng, Xijun Li, Jie Wang, Xiao Li, Yongdong Zhang, and Feng Wu. A deep instance generative framework for MILP solvers under limited data availability. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans Mittelmann, Derya Ozyurt, Ted Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3):443–490, 2021. doi: 10.1007/s12532-020-00194-3.

Ziao Guo, Yang Li, Chang Liu, Wenli Ouyang, and Junchi Yan. ACM-MILP: Adaptive constraint modification via grouping and selection for hardness-preserving MILP instance generation. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16869–16890. PMLR, 2024.

J. N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1(1):33–42, 1995.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026.

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

Weimin Huang, Taoan Huang, Aaron M. Ferber, and Bistra Dilkina. Distributional MIPLIB: a multi-domain library for advancing ML-guided MILP methods. *arXiv preprint arXiv:2406.06954*, 2024.

Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans D. Mittelmann, Ted K. Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011. doi: 10.1007/s12532-011-0025-9.

Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pp. 66–76, 2000.

Sirui Li, Janardhan Kulkarni, Ishai Menache, Cathy Wu, and Beibin Li. Towards foundation models for mixed integer linear programming. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

Haoyang Liu, Jie Wang, Wanbo Zhang, Zijie Geng, Yufei Kuang, Xijun Li, Yongdong Zhang, Bin Li, and Feng Wu. MILP-StuDio: MILP instance generation via block structure decomposition. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Andrea Lodi and Andrea Tramontani. Performance variability in mixed-integer programming. In *Theory Driven by Influential Applications*, INFORMS TutORials in Operations Research, pp. 1–12. INFORMS, 2013. ISBN 978-0-9843378-4-2. doi: 10.1287/educ.2013.0112.

David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In *AAAI*, 1992.

Dimitri J. Papageorgiou, George L. Nemhauser, Joel Sokol, Myun-Seok Cheon, and Ahmet B. Keha. MIRPLib – a library of maritime inventory routing problem instances: Survey, core model, and benchmark results. *European Journal of Operational Research*, 235(2):350–366, 2014. doi: 10.1016/j.ejor.2013.12.013.

Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In *Learning Meets Combinatorial Algorithms Workshop at NeurIPS 2020*, 2020.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Bart Selman, David G Mitchell, and Hector J Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Kate Smith-Miles and Simon Bowly. Generating new test instances by evolving in instance space. *Computers & Operations Research*, 63:102–113, 2015.

Haoyu Peter Wang, Jialin Liu, Xiaohan Chen, Xinshang Wang, Pan Li, and Wotao Yin. DIG-MILP: A deep instance generator for mixed-integer linear programming with feasibility guarantee. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=psDvcWtFdE.

Alinson S. Xavier, Feng Qiu, Xiaoyi Gu, Berkay Becu, and Santanu S. Dey. MIPLearn: An extensible framework for learning-enhanced optimization (version 0.4), 2024.

Tianxing Yang, Huigen Ye, and Hua Xu. Code retrieval for MILP instance generation, 2025. URL https://arxiv.org/abs/2505.11526.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Appendix: Teaching LLMs to Generate Challenging MILP Instances via Solver Feedback

A THE GENERATION PROMPT

A.1 SCOPE AND DELIVERY

Both models OPTISCRIBE-12B and OPTISCRIBE-4B are trained with the *same* prompt. The models only differ only in the underlying base model and in the chat template applied by that model's tokenizer. Concretely, the rendered prompt is passed in as a **single user message, with no system prompt**. Each training row has the form:

```
{"role": "user", "content": [{"type": "text", "text": <rendered prompt>}]}
```

The trainer then applies the model's chat template. As a result, the only input-level difference between the two models is the template wrapper (Gemma's `<start_of_turn>user` versus Qwen's `<|im_start|>user`), not the prompt body.

We render one prompt per training example using a function family $f \in$ {Capacitated Facility Location, Max-cut, Multiple knapsack}, the curriculum bracket b, and an in-context exemplar.

A.2 TEMPLATE

Every prompt is the concatenation of seven blocks in fixed order:

1. a one-line task statement,
2. **FAMILY / MANDATORY MATHEMATICS**: the per-family structural mandate (Listing 2),
3. **SIZE**: the variable-count interval $[\ell_b, u_b]$ for the bracket plus a natural-language geometry hint (Table 4),
4. **FEASIBILITY PROCEDURE**: a per-family construction recipe with one bracket-dependent numeric constant substituted in (Table 4),
5. **WHAT YOU MUST NOT DO**: four shared prohibitions (Listing 3);
6. **PRECEDENCE**: the conflict rule (Listing 3);
7. **EXAMPLE**: one complete in-context exemplar, followed by the strict output-format instruction.

Listing 1: Template skeleton. Braced tokens are substituted per training example.

```
Write ONE mixed-integer linear program in the format shown by the example below.

{MANDATE[family]}

SIZE: the problem must have between {lo} and {hi} variables in total. Aim for {geometry_hint}.
A two-index family x[A,B] contributes |A| x |B| variables; a one-index family x[A] contributes
|A|. Choose the set sizes so the total lands in range.

FEASIBILITY PROCEDURE (follow it; do not merely aim at the outcome):
  {procedure[family] with {caphint} substituted}
  A model with no feasible solution is worth nothing, and so is one whose optimum is obvious.

{BANS}

{PRECEDENCE}

EXAMPLE of the required format and mathematics:

{exemplar}

Now write a NEW problem of the same family at the size requested above.
Output exactly ONE problem.

OUTPUT FORMAT -- this is strict:
  Begin with the MILP header line. End with the DATA block. Stop immediately after the closing
  brace of the DATA block.
```

```
Write NO comments, NO explanation, NO reasoning, NO checks, and no text before or after the
model. Every line you write must be part of the model itself.
```

A.3 PER-FAMILY MANDATES

Listing 2: The three family mandates, verbatim.

```
=== Capacitated facility location ===
FAMILY: tight facility location.
MANDATORY MATHEMATICS (the names are yours; the structure is not):
  - CONTINUOUS flow variables indexed by TWO sets, x[F,D], each with an explicit finite
    upper bound
  - BINARY open variables indexed by one set, y[F]
  - a demand row per customer:   for d in D: sum f in F: x[f,d] >= dem[d]
  - a PER-PAIR link:             for f in F, d in D: x[f,d] - dem[d]*y[f] <= 0
  - a capacity row per depot WITHOUT any binary: for f in F: sum d in D: x[f,d] <= cap[f]
  - a minimisation objective over flow cost plus opening cost

=== Multiple knapsack ===
FAMILY: multiple knapsack (packing items into containers).
MANDATORY MATHEMATICS (the names are yours; the structure is not):
  - one family of BINARY variables indexed by TWO sets, x[I,K] (item, container)
  - each item used at most once:  for i in I: sum k in K: x[i,k] <= 1
    this row MUST be `<=`, never `=` -- with `=` a tight capacity makes the problem
    impossible instead of hard
  - a weighted capacity row per container: for k in K: sum i in I: w[i]*x[i,k] <= cap[k]
  - a MAXIMISATION objective over a two-index profit, p[i,k]

=== Maximum cut, linearised ===
FAMILY: maximum cut on a graph, written in the standard linearised form.
MANDATORY MATHEMATICS (the names are yours; the structure is not):
  - BINARY node variables indexed by one set, x[V]
  - BINARY edge variables indexed by two sets, y[V,V]
  - BOTH of these rows, as a pair, over the same (i,j):
        for i in V, j in V: y[i,j] - x[i] - x[j] <= 0
        for i in V, j in V: y[i,j] + x[i] + x[j] <= 2
    Neither row alone models a cut: the first alone lets y be 0 everywhere, the second
    alone lets y be 1 everywhere.
  - a MAXIMISATION objective over a two-index edge weight, w[i,j]
```

A.4 SHARED BLOCKS

Listing 3: Prohibitions and precedence rule, identical for every family and bracket.

```
WHAT YOU MUST NOT DO:
  1. Do not leave any continuous variable without an explicit finite upper bound.
     `var x[F,D] 0 inf continuous` is invalid. State a real number.
  2. Do not write an aggregated single-binary link -- one binary switching off a SUM of two or
     more continuous columns, as in `sum d in D: x[f,d] - 1000*y[f] <= 0`. A per-pair link,
     `x[f,d] - dem[d]*y[f] <= 0`, is the correct form.
  3. Do not use coefficients spanning more than four orders of magnitude. Keep every nonzero
     within a factor of 10,000 of every other.
  4. Do not omit any mandatory row above, and do not add a row that makes another one
     non-binding.

PRECEDENCE, when these instructions appear to conflict:
  the validity gates dominate these instructions, and these instructions dominate the exemplar.
  The exemplar shows ONE way to satisfy the mandate. It is not the only way, and where it
  differs from the instructions above, the instructions win.
```

A.5 PER-CELL PARAMETERS

Only two things vary with the curriculum bracket: the geometry hint in the **SIZE** line and one numeric constant in the **FEASIBILITY PROCEDURE**. The procedure sentences are otherwise fixed per family:

- **Facility Location**: "Pick every demand `dem[d]` as an integer from 5 to 20. Then set EVERY `cap[f]` to c, and every opening cost `fopen[f]` to an integer from 120 to 260, large enough that opening one more depot is a real decision, not free."

- **Multiple knapsack**: "Pick every weight `w[i]` as an integer from 10 to 30 and every profit `p[i,k]` as an integer from 20 to 60. Then set EVERY `cap[k]` to c. Do not compute it from the weights, just use that number."
- **Max-cut**: "Set `w[i,i]` = 0 on the whole diagonal. For every pair i below j, set `w[i,j]` to an integer from 1 to 20. Every single pair, leaving none at zero. Then mirror it: `w[j,i]` = `w[i,j]`." (No bracket-dependent constant.)

