AI Agents for Distribution-Specific Algorithm Discovery

Anonymous Authors[1]

Abstract

Can an AI agent discover reusable computational structure from examples and turn it into executable code? We study this question through *distribution-aware program learning*, where the learned object is not a predictor but a solver evaluated on fresh instances by both solution quality and runtime. Our central abstraction is a *solver hint*: reusable structure inferred from samples and compiled into specialized solver code.

We give theory showing that samples can improve computation when they identify a shortcut shared by future instances. In fixed solver libraries, the empirically fastest sample-consistent solver generalizes in correctness and runtime; for structured hint classes, identifiable hints can be recovered from polynomially many samples and compiled into faster solvers. Empirically, we instantiate this view with an LLM code agent that proposes hypotheses, writes analysis programs, extracts reusable summaries, and compiles deployment solvers. Across 21 structured combinatorial-optimization target distributions, the synthesized solvers reach mean normalized quality 0.971, improve by $+0.224$ over the average heuristic pool and by $+0.098$ over the highest-quality heuristic, and are $336.9\times$, $342.8\times$, and $16.1\times$ faster than the quality-best heuristic, Gurobi, and the selected time-limited exact backend, respectively. On released PACE 2025 Dominating Set private instances, the synthesized solver is valid on all 100 graphs and runs about two orders of magnitude faster than top competition solvers, with a moderate quality gap. These results provide a concrete testbed for AI-driven discovery, where success is measured by whether inferred structure becomes useful, verifiable computation.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

1. Introduction

Scientific discovery often means finding reusable structure. Scientists do not merely explain one observation; they identify mechanisms, decompositions, laws, or shortcuts that apply to future cases. We study an algorithmic version of this process: can examples from an unknown structured distribution enable an AI agent to infer a reusable computational principle and turn it into solver code?

This gives a controlled setting for studying AI-driven discovery. The agent must do more than generate a plausible explanation or solve isolated instances. It must propose a structural hypothesis, test it on samples, turn the resulting summary into an executable artifact, and improve future computation. The final artifact is therefore a measurable form of discovery: a solver whose quality and runtime can be evaluated on held-out instances.

We formalize this setting as *distribution-aware program learning*. The learner receives samples from an unknown deployment distribution and returns solver code for future instances from the same distribution. Unlike standard prediction problems, success is not measured only by correctness or quality. When the learned object is an executable procedure, computation itself must generalize. Two solvers may both return valid solutions on the deployment distribution while differing substantially in runtime; in that case, they have not learned equally useful procedures.

Our central abstraction is a *solver hint*: reusable structure inferred from samples and compiled into specialized solver code. A hint may encode a SAT backdoor, a graph decomposition, an active-resource pattern in a packing problem, a local repair rule, or geometric structure in a routing problem. The sample-to-solver map factors as $S \longmapsto \widehat{h}_S \longmapsto \widehat{c}_S = \mathrm{Comp}(\widehat{h}_S)$. This factorization is also how we interpret LLM synthesis. The agent is not best viewed as writing a solver in one shot. Rather, it proposes a distributional hypothesis, writes an analysis program to estimate a reusable summary from the sample, and writes a deployment solver conditioned on that summary.

Solver hints are computational shortcuts. In SAT, for example, the hint might be a reusable backdoor set. Correctness need not be learned: a complete solver can always be used as fallback. What the sample learns is which shortcut to

compile so that future instances are solved faster. This separation between correctness and learned computation is central to both our theory and our experiments.

The framing connects to a long-standing question in complexity theory: what does it mean for a problem to be hard *on average*? Worst-case complexity has a clean primary object, the language, but average-case complexity (Levin, 1986; Impagliazzo, 1995; Bogdanov & Trevisan, 2006) also requires a distribution. In practice, deployment distributions often lack closed-form descriptions amenable to classical analysis. Smoothed analysis (Spielman & Teng, 2004), parameterized complexity (Downey & Fellows, 2013), and structural backdoors (Williams et al., 2003) each give hand-designed routes to distribution-specific tractability. We propose a complementary sample-driven route: treat the distribution as accessible only through examples, use an AI agent to discover reusable structure, and evaluate the learned solver as a constructive witness of distribution-specific tractability.

Contributions. This paper makes four contributions.

1. **A setting for AI-driven algorithmic discovery.** We introduce *distribution-aware program learning*, where an AI agent receives examples from an unknown structured distribution and must produce executable solver code for future instances. The setting isolates a core capability of AI scientists: converting repeated observations into a reusable, verifiable computational artifact.

2. **Solver hints as reusable discoveries.** We formalize the intermediate object of discovery as a *solver hint*: distribution-specific structure inferred from samples and compiled into specialized solver code. This separates learning correctness from learning computation. Correctness can be preserved by fallback, while samples identify which shortcut, decomposition, or low-dimensional structure should be used.

3. **Theory for when samples improve computation.** We prove that, for fixed solver libraries, the empirically fastest sample-consistent solver generalizes in both correctness and runtime. For structured hint classes, identifiable hints can be recovered from polynomially many samples and compiled into specialized solvers. A hidden SAT-backdoor model makes the mechanism concrete: recovering the backdoor preserves correctness through fallback while yielding exponential per-instance speedup on the deployment distribution.

4. **An LLM-agent instantiation and empirical evidence.** We instantiate the sample-to-hint-to-solver view with an LLM synthesis loop. Each candidate consists of a structural hypothesis, an analysis program, and a deployment solver. Across 21 structured combinatorial-optimization target distributions, the synthesized solvers reach mean normalized quality 0.971, improve by $+0.224$ over the average heuristic pool and by $+0.098$ over the highest-quality heuristic, and are $336.9\times$, $342.8\times$, and $16.1\times$ faster than the quality-best heuristic, Gurobi, and the selected time-limited exact backend, respectively. Inspection shows that many gains come from changing the computational scale: generated code replaces ambient search or full optimization with distribution-specific sorting, scoring, repair, kernelization, or structured construction.

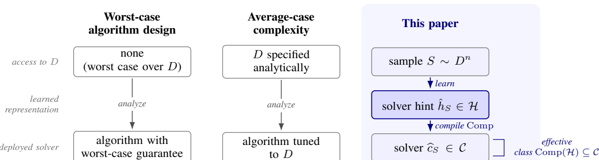

Figure 1. **Three access models for designing solvers against a distribution D.** Worst-case design assumes no distributional information, while average-case complexity assumes an analytic specification of D. We study the intermediate sample-access regime: from $S \sim D^n$, the learner infers a solver hint $\hat{h}_S \in \mathcal{H}$ and compiles it into a deployed solver $\widehat{c}_S = \mathrm{Comp}(\hat{h}_S)$. Thus the effective search space is the structured subfamily $\mathrm{Comp}(\mathcal{H}) \subseteq \mathcal{C}$. Our agents approximate this sample-to-hint-to-solver map.

2. Related Work

Our work connects three themes: AI systems that discover reusable scientific or algorithmic artifacts, classical views of distribution-specific tractability, and program-learning systems whose outputs are executable code. The common thread is that computation can improve when structure is discovered and reused. We study this in a sample-access setting: examples from an unknown deployment distribution are used to infer a reusable solver hint, which is then compiled into solver code and evaluated on fresh instances by both quality and runtime.

AI scientists and autonomous discovery agents. Recent work on AI scientists studies systems that can propose hypotheses, design experiments, execute code, analyze results, and produce scientific artifacts (Lu et al., 2026; Yamada et al., 2025). Our setting gives a concrete algorithmic version of this agenda. The agent proposes a structural hypothesis, writes code to measure it, compiles the resulting summary into a solver, and is evaluated by held-out quality and runtime. Thus, discovery is not assessed only by the plausibility of an explanation, but by whether the inferred structure becomes useful executable computation.

AI-driven algorithm discovery. A growing line of work uses learning or search to discover algorithms and programs. AlphaTensor discovers efficient matrix-multiplication algorithms (Fawzi et al., 2022), AlphaDev discovers low-level

sorting routines (Mankowitz et al., 2023), FunSearch combines language models with systematic evaluation to search over programs for mathematical and algorithmic problems (Romera-Paredes et al., 2024), and AlphaEvolve uses LLM-driven evolutionary code search for scientific and algorithmic discovery (Novikov et al., 2025). These works show that AI systems can discover useful computational artifacts when candidate programs can be evaluated. Our focus is distribution-specific solver discovery from samples: the goal is to infer reusable structure of a deployment distribution and compile it into a solver whose quality and runtime generalize to fresh instances.

Average-case and beyond-worst-case analysis. Average-case complexity (Levin, 1986; Impagliazzo, 1995; Bogdanov & Trevisan, 2006) asks when problems become tractable under input distributions. Smoothed analysis (Spielman & Teng, 2004), parameterized complexity (Downey & Fellows, 2013), and SAT backdoor analyses (Williams et al., 2003) provide influential ways to explain why hard worst-case problems may become easier on structured instances. These traditions typically rely on an analytic distribution, a perturbation model, a parameter, or a hand-specified structural property. We study a complementary sample-access regime: the deployment distribution is observed through examples, and the learned solver serves as a constructive witness that the sampled regime admits specialized computation.

Algorithm selection, portfolios, and hyper-heuristics. Algorithm selection (Rice, 1976) and feature-based portfolios such as SATzilla, Hydra, and AutoFolio (Xu et al., 2008; 2010; Lindauer et al., 2015; Kotthoff, 2014; Kerschke et al., 2019) choose among solvers based on instance or distributional features. This is close to our fixed-library regime, where empirical runtime identifies a fast solver that remains correct on the deployment distribution. Hyper-heuristics and automated heuristic design similarly use data to select, compose, or generate heuristics for optimization problems. Our hint-based regime moves beyond selection from a predefined library: the learner infers reusable structure from samples and compiles it into solver code, potentially producing a solver not present in the original portfolio.

Learning-augmented algorithms, adaptive computation, and amortization. Our deployment criterion aligns with work arguing that worst-case complexity is often too coarse for practical performance (Roughgarden, 2019), and with learning-augmented algorithms, where predictions or advice improve the behavior of an algorithm while robustness is preserved when advice is inaccurate (Lykouris & Vassilvitskii, 2018; Mitzenmacher & Vassilvitskii, 2022). The shared theme is that learned information can improve computation. In our setting, the learned object is not only advice to a fixed algorithmic template; it is a solver hint that can compile into a different executable solver. This also gives an amortization view: synthesis pays a one-time cost on samples so that future instances can be solved with lower per-instance computation.

Program synthesis, sketches, and code agents. Our synthesis regime connects to program synthesis from examples or natural language (Gulwani, 2011; Balog et al., 2017; Devlin et al., 2017), and to work that learns intermediate guides for search, such as sketches (Nye et al., 2019). It is also related to code-generation and agentic coding benchmarks (Hendrycks et al., 2021; Li et al., 2022; Jimenez et al., 2024; Huang et al., 2024). In contrast to benchmarks that primarily evaluate whether code solves a given task instance or issue, our benchmark evaluates whether an agent can use samples to infer structure that improves future computation. Runtime is therefore part of the deployment objective, the intermediate object is a solver hint, and the recovered hypothesis can be inspected after synthesis.

3. Problem Setup

We now formalize the sample-access solver-learning problem introduced above. Let D be an unknown deployment distribution over instances. A learner observes samples $S = (x_1, \dots, x_n) \sim D^n$ and returns executable solver code for future instances drawn from the same distribution.

The key difference from standard statistical learning is that the learned object is an executable procedure. Classical learning theory evaluates generalization by accuracy: a hypothesis is good if it performs well on fresh draws from D (Valiant, 1984; Vapnik & Chervonenkis, 1971; Vapnik, 1998). For solvers, accuracy alone is incomplete. Two solvers may both return valid solutions on the deployment distribution while differing substantially in runtime. We therefore evaluate whether samples can identify a solver whose *computation*, not only its output quality, is specialized to D.

This places the problem between worst-case and average-case analysis. Worst-case algorithm design assumes no distributional information, while average-case complexity assumes an analytic description of the input distribution. We study the intermediate sample-access regime: the distribution is unknown but observed through examples, and the learner must produce solver code that generalizes in both solution quality and runtime.

