DISTRIBUTION-AWARE PROGRAMMING: LEARNING SPECIALIZED SOLVERS FROM EXPERIENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Many optimization problems are solved repeatedly on instances drawn from the same underlying distribution. In this setting, a system can begin with a general-purpose solver and use experience from previous instances to learn a cheaper way to solve future ones. We formalize this as *distribution-aware programming*: samples from an unknown deployment distribution are used to produce executable solver code whose quality and runtime generalize to new instances. A simple analysis shows how sample access interpolates between distribution-oblivious and distribution-informed algorithm design, and when the offline cost of specialization is recovered through lower deployment cost. We instantiate this framework with an LLM agent that proposes structural hypotheses, analyzes training instances, and synthesizes specialized solver code; the LLM is used only before deployment. Across 21 structured combinatorial-optimization distributions, the synthesized solvers achieve high quality while often replacing generic search or optimization with smaller distribution-specific computations. Against released PACE competition solvers, our method remains competitive while using substantially less runtime: 75–125$\times$ less on PACE 2025 Dominating Set, 80–96$\times$ less on Hitting Set, and more than $34{,}000\times$ less on the PACE 2024 OCM exact track. More broadly, the results suggest using AI not only to solve optimization problems, but to learn how recurring distributions should be solved.

1 INTRODUCTION

The fields of optimization and theoretical computer science typically focus on how efficiently a problem can be solved, very often in the worst case. We tackle the common but slightly different setting in which we repeatedly receive new problem instances sampled from a particular distribution or drawn from a specific subfamily of problems. In this setting, the goal is to design an algorithm that minimizes cumulative computation across all such instances. This perspective reveals a strong connection between optimization and learning theory, which we explore further in Section 3. With a few sample instances available before deployment, the central observation is:

The question of average-time complexity can be tackled through statistical learning theory.

Understanding average-time complexity then corresponds to finding, within a hypothesis class of algorithms, one that minimizes expected risk in terms of runtime and correctness. This aligns philosophically with the idea that, when the learned hypothesis is an executable procedure, runtime should matter, not just quality. If two students learn multiplication equally well, but one discovers a faster method, we would say that student learned a better procedure. We term this paradigm *distribution-aware programming*: through empirical risk minimization, find an optimally generalizing (lowest average-time complexity) solver for your distribution of problems. In this article we formalize this setting and discuss when it is tractable and useful as a point of view. After that, we ask:

Can a system based on publicly available LLMs learn this at inference time?
If so, what is a good system design?

The positive results of our experiments have three, we believe, strong implications. First, this perspective suggests a different role for AI in optimization. Much recent work places learned models

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.

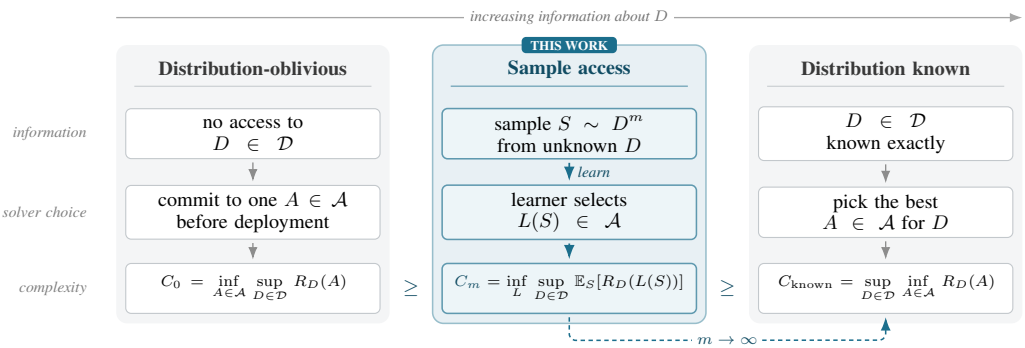


Figure 1: **Three levels of access to the deployment distribution.** With no information about D, a single solver must be fixed before deployment, giving C_0. If D were known exactly, the solver could be specialised to it, giving C_{known}. We study the intermediate regime (centre): D is unknown, but the learner observes a sample $S \sim D^m$ and selects $L(S) \in \mathcal{A}$. Proposition 3.1 shows $C_{\text{known}} \leq C_m \leq C_0$, and that $C_m \to C_{\text{known}}$ for a fixed bounded-description program class.

inside optimization procedures, using them to predict, branch, search, or reason on each new instance. We instead use AI as the engineer behind the solver, not as part of the solver loop itself.

Second, this targets the tradeoff between general-purpose solvers, which handle broad input classes, and customized solvers, which leverage structure but require human design. We ask whether this specialization can instead be learned from experience, and show that *LLM-based systems can do so*.

Third, the cost of solving a problem with unknown structure for the first time is very different from the cost of solving further instances from the same family. As experience accumulates, parts of the generic computation can be replaced by computation tailored to the recurring workload. In this sense, the process is *meta-reasoning*: general-purpose reasoning is used not to solve instances, but to determine how future instances should be solved. Once synthesis is complete, the LLM agent disappears; future instances are handled by ordinary executable code. Expensive general-purpose reasoning is performed offline and compiled into cheaper specialized computation that can be reused many times. In this work, the principle applies both to the generic-to-customized solver pipeline and to the LLM reasoning used to construct those solvers: we improve performance and reduce cost by hard-coding recurring reasoning into our harness.

Contributions. Our main contributions are:

- **Distribution-aware programming.** We formulate repeated algorithm design as a precise learning problem: samples from an unknown deployment distribution are used to synthesize executable solver code, evaluated by solution quality and deployment runtime. We formalize *solver hints* as a sufficient condition for tractability of our distribution of problems.
- **Sample-access complexity.** We formalize the value of observing previous instances through distribution-oblivious, distribution-informed, and sample-conditioned complexity. For bounded program classes, the sample-conditioned cost approaches the distribution-informed optimum as the sample grows, while repeated deployment can amortize the offline learning cost.
- **LLM-based solver synthesis.** We develop an agentic procedure that proposes structural hypotheses, extracts reusable summaries from samples, and synthesizes and refines deployment solvers. The resulting solver requires no LLM inference at deployment.
- **Empirical specialization and external validation.** Across 21 structured distributions spanning seven optimization problems, the synthesized solvers often replace generic search with smaller distribution-specific computations. On released PACE 2025 Dominating Set and Hitting Set and PACE 2024 OCM benchmarks, the same procedure produces fully valid solvers with large runtime advantages over competition code at modest objective gaps.

2 Related Work

Distribution-dependent algorithm design. Average-case complexity, smoothed analysis, and beyond-worst-case analysis explain why hard problems may admit efficient algorithms under particular input distributions or structural assumptions (Levin, 1986; Impagliazzo, 1995; Spielman & Teng, 2004; Roughgarden, 2019). Parameterized and backdoor complexity likewise identify structure that can make otherwise hard problems tractable (Downey & Fellows, 2013; Williams et al., 2003). Speedup learning studies how experience improves the computational efficiency of problem solving (Tadepalli & Natarajan, 1996). Most closely related are self-improving algorithms, which learn from samples of an initially unknown distribution to accelerate future computation (Ailon et al., 2011; Clarkson et al., 2014; Cheng et al., 2020). Our setting follows the same amortization principle but does not assume a hand-designed representation of useful structure or a problem-specific self-improving algorithm: the hint, analysis procedure, and specialized solver may themselves be synthesized.

Average-case optimization and practical specialization. Classical probabilistic analyses obtain polynomial expected bounds for simplex, complementary pivoting, and local-improvement methods under specified input models (Borgwardt, 1982; Smale, 1983; Todd, 1986; Tovey, 1986). Modern average-case optimization relates convergence and halting times to spectral structure and derives distribution-adapted methods (Pedregosa & Scieur, 2020; Scieur & Pedregosa, 2020; Paquette et al., 2023; 2021; Cunha et al., 2022; Domingo-Enrich et al., 2021). In practice, automated tuning targets runtime, solution quality, or anytime performance on representative workloads (Adenso-Díaz & Laguna, 2006; Hutter et al., 2010; López-Ibáñez & Stützle, 2014; López-Ibáñez et al., 2016); commercial solvers such as Gurobi support average-performance tuning across multiple models (Gurobi Optimization, LLC, 2026). Specialization can change the computational procedure itself: generated heuristics, explicit control laws, specialized solver code, learned solution strategies, and warm starts offer different routes to reducing repeated-solve costs (Mascia et al., 2014; Bemporad et al., 2002; Mattingley & Boyd, 2012; Bertsimas & Stellato, 2022; Sambharya et al., 2024).

Algorithm selection and learning-augmented algorithms. Algorithm selection and configuration exploit the fact that different solvers perform best on different instances or distributions (Rice, 1976; Xu et al., 2008; 2010; Lindauer et al., 2015). PAC analyses characterize the sample complexity of selecting high-performing algorithms for an unknown instance distribution (Gupta & Roughgarden, 2017; Balcan et al., 2017; 2021). Complementary work bounds the computational cost of configuration through adaptive evaluation and capped runs, with connections to bandit allocation (Kleinberg et al., 2017; Weisz et al., 2018; 2019; Kleinberg et al., 2019). Learning-augmented algorithms incorporate learned predictions into classical algorithms while seeking robustness to prediction error (Lykouris & Vassilvitskii, 2018; Mitzenmacher & Vassilvitskii, 2022). Our synthesis procedure uses samples before deployment to construct the specialization itself: the structural hypothesis, the analysis program that measures its supporting evidence, and the solver code conditioned on the resulting summary.

Learned computation for optimization. Neural combinatorial optimization learns policies, heuristics, branching rules, or representations for hard optimization problems (Vinyals et al., 2015; Bello et al., 2017; Khalil et al., 2017; Gasse et al., 2019; Bengio et al., 2021; Cappart et al., 2023). Related work on adaptive and test-time computation varies the amount of inference, search, or reasoning performed on each input (Teerapittayanon et al., 2016; Wang et al., 2023; Yao et al., 2023; Snell et al., 2025). Our emphasis is different: expensive learned reasoning occurs before deployment and changes the solver itself. The resulting program may be symbolic or combinatorial and requires no model inference on new instances.

Program synthesis and code generation. Program synthesis from examples has long studied how observations or specifications can be turned into executable programs (Gulwani, 2011; Balog et al., 2017; Devlin et al., 2017; Nye et al., 2019), while large language models have greatly expanded the scope of code generation and autonomous software agents (Hendrycks et al., 2021; Li et al., 2022; Jimenez et al., 2024; Zhuo et al., 2025; Huang et al., 2024; Chan et al., 2025). LLM-guided algorithm discovery also uses execution-based feedback to generate and improve reusable heuristics and algorithms (Romera-Paredes et al., 2024; Liu et al., 2024; Ye et al., 2024; Novikov et al., 2025). Our emphasis is on using samples from a repeated computational workload to synthesize an algorithm whose *computation* is specialized to that workload.

Learning executable hypotheses. Classical learning theory provides the general framework for learning hypotheses that generalize from samples (Valiant, 1984; Vapnik & Chervonenkis, 1971;

Vapnik, 1998; Shalev-Shwartz & Ben-David, 2014), and recent work has considered empirical risk minimization over programs under LLM-induced priors (Singhal et al., 2025). We evaluate executable hypotheses by both solution quality and runtime. Our analysis compares distribution-oblivious, sample-conditioned, and distribution-informed computational costs, and accounts for the amortization of offline learning across repeated solves.

3 Learning and Computational Cost

We begin with a very simple comparison of three settings: choosing a solver without knowing the deployment distribution, choosing one when the distribution is known, and choosing one after observing samples. This also shows when computation spent before deployment can be recovered through lower runtime on future instances.

Defining Distribution-Aware Programming. Fix an input size n, let $\mathcal{D}_n$ be a family of distributions supported on instances of size n, and let $\mathcal{A}$ be a fixed class of correct solver programs. For $A \in \mathcal{A}$, write $R_D(A) = \mathbb{E}_{X\sim D}[T(A, X)]$ for its expected runtime under D. Going forward, we suppress the dependence on n. If the deployment distribution is unknown, the solver must perform well uniformly over $\mathcal{D}_n$. If the distribution were known, one could instead choose a solver specialized to it. These two settings correspond to

$$C_0 = \inf_{A\in\mathcal{A}} \sup_{D\in\mathcal{D}_n} R_D(A), \qquad C_{\text{known}} = \sup_{D\in\mathcal{D}_n} \inf_{A\in\mathcal{A}} R_D(A).$$

The difference is the order in which the algorithm and distribution are chosen. By the minimax inequality, $C_{\text{known}} \le C_0$, so $C_0 - C_{\text{known}}$ measures the potential computational benefit of adapting the solver to the deployment distribution.

In practice, the distribution is not given explicitly. Instead, we may observe previous instances from the same source. Let $S = (X_1, \dots, X_m) \sim D^m$, and let a learning procedure use S to choose a solver for future independent draws from D. Define $C_m = \inf_L \sup_{D\in\mathcal{D}_n} \mathbb{E}_S[R_D(L(S))]$, where the infimum is over learning procedures and the expectation includes any randomness in the learner. The question is whether samples can move the computational cost from the distribution-oblivious quantity C_0 toward the distribution-informed quantity C_{known}.

Samples help. Suppose the possible deployed programs have binary descriptions of length at most ℓ in a fixed programming language. The class is fixed before observing the sample, although the selected program may depend on it. There are at most $N \le 2^{\ell+1} - 1$ such programs. We also allow randomized choices among them; for any fixed D, randomization cannot improve expected runtime beyond that of the best base program. Using standard statistical learning theory tools (Shalev-Shwartz & Ben-David, 2014):

Proposition 3.1 (Sample-conditioned computational complexity)**.** *Suppose $0 \le T(A, X) \le B$ for every base program A, every $D \in \mathcal{D}_n$, and every X in the support of D. For $m \ge 1$, let $\varepsilon_m = B\sqrt{\frac{(\ell+1)\log 2}{2m}}$. Then $C_{\text{known}} \le C_m \le \min\{C_0,\, C_{\text{known}} + \varepsilon_m\}$. In particular, for a fixed program class, $C_m \to C_{\text{known}}$ as $m \to \infty$.*

The proposition shows that samples can approach the computational benefit of knowing the distribution without recovering D itself: it suffices to identify a program with low expected runtime. If $F(m)$ bounds the offline selection cost, reuse over Q instances gives amortized cost at most $F(m)/Q + C_{\text{known}} + \varepsilon_m$, improving on C_0 whenever $F(m)/Q + \varepsilon_m < C_0 - C_{\text{known}}$.

The remaining challenge is search: bounded description length controls generalization but does not make the program class tractable to explore. We therefore use LLM-guided search over structural hypotheses and solver programs. Given the task specification and public samples, the agent proposes recurring structure, estimates it with an analysis program, and generates a solver conditioned on the resulting summary. Validation quality and runtime guide refinement. The analysis above idealizes selection from a fixed class of correct programs; our experiments test whether this guided search can find fast, high-quality specializations in practice.