Every geometry hint in Table 4 lands inside its bracket (CFL: $|F||D| + |F|$; multiple knapsack: $|I||K|$; Max-Cut: $|V|^2 + |V|$). For CFL the constant c also guarantees feasibility for any demand draw: in every cell $|F|\, c \geq 20\, |D|$, so total capacity exceeds the largest possible total demand.

Table 4: Geometry hint and capacity constant c by family and bracket. Multiple knapsack is not trained at 76–110: no screened geometry for that family branched at that size, so the cell is excluded from the curriculum rather than assigned a hint.

Family	Bracket	Geometry hint (**SIZE** line)	c
CFL	76–110	about 8 depots and 12 customers	41
CFL	111–170	about 10 depots and 15 customers	41
CFL	171–225	about 14 depots and 13 customers	26
CFL	226–350	about 16 depots and 17 customers	29
CFL	351–500	about 20 depots and 20 customers	28
Multiple knapsack	76–110	*not trained at this bracket*	
Multiple knapsack	111–170	about 42 items and 4 containers	105
Multiple knapsack	171–225	about 48 items and 4 containers	120
Multiple knapsack	226–350	about 48 items and 6 containers	80
Multiple knapsack	351–500	about 48 items and 8 containers	60
Max-cut	76–110	about 9 nodes	—
Max-cut	111–170	about 12 nodes	—
Max-cut	171–225	about 13 nodes	—
Max-cut	226–350	about 16 nodes	—
Max-cut	351–500	about 20 nodes	—

A.6 In-context exemplar

Each prompt carries exactly one exemplar, drawn uniformly at random from a fixed pool of three per family. Exemplars are rendered at a *fixed* geometry that does not depend on the bracket: CFL's pool has 104, 150 and 104 variables, Multiple knapsack's and Max-cut's pools are at 192 and 182 variables respectively. The exemplar therefore lies inside the requested interval only in a few brackets. In lower brackets it is larger than requested, and in higher brackets smaller. Its data are also not re-drawn with the bracket's constant. The exemplar in Listing 4 has capacities 22–25, whereas the procedure asks for 41. The **PRECEDENCE** block exists to resolve these conflicts in favour of the instructions. Provenance comments are stripped before embedding so that comment lines are not demonstrated as part of the output format.

A.7 A complete rendered prompt

Listing 4 is one prompt exactly as the model receives it (CFL at bracket 111–170), before the chat template is applied.

Listing 4: A complete training prompt: family CFL, bracket 111–170. The `DATA` line is a single long line in the original and is wrapped here.

```
Write ONE mixed-integer linear program in the format shown by the example below.

FAMILY: tight facility location.
MANDATORY MATHEMATICS (the names are yours; the structure is not):
  - CONTINUOUS flow variables indexed by TWO sets, x[F,D], each with an explicit finite
    upper bound
  - BINARY open variables indexed by one set, y[F]
  - a demand row per customer:   for d in D: sum f in F: x[f,d] >= dem[d]
  - a PER-PAIR link:             for f in F, d in D: x[f,d] - dem[d]*y[f] <= 0
```

```
  - a capacity row per depot WITHOUT any binary: for f in F: sum d in D: x[f,d] <= cap[f]
  - a minimisation objective over flow cost plus opening cost

SIZE: the problem must have between 111 and 170 variables in total. Aim for about 10 depots and 15 customers.
A two-index family x[A,B] contributes |A| x |B| variables; a one-index family x[A] contributes
|A|. Choose the set sizes so the total lands in range.

FEASIBILITY PROCEDURE (follow it; do not merely aim at the outcome):
  Pick every demand dem[d] as an integer from 5 to 20. Then set EVERY cap[f] to 41, and every opening cost fopen[f] to
↪  an integer from 120 to 260 -- large enough that opening one more depot is a real decision, not free.
  A model with no feasible solution is worth nothing, and so is one whose optimum is obvious.

WHAT YOU MUST NOT DO:
  1. Do not leave any continuous variable without an explicit finite upper bound.
     `var x[F,D] 0 inf continuous` is invalid. State a real number.
  2. Do not write an aggregated single-binary link -- one binary switching off a SUM of two or
     more continuous columns, as in `sum d in D: x[f,d] - 1000*y[f] <= 0`. A per-pair link,
     `x[f,d] - dem[d]*y[f] <= 0`, is the correct form.
  3. Do not use coefficients spanning more than four orders of magnitude. Keep every nonzero
     within a factor of 10,000 of every other.
  4. Do not omit any mandatory row above, and do not add a row that makes another one
     non-binding.

PRECEDENCE, when these instructions appear to conflict:
  the validity gates dominate these instructions, and these instructions dominate the exemplar.
  The exemplar shows ONE way to satisfy the mandate. It is not the only way, and where it
  differs from the instructions above, the instructions win.

EXAMPLE of the required format and mathematics:

MILP tight_facility min
set F 10
set D 14
par cost[F,D]
par fopen[F]
par dem[D]
par cap[F]
var x[F,D] 0 20 continuous
var y[F] 0 1 binary
obj min sum f in F, d in D: cost[f,d]*x[f,d] + sum f in F: fopen[f]*y[f]
con demand: for d in D: sum f in F: x[f,d] >= dem[d]
con link: for f in F, d in D: x[f,d] - dem[d]*y[f] <= 0
con cap: for f in F: sum d in D: x[f,d] <= cap[f]
DATA: {"cost":[[6,12,1,10,5,1,15,9,9,8,13,14,4,7],[1,8,10,13,14,12,11,1,11,8,10,12,11,15],[1,15,7,11,9,14,6,8,11,13,6,
↪ 14,1,4],[13,14,8,11,9,2,7,2,2,12,12,15,11,6],[12,3,11,10,8,11,11,2,7,8,7,15,14,1],[6,9,9,14,3,13,11,1,15,7,3,15,3,
↪ 2],[6,6,4,4,9,5,6,6,15,8,4,4,5,15],[11,3,7,5,11,8,2,12,14,7,14,14,7,11],[11,6,11,11,11,14,14,6,11,3,12,4,14,8],[4,
↪ 2,8,6,1,13,9,13,14,5,14,10,9,13]],"fopen":[198,172,240,154,202,242,255,175,173,165],"dem":[12,14,12,11,11,20,8,13,
↪ 11,15,18,14,20,20],"cap":[22,23,25,23,23,23,25,23,25,23]}

Now write a NEW problem of the same family at the size requested above.
Output exactly ONE problem.

OUTPUT FORMAT — this is strict:
  Begin with the MILP header line. End with the DATA block. Stop immediately after the closing
  brace of the DATA block.
  Write NO comments, NO explanation, NO reasoning, NO checks, and no text before or after the
  model. Every line you write must be part of the model itself.
```

B THE SOLVER-BASED GRPO REWARD FUNCTION

In this section, we present the details of two reward components that were reffered to Appendix section for more details: Validity gate and Structural diversity.

B.1 THE VALIDITY GATE

The validity gate for instance $\mathcal{I}_i$ generated while training is $V(\mathcal{I}_i) = \prod_{k=1}^{6} \mathbf{1}[g_k(\mathcal{I}_i)]$ with the following conditions.

- g_1 **Parse:** The emitted instance expands under the fixed grammar, with all index sets declared, all referenced parameters present and all rows well formed.
- g_2 **Well posed:** The solver returns a feasible and a finite optimal value within the node cap and the wall-clock limit. Infeasible models, unbounded models and killed solves all fail here.
- g_3 **Bounded variables:** Every continuous variable carries a finite upper bound. Without this gate the policy inflates relaxation gaps with unbounded variables which is a route to apparent hardness that vanishes under standard pre-processing.
- g_4 **No aggregated single-binary link:** No constraint gates a sum of continuous variables through a single binary with one large coefficient. The banned pattern couples many continuous columns to one switch, which we found makes instances easier for the solver, not harder, while inflating superficial structure.
- g_5 **Coefficient range:** The ratio of the largest to the smallest nonzero magnitude over the constraint matrix and objective is at most 10^4. This blocks hardness manufactured from numerical ill-conditioning, which is an artefact of tolerances rather than of combinatorial structure.
- g_6 **Family compliance:** Declared index sets, variable types and row families match the family named in the prompt.

B.2 STRUCTURAL DIVERSITY

GRPO only compares completions *within* the same sampling group, so we also measure structural diversity within a group. We compress each instance to a simple fingerprint $\phi(I)$. For each constraint row we record: (i) the row sense, (ii) the number of nonzeros, (iii) how many of those nonzeros sit in integer columns, and (iv) the set of coefficient signs. The fingerprint is the multiset of these row "types". By design it forgets variable names, coefficient magnitudes, and row order. That is deliberate. Two instances that differ only cosmetically should map to the same fingerprint, which otherwise the diversity term would reward mere renaming, which is exactly the failure mode it is meant to avoid.

Let $\mathcal{G} = \{I_1, \ldots, I_m\}$ denote the *valid* completions in a sampling group, i.e., those that pass the validity gate (invalid completions are dropped and receive score 0). Hence m is at most the nominal group size. For each completion I_k, define

$$c_k \;=\; \big|\{\, j : \phi(I_j) = \phi(I_k) \,\}\big|, \qquad \rho_k \;=\; -c_k, \tag{6}$$

where c_k is the number of group members that share I_k's fingerprint, and ρ_k is a simple "rarity" score (less frequent fingerprints yield larger ρ_k). We then turn these rarities into tie-averaged ranks and rescale to $[0, 1]$:

$$r_k \;=\; L_k + \frac{T_k - 1}{2}, \qquad L_k = \big|\{\, j : c_j > c_k \,\}\big|, \qquad T_k = \big|\{\, j : c_j = c_k \,\}\big|, \tag{7}$$

$$r_{\mathrm{div}}(k) \;=\; \begin{cases} \dfrac{r_k}{m-1}, & m > 1, \\ 0.5, & m = 1. \end{cases} \tag{8}$$

Note that L_k and T_k count *completions* (not distinct fingerprints), and T_k includes k itself. By construction, $r_{\mathrm{div}}(k) \in [0, 1]$.