Formally, let $\mathcal{X}$ be the instance space, and let $\mathrm{V}(x, z) \in \{0, 1\}$ indicate whether z is a valid solution for x. A solver $c \in \mathcal{C}$ is a program mapping $x \mapsto c(x)$, with execution time $\mathrm{T}(c, x) \in \mathbb{R}_{\geq 0}$. We evaluate c by its deployment error $\mathrm{Err}_D(c) := \Pr_{x \sim D}[\mathrm{V}(x, c(x)) = 0]$ and expected deployment runtime $\mathrm{Run}_D(c) := \mathbb{E}_{x \sim D}[\mathrm{T}(c, x)]$.

Correctness is a feasibility constraint: among solvers with $\mathrm{Err}_D(c) = 0$, two solvers may differ arbitrarily in $\mathrm{Run}_D(c)$. For a solver class $\mathcal{C}$, define the class-relative distribution-aware runtime optimum as $\mathrm{Run}^\star_D(\mathcal{C}) := \inf_{c \in \mathcal{C}:\, \mathrm{Err}_D(c)=0} \mathrm{Run}_D(c)$. This is the best expected deployment runtime achievable by a correct solver in $\mathcal{C}$. Unlike worst-case runtime, it depends on D, and the role of the sample is to discover which fast correct solver the present D admits.

Selection from a fixed library. When $\mathcal{C}$ is given in advance, a natural rule is the runtime-aware analogue of empirical risk minimization. Let $\widehat{\mathrm{Err}}_S(c) := \frac{1}{n}\sum_{i=1}^n \mathbf{1}\{\mathrm{V}(x_i, c(x_i)) = 0\}$ and $\widehat{\mathrm{Run}}_S(c) := \frac{1}{n}\sum_{i=1}^n \mathrm{T}(c, x_i)$. We select $\widehat{c}_S \in \arg\min_{c \in \mathcal{C}:\, \widehat{\mathrm{Err}}_S(c)=0} \widehat{\mathrm{Run}}_S(c)$, the empirically fastest sample-consistent solver. Thus fixed-library selection empirically minimizes the sample analogue of $\mathrm{Run}^\star_D(\mathcal{C})$. This rule is appropriate when the solver library is enumerated, but it does not address richer settings where the useful specialization is not already present in $\mathcal{C}$.

Synthesis via solver hints. To move beyond enumerated libraries, we factor the learner through a *solver hint*: reusable structure inferred from samples and compiled into solver code. A hint space $\mathcal{H}$ and compilation map $\mathrm{Comp} : \mathcal{H} \to \mathcal{C}$ split learning into $S \mapsto \widehat{h}_S \mapsto \widehat{c}_S = \mathrm{Comp}(\widehat{h}_S)$. The hint may encode a backdoor, decomposition, active-resource pattern, local repair rule, or other distribution-specific shortcut. We focus on the regime in which $\mathrm{Comp}(h)$ is correct for every $h \in \mathcal{H}$, typically because the compiled solver falls back to a generic complete solver. The sample is then not used to learn correctness, but to identify which shortcut to compile.

4. Method

The setup above defines the object we want to learn: a reusable solver hint compiled into executable solver code. In realistic synthesis, however, the hint space, evidence statistics, and compilation map are unknown. The learner must propose what structure to look for, measure it from samples, and write code that exploits the resulting summary on future instances.

Our method implements this sample-to-hint-to-solver factorization with an LLM code agent. Each candidate contains a hypothesis about the hidden structure, an analysis program that estimates it from public training instances, and a deployment solver conditioned on the resulting summary. Validation selects among candidate factorizations by quality, optimality, and runtime. Additional implementation details, prompt schemas, validation checks, and failure handling appear in Appendix C.

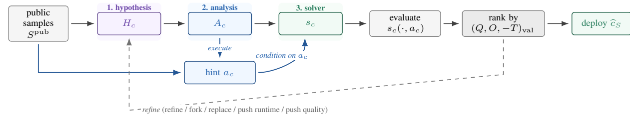

Figure 2. **Method pipeline.** Each candidate contains a hypothesis H_c, an analysis program A_c, and a deployment solver s_c. Executing A_c on the public training sample produces a summary $a_c = A_c(S^{\mathrm{pub}}_{\mathrm{tr}})$, which serves as the recovered hint. The solver $s_c(\cdot, a_c)$ is evaluated on public splits, ranked by validation quality, optimality, and runtime, and refined in a diversity-preserving beam.

Candidate representation. Each candidate has the form $c = (H_c, A_c, s_c)$. The hypothesis H_c is a structured natural-language description of a suspected distributional rule. The analysis program A_c maps the public training sample to a compact JSON-serializable summary, $a_c = A_c(S^{\mathrm{pub}}_{\mathrm{tr}})$, and the deployment solver maps a new public instance and this summary to a solution, $z = s_c(x^{\mathrm{pub}}, a_c)$. Thus a_c is the empirical solver hint, while $s_c(\cdot, a_c)$ is the compiled solver. Neither the hint space nor the compilation map is fixed in advance; both are generated separately for each candidate.

Public instances are stripped of evaluator-only fields, including family identity, hidden-rule metadata, optimum solutions, and optimum objective values. The agent sees only the public instance format, the problem specification, the scoring rule, and samples from the unknown structured distribution.

Three-stage construction. Candidates are generated by three sequential LLM calls. The first produces H_c: the suspected rule, evidence to measure, solver strategy, expected failure modes, and diversity key. The second writes A_c, which measures this evidence and compresses it into a reusable summary. The third is conditioned on H_c, A_c, and the executed summary a_c, and writes the deployment solver s_c. Thus synthesis is sample-conditioned: the solver is written against an actual estimate of distributional structure, not merely a plan to compute one.

Search and selection. Because the relevant hypothesis class is unknown, we search over a beam of candidates. The initial beam is seeded with broad structural directives, such as latent subtypes, separators, bottlenecks, decompositions, higher-order interactions, objective-aware marginal rules, and shortcut-plus-repair strategies. Later rounds refine surviving candidates, fork them into different diversity classes, replace brittle hypotheses, or push them toward higher quality or lower runtime. Each child is conditioned on the parent hypothesis, analysis output, code, validation metrics, and representative public failure cases.

Candidates are ranked lexicographically by $(Q_{\mathrm{val}}, O_{\mathrm{val}}, -T_{\mathrm{val}})$, where Q_{val} is average normalized quality, O_{val} is optimality rate, and T_{val} is average

runtime. Beam survivors are chosen in two passes: first, the best candidate for each diversity key is retained; then remaining slots are filled by the top-ranked candidates overall. Failed candidates receive zero quality and a large failure runtime. After the final iteration, we select the best candidate among all evaluated candidates, rerun its analysis on $S_{\mathrm{tr}}^{\mathrm{pub}}$, and deploy the resulting solver.

5. Why Can Samples Improve Computation?

The method above treats algorithm design as learning: from samples of an unknown distribution, the agent identifies reusable structure and compiles it into solver code. We now abstract this mechanism and ask when such sample-conditioned design generalizes beyond the observed instances.

We study two regimes. First, the learner selects from a fixed solver library, using empirical runtime to identify a fast correct solver for the deployment distribution. Second, the useful specialization is not enumerated in advance; the learner must recover a reusable structural *hint* and compile it into a solver. The results use standard concentration and union-bound arguments (Shalev-Shwartz & Ben-David, 2014). Their purpose is to formalize the core principle: samples can improve computation when they identify a shortcut shared by future instances. Proofs appear in Appendix F.

5.1. Runtime-Aware ERM Over a Fixed Solver Class

When the solver library is fixed, the empirically fastest sample-consistent solver approaches the best correct distribution-specialized solver in the class.

Theorem 5.1 (Runtime-aware generalization for library selection)**.** *Assume* $0 \le \mathrm{T}(c,x) \le \mathrm{T}_{\max}$ *for every* $c \in \mathcal{C}$ *and* $x \in \mathcal{X}$*, and assume* $\mathcal{C}$ *contains at least one solver that is correct almost surely under* D*. Let* π *be a prior on* $\mathcal{C}$ *with* $\sum_{c\in\mathcal{C}} \pi(c) \le 1$*, and write* $\Gamma(c) := \log(1/\pi(c))$*. Let* $\mathcal{C}^{\mathrm{feas}} = \{c \in \mathcal{C} : \mathrm{Err}_D(c) = 0\}$*. For every* $\delta \in (0,1)$*, with probability at least* $1-\delta$ *over* $S \sim D^n$*,* $\mathrm{Err}_D(\widehat{c}_S) \le \frac{\Gamma(\widehat{c}_S)+\log(\frac{2}{\delta})}{n}$ *and* $\mathrm{Run}_D(\widehat{c}_S) \le \inf_{c\in\mathcal{C}^{\mathrm{feas}}} \left\{ \mathrm{Run}_D(c) + 2\mathrm{T}_{\max}\sqrt{\frac{\max\{\Gamma(\widehat{c}_S),\Gamma(c)\}+\log(\frac{4}{\delta})}{2n}} \right\}$.

For a finite uniform library with $\pi(c) = 1/|\mathcal{C}|$, Theorem 5.1 says that $\widehat{c}_S$ approaches $\mathrm{Run}_D^{\star}(\mathcal{C})$ up to an additive $O(\mathrm{T}_{\max}\sqrt{\log|\mathcal{C}|/n})$ term, while sample consistency controls deployment error. Thus runtime-aware ERM is not merely choosing a fast solver on the sample; it estimates the best correct distribution-specialized solver available in the class. The guarantee is class-relative: it does not say that the library contains the right specialization, only that if such a solver is present, empirical runtime minimization can identify it from samples. This motivates the hint-based synthesis regime below, where the specialization itself must be constructed.

5.2. Synthesis via Learnable Hints

In the rich domains we ultimately care about, no fixed library contains the right specialized solver. To make the search tractable, we shift it from full solvers to a smaller space of reusable structure. Let $\mathcal{H}$ be a finite hint space, where each $h \in \mathcal{H}$ induces a distribution D_h over instances sharing that structure. We assume a realizable setting: $D = D_{h^\star}$ for some unknown $h^\star \in \mathcal{H}$, and both training and deployment instances are drawn from $D_{h^\star}$.

Given a score family with a margin separation, hint recovery reduces to a finite-class estimation problem. Suppose score functions $\{\psi_h : \mathcal{X} \to [0,1]\}_{h\in\mathcal{H}}$ satisfy $\mathbb{E}_{x\sim D_h}[\psi_h(x)] \ge \mathbb{E}_{x\sim D_h}[\psi_g(x)] + \gamma$ for every $h \in \mathcal{H}$ and every $g \in \mathcal{H} \setminus \{h\}$ for some $\gamma > 0$. Given $S = (x_1,\ldots,x_n)$, the learner returns $\widehat{h} \in \arg\max_{h\in\mathcal{H}} \widehat{\mu}_S(h)$ with $\widehat{\mu}_S(h) := \frac{1}{n}\sum_{t=1}^{n} \psi_h(x_t)$.

Theorem 5.2 (Exact recovery under identifiable structure)**.** *If* $|\mathcal{H}| = N$ *and the margin is* $\gamma > 0$*, then* $n \ge \frac{2}{\gamma^2}\log\frac{2N}{\delta}$ *samples suffice for* $\widehat{h} = h^\star$ *with probability at least* $1-\delta$*.*

The sample complexity is logarithmic in $|\mathcal{H}|$ and inverse-quadratic in the margin, after which the learner compiles $\widehat{h}$ into a specialized solver. The theorem is intentionally idealized: in realistic domains the hard part is not estimating $\widehat{h}$ for a known score family but *discovering* what the hint should be, what statistic reveals it, and how to compile it into code. This is precisely the regime our experiments target. We use an LLM agent as an approximate procedure for these three sub-tasks and evaluate whether the resulting solvers improve in deployment runtime and quality on structured task distributions. Before turning to that empirical study, we give a formal example where the score family is known and Theorem 5.2 applies directly.