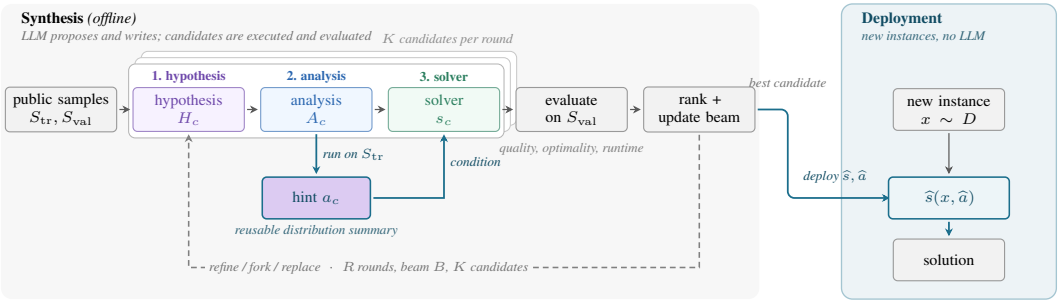


Figure 2: **Offline synthesis and deployment.** For each candidate, the LLM proposes a hypothesis H_c, writes an analysis program A_c, and uses its output a_c to generate a solver s_c. Candidates are evaluated and refined over several rounds. The selected solver $\hat{s}$ and hint $\hat{a}$ are then reused on new instances without LLM inference.

4 METHOD

Our goal is to infer reusable structure from sample instances and compile it into a solver for the same deployment distribution. Because neither the useful structure nor the program that exploits it is known in advance, we use an LLM agent to search jointly over structural hypotheses, analysis programs, and deployment solvers. Figure 2 summarizes the procedure; full prompts, implementation details, validation checks, and failure handling appear in Appendix C.

Candidate construction and search. Each candidate consists of a structural hypothesis H_c, an analysis program A_c, and a deployment solver s_c. The analysis program maps the training sample to a reusable hint $a_c = A_c(S_{\text{tr}})$, and the solver uses (x, a_c) on new instances. The hint representation is task-dependent and may encode bottlenecks, vertex roles, geometric structure, recurring Boolean patterns, or other reusable structure.

We maintain a beam of candidates over several rounds. Candidates are ranked lexicographically by validation quality, optimality rate, and runtime, $(Q_{\text{val}}, O_{\text{val}}, -T_{\text{val}})$. Initial candidates explore diverse structural explanations; later rounds may refine a promising candidate, branch to another explanation, replace a weak hypothesis, or improve quality or runtime using validation feedback and representative failures. To preserve diversity, we retain the best candidate from each hypothesis class before filling the remaining beam slots by validation score. After the final round, we select the best candidate seen during the search, rerun its analysis on the training sample, and deploy the resulting solver. Invalid or failed candidates receive zero quality. Full prompts, scoring, diversity rules, and failure handling appear in Appendix C.

5 EXPERIMENTS

We evaluate whether the synthesis procedure of Section 4 can identify reusable structure from samples and turn it into solvers with better quality–runtime tradeoffs. We compare against classical heuristics and optimization solvers, learned neural baselines, and alternative LLM-based synthesis methods. Full experimental details, benchmark definitions, per-target results, and additional ablations are provided in the appendix.

5.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on both controlled and external benchmarks. Our controlled suite contains 21 structured target distributions spanning seven combinatorial optimization problems, with three hidden distribution families per problem. Each target contains 64 training, 32 validation, and 500 held-out test instances. The learner sees only the public problem instances and specification, not the hidden family identity, generative rule, optimum solutions, or optimum objective values.

We also evaluate on released PACE competition benchmarks: PACE 2025 Dominating Set and Hitting Set (Grobler & Siebertz, 2025) and PACE 2024 One-Sided Crossing Minimization (OCM) (Kin-

dermann et al., 2024). Released public instances are used for synthesis and model selection, while released private instances are reserved for evaluation. These substantially larger, independently constructed datasets provide an external test against highly engineered competition solvers. Detailed Dominating Set statistics, protocol information, and matched-budget results appear in Appendix E.6.

5.1.1 BASELINES

We compare our synthesis agent (Section 4) against classical heuristics and optimization solvers, learned neural methods, and alternative LLM-based synthesis procedures. All methods are evaluated on the same public instances and receive no information about the hidden distribution family or its generative rule.

Classical baselines. For each problem class, we compare against standard problem-specific heuristics and general-purpose or exact optimization systems (e.g., Brélaz, 1979; Chvátal, 1979; Dantzig, 1957; Helsgaun, 2000; Gurobi Optimization, LLC, 2026; Perron & Furnon, 2025; Bestuzheva et al., 2023; HiGHS Development Team, 2024). The complete catalog appears in Appendix D.3.

LLM baselines. We compare against one-shot solver synthesis with OpenAI Codex (OpenAI, 2025) and Anthropic Claude Code (Anthropic, 2025), using the same problem specification and tool access but without iterative refinement. We also use a Gemma 4 best-of-5 baseline (`gemma-4-31b-it`): five independent solvers are generated from the same specification and the best is selected on validation, matching the candidate budget without the refinement loop.

Learned ML baselines. We also compare against one neural baseline for each problem class: physics-inspired GNNs for coloring and independent set (Schuetz et al., 2022b;a), a RUN-CSP-style model for MaxSAT (Tönshoff et al., 2021), a GNN with reinforcement learning for dominating set (Chen et al., 2024), an attention model for TSP (Kool et al., 2019), a reinforcement-learning baseline for multidimensional knapsack (Bushaj & Büyüktahtakın, 2024), and a primal–dual model for packing LP (Park & Van Hentenryck, 2023). These implementations follow the respective model families rather than reproducing the cited systems exactly. They were trained and evaluated on reduced-size per-target datasets drawn from the same target distributions, rather than the full 500-instance test sets used for the main classical comparisons; we therefore treat them as auxiliary model-family baselines. We report inference time on both CPU (AMD EPYC 9554) and GPU (NVIDIA H100 NVL); training time is excluded from deployment runtime, just as synthesis time is excluded for our method.

5.1.2 EVALUATION

The main metric is normalized solution quality, where 1 is optimal and larger is better; invalid or infeasible outputs receive quality 0. We also report optimality rate, feasibility rate, and per-instance wall-clock runtime. The problem-specific quality definitions are given in Table 7 of Appendix D. Results are first averaged over held-out test instances and repeated runs within each target, and then averaged across the 21 targets.

Let Q_{ours} denote the quality of our synthesized solver and Q_b the quality of baseline b. We report the quality difference $\Delta Q_b = Q_{\text{ours}} - Q_b$, so positive values mean our solver has higher quality. Comparisons include Gemma best-of-5, one-shot Codex and Claude Code, the average heuristic, the learned ML baseline, and a *fast high-quality heuristic*. The latter is the fastest heuristic whose average quality is at least 95% of the best heuristic for that problem class. Timeouts receive quality 0 and the 360s runtime. Invalid or failed Gemma best-of-5 outputs are likewise scored as zero quality.

For runtime, we report the speedup $S_b = T_b/T_{\text{ours}}$, where T_b and T_{ours} are the baseline and synthesized-solver runtimes. Thus $S_b > 1$ means our solver is faster. Speedups are computed separately for each target and combined using the geometric mean. We compare against the LLM baselines, CPU and GPU inference for the learned ML models, the fast high-quality heuristic, Gurobi, and the time-limited exact baseline. Failed Gemma best-of-5 runs are excluded from runtime aggregation because their measured time reflects immediate failure rather than solver execution.

Table 1: **Quality and runtime across the 21 target distributions.** Q_{ours} and T_{ours} are the quality and runtime of our synthesized solver. For each baseline b, $\Delta Q_b = Q_{\text{ours}} - Q_b$, so positive values favor our solver, and $S_b = T_b/T_{\text{ours}}$, so $S_b > 1$ means our solver is faster. Quality is normalized so that 1 is optimal.

	Ours		Quality difference ΔQ_b						Speedup $S_b = T_b/T_{\text{ours}}$							
Family	T_{ours}	Q_{ours}	Gem@5	Cod	Cld	Avg	Heur	ML	Gem@5	Cod	Cld	ML-C	ML-G	Heur	Gur	Exact
Coloring	2.7	0.868	+0.085	−0.086	−0.065	+0.217	+0.121	−0.127	0.62×	0.97×	59.0×	0.49×	0.76×	1326.3×	2285.6×	23.1×
MAXSAT	17.1	1.000	+0.004	+0.006	+0.000	+0.122	+0.083	0.000	36.0×	1.52×	1.00×	1.0×	1.0×	212.2×	328.8×	1.2×
MIS	18.8	0.992	+0.009	+0.009	+0.326	+0.218	+0.113	−0.002	1.43×	2.19×	165.8×	0.047×	0.085×	1904.2×	155.8×	41.2×
MDS	13.3	0.973	+0.111	−0.027	+0.022	+0.148	+0.124	+0.021	17.1×	22.1×	4.20×	0.16×	0.42×	1626.8×	443.0×	21.6×
Packing LP	3.3	0.994	+0.215	−0.003	−0.006	+0.301	+0.317	+0.186	1.75×	34.8×	30.2×	0.19×	0.30×	13483.7×	2829.1×	44.1×
MDKP	94.0	0.973	+0.579	−0.010	−0.011	+0.215	+0.009	0.000	0.032×	4.60×	3.69×	0.037×	0.060×	46.4×	33.8×	9.6×
TSP	14.6	0.993	+0.008	−0.002	+0.327	+0.348	−0.007	+0.084	3.34×	3.48×	104.4×	30.0×	61.0×	33.7×	117.5×	37.2×
All (21)	**12.7**	**0.971**	**+0.145**	**−0.016**	**+0.085**	**+0.224**	**+0.109**	**+0.023**	**2.37×**	**4.53×**	**17.4×**	**0.36×**	**0.61×**	**528.6×**	**345.1×**	**16.9×**

T_{ours} is in milliseconds. Avg = average heuristic; Heur = fast high-quality heuristic; ML-C/ML-G = ML inference on CPU/GPU; Gur = Gurobi.

5.2 RESULTS

Running the same synthesis pipeline with Gemma 4 gives mean quality 0.883 versus 0.971 with GPT-5.2, with deployed solvers $1.38\times$ slower on average; full per-family results appear in Appendix E.2.

Main quality–runtime results. Table 1 summarizes the quality–runtime tradeoff across the 21 target distributions; full per-target results are in Appendix E.5. Our synthesized solvers achieve mean normalized quality 0.971, improving by 0.224 over the average heuristic and 0.109 over the fast high-quality heuristic. The reduced-scale learned baselines are reported as an auxiliary comparison; on those runs, our mean quality is 0.023 higher. At the same time, the synthesized solvers are $528.6\times$ faster than the fast high-quality heuristic, $345.1\times$ faster than Gurobi, and $16.9\times$ faster than the time-limited exact baseline, using geometric means of per-target speedups. The heuristic speedup remains $527.2\times$ after excluding methods that hit the time limit on at least half of their instances, and $526.7\times$ after excluding anytime and local-search methods, so the aggregate gain is not driven by budget-consuming heuristics.

Compared with other LLM-generated solvers, one-shot Codex comes closest in quality, exceeding ours by 0.016, but is $4.5\times$ slower. Our solver exceeds one-shot Claude Code by 0.085 in quality and is $17.4\times$ faster; Claude also times out on one MIS and one TSP target. Gemma best-of-5 is $2.4\times$ slower and trails by 0.145 in quality, suggesting that repeated independent sampling alone does not account for the gains from iterative synthesis.

To test whether samples mainly help select among existing algorithms, we use validation performance to choose among the baselines for each problem class. The highest-quality selector reaches 0.9805 mean test quality versus 0.9705 for ours, but is $42.6\times$ slower. The fastest selector matching our validation quality is $27.9\times$ slower and succeeds on only $19/21$ targets. Full results are in Appendix E.4.

The advantage is not uniform across all problems. The fast heuristic achieves slightly higher quality on TSP, and the learned ML model performs better on Coloring and is faster on several families, particularly with GPU inference. Overall, however, no baseline is both faster and higher-quality across the full suite. On MIS, a matched-budget comparison with the OnlineMIS and ReduMIS modes of KaMIS is mixed: our solver is strongest on Core-fringe, while the KaMIS heuristics are stronger on Clique-path and Motif-bridge (Appendix E.7, Table 19).

Against stronger learned baselines on the classes where they apply, our solvers match or exceed compact reproductions of ANYCSP and GFlowNet on $11/12$ targets and are faster on all 12, by $2.5\times$ to over $2{,}800\times$. These matched evaluations use the same 500 held-out instances as our solvers; full results and implementation scope appear in Appendix E.1.

Iteration ablation. We next ask whether iterative refinement improves the deployed solver beyond the initial generated candidate. After each synthesis round, we record the test-time speedup of the best candidate found so far relative to the zero-shot solver.

Figure 3 shows that refinement can improve runtime. TSP, for example, increases from about $8.7\times$ speedup after the first iteration to $40.6\times$ by iteration 4, while Packing LP moves from initially slower

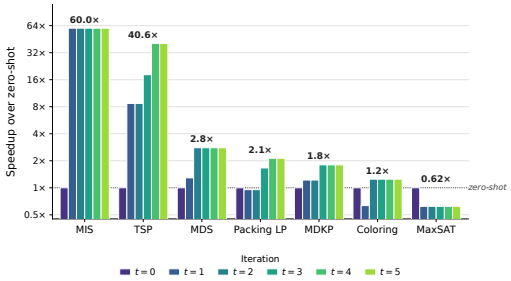


Figure 3: **Runtime improvement across synthesis iterations.** Speedup is measured relative to the zero-shot generated solver; values above $1\times$ indicate faster deployment. Each bar reports the best candidate found up to that iteration.

Table 2: **Representative computation patterns.** Ambient denotes generic computation; generated denotes the distribution-specific computation synthesized from samples. Full expressions and notation appear in Table 20.

Problem	Structure	Ambient	Generated	Problem	Structure	Ambient	Generated
Coloring	ring template	$O^*(\kappa^n)$	$O(m)$ fast path	MDS	star kernel	$O^*(2^n)$	$O(n+m+\sum_j c_j 2^{t_j} + T_{\text{prune}})$
MAXSAT	latent rules	$O^*(2^v)$	$O(\lvert F\rvert \ell + RB\ell\Delta_{\text{occ}})$	Packing LP	bottleneck resource	$T_{\text{LP}}(N,r,L)$	$O(Nr + N\log N)$
MIS	core–fringe	$O^*(2^n)$	$O(n+m+\sum_j 4^{t_j} + kq)$	MDKP	single resource	$O^*(2^N)$	$O(Nr + N\log N + K_{\text{cand}}C_b + T_{\text{repair}})$
TSP	paired ribbon	$O(n^2 2^n)$	$O(n\log n)$ fast path				

candidates to about $2.1\times$ speedup. MDS and MDKP also improve after the first proposal. MaxSAT is the exception, where additional iterations do not improve on the zero-shot solver.

These results suggest that the synthesis loop does more than sample alternative implementations: later rounds can revise an initially poor specialization or turn a useful structural idea into a substantially faster solver. The benefit is not universal, but on several problem families it is a major source of the final runtime gain.

Hint ablation. To isolate explicit structure discovery from iterative code refinement, we compare against a matched GPT-5.2 pipeline with the same beam, candidate budget, validation feedback, refinement, and repair, but no hypothesis/hint stage. The full pipeline matches or exceeds this baseline on all seven representative targets and improves mean quality by 0.040, with the largest gains on Ring-template Coloring ($+0.203$) and Latent-class MDKP ($+0.066$). Full results appear in Appendix E.3.