This definition is intentionally neutral in the two degenerate regimes. If a group collapses to a single fingerprint, then every completion has $c_j = m$, the ranks tie, and $r_{\mathrm{div}} \equiv 0.5$. If every fingerprint is unique, then every completion has $c_j = 1$, the ranks tie again, and $r_{\mathrm{div}} \equiv 0.5$ as well. In both cases the term is constant within the group and therefore cannot create spurious advantage differences. The statistic becomes informative only when a group contains a mix of common and rare structures. Because GRPO normalizes rewards within each group, only within-group variation in r_{div} affects the update. The term is zero-sum around 0.5 and it lowers the advantage of over-represented structures and raises that of under-represented ones, discouraging the policy from concentrating mass on a single structure.

In our experiments, each group is drawn from a single family, and families alternate between optimization steps. This ensures the advantage never pits (say) knapsack against max-cut, so r_{div} always measures *within-family* structural variety.

B.3 Worked example

Consider a completion in a group of 64 at the bracket 171–225, so $n^\star = 198$, which parses, passes every gate, is detected as compliant, and solves to optimality with 196 variables, 1207 nodes, $z^\star = z_{\mathrm{IP}} = 4820$, and root dual bound $z_{\mathrm{cut}} = 4510$. Then

$$\begin{aligned} r_{\mathrm{nodes}} &= \log 1207 / \log 5000 = 0.833, \\ r_{\mathrm{cut}} &= 310/(0.10 \times 4820) = 0.643, \\ H &= 0.75 \cdot 0.833 + 0.25 \cdot 0.643 = 0.786, \\ r_{\mathrm{var}} &= \exp(-3 \cdot 2/198) = 0.970, \end{aligned}$$

With a within-group diversity rank of 0.42,

$$R = 0.70 \cdot 0.786 + 0.15 \cdot 0.970 + 0.15 \cdot 0.42 = 0.759. \tag{9}$$

C COMPACT INDEX-SET TEMPLATE

Each instance is written in two parts. The *structure block* describes the model once, its index sets, parameter tables, variable families, a single objective, and constraint families quantified over the sets. The *data block* is a JSON object that provides the values for each parameter table as nested arrays. We never write the coefficient matrix A in equation 1 and instead, a deterministic parser reconstructs it by expanding each family over its index sets. Table 5 summarizes the grammar statements.

Table 5: **Statements of the index-set template:** Each line of the structure block is one statement. S and T are declared index sets, $i \in S$ and $j \in T$ are their running indices, and `op` is one of `<=`, `>=`, `=`. Indices are bound by `sum` and `for` clauses; in a `var` statement, the running index of each set is its lowercase name.

Statement	Meaning
`MILP <family> <min|max>`	Three-token header with problem class, family name and optimization sense.
`set S <size>`	Index set $S = \{1, \ldots, \|S\|\}$.
`par p[S,T]`	Parameter table $p \in \mathbb{R}^{\|S\|\times\|T\|}$, values given in the data block.
`var v[S,T] [<lb> <ub>] <type>`	Variable family v_{ij}, $(i,j) \in S \times T$, with type `continuous`, `integer` or `binary`. A bound is a number or a parameter entry indexed by the running indices of the family, e.g. `dem[d]`. Bounds are omitted for `binary`, which fixes them to $[0, 1]$.
`obj <min|max> <sum> + <sum>`	Objective, a sum of terms such as `sum i in S, j in T: c[i,j]*v[i,j]`.
`con <name>: for i in S{, j in T}: <lhs> op <rhs>`	Constraint family with one row for every element of the product of the listed sets.
`DATA: {"p": [[...],...], ...}`	Numerical values of every declared parameter, nested row-major over its index sets.

Expansion: The parser applies five rules. First, each variable family is instantiated row-major over its index sets, and the columns of equation 1 are ordered by family in declaration order. Second, each constraint family produces one row per element of its `for` sets, again row-major. Third, a product of a parameter entry and a variable, such as `dem[d]*z[f]`, becomes the numerical coefficient of that variable, so every row stays linear. Fourth, variable terms on the right-hand side are moved to the left with their sign reversed. Fifth, repeated occurrences of the same variable in a row are merged by summing their coefficients, so that, e.g., `x[i] + x[j]` with $i = j$ becomes $2x_i$.

C.1 CAPACITATED FACILITY LOCATION

Let F be the set of facilities and D the set of customers. A binary variable z_f opens facility f at fixed cost o_f, and a continuous variable x_{fd} is the amount customer d receives from facility f at unit cost c_{fd}. With demand δ_d and capacity κ_f,

$$\begin{aligned}
\min_{x,\,z} \quad & \sum_{f\in F} o_f\, z_f + \sum_{f\in F}\sum_{d\in D} c_{fd}\, x_{fd} && && (10)\\
\text{s.t.} \quad & \textstyle\sum_{f\in F} x_{fd} \geq \delta_d, && d \in D, && \text{(demand)}\\
& x_{fd} - \delta_d\, z_f \leq 0, && f \in F,\ d \in D, && \text{(link)}\\
& \textstyle\sum_{d\in D} x_{fd} \leq \kappa_f, && f \in F, && \text{(capacity)}\\
& 0 \leq x_{fd} \leq \delta_d, \quad z_f \in \{0,1\}, && f \in F,\ d \in D. &&
\end{aligned}$$

The instance has $n = |F|(1 + |D|)$ variables, of which $|F|$ are binary, and $m = |D| + |F||D| + |F|$ rows. The link is written per facility and customer pair with coefficient equal to the column bound δ_d, and the capacity row carries no binary. This is the disaggregated form discussed in Section 3, so the instance passes gates g_3 and g_4 of Appendix B.1 by construction. The same model in the template reads

```
MILP facility_location min
set F 2
set D 3
par fopen[F]
par cap[F]
par dem[D]
par cost[F,D]
var x[F,D] 0 dem[d] continuous
var z[F] binary
obj min sum f in F: fopen[f]*z[f] + sum f in F, d in D: cost[f,d]*x[f,d]
con demand: for d in D: sum f in F: x[f,d] >= dem[d]
con link: for f in F, d in D: x[f,d] - dem[d]*z[f] <= 0
con capacity: for f in F: sum d in D: x[f,d] <= cap[f]
DATA: {"fopen":[20,70], "cap":[28,32], "dem":[6,18,6],
       "cost":[[1,2,2],[2,4,2]]}
```

Changing $|F|$ or $|D|$ changes the two `set` lines and the array lengths in `DATA`, and nothing else. For this data the columns are $x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}$ followed by z_1, z_2, and the expansion gives $n = 8$ variables and $m = 11$ rows,

$$\begin{aligned} &x_{11} + x_{21} \geq 6, \qquad x_{12} + x_{22} \geq 18, \qquad x_{13} + x_{23} \geq 6, \\ &x_{11} - 6z_1 \leq 0, \qquad x_{12} - 18z_1 \leq 0, \qquad x_{13} - 6z_1 \leq 0, \\ &x_{21} - 6z_2 \leq 0, \qquad x_{22} - 18z_2 \leq 0, \qquad x_{23} - 6z_2 \leq 0, \\ &x_{11} + x_{12} + x_{13} \leq 28, \qquad x_{21} + x_{22} + x_{23} \leq 32, \end{aligned}$$

with objective $20z_1 + 70z_2 + x_{11} + 2x_{12} + 2x_{13} + 2x_{21} + 4x_{22} + 2x_{23}$ and bounds $x_{fd} \in [0, \delta_d]$. Total demand is 30, which exceeds $\kappa_1 = 28$, so facility 1 cannot serve all customers alone. Opening only facility 2 is feasible but costs 166, whereas opening both costs $v^\star = 144$, attained for instance by $x_{11} = 6$, $x_{12} = 18$, $x_{23} = 6$. The optimum is not unique, since customer 3 costs the same from either facility and any split with $x_{13} \leq 4$ is also optimal. The LP relaxation instead sets $z_1 = 14/15$ and $z_2 = 1/15$, just enough to cover the two units that facility 1 cannot supply, and attains $v_{\mathrm{LP}} = 1202/15 \approx 80.13$. Most of the relaxation gap comes from paying the fixed cost of the second facility fractionally.