5.3. A Formal Example: Hidden SAT Backdoors

SAT illustrates the point cleanly because correctness never requires learning: a complete solver is always available. The value of learning is computational, recovering a reusable backdoor that makes future solving faster. Fix variables $[d]$ and backdoor size k. The hint space is $\mathcal{H} = \binom{[d]}{k}$, and an unknown $B \in \mathcal{H}$ indexes a distribution D_B over CNF formulas on d variables. We assume B is a strong backdoor into a tractable class $\mathcal{T}$: for every F in the support of D_B and every $\alpha \in \{0,1\}^B$, the restricted formula $F|_{B=\alpha}$ lies in $\mathcal{T}$. The learner observes $F^{(1)},\ldots,F^{(m)} \sim D_B$ but not B. Assume membership in B is identifiable from a bounded variable-level salience statistic $\sigma_i(F) \in [0,1]$ with $\mathbb{E}_{F\sim D_B}[\sigma_i(F)] \ge q_1$ for $i \in B$ and $\le q_0$ for $i \notin B$, with margin $\gamma = q_1 - q_0$. The learner estimates

$\widehat{\sigma}_i = \frac{1}{m}\sum_t \sigma_i(F^{(t)})$ and sets $\widehat{B}$ to the top-k variables. The compiled solver enumerates assignments to $\widehat{B}$, solves the residual if it lies in $\mathcal{T}$, and otherwise falls back to a complete base solver—preserving correctness for every $\widehat{B}$ while gaining speed when $\widehat{B} = B$.

Theorem 5.3 (Learning a hidden SAT backdoor from samples)**.** *If* $m \geq 8\,\gamma^{-2}\log\frac{2d}{\delta}$*, then* $\widehat{B} = B$ *with probability at least* $1-\delta$.

On the event $\widehat{B} = B$, the learned solver runs in $O(2^k\text{poly}(|F|))$ on formulas from D_B, with learner-side cost $O(md)$ for the salience scores. The sample is not used to learn SAT correctness; it identifies which structural shortcut to compile. A concrete planted Horn-backdoor family satisfying the separation assumption is given in Appendix B.

6. Experiments

We evaluate whether the synthesis procedure of Section 4 extracts reusable structure from samples and improves the quality–runtime tradeoff over heuristic and solver-backed baselines. The appendices give the full implementation details (Appendix C), benchmark distributions (Appendix D.1), baseline protocols (Appendix D.3), additional setup details (Appendix D), full per-target results (Appendix E.1), distributional-complexity diagnostics (Appendix E.3), perturbation ablations (Appendix E.6), and our results on the PACE Dominating Set competition (Appendix E.2).

6.1. Experimental setup

Benchmarks. The benchmark contains structured combinatorial optimization tasks. Each *target* pairs a problem class with a hidden distribution family. The learner observes sampled public instances, but not the family identity, hidden rule, optimum solutions, optimum objective values, or hidden-rule metadata, which are retained only by the evaluator. The benchmark spans seven problem classes and 21 hidden families, three per class; the full target list is in Table 5 of Appendix D.1. We report aggregate results over all 21 targets. Each family is designed so that recurring cross-instance structure, rather than per-instance heuristic search alone, is the relevant signal.

Baselines. We compare the LLM synthesis agent of Section 4 with the full benchmark baseline catalog, evaluating all methods under the same public-instance protocol. The catalog includes problem-specific heuristics, time-limited optimization backends, and exact or certifying methods when available. The heuristic pool includes DSATUR-style coloring (Brélaz, 1979), greedy set-cover and graph rules for dominating set and independent set variants (Chvátal, 1979), density and relaxation-based rules for packing and knapsack (Dantzig, 1957), and insertion or local-search heuristics for TSP, including two-opt and LKH (Lin, 1965; Rosenkrantz et al., 1977; Helsgaun, 2000). The optimization-backed pool includes time-limited Gurobi formulations (Gurobi Optimization, LLC, 2026), OR-Tools CP-SAT/GLOP formulations (Perron & Furnon, 2025; Perron et al., 2023), PySAT/RC2 MaxSAT solvers (Ignatiev et al., 2019), branch-and-bound routines (Land & Doig, 1960), Held–Karp dynamic programming for TSP (Held & Karp, 1962), and external solvers including SCIP, HiGHS, CBC, Open-WBO, and UWrMaxSAT (Bestuzheva et al., 2023; HiGHS Development Team, 2024; Forrest & Lougee-Heimer, 2005; Martins et al., 2014; Piotrów, 2020). The exhaustive per-problem catalog appears in Table 7 of Appendix D.3.

Heuristic baselines have no access to the hidden family rule. We report Gurobi separately because it uses a 10-second, single-thread budget and may return incumbents rather than certified optima. When several time-limited exact or certifying backends are available, we report the fastest method among those that attain the best certified or validated quality, denoted the *time-limited exact* baseline.

Evaluation. The primary metric is average normalized quality, scaled so that 1.0 is optimal and larger is better; invalid or infeasible outputs receive quality zero. We also report optimality rate, feasibility rate, and average per-instance wall-clock runtime. Per-problem quality definitions appear in Table 6 of Appendix D. Each per-target value averages 10 repeated evaluations per solver and split.

For aggregate quality, optimality, feasibility, and quality-lift values, we compute one value per target distribution by averaging over held-out test instances and repeated runs, and then take the arithmetic mean over the 21 target distributions. For aggregate runtime comparisons, we compute per-target speedup ratios and report their geometric mean. This treats speedup as a multiplicative quantity and avoids having a few high-runtime targets dominate the aggregate. Synthesis methods may use training and validation instances, including representative failure cases from those splits, for refinement. Test instances are never used during synthesis, selection, or refinement.

6.2. Results

Main quality–runtime results. The relevant comparison is the quality–runtime tradeoff: a useful distribution-aware solver should recover high-quality solutions without treating each instance as worst case. Table 1 summarizes this tradeoff over all 21 benchmark target distributions; full per-target results appear in Appendix E.1.

LLM synthesis reaches mean normalized quality 0.971, improving by +0.224 over the average heuristic pool and by +0.098 over the quality-best heuristic. Using geometric

Family	Q_{LLM}	ΔQ_{avg}	ΔQ_{best}	T_{LLM} (ms)	$T_{\text{best}}/T_{\text{LLM}}$	$T_{\text{Gurobi}}/T_{\text{LLM}}$	$T_{\text{exact}}/T_{\text{LLM}}$
Coloring	0.868	+0.217	+0.121	2.7	1326.3×	2285.6×	23.1×
MAXSAT	1.000	+0.122	+0.074	17.1	217.4×	328.8×	1.0×
MIS	0.992	+0.218	+0.106	18.8	530.8×	155.8×	39.7×
MDS	0.973	+0.148	+0.122	13.3	718.9×	443.0×	17.4×
Packing LP	0.994	+0.301	+0.259	3.3	3004.2×	2829.1×	37.2×
MDKP	0.973	+0.215	+0.009	94.0	46.4×	33.8×	12.4×
TSP	0.993	+0.348	−0.007	15.3	32.1×	112.1×	36.6×
All (21)	**0.971**	**+0.224**	**+0.098**	**12.8**	**336.9×**	**342.8×**	**16.1×**

Table 1. **Headline quality–runtime summary over 21 target distributions.** Quality is normalized so higher is better. Quality lifts compare the synthesized solver to the average and best heuristic. T_{LLM} and speedups are geometric means over target distributions; speedups above 1× mean the synthesized solver is faster. Heuristic runtimes are clipped at 10 seconds before ratio computation. Full per-target results, including optimality rates, appear in Table 9.

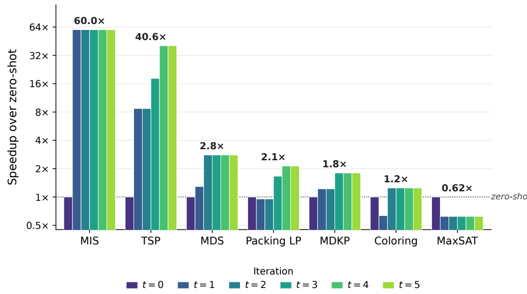

Figure 3. **Runtime speedup relative to the zero-shot generated solver across synthesis iterations.** The bar at $t = 0$ is the zero-shot reference. For $t > 0$, each bar reports the best generated candidate found up to iteration t, with speedup computed as zero-shot runtime divided by candidate runtime. Values above 1× indicate faster execution than zero-shot, while values below 1× indicate slower execution. Tasks are ordered by final speedup.

means of per-target runtime ratios, it is 336.9× faster than the quality-best heuristic, 342.8× faster than Gurobi, and 16.1× faster than the selected time-limited exact backend. The gains are strongest on families where the synthesized solver finds a distribution-specific shortcut, such as Packing LP, MDS, and TSP.

The exceptions are also informative. On TSP, LLM synthesis is much faster but slightly trails the quality-best heuristic by 0.007. On MAXSAT, the synthesized solver matches optimal quality but does not improve over the selected time-limited exact backend in runtime. Thus, the method improves the average quality–runtime frontier, but does not always recover the cheapest or most accurate specialized procedure for every problem family.

Iteration ablation. We next examine how much of the runtime gain comes from iterative synthesis, rather than from the first generated proposal alone. After each iteration, we measure the test-time speedup of the best generated candidate found so far, using the zero-shot generated solver as the reference point. This isolates the effect of search depth on the efficiency of the synthesized solver.

Figure 3 shows that later synthesis iterations often improve runtime beyond the first proposal. Some gains appear immediately, as in MIS, while others require refinement: TSP improves from about 8.7× after the first iteration to 40.6× by iteration 4, and Packing LP recovers from initially slower candidates to about 2.1× speedup. MDKP and MDS also improve after the initial proposal.

Thus, the loop is not merely selecting among equivalent first-pass proposals. Later iterations can recover from poor specializations or refine useful ones into faster implementations. The MaxSAT case is a counterexample: under this search budget, generated candidates remain slower than zero-shot. Iterative synthesis is therefore not uniformly beneficial, but can produce substantial runtime gains when the family contains exploitable structure.

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.3 (specifically Table 12) gives the full per-distribution expressions and notation.

The main pattern is that synthesis often compiles a structural shortcut. For MAXSAT, the generated solvers exploit latent Boolean rules and bounded local repair instead of searching over all assignments. For graph problems, they use planted palettes, motif decompositions, hubs, gateways, or small residual kernels instead of treating each graph as an arbitrary coloring, MIS, or MDS instance. For Packing LP and MDKP, they exploit active resources or bottleneck structure, replacing full LP or exponential knapsack search with sorting, scoring, fractional filling, and bounded repair. For TSP, they exploit clustered or latent geometric structure to construct and improve tours without running full Held–Karp-style dynamic programming.

Thus, the empirical speedups reflect the intended mechanism of distribution-aware program learning: 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 Dominating Set comparison. To test whether the synthesis procedure produces useful solvers beyond our controlled benchmark families, we also evaluate on the PACE 2025 Dominating Set heuristic track. PACE released 100 public instances and, after the competition, 100 private instances generated in a similar way (Grobler & Siebertz, 2025; PACE Challenge, 2025). We split the public set into 50 training and 50 validation instances, and report on the released private set. This comparison is not an official PACE score, but it provides an external test against highly

Structure	Ambient computation	Generated-solver computation
MAXSAT: latent Boolean rules	$O^*(2^v)$	$O(\lvert F\rvert + R(\lvert F\rvert + B\ell^2\Delta_{\text{occ}}))$
Coloring: planted palettes	$O^*(\kappa^n)$	$O(R(n^2 + m + n\kappa) + T_{\text{recolor}})$
MIS: motif structure	$O^*(2^n)$	$O(P(n + m) + T_{\text{local}} + T_{\text{tiny}})$
MDS: coverage kernels	$O^*(2^n)$	$O(n + m + \sum_j c_j 2^{t_j} + T_{\text{prune}})$
Packing / MDKP: resource bottlenecks	$T_{\text{LP}}(N, r, L)$ or $O^*(2^N)$	$O(P(Nr + N\log N) + T_{\text{repair}})$
TSP: latent geometry	$O(n^2 2^n)$	$O(n^2\log n + B_{\text{tsp}}n^2)$

Table 2. **Representative computation patterns discovered by LLM synthesis.** Bounds show dominant runtime terms; full expressions and notation appear in Appendix E.3.