What did the LLM compile? The speedups in Table 1 are not just implementation effects. In many cases, the selected solver changes the effective computation: it replaces an ambient worst-case search or generic optimization call with a distribution-specific procedure inferred from samples. Table 2 gives a compact summary of these computation patterns, while Appendix E.8 (specifically Table 20) gives the full per-distribution expressions and notation.

The main pattern is that synthesis often compiles a structural shortcut. For MAXSAT, generated solvers exploit latent Boolean rules and bounded local repair instead of searching over all assignments. For graph problems, they exploit palettes, motifs, hubs, gateways, or small residual kernels rather than treating each instance as an arbitrary coloring, MIS, or MDS problem. For Packing LP and MDKP, they exploit active resources or bottlenecks, replacing generic optimization with sorting, scoring, fractional filling, and bounded repair. For TSP, they exploit recurring geometry to construct and improve tours without full Held–Karp-style dynamic programming.

Thus, the empirical speedups reflect the intended mechanism of distribution-aware programming: the LLM is not merely producing faster code for the same computation, but often compiling sampled regularities into a lower-dimensional or smaller-residual computation.

External PACE benchmarks. We next test whether the synthesis procedure transfers beyond our controlled distributions to independently constructed competition instances. We evaluate on the released PACE 2025 Dominating Set and Hitting Set heuristic tracks and the PACE 2024 One-Sided Crossing Minimization (OCM) tracks. In all cases, synthesis uses only released public instances, while released private instances are held out for evaluation.

On PACE 2025 Dominating Set, our solver returns valid solutions on all 100 private instances in 2.89 seconds on average. Released competition solvers produce sets about 3% smaller but require

Table 3: **PACE 2025 Dominating Set and Hitting Set.** On both released private benchmarks, our solver is valid on all 100 instances and occupies a low-runtime operating point, trading modestly worse objective values for substantially faster deployment than the released PACE solvers.

Dominating Set

Solver	Valid	Avg. size ↓	Ratio	Time ↓	Speedup
Ours (GPT-5.2)	100/100	231,595	1.00	2.89s	1.0×
Ours (Gemma 4)	100/100	231,667	1.00	6.03s	2.1×
Gemma 4 best@5	100/100	261,741	1.13	2.69s	0.93×
Codex 1-shot	100/100	230,863	0.997	2.43s	0.84×
Claude Code 1-shot	100/100	224,699	0.970	37.29s	12.9×
Released PACE solvers	75–100/100	210k–224k	0.91–0.97	218–360s	75–125×
General heuristics	100/100	235k–242k	1.01–1.04	8.3s	2.9×
Gurobi / exact-style	timeout	—	—	≥ 360s	≥ 124.6×

Hitting Set

Solver	Valid	Avg. size ↓	Time ↓	Speedup
Ours (GPT-5.2)	100/100	197,477	3.75s	1.0×
Claude Code 1-shot	100/100	201,614	23.36s	6.23×
Codex 1-shot	100/100	195,439	10.56s	2.82×
Gurobi (420 s)	100/100	195,596	433.7s	115.7×
Released PACE solvers	100/100	191.5k–192.2k	300.7–360.3s	80.2–96.1×
frequency_greedy	100/100	205,938	6.12s	1.63×
exact (catalog)	0/100	—	0.62s	—

For Dominating Set, Swats statistics use the 75/100 instances on which it returns a valid solution. The Hitting Set exact-catalog method returns no valid solutions, so its runtime is not a successful-solve comparison.

Table 4: **PACE 2024 One-Sided Crossing Minimization.** On both released private tracks, our solver remains close in objective value to the fully valid PACE solvers while running substantially faster, including more than 34,000× on the exact track. Parentheses give objective ratios relative to ours; speedup is baseline runtime divided by ours.

Cutwidth track

Solver	Valid	Obj. ↓ (ratio)	Time ↓	Speedup
Ours (GPT-5.2)	100/100	101,362 (1.000)	0.016	1.0×
Gemma 4 best@5	100/100	101,396 (1.000)	0.014	0.85×
Claude Code 1-shot	100/100	101,209 (0.998)	1.193	74.1×
Codex 1-shot	100/100	101,233 (0.999)	0.402	25.0×
ocmu64	100/100	101,209 (0.998)	0.058	3.6×
pingpong	100/100	101,209 (0.998)	0.179	11.1×
cimat	0/100	—	1.129	70.0×

Exact track

Solver	Valid	Obj. ↓ (ratio)	Time ↓	Speedup
Ours (GPT-5.2)	100/100	148,091 (1.000)	0.0085	1.0×
Gemma 4 best@5	100/100	159,387 (1.076)	0.0055	0.65×
Claude Code 1-shot	100/100	143,207 (0.967)	1.235	145.3×
Codex 1-shot	100/100	143,260 (0.967)	0.234	27.5×
cimat	100/100	143,195 (0.967)	295.039	34,604×
pingpong[a]	97/100	143,978 (0.972)	24.734	2,901×
ocmu64[a]	78/100	118,547 (0.800)	86.764	10,176×

Time is in seconds. Lower objective is better; an objective ratio below 1 means the baseline has a lower objective than GPT-5.2. [a] Objectives are averaged only over instances with valid solutions and are therefore not directly comparable with 100/100 rows.

75–125× more time. The Codex-synthesized solver is slightly better and faster than ours, while the Claude Code-synthesized solver is better but substantially slower. Hitting Set shows a similar tradeoff: our solver is valid on all 100 private instances and averages 3.75 seconds, while released solvers produce sets about 3% smaller but require 80–96× more time (Table 3). On Dominating Set, none of three released anytime solvers matches our all-instance quality at 3s or 10s, although all three surpass it at 30s (Appendix E.6).

On PACE 2024 OCM, our solver is within about 0.2% of the fully valid released solvers on the cutwidth track while running 3.6–11.1× faster. On the exact track, it returns valid solutions on all 100 private instances and is within 3.4% of the best fully valid released solver while running more than 34,000× faster (Table 4). Across the four PACE tasks, the one-shot Codex- and Claude Code-synthesized solvers attain slightly higher normalized quality, while ours is much faster on three; the exception is the Codex-synthesized Dominating Set solver. Together, these results test the same sample-to-solver procedure on external distributions constructed independently of our benchmark suite and against solvers developed specifically for the corresponding competitions.

6 Discussion and Limitations

The broader implication is that repeated computation can become progressively cheaper with experience. Rather than solving each instance from scratch, AI can learn how a recurring workload should be solved and compile that knowledge into executable algorithms. A natural next step is to make this specialization adaptive as workloads evolve.

Several limitations remain. Synthesis cost must be amortized over future instances, specialization may degrade under distribution shift, and the method does not dominate every baseline. Across repeated synthesis runs, quality is relatively stable but runtime varies more (Appendix E.10), suggesting that program search can yield similarly accurate but computationally different solvers. Robustness to distribution shift and synthesis variability remains important future work.

AI USE STATEMENT

Generative AI was used throughout this project as a research and engineering tool. In the solver-synthesis procedure of Section 4, language models propose structural hypotheses, write analysis programs, and generate and refine solver code. Generative AI was also used to help implement the experimental pipeline, including designing and writing code used to generate the 21 benchmark distributions, and to assist with debugging and software development.

Generative AI was further used in editing and polishing the manuscript, to revise portions of the text, improve clarity and presentation, assist in polishing mathematical proofs and arguments, design diagrams and illustrations, and migrate and edit LaTeX formatting.

The authors reviewed all AI-assisted material and take responsibility for the resulting algorithms, code, experiments, mathematical arguments, figures, claims, and text. AI-generated code and solver outputs were tested through the experimental and verification procedures described in the paper, and AI-assisted arguments were checked by the authors before inclusion. Generative AI systems are not listed as authors and bear no responsibility for the contents of this work.

ETHICS STATEMENT

This work studies automated synthesis of specialized optimization software from repeated problem instances. The experiments use synthetic benchmark distributions and released competition instances and involve no human subjects or personal data. The approach may reduce the computational cost of recurring optimization workloads and lower the effort required to construct specialized solvers. As with other automated optimization methods, inappropriate deployment outside the distributions on which a solver was developed may lead to unreliable behavior. We therefore emphasize held-out validation, explicit distributional assumptions, fallback mechanisms where appropriate, and human review of generated code before deployment.

REPRODUCIBILITY STATEMENT

We provide the benchmark definitions, train/validation/test splits, evaluation metrics, baseline catalog, synthesis procedure, candidate-selection and failure-handling rules, compute environment, and full per-target results in the appendix. The controlled benchmarks use fixed training, validation, and held-out test sets, while the PACE experiments use released public instances for synthesis and released private instances only for evaluation. We report the language models and inference settings used for synthesis, hardware for local and learned baselines, deployment-time measurement conventions, and aggregate API and compute usage. Additional ablations report generator dependence, explicit hint recovery, catalog selection, sample access, synthesis variability, execution traces, and sensitivity to graph relabeling.

REFERENCES

Belarmino Adenso-Díaz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research*, 54(1):99–114, 2006. doi: 10.1287/opre.1050.0243. URL https://pubsonline.informs.org/doi/10.1287/opre.1050.0243.

Nir Ailon, Bernard Chazelle, Kenneth L. Clarkson, Ding Liu, Wolfgang Mulzer, and C. Seshadhri. Self-improving algorithms. *SIAM Journal on Computing*, 40(2):350–375, 2011. doi: 10.1137/090766437.

Anthropic. *Claude 3.7 Sonnet and Claude Code*, February 2025. URL https://www.anthropic.com/news/claude-3-7-sonnet. Accessed: 2026-06-21.

David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton University Press, 2006. ISBN 9780691129938.

Florent Avellaneda. A short description of the solver EvalMaxSAT. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins (eds.), *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, pp. 8–9, 2020.

Maria-Florina Balcan, Vaishnavh Nagarajan, Ellen Vitercik, and Colin White. Learning-theoretic foundations of algorithm configuration for combinatorial partitioning problems. In *Proceedings of the 2017 Conference on Learning Theory*, volume 65 of *Proceedings of Machine Learning Research*, pp. 213–274. PMLR, 2017. URL https://proceedings.mlr.press/v65/balcan17a.html.

Maria-Florina Balcan, Dan DeBlasio, Travis Dick, Carl Kingsford, Tuomas Sandholm, and Ellen Vitercik. How much data is sufficient to learn high-performing algorithms? generalization guarantees for data-driven algorithm design. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pp. 919–932. ACM, 2021. doi: 10.1145/3406325.3451036. URL https://doi.org/10.1145/3406325.3451036.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=ByldLrqlx.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations Workshop*, 2017.

Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N. Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3–20, 2002. doi: 10.1016/S0005-1098(01)00174-1. URL https://cse.lab.imtlucca.it/~bemporad/publications/papers/automatica-mpqp.pdf.

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.

Dimitris Bertsimas and Bartolomeo Stellato. Online mixed-integer optimization in milliseconds. *INFORMS Journal on Computing*, 34(4):2229–2248, 2022. doi: 10.1287/ijoc.2022.1181. URL https://pubsonline.informs.org/doi/10.1287/ijoc.2022.1181.

Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, et al. Enabling research through the SCIP optimization suite 8.0. *ACM Transactions on Mathematical Software*, 49(2):1–21, 2023. doi: 10.1145/3585516.

K.-H. Borgwardt. The average number of pivot steps required by the simplex-method is polynomial. *Zeitschrift für Operations Research*, 26:157–177, 1982. doi: 10.1007/BF01917108. URL https://link.springer.com/article/10.1007/BF01917108.

Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979. doi: 10.1145/359094.359101.

Sabah Bushaj and İ. Esra Büyüktahtakın. A k-means supported reinforcement learning framework to multi-dimensional knapsack. *Journal of Global Optimization*, 89(3):655–685, 2024. doi: 10.1007/s10898-024-01364-6.

Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=6s5uXNWGIh.

Mujia Chen, Sihao Liu, and Weihua He. Learn to solve dominating set problem with gnn and reinforcement learning. *Applied Mathematics and Computation*, 474:128717, 2024. doi: 10.1016/j.amc.2024.128717.

Siu-Wing Cheng, Man-Kwun Chiu, Kai Jin, and Man Ting Wong. A generalization of self-improving algorithms. In Sergio Cabello and Danny Z. Chen (eds.), *36th International Symposium on Computational Geometry (SoCG 2020)*, volume 164 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 29:1–29:13, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.SoCG.2020.29.

Adil Chhabra, Marlon Dittes, Ernestine Großmann, Kenneth Langedal, Henrik Reinstädtler, Christian Schulz, Darren Strash, and Henning Woydt. *PACE 2025 Submission for AEG Heidelberg*, 2025. URL https://github.com/KennethLangedal/PACE2025. Solver submission for the PACE 2025 Dominating Set and Hitting Set challenge.

Vašek Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979. doi: 10.1287/moor.4.3.233.

Kenneth L. Clarkson, Wolfgang Mulzer, and C. Seshadhri. Self-improving algorithms for coordinatewise maxima and convex hulls. *SIAM Journal on Computing*, 43(2):617–653, 2014. doi: 10.1137/12089702X.

Jordi Coll, Chu Min Li, Shuolin Li, Djamal Habet, and Felip Manyà. Solving weighted maximum satisfiability with branch and bound and clause learning. *Computers & Operations Research*, 183: 107195, 2025. doi: 10.1016/j.cor.2025.107195.

Leonardo Cunha, Gauthier Gidel, Fabian Pedregosa, Damien Scieur, and Courtney Paquette. Only tails matter: Average-case universality and robustness in the convex regime. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 4474–4491. PMLR, 2022. URL https://proceedings.mlr.press/v162/cunha22a.html.

Guilherme D. da Fonseca, Fabien Feschet, and Yan Gerard. PACE Solver Description: Shadoks Approach to Minimum Hitting Set and Dominating Set. In Akanksha Agrawal and Erik Jan van Leeuwen (eds.), *20th International Symposium on Parameterized and Exact Computation (IPEC 2025)*, volume 358 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 34:1–34:5, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-407-9. doi: 10.4230/LIPIcs.IPEC.2025.34. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.IPEC.2025.34.

George B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–288, 1957. doi: 10.1287/opre.5.2.266.

Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Theory and Applications of Satisfiability Testing – SAT 2013*, volume 7962 of *Lecture Notes in Computer Science*, pp. 166–181. Springer, 2013a. doi: 10.1007/978-3-642-39071-5_13.

Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In *Principles and Practice of Constraint Programming – CP 2013*, volume 8124 of *Lecture Notes in Computer Science*, pp. 247–262. Springer, 2013b. doi: 10.1007/978-3-642-40627-0_21.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 990–998. PMLR, 2017. URL https://proceedings.mlr.press/v70/devlin17a.html.

Carles Domingo-Enrich, Fabian Pedregosa, and Damien Scieur. Average-case acceleration for bilinear games and normal matrices. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=H0syOoy3Ash.

Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013. doi: 10.1007/978-1-4471-5559-1.