C.2 Max-cut linearization

Let V be the vertex set and $w_{ij} \geq 0$ the weight of the ordered pair (i, j), with $w_{ii} = 0$. A binary x_i assigns vertex i to one side of the cut, and a binary y_{ij} may equal one only when the pair (i, j) is cut,

$$\begin{aligned} \max_{x,\,y} \quad & \sum_{i \in V} \sum_{j \in V} w_{ij}\, y_{ij} && && (11) \\ \text{s.t.} \quad & y_{ij} - x_i - x_j \leq 0, && i, j \in V, && \text{(cut_a)} \\ & y_{ij} + x_i + x_j \leq 2, && i, j \in V, && \text{(cut_b)} \\ & x_i \in \{0, 1\}, \quad y_{ij} \in \{0, 1\}, && i, j \in V. \end{aligned}$$

Row (cut_a) sets $y_{ij} = 0$ when both endpoints have $x = 0$, and row (cut_b) sets $y_{ij} = 0$ when both have $x = 1$. For $w_{ij} > 0$ an optimal solution therefore has $y_{ij} = 1$ exactly when $x_i \neq x_j$. For $w_{ij} = 0$, y_{ij} is unconstrained on cut pairs and does not affect the objective. The same two rows force $y_{ii} = 0$. The instance has $n = |V| + |V|^2$ binary variables and $m = 2|V|^2$ rows, and the pair $\{i, j\}$ enters the objective with effective weight $w_{ij} + w_{ji}$. In the template,

```
MILP max_cut max
set V 3
par w[V,V]
var x[V] binary
var y[V,V] binary
obj max sum i in V, j in V: w[i,j]*y[i,j]
con cut_a: for i in V, j in V: y[i,j] - x[i] - x[j] <= 0
con cut_b: for i in V, j in V: y[i,j] + x[i] + x[j] <= 2
DATA: {"w":[[0,4,1],[0,0,3],[2,0,0]]}
```

The expansion gives $n = 12$ and $m = 18$. For the pair $(1, 2)$, for instance, it produces $y_{12} - x_1 - x_2 \leq 0$ and $y_{12} + x_1 + x_2 \leq 2$, and for the diagonal entry $(1, 1)$ it produces $y_{11} - 2x_1 \leq 0$ and $y_{11} + 2x_1 \leq 2$. The effective weights are 4 on $\{1, 2\}$, 3 on $\{2, 3\}$ and $1 + 2 = 3$ on $\{1, 3\}$, so the total off-diagonal weight is $W_{\text{off}} = 10$. A triangle can cut at most two of its three edges, and the optimum $v^\star = 7$ is attained by separating vertex 2 from $\{1, 3\}$, or equally vertex 1 from $\{2, 3\}$. The point $x_i = \frac{1}{2}$, $y_{ij} = 1$ satisfies every row, and $y_{ij} \leq 1$, so the LP relaxation attains $v_{\text{LP}} = \sum_{i,j} w_{ij} = W_{\text{off}} = 10$, using $w_{ii} = 0$. The relative relaxation gap $(v_{\text{LP}} - v^\star)/v_{\text{LP}}$ is then $1 - s$ with cut share $s = v^\star/W_{\text{off}} = 0.7$.

C.3 Length of the compact and expanded encodings

Table 6 compares our template with a fully expanded encoding that spells out every objective term, constraint row, and variable declaration using short identifiers (e.g., `x_3_7`, `y_2_5`), evaluated on the training geometries in Appendix E. For each instance, we sample data uniformly from the prompt ranges and in the max-cut weight matrix, roughly half of the off-diagonal entries are zero. We report lengths in characters. Because the structure block is fixed within each problem family, the compact encoding grows only through its data block. For these dense families the data block and the number of nonzeros in A are of the same order, $O(|F||D|)$ and $O(|V|^2)$ respectively, so the saving is a constant factor, driven by writing each parameter value once instead of spelling out identifiers and operators for every nonzero.

Table 6: **Encoding length in characters:** n and m are the numbers of variables and constraints after expansion. Compact is the index-set template including its data block, and Expanded writes every constraint explicitly. Ratio is Compact divided by Expanded.

Family	Geometry	n	m	Compact	Expanded	Ratio
Facility location	$\lvert F\rvert$=8, $\lvert D\rvert$=12	104	116	786	7,827	0.10
Facility location	$\lvert F\rvert$=10, $\lvert D\rvert$=15	160	175	962	12,279	0.08
Max-cut	$\lvert V\rvert$=9	90	162	438	6,401	0.07
Max-cut	$\lvert V\rvert$=12	156	288	585	11,727	0.05

D Implementation Details

D.1 Size curriculum

We use instance size (number of variables) as the curriculum axis. The direct arms (OptiScribe-12B-D, OptiScribe-4B-D) train only on the target bracket, 171–225 variables. The curriculum arms (OptiScribe-12B, OptiScribe-4B) move through three rungs, 76–110 $\rightarrow$ 111–170 $\rightarrow$ 171–225, training for 61 steps per rung and initializing each rung from the previous rung's checkpoint.

The smaller rungs are mostly about teaching the model to produce outputs of the right length. As instances get larger, they more often hit the token cap. A truncated instance typically fails to parse and therefore gets a score of zero. When Gemma-4-12B is trained directly on the largest bracket, the valid share drops from over 90% to under 60% and training becomes unstable, so we stop that run after 47 of 61 updates (Appendix G.1). Warm-starting each rung from a model that already produces complete instances at a slightly smaller size largely avoids this failure mode. By contrast, both Qwen arms stay near 100% validity throughout, so the curriculum is important for Gemma but not for Qwen.

We also need each rung to contain instances that actually branch, otherwise the node-based reward term is identically zero and GRPO gets no hardness signal. For each rung, we therefore screen candidate index-set sizes with the training solver and keep those whose instances branch at least as often as the target-bracket configuration (Appendix E). In this screen, CFL tends to branch when facilities are few and customers are many, while multiple knapsack never branches at 76–110 variables. As a result, the first rung trains only CFL and Max-Cut, and knapsack is introduced at the second rung.

D.2 Training configuration

Each arm starts from its model's instruction-tuned checkpoint and is fine-tuned with LoRA. We compute GRPO statistics within a single problem family and size bracket, alternating families across steps. All other hyperparameters are held fixed across arms and rungs (Table 7); the only difference is the size-bracket schedule.

Table 7: **Training and reward configuration.** Settings are fixed across arms and rungs. Only the size-bracket schedule differs.

Policy and optimization		*Reward and solver*	
Base models	Gemma-4-12B-it, Qwen3.5-4B	N_{ref}	5,000
Adaptation	LoRA, rank 16	Node cap	50,000
Algorithm	GRPO, group $G = 64$	Wall-clock kill (training)	20 s
Learning rate	5×10^{-5}	In-loop solver	SCIP 10.0
KL coefficient β	0.1	Evaluation-only solvers	HiGHS, Gurobi 13.0.3
Decoding	$T = 1.0$, top-$p = 0.95$		
Steps per rung	61		

D.3 Measurement

We proxy instance hardness using (i) the branch-and-bound node count and (ii) the post-root-cut optimality gap $|z^\star - z_{\text{cut}}|/|z^\star|$. We do not use wall-clock solve time because it is machine-dependent. Within each model, all arms are run on the same prompts with matched random seeds, so we make paired comparisons seed-by-seed. For each table cell we generate completions from three prompts that are identical except for the in-context exemplar. This tests sampling *depth* over exemplars rather than *breadth* over many distinct prompts. Each Gemma cell contains 256 completions and each Qwen cell contains 128.

If an instance hits the node cap, we keep it and record the node count at the cap. Consequently, high-percentile node statistics should be interpreted as lower bounds. If a solve is terminated by the wall-clock limit, we exclude it from the node and gap summaries, but we still count it in the denominators for parse and feasible rates. The difference between these rates therefore equals the share of killed runs. The default evaluation limit is 300 s for two smaller brackets and 3,600 for larger two brackets and for the language-control study.

E RUNG GEOMETRIES

What we mean by a geometry: A curriculum rung sets a target variable-count bracket, but an instance is determined by index-set sizes (facilities/customers, vertices, items/knapsacks). For each family and bracket we pick one fixed *geometry* (index-set cardinalities), which uniquely fixes the number of variables n (e.g., facility location: $n = |F|(1+|D|)$; max-cut: $n = |V|+|V|^2$; knapsack: $n = |I||K|$). We pass this to the model as a single sentence specifying the bracket and a plausible shape (e.g., "111–170 variables; about 10 depots and 15 customers"). The in-context exemplar is always shown at a fixed default geometry (facility location $n = 150$, max-cut $n = 182$, knapsack $n = 192$). At small rungs it can be larger than the requested bracket, so the prompt explicitly tells the model to follow the instructions, not the exemplar.

Why we screen geometries: Some shapes are uninformative. If most instances solve at the root, the node-based reward is near zero and GRPO sees little within-group variation. We therefore want geometries that are not only hard, but also have noticeable spread.

Screening protocol: Before training (no LM involved), we screened candidate geometries using the family's reference instance builder. For each geometry we generated 12 randomized instances and solved them with SCIP under the training configuration (node cap 5×10^4, 20 s time limit, $N_{\text{ref}} = 5000$ in H equation 3). All 336 runs finished. This is only a sanity check. If correctly built instances never branch, model-written ones may not either. Each family already had a pre-curriculum "target-bracket" geometry (171–225 variables) and we treat this as an *anchor* baseline. We kept a candidate geometry only if (i) at least as many of its 12 instances branched as the anchor (facility location/max-cut: 100%; knapsack: 75%), and (ii) the standard deviation of H across the 12 instances exceeded 0.02. Among those that passed, we kept (per family and bracket) the geometry with the largest σ_H.

Outcome: Table 8 lists all 28 rows (3 anchors, 25 candidates). Seven candidates passed and five were kept: facility location $|F|$=8, $|D|$=12 (n=104) and $|F|$=10, $|D|$=15 (n=160); max-cut $|V|$=9 (n=90) and $|V|$=12 (n=156); knapsack $|I|$=42, $|K|$=4 (n=168). For max-cut, the two non-kept passing candidates lost a near-tie (e.g., 0.112 vs. 0.110 in the 76–110 bracket), which 12 instances can barely separate. For facility location, the branching-rate check was the bottleneck and some rejected shapes had higher median H but branched on only 58–92% of instances. No knapsack geometry in the 76–110-variable bracket passed. Across eight candidates with $|K| = 2$ to 5, at most half of the instances branched (vs. the anchor's 75%), and six of the eight have median H exactly zero. This means the median instance is solved at the root with no post-cut gap. Knapsack is therefore absent from the first bracket training and only enters at the second. Since the first rung trains facility location and max-cut only, its curriculum effect is not fully separable from the change in family mix.