Table 3. **PACE 2025 Dominating Set comparison.** Lower size and runtime are better. The synthesized solver is valid on all private instances and much faster, while PACE solvers find smaller dominating sets. This is a local comparison using released instances and solvers, not an official PACE score; setup and official results are described in (Grobler & Siebertz, 2025; PACE Challenge, 2025). Full results appear in Appendix E.2.

Solver	Valid	LLM/solver size	Runtime	LLM speedup
LLM	100/100	1.000	2.89s	1.0×
Fontan–Verger (Fontan & Verger, 2025)	100/100	1.033	286.01s	98.8×
Root (Luo et al., 2025)	100/100	1.033	300.37s	103.8×
Swats (Swat, 2025)	75/100	1.028[†]	287.41s[†]	130.2×[†]
Shadoks (da Fonseca et al., 2025)	100/100	1.033	315.89s	109.2×

[†] Computed only on the 75 instances for which Swats returned verified-valid solutions. "LLM/solver size" is the matched total-size ratio; values above 1 mean the agent returns a larger dominating set.

engineered competition solvers. We compare against the top released Dominating Set heuristic submissions: Fontan–Verger, Root, Swats, and Shadoks (Fontan & Verger, 2025; Luo et al., 2025; Swat, 2025; da Fonseca et al., 2025).

On the private instances, the synthesized solver returns verified-valid solutions on all 100 graphs and runs about two orders of magnitude faster than the released PACE solvers. The tradeoff is quality: the PACE solvers return smaller dominating sets on every matched instance. The aggregate gap, however, is moderate: the synthesized solver's total solution size is about 3.3% larger than Fontan–Verger, Root, and Shadoks, while its average runtime is roughly 99×–109× lower. Thus, the PACE experiment shows a fast-but-lower-quality operating point on a large external benchmark; full details appear in Appendix E.2.

7. Discussion and Limitations

We study learning when the output is executable solver code rather than a prediction rule. The main lesson is that samples can improve computation when they reveal structure reused across future instances. A solver hint captures this structure: it is not a solution to one instance, but information that changes the algorithm used for many later instances. Empirically, LLM synthesis sometimes performs this amortized algorithm design. The generated solvers often change the computational scale, replacing broad search or full optimization with distribution-specific sorting, scoring, bounded repair, kernelization, or structured construction. Thus, the learned object is closer to a specialized algorithm for the deployment regime than to a tuned heuristic for isolated instances.

The same view explains the limitations. The one-time synthesis cost is useful only when amortized over enough future instances. The resulting solver is specialized to the sampled regime, so its advantage may degrade under certain distribution shifts. Finally, because the multi-agent system searches over a rich program space, different runs may recover different hints, useful proxies, or brittle shortcuts. As future work, it would be interesting to understand how to improve stability while preserving the discovery of distribution-specific computation.

440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494

References

Applegate, D. L., Bixby, R. E., Chvátal, V., and Cook, W. J. *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton University Press, 2006. ISBN 9780691129938.

Avellaneda, F. A short description of the solver EvalMaxSAT. In Bacchus, F., Berg, J., Järvisalo, M., and Martins, R. (eds.), *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, pp. 8–9, 2020.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=ByldLrqlx.

Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., van Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., 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.

Bogdanov, A. and Trevisan, L. Average-case complexity. *Foundations and Trends in Theoretical Computer Science*, 2(1):1–106, 2006. doi: 10.1561/0400000004.

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

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

Coll, J., Li, C. M., Li, S., Habet, D., and Manyà, F. Solving weighted maximum satisfiability with branch and bound and clause learning. *Computers & Operations Research*, 183:107195, 2025. doi: 10.1016/j.cor.2025.107195.

da Fonseca, G. D., Feschet, F., and Gerard, Y. PACE Solver Description: Shadoks Approach to Minimum Hitting Set and Dominating Set. In Agrawal, A. and van Leeuwen, E. J. (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.

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

Davies, J. and Bacchus, F. 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.

Davies, J. and Bacchus, F. 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.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. 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.

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

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610:47–53, 2022. doi: 10.1038/s41586-022-05172-4.

Fontan, F. and Verger, G. PACE Solver Description: Reductions and Heuristic Search for the Dominating Set Problem and the Hitting Set Problem. In Agrawal, A. and van Leeuwen, E. J. (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.

Forrest, J., Vigerske, S., Ralphs, T., Hafer, L., Fasano, J., Santos, H. G., Saltzman, M., King, A., Brito, S., et al. coin-or/Clp: Release releases/1.17.7, 2022. URL https://doi.org/10.5281/zenodo.5839302.

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

Grobler, M. and Siebertz, S. The PACE 2025 Parameterized Algorithms and Computational Experiments Challenge: Dominating Set and Hitting Set. In Agrawal, A. and van Leeuwen, E. J. (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.

Gulwani, S. 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.

Gurobi Optimization, LLC. Gurobi optimizer reference manual, 2026. URL https://www.gurobi.com.

Held, M. and Karp, R. M. 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.

Helsgaun, K. 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.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S. Y., He, H., Song, D., and Steinhardt, J. 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/.

Huang, Q., Vora, J., Liang, P., and Leskovec, J. 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/huang24b.html.

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

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

Impagliazzo, R. 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.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. 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.

Kerschke, P., Hoos, H. H., Neumann, F., and Trautmann, H. Automated algorithm selection: Survey and perspectives. *Evolutionary Computation*, 27(1):3–45, 2019. doi: 10.1162/evco_a_00242.

Kotthoff, L. Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 35(3):48–60, 2014. doi: 10.1609/aimag.v35i3.2460.

Lamm, S., Schulz, C., Strash, D., Williger, R., and Zhang, H. 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.

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

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

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

Lin, S. 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.

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

Lu, C., Lu, C., Lange, R. T., et al. Towards end-to-end automation of AI research. *Nature*, 651:914–919, 2026. doi: 10.1038/s41586-026-10265-5.

550 Luo, C., Zhang, Q., Su, Z., and Lü, Z. PACE Solver Description: Weighting-Based Local Search Heuristic for the Hitting Set Problem. In Agrawal, A. and van Leeuwen, E. J. (eds.), *20th International Symposium on Parameterized and Exact Computation (IPEC 2025)*, volume 358 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 40:1–40:4, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-407-9. doi: 10.4230/LIPIcs.IPEC.2025.40. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.IPEC.2025.40.

Lykouris, T. and Vassilvitskii, S. 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.

Mankowitz, D. J., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618:257–263, 2023. doi: 10.1038/s41586-023-06004-9.

Martins, R., Manquinho, V., and Lynce, I. 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.

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

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025.

Nye, M., Hewitt, L., Tenenbaum, J. B., and Solar-Lezama, A. 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.

PACE Challenge. PACE 2025 Results. https://pacechallenge.org/2025/results/, 2025. Accessed 2026-05-07.

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

Perron, L., Didier, F., and Gay, S. 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.

Piotrów, M. 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.

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

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P., and Fawzi, A. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6.

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

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

Shalev-Shwartz, S. and Ben-David, S. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. ISBN 9781107057135. doi: 10.1017/CBO9781107298019.

Spielman, D. A. and Teng, S.-H. 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.

Swat, S. PACE Solver Description: HitS&DoSeS - Exact and Heuristic Solvers for the Dominating Set and Hitting Set Problems. In Agrawal, A. and van Leeuwen, E. J. (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.

Valiant, L. G. 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.

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

Vapnik, V. N. and Chervonenkis, A. Y. 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.

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

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

Xu, L., Hoos, H. H., and Leyton-Brown, K. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pp. 210–216, 2010. doi: 10.1609/aaai.v24i1.7565.

Yamada, Y., Lange, R. T., Lu, C., Hu, S., Lu, C., Foerster, J., Clune, J., and Ha, D. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search. 2025. doi: 10.48550/arXiv.2504.08066.

660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714

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. A Planted Horn-Backdoor Family

We give a concrete distribution satisfying the separation assumption in Section 5.3. Fix an unknown set $B \subseteq [d]$ of size k, and generate a CNF formula $F \sim D_B$ as a conjunction of M independently sampled clauses. Each clause is Horn with probability $1 - \rho$. With probability ρ, it is a non-Horn clause of the form

$$x_i \vee x_j \vee \bigvee_{\ell \in T} \neg x_\ell,$$

where $i \sim \mathrm{Unif}(B)$, $j \sim \mathrm{Unif}([d] \setminus B)$, and $T \subseteq [d] \setminus (B \cup \{j\})$ is sampled by any fixed rule. For every assignment to the variables in B, each non-Horn clause is either satisfied or reduced to a Horn clause over the remaining variables. Thus B is a strong backdoor to Horn-SAT.

Define the salience statistic

$$\sigma_i(F) = \frac{1}{M} \sum_{C \in F} \mathbf{1}\{C \text{ is non-Horn and } x_i \text{ appears positively in } C\}.$$

Then

$$\mathbb{E}[\sigma_i(F)] = \begin{cases} \rho/k, & i \in B, \\ \rho/(d-k), & i \notin B. \end{cases}$$

Hence the variable-level separation assumption holds with $q_1 = \rho/k$, $q_0 = \rho/(d-k)$, and $\gamma = \rho(1/k - 1/(d-k))$, which is positive whenever $k < d/2$.

C. Additional Method Details

This appendix records the experimental protocol behind Section 4. The goal is to make clear what information was available to the synthesis agent, how candidates were represented and selected, and how invalid candidates were handled. The protocol enforces a separation between distribution-level inference and deployment-time solving: the analysis program may use the public training sample to estimate a reusable summary, while the deployed solver must solve new public instances using only that summary and the instance itself.

A single LLM-generated candidate is not treated as the learned algorithm. Instead, the synthesis loop is a proposal-and-selection procedure over solver hints and implementations. Different proposals may encode different structural explanations of the same sample, and some may be brittle, slow, or incorrect. The deterministic evaluator converts this high-variance proposal process into a selected deployed solver by filtering candidates on public validation quality, optimality, and runtime.

Public information and leakage boundary. For each target, the synthesis agent receives public training and validation instances, together with a small task manifest. The manifest specifies the problem type, the required solution format, the normalized-quality metric, and basic instance-size information. It also states that the instances are drawn from a shared but unknown structured distribution.

The public view excludes all fields used to define or score the hidden family. In particular, it does not include the distribution-family identity, the planted rule, optimum solutions, optimum objective values, or evaluator-only metadata. This boundary is intended to model deployment: the learner knows the ambient optimization problem and the scoring rule, but not the generative mechanism that produces the deployment instances.

715
716
717
718
719
720
721
722
723
724
725
726
727
728
729

Stage	Elicited content
Hypothesis	A short title; a concrete rule summary describing the suspected reusable structure; evidence that should be measured from public samples; the solver strategy implied by the rule; expected failure modes; and a diversity key identifying the broad explanation class.
Analysis	A train-time procedure applied to $S_{\mathrm{tr}}^{\mathrm{pub}}$; a description of what the procedure estimates; a compact output requirement, so that the summary is reusable rather than a copy of the training set; and a robustness note favoring aggregate evidence over brittle per-instance artifacts.
Solver	A deployment-time procedure applied to a new public instance; a description of how the recovered summary is used; a fallback for weak or ambiguous inferred structure; and a runtime constraint discouraging repetition of train-time analysis during deployment.

Table 4. **Stage contracts used by the synthesis procedure.** The diversity key is used only for beam selection and is not part of the validation score.

730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769

All synthesis stages use a shared instruction emphasizing that the goal is to infer reusable structure from public samples and compile it into a fast solver. The instruction states that validation quality is the primary selection signal, and that runtime matters once quality is high. It does not reveal hidden-rule metadata or test performance.

Candidate representation and stage contracts. Each candidate has the form $c = (H_c, A_c, a_c, s_c)$, where H_c is a structured hypothesis, A_c is a train-time analysis program, $a_c = A_c(S_{\mathrm{tr}}^{\mathrm{pub}})$ is the recovered summary, and s_c is the deployment-time solver. The analysis program performs distribution-level inference from the public training sample. The solver then maps a new public instance and the recovered summary to a candidate solution, $z = s_c(x^{\mathrm{pub}}, a_c)$. Thus a_c is the empirical analogue of the solver hint in Section 3.