Florian Fontan and Guillaume Verger. PACE Solver Description: Reductions and Heuristic Search for the Dominating Set Problem and the Hitting Set Problem. In Akanksha Agrawal and Erik Jan van Leeuwen (eds.), *20th International Symposium on Parameterized and Exact Computation (IPEC 2025)*, volume 358 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 36:1–36:3, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-407-9. doi: 10.4230/LIPIcs.IPEC.2025.36. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.IPEC.2025.36.

John Forrest, Stefan Vigerske, Ted Ralphs, Lou Hafer, John Fasano, Haroldo Gambini Santos, Matthew Saltzman, Alan King, Samuel Brito, et al. coin-or/Clp: Release releases/1.17.7, 2022. URL https://doi.org/10.5281/zenodo.5839302.

John J. Forrest and Robin Lougee-Heimer. CBC user guide. In *Emerging Theory, Methods, and Applications*, pp. 257–277. INFORMS, 2005. doi: 10.1287/educ.1053.0020.

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. 15554–15566, 2019.

Mario Grobler and Sebastian Siebertz. The PACE 2025 Parameterized Algorithms and Computational Experiments Challenge: Dominating Set and Hitting Set. In Akanksha Agrawal and Erik Jan van Leeuwen (eds.), *20th International Symposium on Parameterized and Exact Computation (IPEC 2025)*, volume 358 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 32:1–32:17, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-407-9. doi: 10.4230/LIPIcs.IPEC.2025.32. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.IPEC.2025.32.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 317–330. ACM, 2011. doi: 10.1145/1926385.1926423.

Rishi Gupta and Tim Roughgarden. A PAC approach to application-specific algorithm selection. *SIAM Journal on Computing*, 46(3):992–1017, 2017. doi: 10.1137/15M1050276. URL https://epubs.siam.org/doi/10.1137/15M1050276.

Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*, 2026. URL https://docs.gurobi.com/projects/optimizer/en/current/index.html. Version 13.0.

Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962. doi: 10.1137/0110015.

Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000. doi: 10.1016/S0377-2217(99)00284-2.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Y. Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html.

HiGHS Development Team. HiGHS: High-performance parallel linear optimization software, 2024. URL https://highs.dev/.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 20271–20309. PMLR, 2024. URL https://proceedings.mlr.press/v235/huang24y.html.

Qi Huangfu and J. A. J. Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018. doi: 10.1007/s12532-017-0130-5.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Automated configuration of mixed integer programming solvers. In Andrea Lodi, Michela Milano, and Paolo Toth (eds.), *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *Lecture Notes in Computer Science*, pp. 186–202. Springer, 2010. doi: 10.1007/978-3-642-13520-0_23. URL https://link.springer.com/chapter/10.1007/978-3-642-13520-0_23.

Alexey Ignatiev, Antonio Morgado, and João Marques-Silva. RC2: An efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019. doi: 10.3233/SAT190116.

Russell Impagliazzo. A personal view of average-case complexity. In *Proceedings of the Tenth Annual Structure in Complexity Theory Conference*, pp. 134–147. IEEE Computer Society, 1995. doi: 10.1109/SCT.1995.514853.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, volume 30, pp. 6348–6358, 2017.

Philipp Kindermann, Fabian Klute, and Soeren Terziadis. The PACE 2024 Parameterized Algorithms and Computational Experiments Challenge: One-Sided Crossing Minimization. In *19th International Symposium on Parameterized and Exact Computation (IPEC 2024)*, volume 321 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 26:1–26:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi: 10.4230/LIPIcs.IPEC.2024.26.

Robert Kleinberg, Kevin Leyton-Brown, and Brendan Lucier. Efficiency through procrastination: Approximately optimal algorithm configuration with runtime guarantees. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pp. 2023–2031, 2017. doi: 10.24963/ijcai.2017/281. URL https://www.ijcai.org/proceedings/2017/281.

Robert Kleinberg, Kevin Leyton-Brown, Brendan Lucier, and Devon Graham. Procrastinating with confidence: Near-optimal, anytime, adaptive algorithm configuration. In *Advances in Neural Information Processing Systems*, volume 32, pp. 8883–8893, 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/hash/538a50fb36b97123adc3627cbdb223bf-Abstract.html.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments*, pp. 144–158. SIAM, 2019. doi: 10.1137/1.9781611975499.12.

Ailsa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960. doi: 10.2307/1910129.

Leonid A. Levin. Average case complete problems. *SIAM Journal on Computing*, 15(1):285–286, 1986. doi: 10.1137/0215020.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44 (10):2245–2269, 1965. doi: 10.1002/j.1538-7305.1965.tb04146.x.

Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015. doi: 10.1613/jair.4726.

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

Manuel López-Ibáñez and Thomas Stützle. Automatically improving the anytime behaviour of optimisation algorithms. *European Journal of Operational Research*, 235(3):569–582, 2014. doi: 10.1016/j.ejor.2013.10.043. URL https://lopez-ibanez.eu/doc/LopStu2014ejor.pdf.

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016. doi: 10.1016/j.orp.2016.09.002. URL https://www.sciencedirect.com/science/article/pii/S2214716015300270.

Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 3296–3305. PMLR, 2018. URL https://proceedings.mlr.press/v80/lykouris18a.html.

Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pp. 438–445. Springer, 2014. doi: 10.1007/978-3-319-09284-3_33.

Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & Operations Research*, 51:190–199, 2014. doi: 10.1016/j.cor.2014.05.020. URL https://lopez-ibanez.eu/doc/MasLopDubStu2014cor.pdf.

Jacob Mattingley and Stephen Boyd. CVXGEN: a code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, 2012. doi: 10.1007/s11081-011-9176-9. URL https://web.stanford.edu/~boyd/papers/code_gen_impl.html.

Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 65(7):33–35, 2022. doi: 10.1145/3528087.

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

Maxwell Nye, Luke Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 4861–4870. PMLR, 2019. URL https://proceedings.mlr.press/v97/nye19a.html.

OpenAI. Introducing Codex. https://openai.com/index/introducing-codex/, 2025. Accessed: 2026-06-21.

Courtney Paquette, Kiwon Lee, Fabian Pedregosa, and Elliot Paquette. SGD in the large: Average-case analysis, asymptotics, and stepsize criticality. In *Proceedings of Thirty Fourth Conference on Learning Theory*, volume 134 of *Proceedings of Machine Learning Research*, pp. 3548–3626. PMLR, 2021. URL https://proceedings.mlr.press/v134/paquette21a.html.

Courtney Paquette, Bart van Merriënboer, Elliot Paquette, and Fabian Pedregosa. Halting time is predictable for large models: A universality property and average-case analysis. *Foundations of Computational Mathematics*, 23:597–673, 2023. doi: 10.1007/s10208-022-09554-y. URL https://link.springer.com/article/10.1007/s10208-022-09554-y.

Seonho Park and Pascal Van Hentenryck. Self-supervised primal-dual learning for constrained optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(4):4052–4060, 2023. doi: 10.1609/aaai.v37i4.25520.

Fabian Pedregosa and Damien Scieur. Acceleration through spectral density estimation. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7553–7562. PMLR, 2020. URL https://proceedings.mlr.press/v119/pedregosa20a.html.

Laurent Perron and Vincent Furnon. OR-Tools, 2025. URL https://developers.google.com/optimization/.

Laurent Perron, Frédéric Didier, and Steven Gay. The CP-SAT-LP solver. In *29th International Conference on Principles and Practice of Constraint Programming*, volume 280 of *Leibniz International Proceedings in Informatics*, pp. 3:1–3:2. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPIcs.CP.2023.3. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2023.3.

Marek Piotrów. UWrMaxSat: Efficient solver for MaxSAT and pseudo-boolean problems. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence*, pp. 132–136, 2020. doi: 10.1109/ICTAI50040.2020.00031.

Adam Polak and Jonas Schmidt. Greeduce: A heuristic solver for the hitting set and dominating set problems. https://github.com/adampolak/greeduce, 2025. PACE 2025 heuristic-track submission.

John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976. doi: 10.1016/S0065-2458(08)60520-3.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

Root team. The Heuristic Solver for the PACE 2025 Challenge. https://github.com/lxily/PACE2025.DS-HS, 2025. PACE 2025 heuristic-track solver for Dominating Set and Hitting Set.

Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, 1977. doi: 10.1137/0206041.

Tim Roughgarden. Beyond worst-case analysis. *Communications of the ACM*, 62(3):88–96, 2019. doi: 10.1145/3232535.

Rajiv Sambharya, Georgina Hall, Brandon Amos, and Bartolomeo Stellato. Learning to warm-start fixed-point optimization algorithms. *Journal of Machine Learning Research*, 25(166):1–46, 2024. URL https://jmlr.org/papers/v25/23-1174.html.

Martin J. A. Schuetz, J. Kyle Brubaker, and Helmut G. Katzgraber. Combinatorial optimization with physics-inspired graph neural networks. *Nature Machine Intelligence*, 4(4):367–377, 2022a. doi: 10.1038/s42256-022-00468-6.

Martin J. A. Schuetz, J. Kyle Brubaker, Zhihuai Zhu, and Helmut G. Katzgraber. Graph coloring with physics-inspired graph neural networks. *Physical Review Research*, 4(4):043131, 2022b. doi: 10.1103/PhysRevResearch.4.043131.

Damien Scieur and Fabian Pedregosa. Universal average-case optimality of Polyak momentum. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 8565–8572. PMLR, 2020. URL https://proceedings.mlr.press/v119/scieur20a.html.

Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. ISBN 9781107057135.

Shivam Singhal, Priyadarsi Mishra, Eran Malach, and Tomer Galanti. LLM priors for ERM over programs, 2025. URL https://arxiv.org/abs/2510.14331.

Stephen Smale. The problem of the average speed of the simplex method. In Achim Bachem, Bernhard Korte, and Martin Grötschel (eds.), *Mathematical Programming: The State of the Art*, pp. 530–539. Springer, 1983. doi: 10.1007/978-3-642-68874-4_20. URL https://link.springer.com/chapter/10.1007/978-3-642-68874-4_20.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4FWAwZtd2n.

Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004. doi: 10.1145/990308.990310.

Sylwester Swat. PACE Solver Description: HitS&DoSeS - Exact and Heuristic Solvers for the Dominating Set and Hitting Set Problems. In Akanksha Agrawal and Erik Jan van Leeuwen (eds.), *20th International Symposium on Parameterized and Exact Computation (IPEC 2025)*, volume 358 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 38:1–38:4, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-407-9. doi: 10.4230/LIPIcs.IPEC.2025.38. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.IPEC.2025.38.

Prasad Tadepalli and Balas K. Natarajan. A formal framework for speedup learning from problems and solutions. *Journal of Artificial Intelligence Research*, 4:445–475, 1996. doi: 10.1613/jair.154. URL https://www.jair.org/index.php/jair/article/view/10167.

Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition*, pp. 2464–2469, 2016. doi: 10.1109/ICPR.2016.7900006.

Michael J. Todd. Polynomial expected behavior of a pivoting algorithm for linear complementarity and linear programming problems. *Mathematical Programming*, 35(2):173–192, 1986. doi: 10.1007/BF01580646. URL https://link.springer.com/article/10.1007/BF01580646.

Jan Tönshoff, Martin Ritzert, Hinrikus Wolf, and Martin Grohe. Graph neural networks for maximum constraint satisfaction. *Frontiers in Artificial Intelligence*, 3:580607, 2021. doi: 10.3389/frai.2020.580607.

Jan Tönshoff, Berke Kisin, Jakob Lindner, and Martin Grohe. One model, any CSP: Graph neural networks as fast global search heuristics for constraint satisfaction. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI-23)*, pp. 4280–4288, 2023. doi: 10.24963/ijcai.2023/476.

Craig A. Tovey. Low order polynomial bounds on the expected performance of local improvement algorithms. *Mathematical Programming*, 35(2):193–224, 1986. doi: 10.1007/BF01580647. URL https://link.springer.com/article/10.1007/BF01580647.

Leslie G. Valiant. A theory of the learnable. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pp. 436–445. ACM, 1984. doi: 10.1145/800057.808710.

V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications*, 16(2):264–280, 1971. doi: 10.1137/1116025.

Vladimir N. Vapnik. *Statistical Learning Theory*. Wiley, 1998. ISBN 9780471030034.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, volume 28, pp. 2692–2700, 2015.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Gellért Weisz, András György, and Csaba Szepesvári. LeapsAndBounds: A method for approximately optimal algorithm configuration. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5257–5265. PMLR, 2018. URL https://proceedings.mlr.press/v80/weisz18a.html.

Gellért Weisz, András György, and Csaba Szepesvári. CapsAndRuns: An improved method for approximately optimal algorithm configuration. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6707–6715. PMLR, 2019. URL https://proceedings.mlr.press/v97/weisz19a.html.

Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pp. 1173–1178. Morgan Kaufmann, 2003.

Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008. doi: 10.1613/jair.2490.

Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pp. 210–216, 2010.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://openreview.net/forum?id=5Xc1ecxO1h.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, pp. 43571–43608, 2024. doi: 10.52202/079017-1381. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/4ced59d480e07d290b6f29fc8798f195-Abstract-Conference.html.

Dinghuai Zhang, Hanjun Dai, Nikolay Malkin, Aaron C. Courville, Yoshua Bengio, and Ling Pan. Let the flows tell: Solving graph combinatorial problems with GFlowNets. In *Advances in Neural Information Processing Systems*, volume 36, pp. 11952–11969, 2023.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=YrycTjllL0.

A Broader Impacts

This work studies methods for learning distribution-specific solvers from examples. Potential positive impacts include reducing the computational cost of repeated optimization workloads, making specialized solvers easier to construct, and improving access to efficient algorithmic tools in scientific, engineering, and logistics settings. Potential negative impacts arise if synthesized solvers are deployed outside the distributions for which they were inferred, where brittle specialization may lead to incorrect or inefficient decisions. More broadly, automated solver synthesis could also reduce the barrier to optimizing objectives in harmful or poorly governed applications. We therefore view validation on held-out instances, explicit distributional assumptions, fallback mechanisms, and human review of generated code as important safeguards.

B Proof of Prop. 3.1

Proposition 3.1 (Sample-conditioned computational complexity)**.** *Suppose* $0 \le T(A, X) \le B$ *for every base program* A*, every* $D \in \mathcal{D}_n$*, and every* X *in the support of* D*. For* $m \ge 1$*, let* $\varepsilon_m = B\sqrt{\frac{(\ell+1)\log 2}{2m}}$*. Then* $C_{\mathrm{known}} \le C_m \le \min\{C_0,\, C_{\mathrm{known}} + \varepsilon_m\}$*. In particular, for a fixed program class,* $C_m \to C_{\mathrm{known}}$ *as* $m \to \infty$*.*

Proof. Fix D, and let $\widehat{A}_S$ minimize the empirical runtime $\widehat{R}_S(A) = m^{-1}\sum_{i=1}^{m} T(A, X_i)$ over the base programs. Hoeffding's lemma gives $\mathbb{E}_S \exp\{\lambda(R_D(A) - \widehat{R}_S(A))\} \le \exp\{\lambda^2 B^2/(8m)\}$. Applying the standard exponential-moment bound to the maximum over N programs yields $\mathbb{E}_S \max_A (R_D(A) - \widehat{R}_S(A)) \le B\sqrt{\log N/(2m)}$.