Table 8: **Screened rung geometries:** Each row is 12 constructed instances solved by SCIP in the training configuration. n is the variable count, which equals the realized count in every row. Nodes and H are medians over the 12 instances, σ_H is the standard deviation of H, and >1 node is the share of instances that branch. Status A marks the family's target-bracket anchor, which sets the branching bar. ✓ marks the kept geometry, ∘ a candidate that passed both conditions but lost the tie-break, and a blank a rejected candidate.

Family	Geometry	n	Bracket	Nodes	H	σ_H	>1 node (%)	Status
Facility location	$\|F\|=14, \|D\|=13$	196	171–225	48.5	0.426	0.155	100.0	A
	$\|F\|=12, \|D\|=7$	96	76–110	11.0	0.165	0.205	75.0	
	$\|F\|=10, \|D\|=9$	100	76–110	9.0	0.251	0.203	75.0	
	$\|F\|=13, \|D\|=7$	104	76–110	10.5	0.221	0.176	58.3	
	$\|F\|=8, \|D\|=12$	104	76–110	7.0	0.173	0.124	100.0	✓
	$\|F\|=13, \|D\|=10$	143	111–170	38.5	0.399	0.170	91.7	
	$\|F\|=12, \|D\|=12$	156	111–170	34.5	0.341	0.155	91.7	
	$\|F\|=10, \|D\|=15$	160	111–170	36.0	0.421	0.081	100.0	✓
	$\|F\|=15, \|D\|=10$	165	111–170	47.0	0.376	0.213	83.3	
Max-cut	$\|V\|=13$	182	171–225	50.5	0.580	0.108	100.0	A
	$\|V\|=9$	90	76–110	7.0	0.407	0.112	100.0	✓
	$\|V\|=10$	110	76–110	12.0	0.427	0.110	100.0	∘
	$\|V\|=11$	132	111–170	18.0	0.504	0.086	100.0	∘
	$\|V\|=12$	156	111–170	30.0	0.549	0.116	100.0	✓
Multiple knapsack	$\|I\|=48, \|K\|=4$	192	171–225	2.0	0.061	0.058	75.0	A
	$\|I\|=48, \|K\|=2$	96	76–110	1.0	0.000	0.080	25.0	
	$\|I\|=32, \|K\|=3$	96	76–110	1.0	0.000	0.030	16.7	
	$\|I\|=24, \|K\|=4$	96	76–110	1.0	0.000	0.030	16.7	
	$\|I\|=34, \|K\|=3$	102	76–110	1.0	0.000	0.065	41.7	
	$\|I\|=36, \|K\|=3$	108	76–110	1.5	0.031	0.129	50.0	
	$\|I\|=27, \|K\|=4$	108	76–110	1.0	0.000	0.093	16.7	
	$\|I\|=55, \|K\|=2$	110	76–110	1.0	0.000	0.029	33.3	
	$\|I\|=22, \|K\|=5$	110	76–110	1.0	0.000	0.017	8.3	
	$\|I\|=36, \|K\|=4$	144	111–170	1.0	0.000	0.043	41.7	
	$\|I\|=40, \|K\|=4$	160	111–170	1.5	0.031	0.118	50.0	
	$\|I\|=42, \|K\|=4$	168	111–170	2.0	0.061	0.131	75.0	✓
	$\|I\|=28, \|K\|=6$	168	111–170	1.0	0.000	0.017	8.3	
	$\|I\|=34, \|K\|=5$	170	111–170	1.0	0.000	0.182	41.7	

F THE P0 GENERATION PROMPT

P0 is the prompt used for the training where reward was designed using pre-cut gap.

F.1 SYSTEM MESSAGE

Listing 5: P0 system message.

```
You are OptiScribe, an expert in operations research and mathematical optimization. You create realistic, well-posed
↪ MIXED-INTEGER Linear Programming (MILP) problems in a COMPACT index-set format that separates structure (written
↪ once) from data (numbers in a JSON block).

You draw from deep knowledge of real-world optimization: manufacturing, logistics, finance, energy systems, healthcare
↪  operations, and more. Your problems have realistic coefficients, meaningful constraints, integer/binary decisions
↪ , and clear optimization objectives.

## Compact index-set format (MILP)

The compact index-set format describes a Linear Program as a STRUCTURE block (text) plus a DATA block (one JSON line).
↪  The constraint matrix is NEVER written out -- it is implied by index patterns and rebuilt by expansion. Write
↪ each variable family and constraint family ONCE; never enumerate variables one by one.

STRUCTURE lines (in this order):
  LP <name> <min|max>                     problem class and objective sense
  set <NAME> <size>                        an index set, e.g.  set I 10
  par <name>[<SET>,...]                    a parameter table; its numbers live in DATA
  var <name>[<SET>,...] <lb> <ub> <type>    a variable family over those sets; type = continuous
  obj <min|max> sum <i in S, j in T,...>: <terms>      objective summed over index sets
  con <label>: for <i in S,...>: sum <j in T,...>: <lhs> <op> <rhs>    a CONSTRAINT FAMILY
  con <op> <rhs>: <idx>:<coef> ...          (optional) one explicit row, vars by 0-based index
  <op> is one of  <=  >=  = .  <rhs> is a number or a parameter ref like cap[i].
  Terms look like  c[i,j]*x[i,j]  (parameter times variable) or just  x[i,j].

DATA line:
  DATA: {"<par>": <nested JSON array indexed by its sets, row-major>, ...}

WORKED EXAMPLE -- a 2x3 transportation problem (6 variables):
  LP transport min
  set P 2
  set M 3
  par c[P,M]
  par sup[P]
  par dem[M]
  var x[P,M] 0 inf continuous
  obj min sum p in P, m in M: c[p,m]*x[p,m]
  con supply: for p in P: sum m in M: x[p,m] <= sup[p]
  con demand: for m in M: sum p in P: x[p,m] >= dem[m]
  DATA: {"c":[[4,6,5],[7,3,8]],"sup":[50,60],"dem":[20,30,25]}
This expands to variables x_0_0..x_1_2, two supply rows and three demand rows.

RULES:
- Index variables in a family range over their declared sets; x[i,j] is the variable at position (i,j).
- The NUMBER OF DECISION VARIABLES equals the product of the variable family's set sizes.
- Every parameter referenced must be declared with `par` and given values in DATA, shaped by its sets.
- The problem must be feasible and bounded.

## MILP extension (integer and binary variables)

This is a MIXED-INTEGER linear program. The format is identical to the LP index-set format above,
with ONE extension to the `var` line's type slot, which is now one of `continuous | integer | binary`:

  var x[I,J] 0 inf continuous     a continuous family (as in LP)
  var y[I]   0 5   integer        a general-integer family, explicit bounds [0,5]
  var z[I,J] binary               a BINARY family -- its bounds are IMPLICIT 0/1 (write no bounds)

RULES for integer/binary variables:
- A `binary` variable is ALWAYS in {0,1}. Do NOT write its bounds, and NEVER write a constraint row
  that restates a 0/1 bound (e.g. `z[i] <= 1`); declare it with the `binary` type instead.
- Put a general integer's bounds in its `var` DECLARATION (`var y[I] 0 5 integer`), never as rows.
- Mixed-type linear rows are allowed and encouraged: a big-M linking row couples a continuous
  variable to a binary decision, e.g. `sum j: x[i,j] - 100*z[i] <= 0` forces all `x[i,.]=0` when
  z[i]=0 and permits flow up to the big-M when z[i]=1.

WORKED MILP EXAMPLE -- capacitated fixed-charge (6 continuous flows + 3 binary open/close):
  MILP fixed_charge min
  set I 3
  set J 2
  par f[I]
```

```
  par c[I,J]
  par d[J]
  var x[I,J] 0 inf continuous
  var z[I] binary
  obj min sum i in I, j in J: c[i,j]*x[i,j] + sum i in I: f[i]*z[i]
  con demand: for j in J: sum i in I: x[i,j] >= d[j]
  con link:   for i in I: sum j in J: x[i,j] - 100*z[i] <= 0
  DATA: {"f":[50,40,60],"c":[[4,6],[7,3],[5,8]],"d":[8,5]}
This is a GENUINE MILP: its LP relaxation opens facilities fractionally, so the integer optimum
differs from the relaxation (a non-trivial integrality gap). `demand` couples the flow variables;
`link` is a big-M family coupling flow to the open/close binaries; no binary is bounded by a row.

## Output Rules

Output ONLY the formulation: the structure lines, then a single final line that starts with "DATA:" and contains the
↪ JSON. Start your output with "MILP ". No explanations, no markdown, no other text.
```

F.2 User message

Braces mark the three per-prompt quantities (Table 9). Everything else is fixed.