The fields elicited at each stage are summarized in Table 4. These stage contracts prevent hypothesis formation, evidence extraction, and solver construction from collapsing into a single opaque generation step. They also make the recovered hint explicit and inspectable after synthesis.

Synthesis loop. Algorithm 1 gives the full synthesis loop. Each round constructs a batch of candidate hint–solver pairs, executes their analysis programs on the public training sample, evaluates their solvers on the public train and validation splits, and updates a diversity-preserving beam. After the final round, the selected candidate is the best evaluated candidate across all rounds, not necessarily the best member of the final beam. Thus the deployed solver is validation-selected from a portfolio of generated candidates rather than taken from a single LLM proposal.

Beam selection and refinement. Each hypothesis includes a diversity key identifying its broad explanation class, such as a separator-based explanation, a bottleneck-resource explanation, a latent-cluster explanation, or a geometric-template explanation. After each round, candidates are ranked by the validation-aware score in Algorithm 1. The next beam is then formed in two passes. First, the best candidate for each diversity key is retained. Second, any remaining slots are filled by the highest-scoring candidates overall.

This rule encourages exploration early in synthesis without preventing a strong candidate from surviving once diversity has been exhausted. Refinement prompts include the parent hypothesis, the recovered summary, aggregate train and validation metrics, and representative failure cases. They do not include hidden-rule metadata or test performance. The refinement action asks the agent to refine the current explanation, fork to a different explanation, improve quality, reduce runtime, or replace a hypothesis that appears to rely on an accidental shortcut.

This design is important because LLM synthesis is stochastic. The search loop is not only an optimization over code implementations; it is also an exploration over competing explanations of the distribution. Diversity keys keep multiple structural hypotheses alive, while validation selection determines which hypothesis actually compiles into a reliable and fast solver.

Recovered summaries. The recovered summary a_c is not restricted to a fixed representation. Its form depends on the structure inferred by the candidate. In graph problems, summaries may contain vertex-role scores, cluster assignments, separator candidates, or recurring motif statistics. In MaxSAT, they may contain variable-salience scores, polarity statistics, backbone estimates, or clause-position signals. In packing and knapsack problems, they may contain estimated resource bottlenecks, item-class prototypes, or active-constraint patterns. In TSP-like tasks, they may contain geometric prototypes, coarsened tour skeletons, or detected coordinate transformations.

770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824

Algorithm 1 LLM-based distribution-to-algorithm synthesis

Require: Public train/validation sets $S_{\text{tr}}^{\text{pub}}, S_{\text{val}}^{\text{pub}}$; iterations R, beam width B, candidate budget K
Ensure: Learned solver $\hat{s}$ and recovered summary $\hat{a}$
1: Initialize beam $\mathcal{B}_0$ with diverse hypothesis directives
2: Initialize evaluated set $\mathcal{E} \leftarrow \emptyset$
3: **for** $r = 0, \ldots, R-1$ **do**
4: Build K candidate prompts from $\mathcal{B}_r$ {seed / refine / fork / replace / push runtime / push quality}
5: **for** each prompt p **do**
6: Generate hypothesis H_c
7: Generate analysis program A_c
8: Execute analysis $a_c \leftarrow A_c(S_{\text{tr}}^{\text{pub}})$
9: Generate solver s_c conditioned on $(H_c,\ A_c,\ a_c)$
10: Evaluate $s_c(\cdot, a_c)$ on $S_{\text{tr}}^{\text{pub}}$ and $S_{\text{val}}^{\text{pub}}$
11: Add $c = (H_c, A_c, a_c, s_c)$ to $\mathcal{E}$
12: **end for**
13: Score each $c \in \mathcal{E}$ by

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

14: Beam update: set $\mathcal{B}_{r+1}$ using a diversity-preserving rule: keep the best candidate per diversity key, then fill remaining slots by score
15: Extract summaries, scores, and failure cases from $\mathcal{B}_{r+1}$ for the next refinement round
16: **end for**
17: Select $c^\star \in \mathcal{E}$ with highest $\text{Score}(c^\star)$
18: Re-run analysis $\hat{a} \leftarrow A_{c^\star}(S_{\text{tr}}^{\text{pub}})$
19: **Return** $\hat{s} := s_{c^\star}$ and $\hat{a}$

These summaries are not directly supervised or scored. They matter only through the quality and runtime of the solver that uses them. Their role is to amortize distribution-level inference: the analysis program may spend computation once on the public training sample, while the deployed solver should use the resulting summary cheaply on each new instance. This separation is what distinguishes the method from per-instance prompting or repeated test-time reasoning: the expensive distribution-level inference happens before deployment, and the deployed solver is ordinary executable code conditioned on the recovered summary.

Failure handling. A candidate can fail during analysis, solver construction, or instance-level execution. If the analysis program A_c raises an error or exceeds its time budget, the candidate receives zero quality on every instance and is assigned a large failure runtime. The same convention applies if the solver s_c fails to load, for example because of a syntax error or unavailable dependency.

If both A_c and s_c load successfully but the solver returns an invalid output on some instances, only those instances receive normalized quality zero. The remaining instances are scored normally. This convention keeps the synthesis loop fully automatic: malformed candidates are not manually repaired, and partial failures degrade the validation score rather than requiring special-case intervention.

Connection to the theoretical framework. The protocol implements the hint-learning decomposition from Section 3. The hypothesis H_c proposes a possible family of reusable structure, the analysis program A_c estimates a hint from samples, and the solver s_c compiles the estimated hint into an executable algorithm. Unlike the idealized theory, the hint space and evidence statistics are not fixed in advance. They are proposed by the LLM, and the resulting candidates are selected by deterministic train-validation evaluation. Thus the method is better viewed as search over a proposal distribution of candidate hints and solvers, rather than as a deterministic sample-to-solver map. Candidates survive only when their inferred summaries yield high-quality solutions with low deployment runtime on public validation instances.

825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847

Problem	Distribution family	Instance size	Objective
Coloring	Ring-template	169 vertices	minimize colors
	Overlapping-palette	340 vertices	
	Separator-trap	238 vertices	
MAXSAT	Community-parity	240 variables / 960 clauses	maximize satisfied clauses
	Last-clause signal	280 variables / 1120 clauses	
	Latent-backdoor	128 variables / 512 clauses	
MIS	Clique-path	190 vertices	maximize set size
	Core-fringe trap	1000 vertices	
	Motif-bridge	195 vertices	
MDS	Gateway-hub	2800 vertices	minimize set size
	Geometric-anchor	1600 vertices	
	Star-kernel	2800 vertices	
Packing LP	Block-coupled	1200 items / 40 resources	maximize value
	Active-resource	1200 items / 40 resources	
	Single-bottleneck	1200 items / 40 resources	
MDKP	Decoy-complement	1040 items / 48 resources	maximize value
	Latent-class	520 items / 32 resources	
	Single-resource	1040 items / 48 resources	
TSP	Clustered Euclidean	120 cities	minimize tour length
	Latent-metric	120 cities	
	Paired-ribbon zigzag	320 cities	

848
849
850

Table 5. **Benchmark target distributions.** The benchmark contains 21 targets: seven problem classes with three hidden distribution families per class. Each target contains 64 training instances, 32 validation instances, and 500 held-out test instances. Sizes are listed in the same order as the distribution families.

851
852
853
854
855

D. Additional Experimental Details

D.1. Benchmark Distributions

856
857
858
859
860
861
862

Table 5 summarizes the benchmark targets. The benchmark contains 21 designed targets, spanning seven problem classes with three hidden distribution families per class. The benchmark contains 21 target distributions, each with 64 training instances, 32 validation instances, and 500 held-out test instances. Each target defines a distribution over structured optimization instances, rather than a collection of arbitrary worst-case instances. The goal is to test whether a synthesis procedure can infer reusable regularities from a small sample and compile them into a specialized solver. Each completed target contains 64 training instances, 32 validation instances, and 500 held-out test instances.

863
864
865

The distributions are designed to cover several kinds of exploitable structure: planted assignments, latent classes, recurring bottlenecks, geometric layouts, motif decompositions, and hidden resource constraints. We briefly describe each family below.

866
867
868
869
870
871

Graph coloring. The coloring targets contain graphs with planted low-color structure. In *Ring-template*, vertices are partitioned into local blocks with compatible palette assignments and ring-style bridges between neighboring blocks. In *Overlapping-palette*, blocks share shifted and overlapping palettes, so a solver must recover the global palette rather than color each block independently. In *Separator-trap*, separator vertices create long-range color-reuse constraints, making local coloring rules unreliable unless the global palette structure is inferred.

872
873
874
875
876

MAXSAT. The MAXSAT targets contain formulas with hidden assignments induced by small latent rules. In *Community-parity*, variables are partitioned into communities whose values are controlled by parity-style functions of a few anchor bits. In *Last-clause signal*, the final clause reveals an anchor pattern, while the remaining variables copy or negate anchor bits according to a hidden rule table. In *Latent-backdoor*, each instance comes from one of several regimes in which variable blocks are governed by anchor-dependent Boolean rules, with additional noise and bridge clauses.

877
878
879

Maximum independent set. The MIS targets are block-structured graph distributions. In *Clique-path*, blocks alternate between clique-like and path-like motifs, with sparse bridge edges between adjacent blocks. In *Motif-bridge*, each instance

Problem	Normalized quality
Coloring	$\chi(G)/k_{\text{alg}}$
MAXSAT	sat_{alg} / sat_{opt}
MIS	$\mid \text{IS}_{\text{alg}} \mid / \mid \text{IS}_{\text{opt}} \mid$
MDS	$\mid \text{DS}_{\text{opt}} \mid / \mid \text{DS}_{\text{alg}} \mid$
Packing LP, MDKP	obj_{alg} / obj_{opt}
TSP	len_{opt} / len_{alg}

Table 6. **Normalized quality metrics used for evaluation.** Higher is better, and optimal solutions have quality 1. For maximization problems, quality is the algorithm value divided by the optimum value; for minimization problems, it is the optimum value divided by the algorithm value. Invalid or infeasible outputs receive quality 0.

samples a latent sequence of motifs, including cliques, cycles, bicliques, and crown-like gadgets. Good solvers should decompose the graph into motifs and account for bridge conflicts.

Minimum dominating set. The MDS targets emphasize different coverage structures. In *Gateway-hub*, clusters contain hubs and gateways, where gateways provide overlapping coverage across neighboring clusters. In *Geometric-anchor*, vertices come from geometric clusters with heterogeneous density and connector edges. In *Star-kernel*, each cluster has a stable hub that dominates most local vertices, with sparse connectors between hubs. These targets test whether the solver can identify high-coverage vertices rather than relying only on generic degree heuristics.

Packing LP. The packing LP targets contain continuous packing problems with recurring resource structure. In *Block-coupled*, items belong to latent resource blocks, but a shared coupling resource limits the combination of otherwise attractive block-local items. In *Active-resource*, each instance has a hidden active-resource regime with recurring dual-price patterns. In *Single-bottleneck*, one resource is consistently much tighter than the others, so the LP optimum is largely governed by value per unit of the hidden bottleneck resource.

Multidimensional knapsack. The MDKP targets are integer packing problems with latent item and resource structure. In *Decoy-complement*, high-value decoy items look attractive locally but consume a scarce hidden resource, while complementary item classes combine better across resources. In *Latent-class*, items come from hidden resource-consumption classes, and the active bottleneck changes across instances. In *Single-resource*, one hidden resource is typically tight, making density with respect to that resource a strong but not always exact rule.

Traveling salesperson problem. The TSP targets contain Euclidean or Euclidean-like geometric structure. In *Clustered Euclidean*, cities are sampled from balanced clusters arranged around a ring, so good tours combine short intra-cluster paths with the correct inter-cluster order. In *Latent-metric*, each instance comes from one of several geometric regimes, including ring clusters, paired ribbons, and barrier-bridge layouts. This target tests whether the solver can classify the latent geometry and choose an appropriate tour construction strategy.

Together, these targets test the central hypothesis of the benchmark: sampled instances can reveal distribution-specific algorithmic structure, and a solver that identifies this structure can improve the quality–runtime tradeoff relative to generic heuristics and off-the-shelf optimization backends.