Let $A_D^\star$ minimize $R_D(A)$. Since $\widehat{R}_S(\widehat{A}_S) \le \widehat{R}_S(A_D^\star)$, taking expectations gives

$$\mathbb{E}_S[R_D(\widehat{A}_S)] \le \inf_{A \in \mathcal{A}} R_D(A) + B\sqrt{\frac{\log N}{2m}}.$$

Taking the supremum over D and using $\log N \le (\ell+1)\log 2$ gives the stated upper bound. A learner may always ignore its sample, so $C_m \le C_0$. Conversely, for each fixed D, no learner can achieve expected deployment runtime below $\inf_A R_D(A)$, which gives $C_{\mathrm{known}} \le C_m$. □

C Additional Method Details

This appendix gives the protocol details omitted from Section 4: what information the synthesis agent receives, how candidates are represented and selected, and how failures are handled. The analysis program may use the public training sample to compute a reusable summary, but the deployed solver receives only that summary and the new instance.

Each LLM proposal is treated as a candidate rather than the learned algorithm itself. The search produces alternative structural hypotheses and solver implementations, some of which may be slow, brittle, or incorrect. Candidates are executed and ranked using public validation quality, optimality, and runtime, and the selected candidate is deployed.

Public information and leakage boundary. For each target, the agent receives the public training and validation instances and a task manifest specifying the problem, output format, quality metric, and basic instance sizes. It is also told that the instances come from a shared but unknown structured distribution.

The agent does not see the distribution family, generative rule, optimum solutions or values, test instances, or evaluator-only metadata. Thus it knows the optimization problem and how solutions are scored, but not the mechanism generating the instances.

All synthesis stages share the same objective: infer structure from the public samples and use it to produce a fast solver. Validation quality is the primary selection criterion, with runtime used to distinguish high-quality candidates.

Table 5: **Synthesis-stage outputs.** The diversity key is used for beam selection, not validation scoring.

Stage	Output
Hypothesis	Suspected reusable structure; evidence to measure; the corresponding solver strategy; likely failure modes; and a diversity key identifying the broad hypothesis class.
Analysis	A program applied to $S_{\text{tr}}^{\text{pub}}$ that estimates the relevant structure and returns a compact reusable summary rather than the training instances themselves.
Solver	A deployment program for new instances that uses the recovered summary, with fallback behavior for weak or ambiguous structure and without repeating the training-time analysis.

Algorithm 1 LLM-based solver synthesis

Require: Public train/validation sets $S_{\text{tr}}^{\text{pub}}, S_{\text{val}}^{\text{pub}}$; rounds R, beam width B, candidate budget K
Ensure: Solver $\hat{s}$ and summary $\hat{a}$

Initialize beam $\mathcal{B}_0$ with diverse hypotheses
$\mathcal{E} \leftarrow \emptyset$
for $r = 0, \ldots, R-1$ **do**
  Form K candidate prompts from $\mathcal{B}_r$
  for each candidate prompt **do**
    Hypothesis H_c
    Analysis A_c
    Run analysis $a_c \leftarrow A_c(S_{\text{tr}}^{\text{pub}})$
    Solver $s_c \leftarrow \text{LLM}(H_c, A_c, a_c)$
    Evaluate $s_c(\cdot, a_c)$ on $S_{\text{tr}}^{\text{pub}}$ and $S_{\text{val}}^{\text{pub}}$
    Add $c = (H_c, A_c, a_c, s_c)$ to $\mathcal{E}$
  end for
  Rank candidates by

$$\big(Q_{\text{val}}(c),\ O_{\text{val}}(c),\ -T_{\text{val}}(c)\big)$$

  Set $\mathcal{B}_{r+1}$ by keeping the best candidate from each diversity class, then filling remaining slots by score
end for
$c^\star \leftarrow \arg\max_{c \in \mathcal{E}} \text{Score}(c)$
$\hat{a} \leftarrow A_{c^\star}(S_{\text{tr}}^{\text{pub}})$
return $\hat{s} := s_{c^\star}$ and $\hat{a}$

Candidate representation. Each candidate is $c = (H_c, A_c, a_c, s_c)$. Here H_c is a hypothesis about recurring structure, A_c is an analysis program, $a_c = A_c(S_{\text{tr}}^{\text{pub}})$ is the resulting solver hint, and s_c is the deployment solver. Given a new instance x^{pub}, the solver returns $z = s_c(x^{\text{pub}}, a_c)$.

Table 5 summarizes what is produced at each stage.

Synthesis loop. Algorithm 1 summarizes the search procedure. In each round, the agent generates candidate hypotheses, analysis programs, and solvers; the resulting solvers are evaluated on the public training and validation sets, and the beam is updated. We keep track of all evaluated candidates and deploy the best one found over the full search.

Beam selection and refinement. Each hypothesis is assigned a broad class, such as separator-based, bottleneck-resource, latent-cluster, or geometric-template. After each round, candidates are ranked by the validation score in Algorithm 1. We retain the best candidate from each class and fill the remaining beam slots by score.

Refinement uses the parent hypothesis, recovered summary, train and validation metrics, and representative failures. The agent may revise the current hypothesis, branch to another explanation, or improve quality or runtime. Hidden distribution metadata and test results are never provided. The

Table 6: **Benchmark target distributions.** Seven problem classes, each with three hidden distribution families, giving 21 targets. Each target is split into 64 training, 32 validation, and 500 held-out test instances.

Problem *objective*	**Hidden distribution families** *(name, instance size)*		
Coloring *min. colors*	Ring-template 169 vertices	Overlapping-palette 340 vertices	Separator-trap 238 vertices
MAXSAT *max. sat. clauses*	Community-parity 240 var. / 960 cl.	Last-clause signal 280 var. / 1120 cl.	Latent-backdoor 128 var. / 512 cl.
MIS *max. set size*	Clique-path 190 vertices	Core-fringe trap 1000 vertices	Motif-bridge 195 vertices
MDS *min. set size*	Gateway-hub 2800 vertices	Geometric-anchor 1600 vertices	Star-kernel 2800 vertices
Packing LP *max. value*	Block-coupled 1200 items / 40 res.	Active-resource 1200 items / 40 res.	Single-bottleneck 1200 items / 40 res.
MDKP *max. value*	Decoy-complement 1040 items / 48 res.	Latent-class 520 items / 32 res.	Single-resource 1040 items / 48 res.
TSP *min. tour length*	Clustered Euclidean 120 cities	Latent-metric 120 cities	Paired-ribbon zigzag 320 cities

diversity rule simply keeps several structural explanations in play while validation determines which candidates survive.

Recovered summaries. The summary a_c has no fixed representation. Depending on the problem, it may contain vertex-role scores, clusters, separators, Boolean statistics, resource bottlenecks, item classes, or geometric structure. It is not scored directly; only the quality and runtime of the resulting solver matter.

The analysis program computes a_c once from the training sample, and the deployed solver reuses it on future instances. Thus distribution-level inference is performed before deployment rather than repeated for every solve.

Failure handling. Candidates may fail during analysis, solver construction, or execution. If A_c fails or exceeds its time budget, or if s_c cannot be loaded, the candidate receives zero quality and a large failure runtime. If a running solver produces an invalid output on only some instances, those instances receive zero quality and the rest are scored normally. Failed candidates are not manually repaired.

Connection to the analysis. The protocol is a practical version of the sample-conditioned selection problem in Section 3. The hypothesis H_c proposes reusable structure, A_c estimates it from samples, and s_c turns the resulting summary into a deployment algorithm. Unlike the idealized analysis, neither the relevant structure nor its representation is fixed in advance: both are proposed by the LLM and selected through validation. The search therefore approximates a sample-to-solver map by generating and testing candidate specializations.

D Additional Experimental Details

D.1 Benchmark Distributions

Table 6 summarizes the 21 benchmark targets: seven problem classes, each with three hidden distribution families. Every target is split into 64 training, 32 validation, and 500 held-out test instances. The distributions are structured rather than adversarial, with recurring regularities that a solver may exploit after seeing a small sample. These include planted assignments, latent classes, bottlenecks, geometric layouts, motifs, and hidden resource constraints.

Table 7: **Normalized solution quality.** Quality is the ratio between the returned and optimal objective, oriented so that higher is always better: optimal solutions have quality 1, and invalid or infeasible outputs receive 0.

Problem	Measured quantity	Normalized quality
Maximization ***quality*** $=$ ***returned*** $/$ ***optimum***		
MAXSAT	satisfied clauses sat	$\mathrm{sat}_{\mathrm{alg}}/\mathrm{sat}_{\mathrm{opt}}$
MIS	independent-set size $\lvert\mathrm{IS}\rvert$	$\lvert\mathrm{IS}_{\mathrm{alg}}\rvert/\lvert\mathrm{IS}_{\mathrm{opt}}\rvert$
Packing LP, MDKP	objective value obj	$\mathrm{obj}_{\mathrm{alg}}/\mathrm{obj}_{\mathrm{opt}}$
Minimization ***quality*** $=$ ***optimum*** $/$ ***returned***		
Coloring	colors used k	$k_{\mathrm{opt}}/k_{\mathrm{alg}}$
MDS	dominating-set size $\lvert\mathrm{DS}\rvert$	$\lvert\mathrm{DS}_{\mathrm{opt}}\rvert/\lvert\mathrm{DS}_{\mathrm{alg}}\rvert$
TSP	tour length len	$\mathrm{len}_{\mathrm{opt}}/\mathrm{len}_{\mathrm{alg}}$

Graph coloring. *Ring-template* partitions vertices into local blocks with compatible palettes and ring-like bridges. *Overlapping-palette* uses blocks with shifted, overlapping palettes, so local coloring alone is insufficient. *Separator-trap* introduces separator vertices that create long-range color-reuse constraints.

MAXSAT. *Community-parity* partitions variables into communities controlled by parity-style functions of a few anchor bits. In *Last-clause signal*, the final clause reveals an anchor pattern and the remaining variables copy or negate anchor bits according to a hidden rule table. *Latent-backdoor* mixes several regimes in which variable blocks follow anchor-dependent Boolean rules, with additional noise and bridge clauses.

Maximum independent set. *Clique-path* alternates clique-like and path-like blocks connected by sparse bridges. *Core-fringe trap* places a low-degree fringe around a dense core, making naive low-degree greedy rules misleading. *Motif-bridge* draws instances from latent sequences of cliques, cycles, bicliques, crown-like gadgets, and sparse inter-motif bridges.

Minimum dominating set. *Gateway-hub* contains clusters with hubs and gateways whose coverage overlaps across neighboring clusters. *Geometric-anchor* uses geometric clusters of varying density connected by sparse edges. *Star-kernel* consists of clusters with stable local hubs and sparse inter-cluster connectors.

Packing LP. *Block-coupled* groups items into latent resource blocks linked by a shared coupling constraint. *Active-resource* varies the active resource regime across instances, producing recurring dual-price patterns. In *Single-bottleneck*, one resource is consistently tighter than the others, making value per unit of that resource highly informative.

Multidimensional knapsack. *Decoy-complement* includes high-value decoys that consume a scarce hidden resource, while complementary item classes combine more efficiently. *Latent-class* draws items from hidden resource-consumption classes and varies the active bottleneck across instances. In *Single-resource*, one resource is usually dominant, making density with respect to it a strong but imperfect guide.

Traveling salesperson problem. *Clustered Euclidean* samples cities from balanced clusters arranged around a ring, favoring tours with short intra-cluster paths and the correct inter-cluster order. *Latent-metric* mixes several geometric regimes, including ring clusters, paired ribbons, and barrier-bridge layouts. In *Paired-ribbon zigzag*, cities lie on two nearly parallel rails, so the best tours traverse one rail and return along the other.

D.2 Evaluation Metrics

We evaluate solvers by normalized solution quality, optimality rate, and runtime. Because the benchmark mixes maximization and minimization problems, we map objective values to a common quality score in $[0, 1]$, with 1 denoting an optimum. Table 7 gives the normalization for each problem class.

Normalized quality. For an instance x and solver A, define

$$q_A(x) = \begin{cases} \mathrm{val}_A(x)/\mathrm{val}_{\mathrm{opt}}(x), & \text{for maximization,} \\ \mathrm{cost}_{\mathrm{opt}}(x)/\mathrm{cost}_A(x), & \text{for minimization,} \end{cases}$$

when the returned solution is feasible. Invalid, malformed, or infeasible outputs receive $q_A(x) = 0$. All benchmark optima are positive, so these ratios are well defined.

Optimality rate. The optimality rate is the fraction of held-out instances with $q_A(x) = 1$. Equality is checked by the benchmark verifier, with numerical tolerance for Packing LP. Invalid or infeasible outputs are not counted as optimal.

Runtime. Runtime is measured per instance in milliseconds and includes all deployment-time computation, including verification, repair, fallback, and local search. Offline synthesis is excluded.

Aggregation. Each of the 21 target distributions receives equal weight. We first average quality, optimality, feasibility, and runtime over held-out instances and repeated runs within each target. Additive quantities, including quality differences $\Delta Q_b = Q_{\mathrm{ours}} - Q_b$, are then averaged across targets. Runtime comparisons use $S_b(\tau) = \frac{T_b(\tau)}{T_{\mathrm{ours}}(\tau)}$, so $S_b > 1$ means our solver is faster; reported speedups are geometric means over targets. Comparator definitions are given in Section 5.

D.3 Baseline Catalog and Reporting Protocol

The main text summarizes the baseline groups; this appendix gives the full per-problem catalog and execution details. Table 8 lists 78 registered entries across seven problem classes, corresponding to 72 unique implementations after removing aliases. We retain aliases because they are executable registry entries; for example, several `exact` entries point to problem-specific implementations such as OR-Tools formulations or Held–Karp dynamic programming.

Classical baselines fall into three groups. *Heuristics* are local problem-specific methods such as greedy, local-search, insertion, rounding, and randomized rules. *Solver-backed* methods call general-purpose optimization software under the benchmark protocol. Gurobi is reported separately with a 10-second single-thread budget and may return an incumbent without proving optimality. *Exact* and *external exact* methods provide certified comparisons when they terminate with an optimum.

Each problem class also has one architecture-matched learned baseline rather than a single shared model: GNN-based methods for coloring, MIS, and MDS; RUN-CSP-style models for MaxSAT; an attention model for TSP; and learned packing methods for MDKP and Packing LP. The LLM baselines are one-shot Codex, one-shot Claude Code, and best-of-5 Gemma 4; these are described in the main text.

In aggregate tables, Heur denotes the fastest heuristic within 95% of the best heuristic quality for that class, avg the average heuristic, Gur Gurobi, and Exact the fastest certified exact method attaining the best validated quality. All baselines use the same public-instance protocol and receive no information about the hidden distribution family or generative rule.

D.4 Compute and API Usage

Our primary solvers were synthesized with GPT-5.2 Thinking through a remote API, using high reasoning effort and structured outputs; temperature, top-p, and output-length settings were left at their defaults. Across the 21 target distributions, synthesis used 463 successful API calls and 16.1M tokens, averaging 22.0 calls and 767K tokens per target. Of these, 5.95M were prompt tokens and 10.16M completion tokens, including 8.05M reasoning tokens.

We also run the same pipeline with Gemma 4 (`gemma-4-31b-it`) on a single NVIDIA H100 NVL, both for the full synthesis pipeline (Table 11) and for the best-of-5 baseline. The latter used 3.4M tokens across the 21 targets and about 1.9 GPU-hours. The main benchmark run took approximately 4.12 hours.