Listing 6: P0 user message. {exemplar} is the few-shot instance of Listing 7.

```
Here are examples of MILP problems in the COMPACT index-set format:

<example_1>
{exemplar}
</example_1>

Generate a NEW, original MIXED-INTEGER LP problem about **{domain}** in the SAME compact index-set format.

SPECIFICATIONS:
- The EXPANDED problem must have about {n} decision variables. Choose index-set sizes whose PRODUCT is about {n} (e.g.
↪  for ~100 use two sets of size 10; for ~12 use sizes 3 and 4).
- About {k} of those decision variables should be INTEGER or BINARY (declare them with the `integer`/`binary` type).
↪ The rest are continuous.
- It must be a GENUINE MILP: the integer/binary decisions must actually matter, so that the LP relaxation (dropping
↪ integrality) is NOT already integral. Use integer/binary variables to model indivisible choices -- open/close,
↪ select, count, assign -- not just continuous quantities rounded.
- Use 2 to 4 DISTINCT constraint families with DIFFERENT index patterns. At least TWO must be COUPLING families --
↪ families whose expanded rows each sum over an index set of size >= 2, so every row links MULTIPLE variables (a row
↪ -sum family, a column-sum family, a weighted budget/knapsack `sum i: w[i]*z[i] <= C`, or a set-cover/assignment
↪ family `sum i: z[i,j] = 1`).
- Demonstrate BIG-M LINKING as a first-class pattern: couple a continuous variable to a binary decision with a row
↪ like `for i in I: sum j in J: x[i,j] - M[i]*z[i] <= 0`, where the big-M is a NUMERIC constant (e.g. `- 1000*z[i]`)
↪  or an indexed parameter `M[i]` declared with `par M[I]`, so that z[i]=0 shuts the continuous variables off and z[
↪ i]=1 permits them up to the big-M.
- Put simple per-variable limits in the `var` DECLARATION as bounds (`var x[I,J] 0 50 continuous`, `var y[I] 0 5
↪ integer`), NOT as constraint rows. NEVER write a constraint row that restates a declared bound. FORBIDDEN: writing
↪  a binary's 0/1 bounds as rows (e.g. `z[i] <= 1`) -- declare it with the `binary` type, which makes 0/1 implicit.
- Express the objective and constraints as index-set families (`sum i in S, j in T: ...`, `for i in S: sum j in T:
↪ ...`) so the structure is written ONCE -- do NOT enumerate variables individually.
- Put every numeric coefficient in the final DATA line as JSON arrays shaped by the sets.
- Use realistic coefficients for the domain.
- The problem MUST be feasible and bounded.

HOW CONSTRAINT ROWS EXPAND (your output is expanded and solved as a MILP):
- `con lbl: for i in S: sum j in T: ... <op> ...` expands to |S| rows; each row sums |T| variables.
- `con lbl: for i in S, j in T: ... <op> ...` expands to |S|*|T| single-variable rows -- use sparingly, only for
↪ genuinely per-cell limits.
- `con lbl: sum i in S, j in T: ... <op> ...` (no `for`) is 1 global row over all |S|*|T| variables.

HOW TO ENSURE FEASIBILITY (your output is expanded and solved by a MIP solver):
Build the numbers around a feasible integer operating point so that adding more constraints never makes the problem
↪ infeasible:
1. Put simple bounds in the var DECLARATION; keep binaries as the `binary` type (implicit 0/1).
2. Pick a concrete feasible point: choose which binaries are 1 (e.g. enough facilities open), then a modest continuous
↪ /integer allocation consistent with those choices.
3. Set each right-hand side FROM that point with a little slack:
   - for a `<=` row, choose rhs >= (the row's left-hand side at the point);
   - for a `>=` row, choose rhs <= (the row's left-hand side at the point);
   - for an `=` row, choose rhs = (the row's left-hand side at the point).
4. For a big-M row `sum x - M*z <= 0`, pick M at least as large as the largest total the continuous variables can
↪ reach, so an open (z=1) facility is not artificially throttled.
5. Do not put contradictory families on the same variables.

Output the structure lines, then a single final line starting with "DATA:". Start with "MILP " -- nothing else before
↪ or after.
```

F.3 FEW-SHOT EXEMPLAR

Every P0 training prompt at *every* curriculum bracket carried the same 5-variable instance (Listing 7).

Listing 7: The fixed training exemplar: 5 variables, 21 lines.

```
MILP optmath_milp_86390 max
set S0 4
set S1 1
par c_x1[S1]
var z0[S0] binary
var x1[S1] 0 inf continuous
obj max sum i in S1: c_x1[i]*x1[i]
con = 2: 0:1 1:1 2:1 3:1
con <= 2000012: 4:1 0:1000000 1:1000000
con <= 2000011: 4:1 0:1000000 2:1000000
con <= 2000013: 4:1 0:1000000 3:1000000
con <= 2000013: 4:1 1:1000000 0:1000000
con <= 2000012: 4:1 1:1000000 2:1000000
con <= 2000011: 4:1 1:1000000 3:1000000
con <= 2000012: 4:1 2:1000000 0:1000000
con <= 2000015: 4:1 2:1000000 1:1000000
con <= 2000013: 4:1 2:1000000 3:1000000
con <= 2000011: 4:1 3:1000000 0:1000000
con <= 2000011: 4:1 3:1000000 1:1000000
con <= 2000015: 4:1 3:1000000 2:1000000
DATA: {"c_x1":[1.0]}
```

F.4 PER-PROMPT SAMPLED QUANTITIES

The integer target is stated as a **count** ("About k of those decision variables…"), not a percentage. Sampling ρ uniformly and rounding would request fractions that do not exist at small n, at $n = 14$ the attainable values are $k/14$, spaced 0.071 apart, so the model would be penalised for a rounding error it cannot avoid. Drawing k first removes that.

Table 9: The three quantities resampled for every P0 prompt. n is drawn uniformly from the current curriculum bracket; the integer *count* k is drawn first and the fraction derived as $\rho^\star = k/n$, so that the requested fraction is always attainable at that n.

Symbol	Meaning	Distribution
n	target variable count	$\mathcal{U}$(curriculum bracket)
k	target integer/binary count	$\mathcal{U}\{\lceil 0.05n \rceil, \ldots, \lfloor 0.50n \rfloor\}$
$\rho^\star$	implied integer fraction	k/n (derived, not sampled)
domain	application domain string	uniform over a fixed pool of 25 domains

G Additional Results

G.1 Training dynamics

Figure 5 tracks each reward term during training, with rejected completions scored zero. By construction the diversity term averages exactly 0.5 over valid completions (Appendix B.2), so its curve is half the valid share. It shows how direct training fails on Gemma. The valid share of OptiScribe-12B-D falls from above 90% to below 60%, and its node score falls with it. The run became unstable and we stopped it after 47 updates. OptiScribe-12B dips when it moves to a larger rung and recovers within that rung. Both Qwen models stay near a valid share of 100%, including OptiScribe-4B-D, which trains on the target size from the start. This is why the curriculum matters for Gemma and not for Qwen.

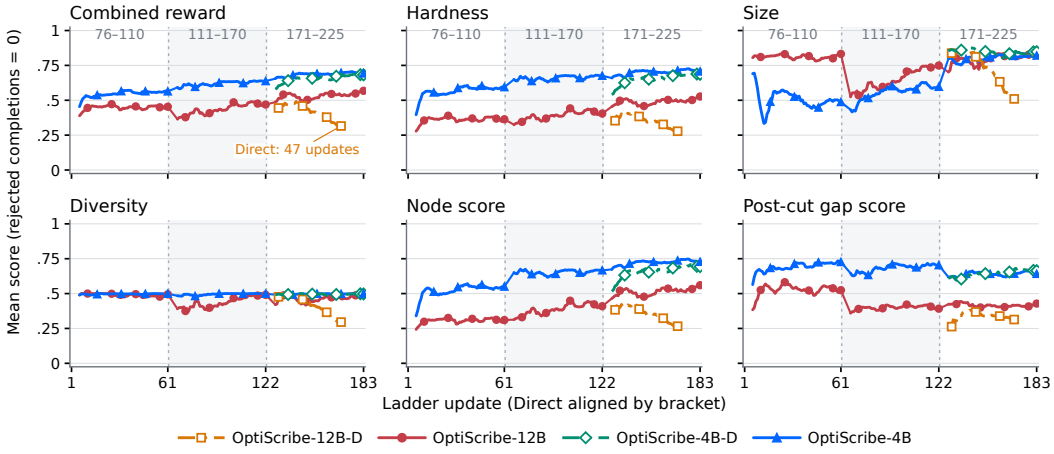

Figure 5: **Training curves:** Mean score per update over all completions, with rejected completions scored 0. Panels show the combined reward R, the hardness H, the size term r_{var}, the diversity term r_{div}, the node score r_{node} and the post-cut gap score r_{cut}. Curriculum models train 61 updates on each of 76–110, 111–170 and 171–225. Direct models train on 171–225 only and are aligned with the last rung. OptiScribe-12B-D became unstable and was stopped after 47 updates. The diversity term averages exactly 0.5 over valid completions, so its curve is half the valid share.

G.2 Multiple knapsack

Table 10 reports multiple knapsack, the third training family, at the two brackets on which it was trained. No model makes it hard. Median node counts stay between 3 and 5 and median gaps at 4–5 $\times 10^{-3}$ for every model. Knapsack also sat out the first curriculum rung because no geometry at 76–110 variables branched (Appendix E). We regard it as an open case for the method.

Table 10: **Multiple knapsack under SCIP at the two trained brackets.** Columns as in Table 2. No model generates hard knapsack instances, and training leaves node counts and gaps essentially unchanged.

Bracket	Arm	Parse ↑	Feas ↑	Nodes ↑	Gap ↑
111–170	Gemma-4-12B	88.3	85.5	3	5
	OptiScribe-12B-D	77.0	75.4	3	5
	OptiScribe-12B	**93.4**	**91.4**	**5**	5
171–225	Gemma-4-12B	**91.8**	**90.2**	**5**	4
	OptiScribe-12B-D	69.1	64.5	3	4
	OptiScribe-12B	87.5	82.8	**5**	4

G.3 STANDARD SYNTHETIC GENERATORS

Learning-for-MILP papers often benchmark on four synthetic families, i.e., set cover, independent set, combinatorial auction, and capacitated facility location, available in Ecole (Gasse et al., 2019; Prouvost et al., 2020). We ran these generators and solved them with the same SCIP settings as our instances (Table 11). In the 111–500 variable range, 55% solve at the root and none exceeds 189 nodes. Even at the published sizes (up to 10,100 variables) the median is 21 nodes and the maximum 853. Facility location is the closest structural match, but despite a similar integer share (generator: 0.059, our CFL: 0.063), OPTISCRIBE-12B needs much more search (median 347 vs 1 node), likely because these generators were tuned around SCIP 6 while SCIP 10 presolves and cuts more aggressively.