D.2. Evaluation Metrics

We evaluate each solver on held-out instances using three quantities: normalized quality, optimality rate, and runtime. The goal is to compare solvers across problem classes with different objective scales and directions. Some tasks are maximization problems, such as MAXSAT, MIS, Packing LP, and MDKP. Other tasks are minimization problems, such as Coloring, MDS, and TSP. We therefore convert all objective values to a common normalized quality score in $[0, 1]$, where larger is better. Table 6 summarizes the normalization used for each problem class.

Normalized quality. For each instance x, let $q_A(x)$ denote the normalized quality of solver A. For maximization problems, if $A(x)$ is feasible and the optimum value is positive, we set

$$q_A(x) = \frac{\text{val}_A(x)}{\text{val}_{\text{opt}}(x)}.$$

935 For minimization problems, if $A(x)$ is feasible and the returned cost is positive, we set
936
937
$$q_A(x) = \frac{\mathrm{cost}_{\mathrm{opt}}(x)}{\mathrm{cost}_A(x)}.$$
938
939
940 Thus $q_A(x) = 1$ means that the solver attains the optimum objective value, while $q_A(x) < 1$ measures the multiplicative
941 gap from optimum. Invalid, malformed, or infeasible outputs are assigned $q_A(x) = 0$. In the benchmark distributions used
942 here, optimum objective values are positive, so the normalizations in Table 6 are well-defined.
943
944 For Coloring, $\chi(G)$ is the chromatic number of the instance graph G, and k_{alg} is the number of colors used by the returned
945 coloring. For MAXSAT, $\mathrm{sat}_{\mathrm{alg}}$ is the number of clauses satisfied by the returned assignment, and $\mathrm{sat}_{\mathrm{opt}}$ is the maximum
946 satisfiable number of clauses. For MIS, $\mathrm{IS}_{\mathrm{alg}}$ and $\mathrm{IS}_{\mathrm{opt}}$ are the returned and maximum independent sets. For MDS, $\mathrm{DS}_{\mathrm{alg}}$
947 and $\mathrm{DS}_{\mathrm{opt}}$ are the returned and minimum dominating sets. For Packing LP and MDKP, $\mathrm{obj}_{\mathrm{alg}}$ and $\mathrm{obj}_{\mathrm{opt}}$ are the returned
948 and optimal objective values. For TSP, $\mathrm{len}_{\mathrm{alg}}$ is the length of the returned tour and $\mathrm{len}_{\mathrm{opt}}$ is the optimal tour length.

949 **Optimality rate.** The optimality rate measures the fraction of held-out instances on which a solver attains the optimum
950 value. Equivalently, for a test set $\mathcal{D}_{\mathrm{test}}$, we report
951
952
$$\frac{1}{|\mathcal{D}_{\mathrm{test}}|} \sum_{x \in \mathcal{D}_{\mathrm{test}}} \mathbf{1}\{q_A(x) = 1\}.$$
953
954
955 In practice, the equality check is implemented using the benchmark verifier and the stored optimal objective value, with
956 the appropriate numerical tolerance for continuous Packing LP objectives. Invalid or infeasible outputs are not counted as
957 optimal.
958
959 **Runtime.** Runtime is measured per instance in milliseconds and averaged over the held-out test set. The runtime includes
960 the deployed solver computation on the instance, including any verification, repair, fallback, or local-search steps used by
961 the selected generated program. It does not include the earlier synthesis process that produced the solver, since synthesis
962 is an offline training-time cost. When we report aggregate rows over several benchmark distributions, we first compute
963 the mean metric for each distribution and then average across distributions, so each distribution contributes equally to the
964 reported problem-level or overall aggregate.

965 **Aggregation protocol.** All reported aggregate quantities are computed over the 21 benchmark target distributions.
966 For each target distribution, we first average over held-out test instances and repeated runs to obtain a single quality,
967 optimality, feasibility, and runtime value for each method. Quality lifts are then computed at the target level, for example
968 $Q_{\mathrm{LLM}} - Q_{\mathrm{avg\ heur}}$ and $Q_{\mathrm{LLM}} - Q_{\mathrm{best\ heur}}$.
969
970 For additive quantities such as quality, optimality, feasibility, and quality lift, aggregate values are arithmetic means over the
971 completed target distributions. Runtime comparisons are aggregated as speedup ratios. For each completed target τ, we
972 compute
973
$$\frac{T_{\mathrm{baseline}}(\tau)}{T_{\mathrm{LLM}}(\tau)},$$
974
975 so that values larger than 1 indicate that the LLM-synthesized solver is faster. Because runtime ratios are multiplicative, we
976 aggregate them using geometric means, equivalently by averaging log-speedups and exponentiating. All aggregate speedups
977 are computed over the same 21 target distributions.
978
979
D.3. Baseline Catalog and Reporting Protocol
980
981 The main text summarizes the baseline families used in our experiments. Here we give the complete per-problem baseline
982 catalog used in the benchmark. Each entry specifies the problem class, solver family, backend, category, aliases when
983 applicable, and implementation notes. The benchmark protocol also specifies the execution details, including formulations,
984 solver wrappers, time limits, thread settings, external-solver requirements, and alias mappings.
985
986 Table 7 lists all baseline entries used by the benchmark. The catalog contains 78 registered baseline entries across the seven
987 problem classes, corresponding to 72 unique implementations after removing registry aliases. We nevertheless list aliases
988 explicitly because they are executable entries in the registry. In particular, several problems include an `exact` entry that
989 aliases a problem-local exact implementation, such as an OR-Tools formulation or the Held–Karp dynamic program.

Problem	Heuristic baselines	Solver-backed and exact baselines
Coloring	DSATUR and greedy coloring variants, including largest-degree, random-order, and smallest-last orderings (Brélaz, 1979)	OR-Tools CP-SAT coloring (Perron & Furnon, 2025; Perron et al., 2023); `exact` alias; DSATUR branch-and-bound coloring (Brélaz, 1979; Land & Doig, 1960); Gurobi timed coloring (Gurobi Optimization, LLC, 2026); HiGHS coloring MIP (HiGHS Development Team, 2024; Huangfu & Hall, 2018); PySAT SAT-based coloring; SCIP coloring MIP (Bestuzheva et al., 2023)
MaxSAT	Greedy flip local search; literal-majority polarity assignment; random Boolean assignment	EvalMaxSAT (Avellaneda, 2020); Gurobi timed MaxSAT (Gurobi Optimization, LLC, 2026); MaxHS (Davies & Bacchus, 2013a;b); Open-WBO (Martins et al., 2014); PySAT RC2 with CaDiCaL 1.9.5, Glucose4, MiniSat 2.2, and the default RC2 backend (Ignatiev et al., 2019); UWrMaxSAT (Piotrów, 2020); WMaxCDCL (Coll et al., 2025)
MDKP	LP-relaxation rounding, redundancy-improved greedy, and value-density greedy heuristics (Dantzig, 1957)	Custom MDKP branch-and-bound (Land & Doig, 1960); CBC MDKP MIP (Forrest & Lougee-Heimer, 2005); OR-Tools CP-SAT MDKP (Perron & Furnon, 2025; Perron et al., 2023); `exact` alias; Gurobi timed MDKP (Gurobi Optimization, LLC, 2026); HiGHS MDKP MIP (HiGHS Development Team, 2024; Huangfu & Hall, 2018); SCIP MDKP MIP (Bestuzheva et al., 2023)
MDS	High-degree, marginal-gain, and redundancy-aware greedy dominating-set heuristics (Chvátal, 1979)	CBC MDS MIP (Forrest & Lougee-Heimer, 2005); OR-Tools CP-SAT MDS (Perron & Furnon, 2025; Perron et al., 2023); `exact` alias; Gurobi timed MDS (Gurobi Optimization, LLC, 2026); HiGHS MDS MIP (HiGHS Development Team, 2024; Huangfu & Hall, 2018); SCIP MDS MIP (Bestuzheva et al., 2023); set-cover branch-and-bound exact MDS (Chvátal, 1979; Land & Doig, 1960)
MIS	Local-improvement, minimum-degree, random greedy, and ratio-based greedy MIS heuristics	Clique branch-and-bound exact MIS (Land & Doig, 1960); OR-Tools CP-SAT MIS (Perron & Furnon, 2025; Perron et al., 2023); `exact` alias; Gurobi timed MIS (Gurobi Optimization, LLC, 2026); HiGHS MIS MIP (HiGHS Development Team, 2024; Huangfu & Hall, 2018); KaMIS exact via vertex cover (Lamm et al., 2019); SCIP MIS MIP (Bestuzheva et al., 2023)
Packing LP	Density-based fractional packing and uniform-fraction LP heuristics (Dantzig, 1957)	CLP LP (Forrest et al., 2022); OR-Tools GLOP simplex (Perron & Furnon, 2025); `exact` alias; Gurobi timed LP (Gurobi Optimization, LLC, 2026); HiGHS simplex and interior-point LP (HiGHS Development Team, 2024; Huangfu & Hall, 2018); SCIP LP (Bestuzheva et al., 2023)
Euclidean TSP	Random tour; nearest neighbor; nearest insertion; farthest insertion; multi-start two-opt; nearest-neighbor plus two-opt; farthest-insertion plus two-opt (Lin, 1965; Rosenkrantz et al., 1977); LKH (Helsgaun, 2000)	CBC MTZ TSP MIP (Forrest & Lougee-Heimer, 2005); Concorde (Applegate et al., 2006); OR-Tools CP-SAT TSP (Perron & Furnon, 2025; Perron et al., 2023); Held–Karp dynamic programming (Held & Karp, 1962); `exact` alias; Gurobi timed TSP (Gurobi Optimization, LLC, 2026); SCIP MTZ TSP MIP (Bestuzheva et al., 2023)

Table 7. **Baseline catalog used in the benchmark registry.** The catalog includes lightweight problem-specific heuristics, time-limited solver-backed baselines, and exact or certifying backends when available. The `exact` entries are registry aliases for problem-local exact implementations. Citations for heuristic variants refer to the closest classical heuristic family or solver framework; exact implementation details are defined in the registry.

We distinguish three categories of baselines. *Heuristic* baselines are problem-specific procedures implemented locally in the benchmark, such as greedy, local-search, insertion, rounding, and randomized rules. *Solver-backed* baselines are calls to general-purpose optimization software under the benchmark protocol. In particular, Gurobi is reported separately because it is run with a fixed 10-second, single-thread budget and may return an incumbent solution rather than a certified optimum. *Exact* and *external exact* baselines are used as certified exact comparisons when they return certified optima under the evaluation protocol.

In the aggregate result tables, "best heuristic" denotes the strongest heuristic baseline for that problem, "Gurobi (10s)" denotes the time-limited Gurobi baseline, and "best exact" denotes the fastest certified exact solver available for that problem. All baselines are evaluated under the same public-instance protocol as the LLM synthesis agent. The baselines are given the benchmark instances but not direct access to the hidden family-generation rule.

D.4. Compute and API Usage

Solver synthesis was performed with GPT-5.2 Thinking through remote API calls. We used fixed high reasoning effort and structured outputs. Temperature, top-p, and maximum output length were left at service defaults. Across the 21 target distributions, synthesis used 463 successful API calls and 16.1M total tokens, where total tokens are prompt plus completion tokens. This amounts to 22.0 calls and 767K tokens per distribution on average. Of the 16.1M total tokens, 5.95M were prompt tokens and 10.16M were completion tokens; the recorded 8.05M reasoning tokens are a subset of the completion tokens.

All generated solvers and benchmark baselines were executed locally on CPUs. No local GPU inference, GPU-backed

Quantity	Value
Target distributions	21
Total API calls	463
Mean API calls per distribution	22.0
Prompt tokens	5.95M
Completion tokens	10.16M
Reasoning tokens	8.05M
Total tokens	16.11M
Mean tokens per distribution	767K
Aggregate synthesis-stage time	73.4 h
Aggregate baseline pre-synthesis time	417.3 h
Additional local work inside synthesis	0.6 h
CPU	$2\times$ AMD EPYC 9554
Physical / logical CPUs	128/256
RAM	1.1 TiB
Stored benchmark artifacts	83 GB
GPU training or fine-tuning	None

Table 8. **Compute summary.** API usage is for solver synthesis. Timing values are aggregate recorded per-target stage times. Generated solvers and baselines were evaluated locally on CPUs.

solver, model training, or fine-tuning was used. Experiments were run on a dual-socket AMD EPYC 9554 machine with 128 physical cores, 256 logical CPUs, and approximately 1.1TiB of RAM. The main benchmark was parallelized across target distributions. Summed over target distributions, the recorded synthesis-stage time was 73.4 hours, while baseline pre-synthesis evaluation took 417.3 hours. Additional local work inside synthesis, including analysis execution, solver construction, training evaluation, and validation evaluation, took approximately 0.6 hours. These times are aggregate recorded per-target stage times rather than raw calendar elapsed time.