All deployed solvers and classical baselines run locally on CPU, with no GPU use, training, or fine-tuning. Experiments used a dual-socket AMD EPYC 9554 machine with 128 physical cores, 256 logical CPUs, and approximately 1.1 TiB RAM. Summed across targets, recorded synthesis-stage time was 73.4 hours, classical baseline evaluation 417.3 hours, and other local synthesis work

Table 8: **Classical and learned baseline catalog.** The top table lists problem-specific heuristics, exact or certifying methods, and the learned baseline for each problem class. The bottom table lists general-purpose backends shared across classes. Each class also has an `exact` alias for its designated exact backend. LLM baselines are described in the main text.

Problem	Heuristics	Exact / certifying	Learned
Coloring	DSATUR and greedy variants: largest-degree, random-order, and smallest-last orderings (Brélaz, 1979)	DSATUR branch-and-bound (Brélaz, 1979; Land & Doig, 1960)	PI-GNN (Schuetz et al., 2022b;a)
MaxSAT	Greedy flip local search; literal-majority polarity; random assignment	EvalMaxSAT (Avellaneda, 2020); MaxHS (Davies & Bacchus, 2013a;b); Open-WBO (Martins et al., 2014); UWrMaxSAT (Piotrów, 2020); WMaxCDCL (Coll et al., 2025)	RUN-CSP (Tönshoff et al., 2021)
MIS	Local improvement; minimum-degree, random, and ratio-based greedy	Clique branch-and-bound (Land & Doig, 1960); KaMIS via exact vertex cover (Lamm et al., 2019)	PI-GNN (Schuetz et al., 2022a)
MDS	High-degree, marginal-gain, and redundancy-aware greedy (Chvátal, 1979)	Set-cover branch-and-bound (Chvátal, 1979; Land & Doig, 1960)	GNN+RL (Chen et al., 2024)
Packing LP	Density-based fractional packing; uniform-fraction LP (Dantzig, 1957)	Generic LP backends only	Primal–dual net (Park & Van Hentenryck, 2023)
MDKP	LP-relaxation rounding; redundancy-improved greedy; value-density greedy (Dantzig, 1957)	Custom branch-and-bound (Land & Doig, 1960)	Deep RL agent (Bushaj & Büyüktahtakın, 2024)
Euclidean TSP	Random tour; nearest neighbor; nearest and farthest insertion; multi-start two-opt; nearest-neighbor and farthest-insertion plus two-opt (Lin, 1965; Rosenkrantz et al., 1977); LKH (Helsgaun, 2000)	Concorde (Applegate et al., 2006); Held–Karp DP (Held & Karp, 1962); MTZ formulations via CBC and SCIP	Attention model (Kool et al., 2019)

Shared backend	Color.	MaxSAT	MIS	MDS	Pack. LP	MDKP	TSP
Gurobi, 10 s single-thread (Gurobi Optimization, LLC, 2026)	✓	✓	✓	✓	✓	✓	✓
OR-Tools CP-SAT; GLOP for LP (Perron & Furnon, 2025; Perron et al., 2023)	✓		✓	✓	✓	✓	✓
SCIP MIP; LP (Bestuzheva et al., 2023)	✓		✓	✓	✓	✓	✓
HiGHS MIP; simplex and interior point (HiGHS Development Team, 2024; Huangfu & Hall, 2018)	✓		✓	✓	✓	✓	
CBC MIP (Forrest & Lougee-Heimer, 2005)				✓		✓	✓
CLP LP (Forrest et al., 2022)					✓		
PySAT: SAT-based; RC2 with CaDiCaL 1.9.5, Glucose4, MiniSat 2.2, and default backend (Ignatiev et al., 2019)	✓	✓					

approximately 0.6 hours. These are aggregate per-target stage times rather than elapsed wall-clock time.

GPU use is limited to Gemma generation and the learned ML baselines, both on the same H100 NVL. The ML models are trained separately for each target, using approximately 6.0 GPU-hours in total, and timed at inference on both CPU and GPU. As with GPT-5.2 synthesis, these offline costs are excluded from deployment-runtime comparisons.

E Additional Empirical Results

E.1 ANYCSP and GFlowNet Baselines

To strengthen the learned comparison on the problem classes where these methods naturally apply, we implement compact reproductions of ANYCSP (Tönshoff et al., 2023) for Coloring and MaxSAT and a GFlowNet (Zhang et al., 2023) combinatorial-optimization model for MIS and MDS. Each model is trained separately for each target using only the public training and validation splits and is evaluated on the same 500 held-out instances as our synthesized solver. These are reduced-budget reproductions in the style of the corresponding model families rather than the authors' original implementations; by construction they cover four of the seven problem classes.

Across the 12 matched targets, our solver matches or exceeds baseline quality on $11/12$ and is faster on all 12, with speedups ranging from $2.5\times$ to over $2{,}800\times$. The single quality loss is Gateway-hub MDS, where GFlowNet obtains 0.9250 versus 0.9197 for ours. These comparisons should be

Table 9: **Compute summary.** All costs below are incurred once, before deployment, and are excluded from the runtime ratios reported in the main text — as are the training costs of the learned ML baselines. Timing values are aggregate recorded per-target stage times, not calendar elapsed time.

Offline stage	**Runs on**	**Generation cost**	**Time**
GPT-5.2 synthesis 21 targets; 22.0 calls and 767K tokens per target	remote API + local CPU	463 calls, 16.11M tokens 5.95M prompt / 10.16M completion, of which 8.05M reasoning	73.4 h recorded stage time
Gemma 4 generation, best-of-5	1× H100	3.39M tokens	≈1.9 GPU-h
ML baseline training, all targets	1× H100	—	≈6.0 GPU-h
Classical baseline pre-synthesis evaluation	CPU	—	417.3 h
Other local work inside synthesis analysis execution, solver construction, train/val evaluation	CPU	—	≈0.6 h
Deployment	CPU	**no LLM calls, no training**	**ms / instance**

Hardware: 2× AMD EPYC 9554 (128 physical / 256 logical CPUs), ≈1.1 TiB RAM, 1× NVIDIA H100 NVL; 83 GB stored benchmark artifacts. Experiments were parallelized across target distributions.

Table 10: **Matched learned baselines on full-size held-out targets.** ANYCSP is evaluated on Coloring and MaxSAT, and GFlowNet on MIS and MDS. Q is normalized quality; runtime is per instance in milliseconds. Speedup is $T_{\text{baseline}}/T_{\text{ours}}$.

Problem	**Target**	**Baseline**	Q_b	Q_{ours}	T_b	T_{ours}	**Speedup**
Coloring	Ring-template	ANYCSP	0.7770	1.0000	42.5	0.2	212.5×
Coloring	Overlapping-palette	ANYCSP	0.7980	0.8040	60.2	24.0	2.5×
Coloring	Separator-trap	ANYCSP	0.7740	0.8000	45.1	3.9	11.6×
MaxSAT	Community-parity	ANYCSP	0.9985	0.9999	4922	24.0	205.1×
MaxSAT	Last-clause signal	ANYCSP	1.0000	1.0000	834	5.7	146.3×
MaxSAT	Latent-backdoor	ANYCSP	0.9980	0.9991	2496	40.0	62.4×
MIS	Clique-path	GFlowNet	0.8660	0.9914	148	15.0	9.9×
MIS	Core-fringe	GFlowNet	1.0000	1.0000	2942	7.2	408.6×
MIS	Motif-bridge	GFlowNet	0.9130	0.9860	397	57.0	7.0×
MDS	Gateway-hub	GFlowNet	0.9250	0.9197	9650	5.5	1754.5×
MDS	Geometric-anchor	GFlowNet	0.8410	1.0000	6894	128.0	53.9×
MDS	Star-kernel	GFlowNet	1.0000	1.0000	9951	3.5	2843.1×

read as compact model-family reproductions rather than claims against the authors' fully tuned implementations.

E.2 Generator Model Ablation

We test how much the synthesis results depend on the model used to generate hypotheses, analysis code, and solvers. We rerun the same pipeline with Gemma 4 (`gemma-4-31b-it`) in place of GPT-5.2, keeping the synthesis procedure and evaluation protocol fixed.

GPT-5.2 produces higher-quality solvers in every problem family, with mean normalized quality 0.971 versus 0.883 for Gemma 4. Deployment runtime differs less: GPT-generated solvers are 1.38× faster overall, but Gemma produces faster solvers for MDS and, especially, MDKP. Thus the generator affects both the specialization found and the resulting quality–runtime tradeoff; the advantage of the stronger model is primarily in solution quality rather than uniformly lower runtime.

E.3 Matched Hint Ablation

We next isolate the contribution of explicit hint recovery from iterative code refinement. The matched baseline uses the same GPT-5.2 model, beam search, candidate budget, validation feedback, refinement procedure, repair, and test splits, but omits the hypothesis/hint stage. It still uses a generic analysis program, so the comparison tests explicit structural factorization rather than analysis itself.

Table 11: **Generator-model ablation.** The same synthesis pipeline run with GPT-5.2 and with Gemma 4. $\Delta Q = Q_{\mathrm{GPT}} - Q_{\mathrm{Gem}}$, so positive values favor GPT-5.2; $S = T_{\mathrm{Gem}}/T_{\mathrm{GPT}}$, so $S > 1$ means the GPT-generated solver is faster. The final row averages over all 21 targets.

	Quality Q			**Runtime** T (ms)		
Family	GPT-5.2	Gemma 4	ΔQ	GPT-5.2	Gemma 4	S
Coloring	0.868	0.798	+0.070	2.7	15.5	5.73×
MAXSAT	1.000	0.910	+0.090	17.1	38.2	2.23×
MIS	0.992	0.982	+0.010	18.8	25.8	1.37×
MDS	0.973	0.938	+0.035	13.3	10.2	0.77×
Packing LP	0.994	0.890	+0.104	3.3	12.5	3.80×
MDKP	0.973	0.740	+0.233	94.0	9.3	0.099×
TSP	0.993	0.926	+0.067	14.6	28.0	1.92×
All (21)	**0.971**	**0.883**	**+0.088**	**12.7**	**17.5**	**1.38×**

Table 12: **Ablation of explicit hint recovery.** One target per problem family. The no-hint variant removes the explicit hypothesis stage but retains a generic analysis program; model, search loop, candidate budget, validation feedback, refinement, repair, and test data are otherwise matched. $\Delta Q = Q_{\mathrm{full}} - Q_{\mathrm{no\text{-}hint}}$; runtime is normalized by the full pipeline.

Problem	**Target distribution**	**Full**	**No hint**	$\boldsymbol{\Delta Q}$
Coloring	Ring-template	1.0000	0.7973	+0.2027
MDKP	Latent-class	0.9684	0.9026	+0.0658
TSP	Clustered Euclidean	0.9872	0.9806	+0.0066
MIS	Clique-path	0.9914	0.9865	+0.0049
MDS	Geometric-anchor	1.0000	0.9974	+0.0026
MAXSAT	Latent-backdoor	0.9991	0.9991	0.0000
Packing LP	Block-coupled	1.0000	1.0000	0.0000
Mean quality		**0.9923**	**0.9519**	**+0.0404**
Runtime / full (geom. mean)		1.00×	**0.52×**	—

Table 12 shows a quality–runtime tradeoff. Explicit hint recovery improves mean quality from 0.9519 to 0.9923, with the largest gains on Ring-template Coloring and Latent-class MDKP, but the no-hint solvers are about $1/0.52 \approx 1.9\times$ faster in geometric-mean deployment runtime. On MAXSAT and Packing LP, both pipelines already reach essentially optimal quality. The hint stage therefore buys substantial quality on the targets where explicit structure recovery matters, rather than providing a uniform runtime improvement.

E.4 Sample-Conditioned Catalog Selection

A natural alternative to synthesizing a new solver is to use the public sample to choose among existing ones. For each target, we therefore select from the problem-specific entries in our 78-entry cross-family catalog using validation performance. One selector maximizes validation quality; a second minimizes validation runtime among solvers whose quality matches or exceeds that of our synthesized solver.

Table 13 shows that selecting from the existing solver catalog does not match the quality–runtime tradeoff of synthesis. The maximum-quality selector raises mean quality from 0.9705 to 0.9805, but is 42.6× slower. On the 19 covered targets, the fastest selector whose validation quality matches ours reaches mean test quality 0.9911 but remains 27.9× slower; no qualifying solver exists on the other two targets. Even the per-instance oracle virtual best reaches 0.9999 quality at 21.4× our runtime. Thus the gain is not explained simply by using samples to choose among existing algorithms.

The same distinction is sharper on PACE 2025 Dominating Set. Table 14 compares our synthesized solver with the best valid choices from the existing catalog on the 100 private instances.

Table 13: **Sample-conditioned selection from the existing solver catalog.** Selectors use validation data and are evaluated on held-out test instances. Runtime is reported relative to our synthesized solver. Coverage is the number of targets for which the selection criterion can be satisfied; for incomplete coverage, mean quality and runtime are computed over covered targets only.

Method	Mean quality	Runtime / ours	Coverage
Ours	0.9705	1.0×	21/21
Maximum-quality selector	0.9805	42.6×	21/21
Fastest selector at our quality	0.9911	27.9×	19/21
Per-instance oracle virtual best	0.9999	21.4×	21/21

Table 14: **PACE 2025 Dominating Set: synthesis versus catalog selection.** Lower set size and runtime are better. Exact catalog methods and Gurobi return no valid solution within the evaluation budget.

Method	Valid	Avg. set size	Runtime
Ours	100/100	231,595	2.89 s
Best catalog heuristic	100/100	234,674	8.3 s
Oracle best-valid catalog pick	100/100	234,673	8.3 s
Catalog exact / Gurobi	0/100	—	timeout

On PACE, the best valid catalog heuristic is both slower and about 1.3% worse in set size, while exact catalog methods and Gurobi return no valid solution within the budget. This provides an external-scale example where selecting an existing solver does not reproduce the synthesized solver's operating point.

E.5 Full Per-Target Results

Table 15 reports the full held-out test results behind the headline summary in Table 1, giving per-target normalized quality, optimality rate, and mean runtime for every baseline group. Two points are worth noting when reading it against the headline table. First, the heuristic column here is the *quality-best* heuristic per target, whereas the headline runtime ratio R_{Heur} uses the *fast high-quality* heuristic (the fastest within 95% of best quality), so the two heuristic runtimes are not directly comparable. Second, the ML baseline is fast and high-quality on several graph families but degrades on Packing LP and TSP, consistent with its aggregate position in Table 1. The aggregate quality lifts and runtime ratios in the headline table are computed directly from these 21 target-level values.

E.6 PACE 2025 Dominating Set Comparison

We evaluate on the PACE 2025 Dominating Set heuristic track, where the goal is to return a small dominating set of an undirected graph. PACE released 100 public instances and, after the competition, 100 private instances from a similar distribution. We split the public set $50/50$ for training and validation and evaluate only on the released private set. These results are local comparisons with released PACE instances and solvers, not official competition scores.

The private graphs are large and sparse, ranging from tens of thousands to more than four million vertices, with median size about 6.5×10^5 vertices and 9.4×10^5 edges. Table 16 gives the full size distribution. The learned ML baselines are not included here because the largest graphs exceed the scale handled by those models.