Table 11: **Standard synthetic generators vs. OPTISCRIBE-12B under SCIP.** Four families from Gasse et al. (2019) (Ecole; Prouvost et al., 2020). Top block rescales to 111–500 variables, bottom uses published sizes. *Variables* is the median variable count. *Branching* is the share needing > 1 node. SCIP uses a 50,000-node cap and a 300 s limit. No standard-generator instance hits either. OPTISCRIBE-12B node stats exclude 300 s timeouts (2.6% CFL, 6.4% Max-Cut).

Source	Variables	Nodes p50	Nodes p90	Nodes max	Branching (%)
Standard generators, 111–500 *variables,* 250 *instances each*					
Combinatorial auction	319	4	11	70	78.0
Independent set	299	1	15	189	48.8
Set cover	304	1	5	28	33.6
Facility location	272	1	3	20	18.8
Pooled	–	1	8	189	44.8
Standard generators, published sizes, 25 *instances each*					
Combinatorial auction	500	13	50	120	100.0
Independent set	500	15	229	853	84.0
Set cover	1,000	37	387	590	100.0
Facility location	10,100	136	384	492	100.0
Pooled	–	21	271	853	96.0
OPTISCRIBE-*12B,* 111–500 *variables, standard prompt*					
CFL	196	347	4,655	50,000	96.3
Max-Cut	210	634	2,643	14,678	100.0

G.4 PUBLIC BENCHMARK SOURCES

Table 12 lists the public sources pooled in Section 4.2. From each source we keep instances with 111–500 variables. We then split them by variable type: mixed-integer instances are compared against our CFL pools, and pure-integer instances against our Max-Cut pools.

Table 12: Public MILP instance sources used in Section 4.2.

Source	Description
MIPLIB 2010 (Koch et al., 2011)	Fifth edition of the Mixed Integer Programming Library.
MIPLIB 2017 (Gleixner et al., 2021)	Sixth edition of the Mixed Integer Programming Library, compiled with a data-driven selection procedure.
MILP-Evolve (Li et al., 2025)	MILP problem classes generated by an evolutionary framework based on large language models (LLMs).
D-MIPLIB (Huang et al., 2024)	Distributional MIPLIB: a multi-domain library of MILP instance distributions for machine-learning-guided methods.
DIG-MILP (Wang et al., 2024)	Instances from the DIG-MILP study, which uses a deep generator (a variational autoencoder, VAE) that guarantees feasibility.
MIPcc23 (Bolusani et al., 2024)	MIP Workshop 2023 Computational Competition on reoptimization. Each instance series contains related instances of the same size.
MIPLearn (Xavier et al., 2024)	Benchmark problems distributed with the MIPLearn framework for learning-enhanced optimization.
MIRPLIB (Papageorgiou et al., 2014)	Library of maritime inventory routing problem instances.

G.5 NODE COUNTS OF ALL MODELS UNDER THREE SOLVERS

Table 13 extends Figure 3 to all three Gemma models. The ordering of Table 2 holds under each solver. OPTISCRIBE-12B has the highest median in every cell except CFL at 111–170 under Gurobi, where the median instance of every model is solved at the root. On CFL, OPTISCRIBE-12B-D is at or below the base model under every solver and in every bracket. Absolute counts differ by solver. Gurobi needs far fewer nodes than SCIP on CFL and more on Max-Cut.

Table 13: **Median branch-and-bound nodes under three solvers.** All three Gemma models, both families and all four brackets. SCIP is the training solver. HiGHS and Gurobi never enter training. Max-Cut instances are solved as maximization. Bold marks the highest median in each bracket and solver. OPTISCRIBE-12B is highest in every cell except CFL at 111–170 under Gurobi, where the median instance of every model is solved at the root.

		CFL			Max-cut)		
Bracket	Arm	SCIP ↑	HiGHS ↑	Gurobi ↑	SCIP ↑	HiGHS ↑	Gurobi ↑
111–170	Gemma-4-12B	22	19	1	41	27	10
	OPTISCRIBE-12B-D	21	16	1	41	29	15
	OPTISCRIBE-12B	**109**	**79**	1	**142**	**59**	**214**
171–225	Gemma-4-12B	224	277	58	81	41	36
	OPTISCRIBE-12B-D	170	179	2	85	41	44
	OPTISCRIBE-12B	**391**	**491**	**69**	**364**	**158**	**982**
226–350	Gemma-4-12B	293	480	53	423	200	412
	OPTISCRIBE-12B-D	211	194	1	369	163	302
	OPTISCRIBE-12B	**710**	**956**	**126**	**1002**	**873**	**4379**
351–500	Gemma-4-12B	420	539	69	1350	1826	4363
	OPTISCRIBE-12B-D	194	230	1	1096	1303	3353
	OPTISCRIBE-12B	**1014**	**1539**	**176**	**2578**	**5041**	**6101**

G.6 TRIVIAL INSTANCES

Figure 6 reports the share of instances each solver solves at the root node. For OPTISCRIBE-12B, which was trained with SCIP, SCIP and HiGHS solve at most 7% of CFL instances at the root and almost no Max-Cut instances. Gurobi solves 35–52% of CFL instances at the root but almost no Max-Cut instances. The pre-cut model, trained with HiGHS, behaves differently. HiGHS solves at most 6% of its instances at the root, while SCIP solves 33–57% and Gurobi 38–49%. Its instances are still solved at the root less often than the base model's on the same prompt, which SCIP and Gurobi solve at the root 85–98% of the time.

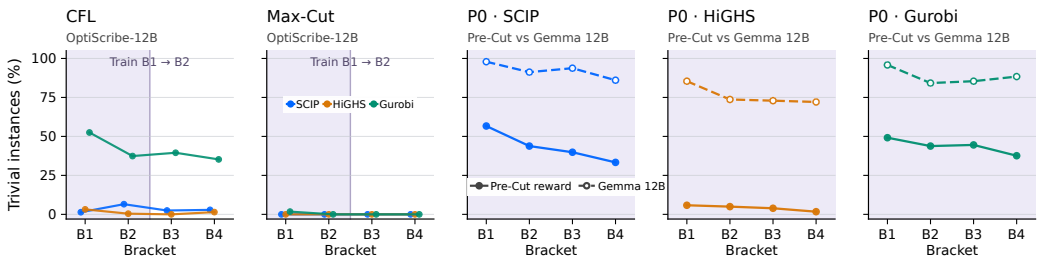

Figure 6: **Share of instances solved at the root under three solvers.** The two left panels show OPTISCRIBE-12B on CFL and Max-Cut, with shading on the training brackets. The three right panels show prompt P0 under SCIP, HiGHS and Gurobi, for the pre-cut model (solid) and untrained Gemma-4-12B (dashed). The pre-cut model was trained with HiGHS in the loop. HiGHS rarely solves its instances at the root, but SCIP and Gurobi often do.

G.7 SENSITIVITY TO THE NODE REFERENCE

Figure 7 compares two OPTISCRIBE-4B runs that differ in N_{ref} and in the training node cap. Inside the training range their node and gap gains are similar. Beyond it they differ on nodes and parsing but not on the gap. On CFL, $N_{\text{ref}} = 50{,}000$ gives larger node gains at B3 and B4 ($24.5\times$ and $5.7\times$ against $7.6\times$ and $1.6\times$). On Max-Cut it gives a smaller node gain at B4 ($6.6\times$ against $11.6\times$), and its parse rate falls 13–15 points below the base model at B3 and B4. Post-cut gap gains stay close to each other at every size. Neither setting is better on every metric.

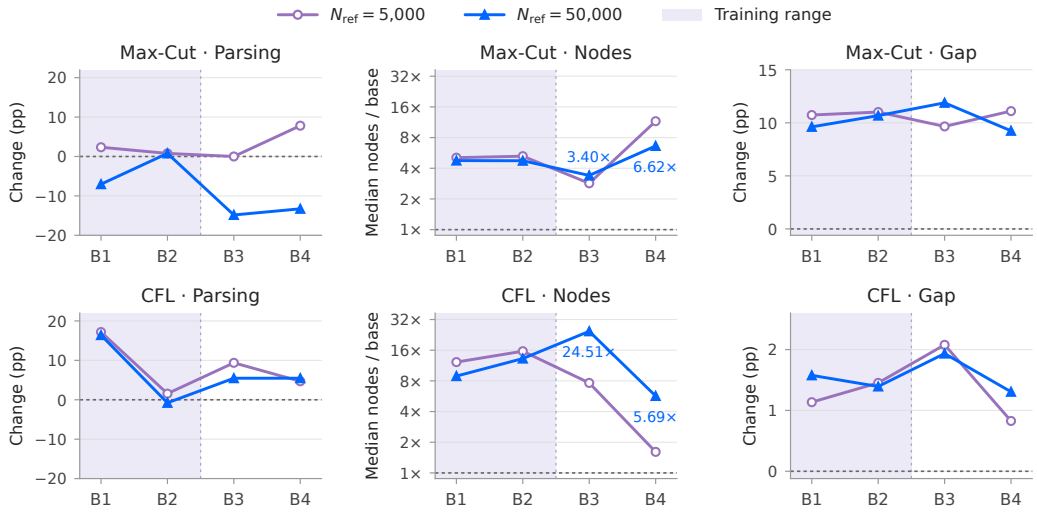

Figure 7: **Sensitivity to the node reference N_{ref} for OPTISCRIBE-4B.** Each panel shows the change against untrained Qwen3.5-4B for a model trained with $N_{\text{ref}} = 5{,}000$, the default, and one trained with $N_{\text{ref}} = 50{,}000$. The top row is Max-Cut and the bottom row is CFL. Parsing and Gap are changes in percentage points, where Gap is the median post-root-cut gap. Nodes is the ratio of median node counts. Shading marks the training brackets. The two runs also differ in their training node cap.