E. Additional Empirical Results

E.1. Full Per-Target Results

Table 9 reports the full held-out test results for all 21 benchmark target distributions. Unlike the headline summary in Table 1, this table gives the per-target normalized quality, optimality rate, and mean wall-clock runtime for each baseline group. Runtime is reported in milliseconds.

For each target distribution, "Quality-best heuristic" denotes the heuristic baseline with the highest normalized quality, breaking ties by optimality rate and then runtime. "Avg. heuristic" averages over all heuristic baselines for that target. "Time-limited exact" denotes the strongest exact-formulation or certifying backend under the benchmark time limit, selected by the same quality-first rule. The aggregate statistics in Table 1 are computed from these 21 target-level results.

E.2. PACE 2025 Dominating Set Comparison

We also evaluate on instances from the PACE 2025 Dominating Set heuristic track. The task is to output a dominating set in an undirected graph, with smaller solution size corresponding to higher quality. PACE released a public set of 100 heuristic instances and, after the competition, the private evaluation set of 100 instances. The organizers state that the private instances are similar to the public instances. We therefore use the public set as our development distribution, split into 50 training instances and 50 validation instances, and report final results on the released private set of 100 instances. We emphasize that this is a local comparison against released PACE solvers and instances, not an official PACE score.

The private test graphs are large and sparse. Table 10 summarizes their sizes. The graphs range from tens of thousands to several million vertices, with median size about 6.5×10^5 vertices and 9.4×10^5 edges.

For each test instance, we run our synthesized solver and compare it with the released PACE heuristic solvers Fontanf, Root, Shadoks, and Swats. We verify validity of every returned solution. Since this is a minimization problem, a solver has better quality when it returns a smaller dominating set. Runtime is measured locally. Root uses the parallel rerun for instances `private_heuristic_001–046` and the original completed run for `private_heuristic_047–100`.

		Quality-best heuristic			Avg. heuristic			Gurobi (10s)			Time-limited exact			**LLM synthesis**		
Problem	Target distribution	Q	opt.	T (ms)	Q	opt.	T (ms)	Q	opt.	T (ms)	Q	opt.	T (ms)	Q	opt.	T (ms)
Coloring	Ring-template	0.746	0.002	3698.89	0.655	0.001	3634.70	0.976	0.882	4136.04	1.000	1.000	31.13	1.000	1.000	0.21
	Overlapping-palette	0.747	0.004	3476.27	0.651	0.001	3587.13	0.878	0.392	8365.97	1.000	1.000	79.93	0.804	0.022	24.25
	Separator-trap	0.749	0.000	3615.68	0.648	0.000	3587.68	0.933	0.666	6876.72	1.000	1.000	98.68	0.800	0.000	3.91
MAXSAT	Community-parity	0.914	0.000	3732.89	0.862	0.000	4367.53	1.000	0.992	8711.96	1.000	1.000	26.38	1.000	0.876	23.81
	Last-clause signal	0.932	0.000	3645.47	0.880	0.000	4491.54	1.000	1.000	8325.44	1.000	1.000	7.76	1.000	1.000	5.58
	Latent-backdoor	0.931	0.000	3791.20	0.891	0.000	3614.69	1.000	0.996	2461.24	1.000	1.000	27.02	1.000	0.896	37.80
MIS	Clique-path	0.876	0.040	10000.00	0.833	0.018	10000.00	1.000	1.000	2404.74	1.000	1.000	1355.27	0.991	0.594	15.29
	Core-fringe trap	0.908	0.908	10000.00	0.664	0.451	10000.00	1.000	1.000	1555.43	1.000	1.000	144.56	1.000	1.000	7.67
	Motif-bridge	0.874	0.044	10000.00	0.826	0.018	10000.00	0.998	0.890	6756.90	0.994	0.994	2136.32	0.986	0.384	57.00
MDS	Gateway-hub	0.839	0.000	10000.00	0.811	0.000	10000.00	1.000	1.000	8098.52	1.000	1.000	112.77	0.920	0.000	5.46
	Geometric-anchor	0.784	0.298	8706.31	0.748	0.199	8725.88	1.000	1.000	3082.80	1.000	1.000	2093.16	1.000	1.000	129.61
	Star-kernel	0.930	0.930	10000.00	0.918	0.918	10000.00	1.000	1.000	8162.42	1.000	1.000	52.26	1.000	1.000	3.31
Packing LP	Block-coupled	0.792	0.000	10000.00	0.718	0.000	10000.00	1.000	1.000	8627.91	1.000	1.000	125.33	1.000	1.000	1.51
	Active-resource	0.610	0.000	10000.00	0.572	0.000	10000.00	1.000	1.000	12675.99	1.000	1.000	124.13	0.981	0.000	17.49
	Single-bottleneck	0.803	0.000	10000.00	0.788	0.000	10000.00	1.000	1.000	7635.97	1.000	1.000	121.99	1.000	1.000	1.40
MDKP	Decoy-complement	0.967	0.016	5329.50	0.558	0.005	6498.60	1.000	1.000	2498.77	1.000	1.000	1083.96	0.951	0.000	40.54
	Latent-class	0.966	0.010	3660.12	0.867	0.003	3792.81	1.000	0.746	4955.21	1.000	0.544	6306.12	0.968	0.000	116.31
	Single-resource	0.960	0.666	4263.46	0.851	0.222	5585.39	1.000	1.000	2593.35	1.000	1.000	232.50	1.000	1.000	176.39
TSP	Clustered Euclidean	1.000	0.910	1543.61	0.653	0.114	8942.95	1.000	0.930	1552.81	1.000	0.662	135.53	0.987	0.002	66.68
	Latent-metric	1.000	0.848	1490.05	0.657	0.228	8936.26	1.000	0.966	1565.87	1.000	0.758	117.86	0.992	0.306	232.10
	Paired-ribbon zigzag	1.000	1.000	51.39	0.626	0.402	8756.42	1.000	1.000	2071.28	0.499	0.000	10992.61	1.000	1.000	0.23

Table 9. **Per-target held-out test results for all 21 benchmark target distributions.** Here Q denotes normalized quality, scaled so that higher is better; opt. is the fraction of instances solved to optimality; and T is average wall-clock runtime per instance in milliseconds. "Quality-best heuristic" selects, within each target, the heuristic with highest normalized quality, breaking ties by optimality and then runtime. "Avg. heuristic" reports the arithmetic average over the heuristic baselines for that target. For the two heuristic columns, reported runtimes are clipped at 10,000 ms before averaging or reporting. "Time-limited exact" selects the strongest exact backend under the benchmark time limit using the same quality-first rule. The Gurobi and exact backends are run with a 10 second solver limit, but the reported T values are measured wall-clock runtimes and may include wrapper overhead. The headline speedups in Table 1 are computed from these per-target runtime ratios using geometric means.

Quantity	Min.	25th perc.	Median	75th perc.	Max.
Number of vertices $\lvert V\rvert$	18,046	21,400	653,867	1,261,963	4,221,675
Number of 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

Table 10. **PACE private-test graph statistics.** Statistics are computed over the 100 released private Dominating Set heuristic instances used for evaluation.

The comparison shows a clear speed–quality tradeoff. The synthesized solver is not competitive with the strongest PACE submissions in solution quality: every valid PACE baseline returns a smaller dominating set on every matched instance. However, the gap is moderate in aggregate. Against Fontanf, Root, and Shadoks, the agent's total solution size is about 3.3% larger, while its average runtime is roughly $99\times$–$109\times$ smaller. Thus, on this benchmark the synthesized solver discovers a very fast distribution-specific heuristic, but not one that matches the quality of highly engineered competition solvers.

E.3. Distributional Algorithmic Complexity

This appendix explains the computational mechanisms behind the generated solvers in Table 12. The comparison is distributional. We do not claim that the generated programs improve the worst-case complexity of MaxSAT, graph coloring, minimum dominating set, maximum independent set, Packing LP, multidimensional knapsack, or TSP as abstract problem classes. Instead, the generated programs exploit regularities of the benchmark distributions and replace an ambient generic computation by a smaller distribution-specific computation on instances drawn from the same distribution.

The central contrast is between an ambient solver and a distribution-aware solver. A generic exact baseline must remain valid for arbitrary instances of the problem class. It therefore searches over color assignments for graph coloring, Boolean assignments for MaxSAT, vertex subsets for MIS and MDS, multidimensional item subsets for MDKP, or tours for TSP. For Packing LP, the ambient problem is polynomial-time solvable, but a generic LP backend still solves the full N-variable, r-constraint linear program. By contrast, a generated solver is selected after seeing samples from a fixed distribution. When it identifies reusable structure, it can verify a template, seed a local search, restrict a candidate set, use a surrogate score, or

Table 11. **PACE 2025 Dominating Set comparison.** Lower solution size and lower runtime are better. The synthesized solver is valid on all instances and is roughly two orders of magnitude faster than the PACE baselines, but the PACE solvers return consistently smaller dominating sets. Ratios are computed on matched verified-valid instances.

Solver	Valid	Total size ↓	Avg. size ↓	Agent/solver size	Avg. runtime ↓	Agent speedup	Quality wins vs. agent
Agent	100/100	23.16M	231,594.7	1.000	2.89s	1.0×	–
Fontanf	100/100	22.41M	224,103.9	1.033	286.01s	98.8×	100/100
Root	100/100	22.41M	224,123.7	1.033	300.37s	103.8×	100/100
Shadoks	100/100	22.43M	224,275.6	1.033	315.89s	109.2×	100/100
Swats	75/100	15.77M	210,235.7	1.028	287.41s	130.2×	75/75

Table 12. Distribution-aware computation for all benchmark distributions.

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

construct a small set of structured candidates. This changes the effective deployed computation, even though it does not change the worst-case complexity of the ambient problem.

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

The symbols B, R, P, F, S, and K in Table 12 denote fixed implementation budgets: repair steps, restarts, local-search rounds, coordinate frames, candidate families, pricing variants, improvement passes, or bounded candidate pools. These quantities are fixed by the generated program or by the benchmark analysis file, not by the asymptotic input size. We keep them visible because they identify the bounded computation that replaces the ambient search.

More specifically, B_{rep} is the bounded coloring-repair budget; R_{ds} is the number of DSATUR-style restarts; P_{elim} is the number of color-elimination passes; and B_{recolor} is the capped local recoloring budget. For MaxSAT, R_{ws} is the number of WalkSAT-style restarts, B_{flip} is the flip budget per restart, and B_{patch} is the bounded patch-search budget. For MIS, P_{ord} is the number of vertex-ordering rules, P_{gr} is the number of greedy candidate constructions, B_{loc} is the local-search budget, B_{kick} is the number of kick-repair attempts, and T_{ARW} and T_{tiny} denote bounded ARW-style and tiny time-limited

1210 improvement subroutines. In the Core-fringe trap row, t_j is the size of residual fringe component j, k is the number of core
1211 choices considered, and q is the number of candidate configurations scored.
1212
1213 For MDS, $T_{\text{heap-greedy}}$, T_{greedy}, and T_{prune} denote the bounded heap-greedy, greedy-completion, and pruning subroutines;
1214 B_{geo} is the bounded geometric-anchor completion budget. In the Star-kernel row, t_j is the number of residual target vertices
1215 in component j, and c_j is the number of candidate cover vertices for that component. For Packing LP, P_{LP} is the number
1216 of density or active-resource rules evaluated. For MDKP, P_{score} is the number of surrogate scoring rules, P_{price} is the
1217 number of resource-price vectors, T_{iter} is one bounded price-update and repair iteration, K_{cand} is the restricted candidate-set
1218 size, and C_b is the effective one-dimensional bottleneck capacity used by the guarded DP. For TSP, B_{cand} and B_{full} are
1219 bounded candidate-2-opt and full-2-opt budgets, F is the number of coordinate frames, k is the number of stripes or groups
1220 in the latent-metric construction, S is the number of selected stripe frames, P_{imp} is the number of candidate tours sent to
1221 improvement, B_{imp} is the improvement budget per candidate, and B_{2opt} is the bounded 2-opt budget in the paired-ribbon
1222 fallback.