For each private instance, we run our synthesized solver and compare it with the released PACE heuristic solvers: Fontan–Verger, Root, Shadoks, AEG Heidelberg, Greeduce, and Swats. We verify every returned solution and report locally measured runtime. Because Dominating Set is a minimization problem, smaller returned sets indicate better quality.

The results show a clear speed–quality tradeoff. The released PACE solvers generally return smaller dominating sets, with our solutions about 3.3% larger than the strongest fully valid submissions. In

Table 15: **Held-out results on all 21 target distributions.** Q is normalized quality, Opt. is optimality rate, and T is mean per-instance runtime in milliseconds.

	Best heuristic			ML			Gurobi (10 s)			Exact (10 s)			Ours		
Target distribution	Q	Opt.	T	Q	Opt.	T	Q	Opt.	T	Q	Opt.	T	Q	Opt.	T
Coloring															
Ring-template	.746	.002	3699	.992	.962	2.2	.976	.882	4136	1.00	1.00	31	1.00	1.00	0.2
Overlapping-palette	.747	.004	3476	.999	.997	1.9	.878	.392	8366	1.00	1.00	80	.804	.022	24
Separator-trap	.749	.000	3616	.995	.974	2.1	.933	.666	6877	1.00	1.00	99	.800	.000	4
MAXSAT															
Community-parity	.914	.000	3733	1.00	.986	21	1.00	.992	8712	1.00	1.00	26	1.00	.876	24
Last-clause signal	.932	.000	3645	1.00	1.00	14	1.00	1.00	8325	1.00	1.00	8	1.00	1.00	6
Latent-backdoor	.931	.000	3791	1.00	.999	20	1.00	.996	2461	1.00	1.00	27	1.00	.896	38
MIS															
Clique-path	.876	.040	10000	.996	.965	1.6	1.00	1.00	2405	1.00	1.00	1355	.991	.594	15
Core-fringe trap	.908	.908	10000	1.00	1.00	1.6	1.00	1.00	1555	1.00	1.00	145	1.00	1.00	8
Motif-bridge	.874	.044	10000	.988	.878	1.6	.998	.890	6757	.994	.994	2136	.986	.384	57
MDS															
Gateway-hub	.839	.000	10000	.935	.702	6.6	1.00	1.00	8099	1.00	1.00	113	.920	.000	5
Geometric-anchor	.784	.298	8706	.945	.790	4.5	1.00	1.00	3083	1.00	1.00	2093	1.00	1.00	130
Star-kernel	.930	.930	10000	.977	.959	5.8	1.00	1.00	8162	1.00	1.00	52	1.00	1.00	3
Packing LP															
Block-coupled	.792	.000	10000	.796	.000	1.0	1.00	1.00	8628	1.00	1.00	125	1.00	1.00	1.5
Active-resource	.610	.000	10000	.724	.000	1.0	1.00	1.00	12676	1.00	1.00	124	.981	.000	17
Single-bottleneck	.803	.000	10000	.903	.000	1.0	1.00	1.00	7636	1.00	1.00	122	1.00	1.00	1.4
MDKP															
Decoy-complement	.967	.016	5330	.982	.319	6.5	1.00	1.00	2499	1.00	1.00	1084	.951	.000	41
Latent-class	.966	.010	3660	.981	.291	6.5	1.00	.746	4955	1.00	.544	6306	.968	.000	116
Single-resource	.960	.666	4263	.958	.019	4.2	1.00	1.00	2593	1.00	1.00	233	1.00	1.00	176
TSP															
Clustered Euclidean	1.00	.910	1544	.782	.000	888	1.00	.930	1553	1.00	.662	136	.987	.002	67
Latent-metric	1.00	.848	1490	.970	.282	891	1.00	.966	1566	1.00	.758	118	.992	.306	235
Paired-ribbon zigzag	1.00	1.00	51	.976	.700	888	1.00	1.00	2071	.499	.000	10993	1.00	1.00	0.2

Table 16: **PACE private-test graph statistics,** over the 100 released private Dominating Set heuristic instances. Quantiles are computed separately for each row, so entries in a column need not come from the same instance; in particular, the average-degree row is not the ratio of the two rows above it.

Quantity	**Min.**	**25th pct.**	**Median**	**75th pct.**	**Max.**
Vertices $\lvert V\rvert$	18,046	21,400	653,867	1,261,963	4,221,675
Edges $\lvert E\rvert$	54,138	382,083	939,263	1,831,990	10,356,300
Average degree $2\lvert E\rvert/\lvert V\rvert$	2.28	2.57	3.74	13.00	48.00

return, our solver is roughly $100\times$ faster and returns valid solutions on all 100 instances. Greeduce outperforms it on $91/100$ instances, while Swats is valid on only $75/100$. Thus, on this benchmark, synthesis produces a fully valid distribution-specific heuristic that gives up a few percent in solution quality for roughly two orders of magnitude in runtime.

Matched-budget anytime comparison. Because the released PACE solvers are anytime methods, we also run Root, Greeduce, and Fontan–Verger under nominal 3, 10, and 30 second budgets. Table 18 reports the mean per-instance normalized quality score over all 100 private instances, assigning zero quality to invalid outputs; our solver is shown at its natural 2.90s operating point.

Table 18 shows that none of the three released solvers matches our all-instance quality at either the 3s or 10s budget. At 30s, all three surpass it. The synthesized solver therefore provides a strong low-latency operating point, while the competition solvers benefit from additional search time.

Table 17: **PACE 2025 Dominating Set results.** Lower size and runtime are better. Our solver is valid on all 100 private instances and is roughly 75–125× faster than the released PACE solvers, whose dominating sets are about 3% smaller. Size ratio is our size divided by the comparator size. Swats statistics are computed over the 75 instances on which it returns a valid solution.

Solver	Valid	Total size ↓	Avg. size ↓	Size ratio	Time (s) ↓	Speedup	Wins vs. ours
Ours	100/100	23.16M	231,595	1.000	2.89	1.0×	–
AEG Heidelberg (Chhabra et al., 2025)	100/100	22.41M	224,086	1.034	350.14	121.0×	99/100
Fontan–Verger (Fontan & Verger, 2025)	100/100	22.41M	224,107	1.033	286.24	98.9×	100/100
Root (Root team, 2025)	100/100	22.41M	224,108	1.033	360.42	124.5×	100/100
Shadoks (da Fonseca et al., 2025)	100/100	22.43M	224,306	1.032	316.07	109.2×	100/100
Greeduce (Polak & Schmidt, 2025)	100/100	22.47M	224,699	1.031	300.86	104.0×	91/100
Swats (Swat, 2025)	75/100	15.77M	210,237	1.028	218.11	75.4×	75/75

Table 18: **PACE 2025 Dominating Set under matched solver budgets.** Entries report the mean per-instance normalized quality score over all 100 private instances; larger is better and invalid outputs receive zero. Our solver has $Q = 0.2285$ at 2.90s.

Solver	3 s	10 s	30 s
Root	0.1898	0.2244	0.2373
Greeduce	0.0805	0.2134	0.2326
Fontan–Verger	0.1847	0.2133	0.2348
Ours	0.2285 at 2.90 s		

E.7 Matched-Budget MIS Comparison

We also compare against the OnlineMIS and ReduMIS heuristic modes of KaMIS on the three MIS targets. Table 19 reports representative operating points using full wall-clock time.

The comparison is mixed. On Core-fringe, our solver reaches optimal quality in 7.2 ms, about 5× faster than either KaMIS mode. OnlineMIS is slightly stronger at the same latency on Clique-path and is both faster and higher-quality on Motif-bridge; ReduMIS reaches near-optimal or optimal quality on all three targets but uses substantially more time on Clique-path and Motif-bridge.

E.8 Distributional Algorithmic Complexity

This appendix expands the computation patterns summarized in Table 20. The comparison is distributional rather than worst-case: the synthesized programs do not improve the worst-case complexity of the underlying problem classes. Instead, they exploit regularities shared by instances from a fixed distribution to replace generic search with smaller, distribution-specific computations.

A generic exact method must remain valid on arbitrary instances and therefore searches the full space of colorings, Boolean assignments, vertex subsets, item subsets, or tours. Packing LP is different in that the ambient problem is polynomial-time solvable, but a generic LP solver still handles the full N-variable, r-constraint program. After observing samples, the synthesized solvers can instead exploit recurring structure through templates, restricted candidate sets, local search, surrogate scores, or a small collection of structured candidates. The speedups in Table 20 come from these changes in deployed computation.

Notation. For graph problems, n and m denote the numbers of vertices and edges, $\Delta_{\max}$ the maximum degree, and κ the number of colors considered by the generic Coloring search. For MaxSAT, v is the number of variables, $|F|$ the number of clauses, ℓ the maximum clause width, and Δ_{occ} the maximum variable-occurrence count. Packing LP and MDKP use N items, r resource constraints, and input bit-length L; TSP uses n cities.

The remaining symbols denote fixed budgets used by the synthesized programs. They do not grow with the input size and typically bound restarts, repair steps, local-search passes, or candidate sets:

Table 19: **Matched-budget comparison with KaMIS heuristic modes.** Each entry reports normalized quality and wall-clock runtime.

Target	Ours	OnlineMIS	ReduMIS
Clique-path	0.9914 @ 15.3 ms	0.9952 @ 15.1 ms	0.9999 @ 551 ms
Core-fringe	1.0000 @ 7.2 ms	1.0000 @ 37 ms	1.0000 @ 37 ms
Motif-bridge	0.9860 @ 57.3 ms	0.9941 @ 13.9 ms	1.0000 @ 1056 ms

Table 20: **Computation used by the synthesized solvers across the 21 target distributions.** For each problem class, we show the generic exact-search complexity and the distribution-specific computation used by the synthesized solver. A slash denotes a learned fast path followed by its fallback. Implementation budgets such as B, R, P, F, S, and K are fixed independently of input size. Corresponding exact-solver runtimes are reported in Table 15.

Distribution	Synthesized computation
Coloring *generic:* $O^*(\kappa^n)$	
Ring-template	$O(m)$ / $O(n^2 + m + B_{\mathrm{rep}}\Delta_{\max})$
Overlapping-palette	$O\big(n + m + R_{\mathrm{ds}}(n^2 + m) + P_{\mathrm{elim}}\kappa(n + m)\big)$
Separator-trap	$O\big(n + m + R_{\mathrm{ds}}(n^2 + m) + B_{\mathrm{recolor}}\big)$
MAXSAT *generic:* $O^*(2^v)$	
Community-parity	$O(\lvert F\rvert \ell + R_{\mathrm{ws}} B_{\mathrm{flip}} \ell \Delta_{\mathrm{occ}})$
Last-clause signal	$O(\lvert F\rvert \ell + R_{\mathrm{ws}} B_{\mathrm{flip}} \ell \Delta_{\mathrm{occ}} + B_{\mathrm{patch}} \ell \Delta_{\mathrm{occ}})$
Latent-backdoor	$O(\lvert F\rvert \ell + R_{\mathrm{ws}} B_{\mathrm{flip}} \ell \Delta_{\mathrm{occ}})$
MIS *generic:* $O^*(2^n)$	
Clique-path	$O(n + m + P_{\mathrm{ord}} n \log n + P_{\mathrm{gr}}(n + m) + T_{\mathrm{ARW}})$
Core-fringe trap	$O\Big(n + m + \sum_j 4^{t_j} + kq\Big)$
Motif-bridge	$O(n + m + P_{\mathrm{gr}}(n + m) + B_{\mathrm{loc}}(n + m) + B_{\mathrm{kick}}(n + m) + T_{\mathrm{tiny}})$
MDS *generic:* $O^*(2^n)$	
Gateway-hub	$O(n + m + T_{\text{heap-greedy}} + T_{\mathrm{prune}})$
Geometric-anchor	$O(n + m)$ / $O((1 + B_{\mathrm{geo}})(n + m))$
Star-kernel	$O\Big(n + m + \sum_j c_j 2^{t_j} + T_{\mathrm{greedy}} + T_{\mathrm{prune}}\Big)$
Packing LP *generic:* $T_{\mathrm{LP}}(N, r, L)$	
Block-coupled	$O(Nr + N \log N)$
Active-resource	$O(Nr + P_{\mathrm{LP}}(Nr + N \log N))$
Single-bottleneck	$O(Nr + N \log N)$
MDKP *generic:* $O^*(2^N)$	
Decoy-complement	$O\big(Nr + P_{\mathrm{score}}(Nr + N \log N) + T_{\mathrm{add/drop}} + T_{\mathrm{repair}}\big)$
Latent-class	$O(Nr + N \log N + P_{\mathrm{price}} T_{\mathrm{iter}}(Nr + N \log N + T_{\mathrm{repair}}))$
Single-resource	$O(Nr + N \log N + K_{\mathrm{cand}} C_b + T_{\mathrm{repair}})$
TSP *generic:* $O(n^2 2^n)$	
Clustered Euclidean	$O\big(n^2 \log n + B_{\mathrm{cand}} n^2 + B_{\mathrm{full}} n^2\big)$
Latent-metric	$O\big(n^2 + F(kn^2 + n \log n) + S(2^k k + kn) + P_{\mathrm{imp}} B_{\mathrm{imp}} n^2\big)$
Paired-ribbon zigzag	$O(n \log n)$ / $O\big((1 + B_{\mathrm{2opt}}) n^2\big)$

- *Coloring:* B_{rep} is the repair budget, R_{ds} the number of DSATUR-style restarts, P_{elim} the number of color-elimination passes, and B_{recolor} the recoloring budget.
- *MaxSAT:* R_{ws} is the number of WalkSAT-style restarts, B_{flip} the number of flips per restart, and B_{patch} the patch-search budget.

- *MIS:* P_{ord} and P_{gr} count ordering rules and greedy constructions, B_{loc} and B_{kick} bound local search and kick-repair, and T_{ARW} and T_{tiny} denote bounded auxiliary subroutines. For Core-fringe, t_j is the size of residual fringe component j, k the number of core choices, and q the number of candidate configurations scored.
- *MDS:* $T_{\text{heap-greedy}}$, T_{greedy}, and T_{prune} denote greedy and pruning subroutines, while B_{geo} bounds geometric-anchor completion. For Star-kernel, t_j is the number of residual targets in component j, and c_j the number of candidate covers.
- *Packing LP:* P_{LP} is the number of density or active-resource rules considered.
- *MDKP:* P_{score} is the number of surrogate scores, P_{price} the number of resource-price vectors, T_{iter} the cost of one price-update and repair iteration, K_{cand} the candidate-set size, C_b the effective one-dimensional bottleneck capacity, and $T_{\mathrm{add/drop}}$ and T_{repair} the bounded improvement and repair costs.
- *TSP:* B_{cand} and B_{full} bound candidate and full 2-opt search, F is the number of coordinate frames, k the number of stripes or groups, S the number of selected stripe frames, P_{imp} the number of tours improved, B_{imp} the number of improvement passes per tour, and $B_{2\mathrm{opt}}$ the paired-ribbon fallback budget.