G.8 LANGUAGE CONTROL DETAILS

Table 14 lists the prompt conditions. On Max-Cut the density request is a reliable dial. At requested densities of 0.5 and 0.2, every seed of both models lowers its density against the same seed under the control prompt (Figure 8b). At 0.8 the change is small, and 73% of base seeds and 83% of OPTISCRIBE-12B seeds move by at least 0.02. Sparse instances also solve roughly ten times faster for both models (Figure 8a).

On CFL the density sentence asks that only a fraction of facilities be economical for each customer. The base model follows it, with median κ of 0.80, 0.53 and 0.07 for requests of 0.8, 0.5 and 0.2. OPTISCRIBE-12B does not, giving 0.97, 0.84 and 0.88. It keeps serving costs close together, which is the opposite of what the sentence asks. Its instances still become easy under this sentence, with median nodes falling from 203 to between 1 and 5, through a change we have not identified. On CFL the word "harder" raises median nodes for both models, from 142 to 215 and from 203 to 464, but with 48 completions neither paired change is significant.

Instructions cost the trained model less validity. Appending any sentence lowers the base model's CFL parse rate from 94% to as low as 67%. For OPTISCRIBE-12B it falls from 98% to no lower than 90%. On Max-Cut both models parse every completion under every sentence. OPTISCRIBE-12B writes the complete header in all of them, against 40–85% for the base model.

Table 14: **Prompt conditions for the language-control study.** Each condition appends one sentence to the production prompt at a fixed position. Everything else is byte-identical across conditions and models. This includes the row form, the size request, the exemplar pool, the token cap, the decoding settings and the seed of each completion. κ is the coupling density. On Max-Cut it is the fraction of vertex pairs with nonzero effective weight $w_{ij} + w_{ji}$. On CFL it is the mean fraction of facilities whose serving cost is within $10\times$ of each customer's cheapest option. κ^{req} is the value the sentence asks for.

Tag	κ^{req}	Sentence appended to the production prompt
control	–	(nothing appended)
harder	–	Make this instance as hard as possible for a branch-and-bound solver to prove optimal.
easier	–	Make this instance easy, a warm-up example that a solver should finish almost immediately.
Max-Cut		
sparser	$0.8/0.5/0.2$	Use a graph in which about P percent of the vertex pairs are joined by an edge; every remaining pair must have weight zero.
sparser + harder	0.5	The sentence above at $P = 50$, followed by the difficulty sentence.
CFL		
sparser	$0.8/0.5/0.2$	For each customer only about P percent of the facilities should be economical; every other facility must cost that customer at least ten times its cheapest option.
sparser + harder	0.5	The sentence above at $P = 50$, followed by the difficulty sentence.

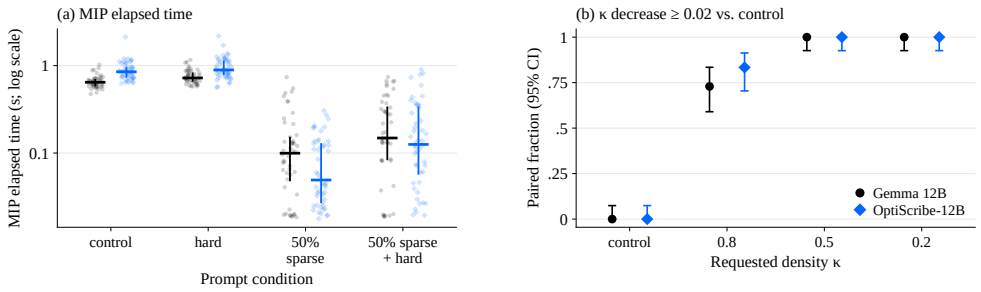

Figure 8: **More on the Max-Cut density instruction.** Bracket 171–225, 48 completions per model and prompt. (a) SCIP solve time per instance on a log scale. Sparse requests cut solve time by roughly an order of magnitude for both models. (b) Fraction of seeds whose density falls by at least 0.02 against the same seed under the control prompt, with 95% confidence intervals. Every seed moves at requested densities of 0.5 and 0.2.

G.9 Instance properties by bracket

Figures 9 and 10 break the results down by bracket for all six models. Three points matter for reading the main tables. First, feasibility given a parse stays between about 86% and 100% for every model, so validity differences come mostly from parsing. Second, constraint-matrix density falls with size at the same rate for every model, so trained models do not buy hardness with denser matrices. Third, size targeting weakens at the largest bracket. At 351–500, OptiScribe-12B lands 56% of CFL and 75% of Max-Cut instances inside the requested bracket. OptiScribe-4B misses the Max-Cut bracket often at every size, landing inside it for only 18–71% of instances.

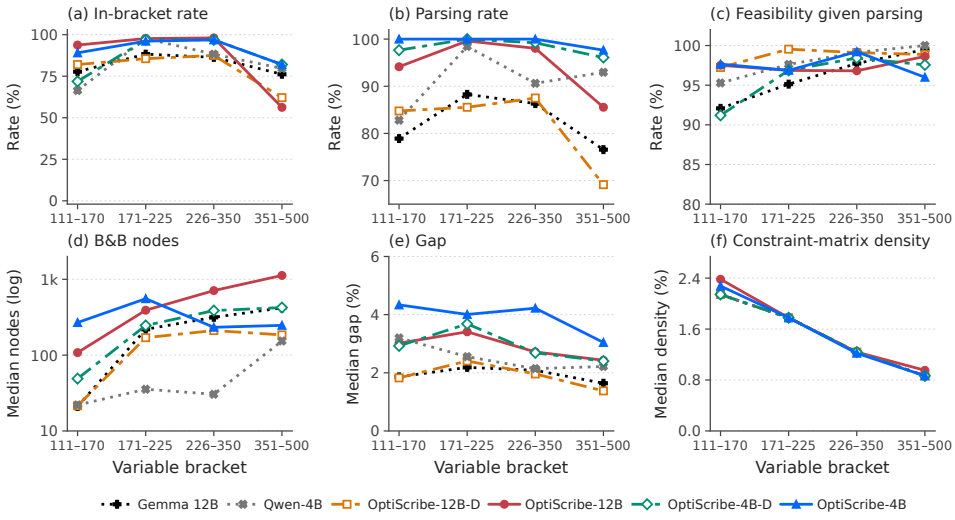

Figure 9: **CFL instance properties by size bracket under SCIP, for all six models.** (a) Share of instances whose variable count falls inside the requested bracket. (b) Parse rate. (c) Feasible rate among parsed completions. (d) Median branch-and-bound nodes. (e) Median post-root-cut gap in percent. (f) Median constraint-matrix density, the share of nonzero entries. Curriculum models are trained on 76–225 variables and direct models on 171–225.

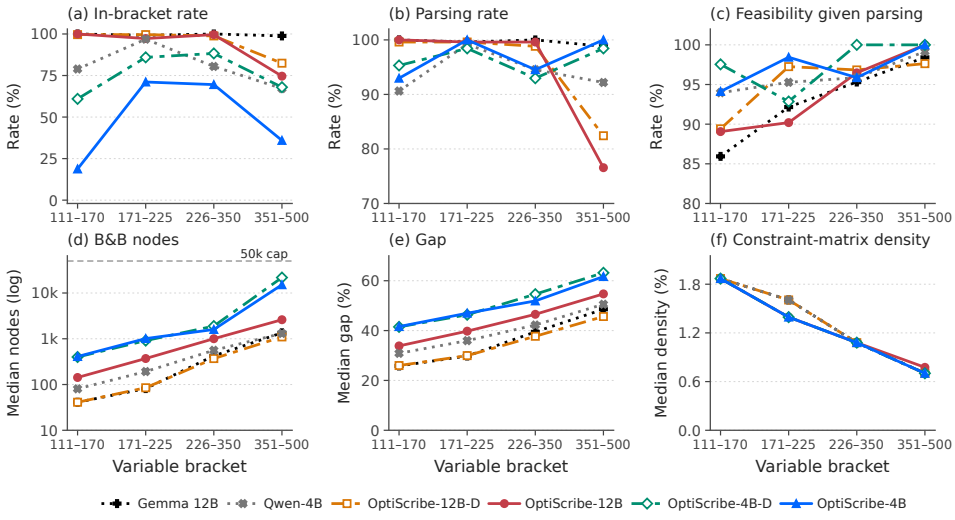

Figure 10: **Max-Cut instance properties by size bracket under SCIP, for all six models.** Panels as in Figure 9. The dashed line in (d) is the 50,000 node cap.

G.10 The sense token on Max-Cut

The Max-Cut header is `MILP max_cut max`. Untrained Gemma-4-12B writes the final sense token in only 48–69% of completions, against 94–98% for OptiScribe-12B-D and 100% for OptiScribe-12B. Without the token our parser reads the objective as a minimization, whose optimum is zero and is found at the root. The prompt always asks for maximization, so in evaluation we solve every Max-Cut instance as a maximization. The rule applies to all models alike, so the Max-Cut gains in Table 2 do not come from the header. We suspect the base model drops the token because the family name already contains the word max, but we have not tested this.