Graph coloring. The generic search space has size $O^*(\kappa^n)$. The synthesized solvers instead rely on DSATUR-style construction and bounded repair, where $T_{\mathrm{DSATUR}}(n, m, \kappa) = O(n^2 + m + n\kappa) = O(n^2 + m)$ for $\kappa \le n$. For *Ring-template*, the main path checks a learned coloring pattern against the edge set in $O(m)$; if the pattern fails, the solver invokes bounded repair or the greedy fallback, for $O(n^2 + m + B_{\mathrm{rep}}\Delta_{\max})$. *Overlapping-palette* uses a bounded number of DSATUR attempts followed by color elimination,

$$O\big(n + m + R_{\mathrm{ds}}(n^2 + m) + P_{\mathrm{elim}}\kappa(n + m)\big)\,,$$

whereas *Separator-trap* uses capped recoloring, giving $O(n + m + R_{\mathrm{ds}}(n^2 + m) + B_{\mathrm{recolor}})$.

MaxSAT. The ambient search ranges over 2^v assignments. Here the synthesized solvers use statistics from the training instances to choose an initial assignment and then make a bounded number of local repairs. Building the occurrence lists and initial assignment costs $O(|F|\ell)$, and each flip costs $O(\ell\Delta_{\mathrm{occ}})$. Thus *Community-parity* and *Latent-backdoor* require $O(|F|\ell + R_{\mathrm{ws}}B_{\mathrm{flip}}\ell\Delta_{\mathrm{occ}})$, while *Last-clause signal* adds $O(B_{\mathrm{patch}}\ell\Delta_{\mathrm{occ}})$ to repair variables near the remaining unsatisfied clauses. The useful structure here is therefore not a smaller worst-case search space, but an initialization for which bounded local search is usually enough.

Maximum independent set. Rather than search all 2^n vertex subsets, the synthesized solvers construct and refine a small collection of candidate independent sets. *Clique-path* combines learned vertex orderings, greedy maximal sets, and ARW-style improvement, for $O(n + m + P_{\mathrm{ord}}n \log n + P_{\mathrm{gr}}(n+m) + T_{\mathrm{ARW}})$. *Core-fringe trap* decomposes the fringe into small components and enumerates only within those components:

$$O\Big(n + m + \textstyle\sum_j 4^{t_j} + kq\Big)\,.$$

The exponential term therefore depends on the residual component sizes t_j, not on the full graph size n. *Motif-bridge* instead uses greedy constructions, bounded local improvement, kick moves, and an optional small exact step, with $O(n + m + P_{\mathrm{gr}}(n + m) + B_{\mathrm{loc}}(n + m) + B_{\mathrm{kick}}(n + m) + T_{\mathrm{tiny}})$.

Minimum dominating set. The generic search considers 2^n vertex subsets. *Gateway-hub* exploits high-degree hubs, leaves, and isolated vertices, followed by greedy completion and pruning, for $O(n + m + T_{\text{heap-greedy}} + T_{\mathrm{prune}})$. *Geometric-anchor* checks a small anchor pattern in $O(n + m)$ and uses bounded greedy completion when needed, giving $O((1 + B_{\mathrm{geo}})(n + m))$. *Star-kernel* first applies forced leaf-neighbor choices and then solves only the remaining small components. Its residual work is $\sum_j c_j 2^{t_j}$, so the exponential part is confined to those components.

Packing LP. Packing LP is different because the underlying problem is already polynomial-time solvable. The comparison is between a generic LP solve, $T_{\mathrm{LP}}(N, r, L)$, and a simpler rule adapted to the observed resource structure. For *Block-coupled* and *Single-bottleneck*, the solver scores items against the effective resource and sorts them in $O(Nr + N \log N)$. *Active-resource* evaluates P_{LP} such scoring rules, for $O(Nr + P_{\mathrm{LP}}(Nr + N \log N))$. In these distributions, a small set of density rules is sufficient to avoid a full general-purpose LP solve.

Multidimensional knapsack. The ambient search has size 2^N. The synthesized solvers instead use surrogate scores, restricted candidate sets, bounded dynamic programming, and repair. *Decoy-complement* scores and greedily packs items before bounded add/drop improvement, for $O(Nr + P_{\text{score}}(Nr + N\log N) + T_{\text{add/drop}} + T_{\text{repair}})$. *Latent-class* iterates over a small set of resource-price vectors,

$$O(Nr + N\log N + P_{\text{price}}T_{\text{iter}}(Nr + N\log N + T_{\text{repair}}))\,,$$

while *Single-resource* identifies a dominant constraint and runs one-dimensional DP on a bounded candidate set, with $O(Nr + N\log N + K_{\text{cand}}C_b + T_{\text{repair}})$.

Traveling salesman problem. Held–Karp takes $O(n^2 2^n)$ time. The synthesized solvers instead exploit the recurring geometry of each distribution to construct a small set of tours and improve them locally. *Clustered Euclidean* builds geometric or nearest-neighbor tours and applies bounded candidate and full 2-opt, for $O(n^2\log n + B_{\text{cand}}n^2 + B_{\text{full}}n^2)$. *Latent-metric* evaluates F coordinate frames with k groups, solves a small stripe-order problem for the selected S frames, and improves a bounded number of tours:

$$O\big(n^2 + F(kn^2 + n\log n) + S(2^k k + kn) + P_{\text{imp}}B_{\text{imp}}n^2\big)\,.$$

Since k is fixed and small, the 2^k term is independent of the asymptotic growth in n. Finally, *Paired-ribbon zigzag* detects the two-rail structure, sorts points along the rails, and pairs endpoints in $O(n\log n)$. When that structure is not detected, it falls back to nearest-neighbor construction followed by bounded 2-opt, for $O((1 + B_{\text{2opt}})n^2)$.

Summary. The same pattern appears across the seven problem classes: samples reveal structure that lets the deployed solver work with templates, good initializations, small residual components, dominant resources, restricted candidate sets, or geometric orderings instead of carrying out the full generic computation. These procedures do not improve the worst-case complexity of the underlying problems, and several retain bounded repair or fallback paths. Their advantage is specific to the distributions on which they were synthesized.

E.9 Zero-Sample versus Sample-Conditioned Synthesis

We isolate the contribution of public samples with a paired ablation. The zero-sample variant runs the same synthesis pipeline with no public training instances; the sample-conditioned variant uses the default $m = 64$ training instances from our main experiments (we fix this rather than tuning the sample size). Each row averages 10 paired synthesis seeds, so the two variants are compared under matched randomness.

Table 21 shows quality is near-saturated under both variants, so the observable effect of sample access on these distributions is not feasibility or final quality but *deployment runtime*. The sample-conditioned solver is at least as fast on $10/12$ targets and strictly faster on $9/12$, with large gains on Single-resource MDKP ($36\times$), Paired-ribbon zigzag TSP ($18\times$), and Separator-trap Coloring ($6\times$). The one substantial regression is Decoy-complement MDKP, where the sample-conditioned solver is slower ($0.36\times$) despite a marginal quality gain. We read this as evidence for sample-conditioned compilation: public samples often help synthesis discover faster deployment code even when zero-sample synthesis already attains high quality—though the effect is distribution-dependent rather than a monotone function of sample count.

E.10 Synthesis Variance

We repeated GPT-5.2 synthesis three times on seven representative targets. Across-run quality is relatively stable: four of the seven targets have quality standard deviation at most 0.005, and the mean standard deviation is 0.018. Runtime varies more substantially, reflecting both different synthesized implementations and, in these sweep measurements, possible machine contention. With only three runs per target, we treat this as an aggregate stability diagnostic rather than a precise estimate of per-target variance: solution quality appears more stable than the exact deployment implementation and runtime.

Table 21: **Zero-sample versus sample-conditioned synthesis.** The same pipeline run with no training instances ($m = 0$) and with 64 ($m = 64$) on 12 targets from four problem families, averaged over 10 paired synthesis seeds. Quality is near-identical in the two settings, so we report $\Delta Q = 10^3(Q_{64} - Q_0)$; speedup $T_0/T_{64} > 1$ favors sample conditioning.

Target distribution	$\Delta Q \times 10^3$	Runtime (ms)		Speedup
		m=0	m=64	T_0/T_{64}
Coloring				
Ring-template	0.00	0.101	0.072	1.40×
Overlapping-palette	−0.08	0.076	0.077	0.99×
Separator-trap	0.00	0.603	0.097	**6.23**×
MDS				
Gateway-hub	0.00	0.103	0.102	1.00×
Geometric-anchor	−0.33	0.130	0.104	1.26×
Star-kernel	0.00	0.086	0.080	1.08×
MDKP				
Decoy-complement	+0.57	90.79	251.35	0.36×
Latent-class	−0.07	116.04	97.13	1.19×
Single-resource	0.00	7.92	0.22	**35.95**×
TSP				
Clustered Euclidean	0.00	6.08	1.60	3.80×
Latent-metric	0.00	6.11	4.31	1.42×
Paired-ribbon zigzag	0.00	0.65	0.04	**17.53**×

Table 22: **Execution traces of the synthesized solvers on held-out instances.** Rates and counts are averaged within each target and then across the three targets in each problem family. The two rates need not sum to 100%: remaining instances take the solver's own non-shortcut path rather than a generic fallback. Residual size is comparable only within a family, so it is not aggregated.

	Control flow (% of instances)		**Residual work**	
Problem family	Fast path	Fallback	Residual size	Repair iters.
Coloring	67.4	0.0	0.00	0.00
MAXSAT	88.1	7.4	26.24	0.00
MIS	66.7	0.0	235.92	0.33
MDS	86.4	13.6	441.72	0.00
Packing LP	100.0	0.0	346.78	0.00
MDKP	100.0	0.0	321.03	2.67
TSP	100.0	0.0	906.74	0.00
All 21	**86.9**	**3.0**	—	**0.43**

E.11 Diagnostic Traces for Synthesized Solvers

We record four traces during held-out evaluation to characterize how the synthesized solvers behave at deployment. *Shortcut* indicates that the solver uses its distribution-specific fast path, such as template checking in Coloring, a seeded assignment in MAXSAT, density sorting in Packing LP, restricted scoring or DP in MDKP, or structured tour construction in TSP. *Fallback* indicates that the solver instead invokes its generic safety routine. *Residual size* measures the subproblem remaining after specialization; because its meaning differs across problem classes, it should only be compared within a class. *Repair iterations* counts the bounded local improvements made after the initial construction.

These traces describe the execution of the deployed solvers rather than causal ablations: we do not disable individual mechanisms and rerun the evaluation. We average each trace within a target and then across the three targets in each problem class.

The synthesized solvers use their distribution-specific fast paths on 86.9% of test instances on average, including 100% for Packing LP, MDKP, and TSP. This is consistent with the computation patterns in Table 20: density rules in Packing LP, surrogate scoring or restricted DP in MDKP, and structured tour construction in TSP are used directly on most test instances.

Fallback is uncommon, occurring on 3.0% of instances overall. It is most frequent for MDS (13.6%) and MAXSAT (7.4%), where the synthesized strategies combine learned structure with a generic fallback when that structure is insufficient. Repair is also limited, averaging 0.43 iterations, although MDKP uses more repair (2.67 iterations) as part of its packing procedure. Residual size provides an

Table 23: **Aggregate graph-relabeling perturbation ablation.** Means over the three target distributions in each graph problem, and over all nine targets in the final row. The columns "Qual. changed," "Obj. changed," and "Feas. changed" report the fraction of instances whose quality, objective value, or feasibility status changes under relabeling, respectively. The runtime ratio is the geometric mean of $t_{\mathrm{pert}}/t_{\mathrm{orig}}$. Feasibility is invariant throughout.

		Quality			**Change rate**			**Runtime**
Problem	**Targets**	Q_{orig}	Q_{pert}	ΔQ	Qual. changed	Obj. changed	Feas. changed	Ratio
Coloring	3	0.868	0.791	−0.077	0.342	0.342	0.000	1.38×
MDS	3	0.973	0.819	−0.155	0.419	0.333	0.000	1.39×
MIS	3	0.992	0.992	−0.001	0.258	0.191	0.000	1.31×
All	9	**0.945**	**0.867**	**−0.077**	**0.340**	**0.289**	**0.000**	**1.36×**

additional within-family measure of how much of the original problem remains after specialization; because its meaning differs across problem classes, the values should not be compared across families.

Overall, the traces show that the synthesized solvers usually execute the specialized computation identified during synthesis rather than relying on their fallback routines. Fallback and repair remain available for instances on which the learned structure does not fully determine a solution.

E.12 Perturbation Robustness Ablation

To test sensitivity to graph presentation, we randomly permute the vertex labels of each held-out instance. This leaves the graph unchanged up to isomorphism but changes the vertex identifiers and input order. Solvers that rely on graph structure should therefore be largely unaffected, whereas solvers that exploit labeling or ordering may change their behavior.

We evaluate each selected solver on both the original and relabeled instances. We report Q_{orig}, Q_{pert}, their difference $\Delta Q = Q_{\mathrm{pert}} - Q_{\mathrm{orig}}$, the fraction of instances for which quality, objective value, or feasibility changes, and the runtime ratio $t_{\mathrm{pert}}/t_{\mathrm{orig}}$. Runtime ratios are aggregated by geometric mean. Relabeling is used only for this diagnostic and plays no role in solver selection.

Feasibility is unchanged by relabeling on all nine targets. Mean quality drops from 0.945 to 0.867, but almost all of the loss comes from two targets: Ring-template in Coloring and Geometric-anchor in MDS. The other seven show little change in mean quality, suggesting that most synthesized solvers rely primarily on structure that survives relabeling, while a few also exploit presentation-specific cues.

Runtime is less stable. The geometric-mean slowdown is $1.36\times$, with Ring-template slowing by $9.65\times$ and Core-fringe trap by $2.19\times$. In the latter case, quality is unchanged, showing that relabeling can alter the solver's execution path even when the final solution is unaffected.

Overall, relabeling leaves feasibility unchanged and has little effect on quality for most targets. The two exceptions, Ring-template and Geometric-anchor, show that some synthesized solvers also exploit vertex labeling or input order rather than relying entirely on isomorphism-invariant graph structure.

Table 24: **Graph-relabeling robustness by target.** Each instance is evaluated before and after a random permutation of its vertex labels. Runtime is reported as $t_{\mathrm{pert}}/t_{\mathrm{orig}}$; bold marks the two largest slowdowns.

		Quality			Change rate			Runtime
Problem	**Target distribution**	Q_{orig}	Q_{pert}	ΔQ	**Qual. changed**	**Obj. changed**	**Feas. changed**	**Ratio**
Coloring	Ring-template	1.000	0.772	−0.228	1.000	1.000	0.000	**9.65**×
	Overlapping-palette	0.804	0.801	−0.004	0.026	0.026	0.000	0.28×
	Separator-trap	0.800	0.800	0.000	0.000	0.000	0.000	0.98×
MDS	Gateway-hub	0.920	0.920	0.000	0.256	0.000	0.000	1.10×
	Geometric-anchor	1.000	0.536	−0.464	1.000	1.000	0.000	2.18×
	Star-kernel	1.000	1.000	0.000	0.000	0.000	0.000	1.11×
MIS	Clique-path	0.991	0.991	−0.001	0.386	0.350	0.000	1.01×
	Core-fringe trap	1.000	1.000	0.000	0.000	0.000	0.000	**2.19**×
	Motif-bridge	0.986	0.985	−0.001	0.388	0.224	0.000	1.02×