1223 For Coloring, the DSATUR implementation used by the generated solvers scans all uncolored vertices at each step. Thus
1224
1225 $$T_{\text{DSATUR}}(n, m, \kappa) = O(n^2 + m + n\kappa),$$
1226
1227 and $T_{\text{DSATUR}}(n, m, \kappa) = O(n^2 + m)$ when $\kappa \leq n$. This is the main polynomial term replacing the ambient $O^*(\kappa^n)$ search
1228 in the DSATUR-based coloring rows.

1229 **Graph coloring.** A generic exact coloring solver must handle arbitrary graphs and therefore faces an ambient $O^*(\kappa^n)$
1230 assignment space, even if practical solvers use pruning, SAT encodings, or branch-and-bound. The generated solvers exploit
1231 the fact that the benchmark graphs are not arbitrary.
1232
1233 In the Ring-template distribution, the dominant fast path is template checking: the solver verifies a learned coloring
1234 pattern against the edge set in $O(m)$. If this path fails or leaves conflicts, the solver uses bounded repair or falls back to
1235 a polynomial-time greedy coloring routine, summarized in Table 12 by $O(n^2 + m + B_{\text{rep}}\Delta_{\max})$. The gain is that most
1236 instances can be handled by verification and bounded repair rather than by color-assignment search.

1237 In the Overlapping-palette and Separator-trap distributions, the selected solvers do not reduce to pure template checking.
1238 They run a bounded number of DSATUR-style attempts and cleanup passes. For Overlapping-palette, the cleanup includes
1239 bounded color-elimination passes, yielding
1240
1241 $$O(n + m + R_{\text{ds}}(n^2 + m) + P_{\text{elim}}\kappa(n + m)).$$
1242
1243 For Separator-trap, the cleanup is capped recoloring repair, yielding
1244
1245 $$O(n + m + R_{\text{ds}}(n^2 + m) + B_{\text{recolor}}).$$
1246
1247 These rows are bounded polynomial-time heuristics rather than exact reductions, but they still replace ambient exponential
1248 search by a small number of structured attempts.

1249 **MaxSAT.** For MaxSAT, a generic exact backend must reason over the full 2^v Boolean assignment space and certify
1250 optimality without assuming planted structure. The generated solvers instead use distributional statistics to construct strong
1251 initial assignments, then run bounded local repair. Building occurrence lists, initializing scores, and evaluating the seed
1252 costs $O(|F|\ell)$. A flip updates clauses incident to the flipped variable, which costs $O(\ell\Delta_{\text{occ}})$ with maintained clause counts.
1253
1254 The Community-parity and Latent-backdoor rows therefore have dominant cost
1255
1256 $$O(|F|\ell + R_{\text{ws}}B_{\text{flip}}\ell\Delta_{\text{occ}}),$$
1257
1258 where R_{ws} is the number of restarts and B_{flip} is the flip budget per restart. The Last-clause signal row adds a bounded patch
1259 term,
1260
$$O(B_{\text{patch}}\ell\Delta_{\text{occ}}),$$
1261
1262 because it explicitly repairs variables around remaining unsatisfied clauses. In all three cases, the computational advantage
1263 is not a new MaxSAT algorithm for arbitrary formulas. The advantage is that the distributional seed moves the solver close
1264 enough to a high-quality assignment that bounded local search is often sufficient.

Maximum independent set. For MIS, an exact generic solver must handle arbitrary vertex subsets, giving an ambient $O^*(2^n)$ search space. The selected generated solvers are not certified exact MIS algorithms. They build small pools of candidate independent sets using orderings or decompositions suggested by the distribution, then apply bounded local improvement or small residual enumeration.

In the Clique-path distribution, the solver constructs candidates from analysis-guided vertex orders, greedy maximal independent sets, randomized restarts, and bounded ARW-style local improvement. This gives

$$O(n + m + P_{\text{ord}} n \log n + P_{\text{gr}}(n + m) + T_{\text{ARW}}).$$

The benefit is that the solver spends its budget on a small set of distributionally promising independent sets rather than on the full subset space.

In the Core-fringe trap distribution, the solver uses the high-degree core and low-degree fringe structure. On the structured path, it decomposes the fringe into tiny components, enumerates exact choices inside each component, and scores a bounded set of candidate core decisions. This gives

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

where t_j is the size of residual fringe component j, k is the number of core choices considered, and q is the number of candidate configurations scored. The exponential term is over tiny residual pieces, not over the full graph.

In the Motif-bridge distribution, the solver uses multiple greedy constructions, bounded local improvement, bounded kick moves, and an optional tiny time-limited exact improvement step:

$$O(n + m + P_{\text{gr}}(n + m) + B_{\text{loc}}(n + m) + B_{\text{kick}}(n + m) + T_{\text{tiny}}).$$

This is again a bounded heuristic, but its search is restricted to a small set of distributionally meaningful candidates.

Minimum dominating set. For MDS, a generic exact solver faces an ambient $O^*(2^n)$ subset-selection problem. The generated solvers exploit distribution-specific structure in how domination is achieved.

In Gateway-hub, the main computation is graph processing, greedy completion, and redundant-vertex pruning:

$$O(n + m + T_{\text{heap-greedy}} + T_{\text{prune}}).$$

The advantage is that high-degree hubs and leaf or isolate structure make much of the dominating set apparent without enumerating subsets.

In Geometric-anchor, the fast path is $O(n + m)$, because the solver checks a small anchor pattern induced by a residue-class structure. If the pattern fails, it uses bounded greedy completion, giving

$$O((1 + B_{\text{geo}})(n + m)).$$

In Star-kernel, the solver uses forced leaf-neighbor structure, then solves only tiny residual components. The residual exact work is

$$\sum_j c_j 2^{t_j},$$

where t_j is the number of residual target vertices in component j, and c_j is the number of candidate cover vertices relevant to that component. This is exponential only in the tiny residual size, not in the original graph size n.

Packing LP. Packing LP is different from the discrete NP-hard families. The ambient problem is already polynomial-time solvable, so the comparison is not exponential versus polynomial. The comparison is between a generic LP solve, $T_{\text{LP}}(N, r, L)$, and a specialized sorting-based computation. The generated solvers exploit the fact that the benchmark LP distributions often have a small number of dominant or active resources.

In the Block-coupled and Single-bottleneck distributions, the solver identifies a bottleneck or effective resource score and sorts items by density, giving

$$O(Nr + N \log N).$$

1320 The Nr term comes from evaluating feasibility and resource usage across constraints, while $N \log N$ comes from sorting.
1321 In Active-resource, the solver evaluates a small portfolio of P_{LP} density rules, giving
1322
1323
$$O(Nr + P_{\text{LP}}(Nr + N \log N)).$$

1324 The advantage over $T_{\text{LP}}(N, r, L)$ is not a worst-case complexity improvement in the abstract LP model, but a deployment
1325 advantage on these distributions: a small portfolio of density rules captures the active-resource structure without invoking a
1326 full general-purpose optimizer.
1327

1328 **Multidimensional knapsack.** For MDKP, a generic exact baseline is a MIP or branch-and-bound solver over 2^N possible
1329 item subsets. The generated solvers replace this ambient search by surrogate scoring, candidate restriction, bounded dynamic
1330 programming, add/drop improvement, and repair.

1331 In Decoy-complement MDKP, the solver evaluates a small portfolio of surrogate scores, greedily packs feasible candidates,
1332 applies bounded add/drop improvement, and repairs violations:
1333
1334
$$O(Nr + P_{\text{score}}(Nr + N \log N) + T_{\text{add/drop}} + T_{\text{repair}}).$$
1335

1336 In Latent-class MDKP, the solver uses adaptive price vectors and iterated surrogate packing:
1337
1338
$$O(Nr + N \log N + P_{\text{price}} T_{\text{iter}}(Nr + N \log N + T_{\text{repair}})).$$

1339 In Single-resource MDKP, the solver identifies a dominant resource, filters or sorts items, and runs one-dimensional DP on a
1340 bounded candidate set:
1341
$$O(Nr + N \log N + K_{\text{cand}} C_b + T_{\text{repair}}),$$
1342

1343 where K_{cand} is the candidate-set size and C_b is the effective one-dimensional capacity used by the guarded DP. These rows
1344 are bounded heuristics or restricted dynamic programs rather than exact reductions for general MDKP. Their advantage is
1345 that they search over a small number of distributionally meaningful item orders or candidates instead of exploring the full
1346 subset space.

1347 **Traveling salesman problem.** For TSP, a generic exact dynamic program such as Held-Karp has cost $O(n^2 2^n)$, and exact
1348 branch-and-bound solvers must still be prepared for arbitrary tour structure. The generated solvers do not implement exact
1349 TSP dynamic programming. They exploit geometric structure to generate a small set of candidate tours and then improve
1350 them by bounded local search.

1351 In Clustered Euclidean TSP, the solver builds geometric or nearest-neighbor candidate tours and candidate neighborhoods.
1352 Sorting neighborhoods contributes the $O(n^2 \log n)$ term, while bounded candidate 2-opt and bounded full 2-opt cleanup
1353 contribute quadratic improvement terms:
1354
1355
$$O(n^2 \log n + B_{\text{cand}} n^2 + B_{\text{full}} n^2).$$
1356

1357 Thus the effective computation is structured candidate generation plus bounded quadratic local improvement, rather than
1358 exponential tour search.

1359 In Latent-metric TSP, the solver builds several structured candidate families. The term $F(kn^2 + n \log n)$ comes from
1360 evaluating F coordinate frames and computing stripe or projection structure with k groups. The term $S(2^k k + kn)$ comes
1361 from using the top S stripe frames and solving the small stripe-order subproblem over k stripes. Since k is fixed and small
1362 in the selected program, this is not an exponential dependence on the number of cities. Finally, $P_{\text{imp}} B_{\text{imp}} n^2$ summarizes
1363 improving at most P_{imp} candidate tours for a bounded number B_{imp} of local improvement passes:
1364
1365
$$O(n^2 + F(kn^2 + n \log n) + S(2^k k + kn) + P_{\text{imp}} B_{\text{imp}} n^2).$$
1366

1367 In Paired-ribbon zigzag TSP, the solver has a fast structured path based on the two-rail geometry of the distribution. It
1368 detects the rail structure, sorts points along the rails, and pairs endpoints, giving
1369
1370
$$O(n \log n).$$
1371

1372 If the structured path is not used, it falls back to nearest-neighbor construction and bounded 2-opt improvement:
1373
1374
$$O(n^2 + B_{\text{2opt}} n^2).$$

Problem	Target distribution	Q_0	Q_{64}	$\Delta Q\times10^3$	T_0 (ms)	T_{64} (ms)	T_0/T_{64}
Coloring	Cluster-ring mix	1.0000	1.0000	+0.00	0.101	0.072	1.40×
Coloring	Planted-palette overlap	1.0000	0.9999	−0.08	0.076	0.077	0.99×
Coloring	Separator-palette trap	1.0000	1.0000	+0.00	0.603	0.097	6.23×
MDS	Gateway-overlap cover	1.0000	1.0000	+0.00	0.103	0.102	1.00×
MDS	Geometric-cluster cover	1.0000	0.9997	−0.33	0.130	0.104	1.26×
MDS	Star-cluster cover	1.0000	1.0000	+0.00	0.086	0.080	1.08×
MDKP	Decoy-complement	0.9990	0.9995	+0.57	90.792	251.347	0.36×
MDKP	Latent-class	0.9993	0.9992	−0.07	116.035	97.130	1.19×
MDKP	Single-resource	1.0000	1.0000	+0.00	7.916	0.220	35.95×
TSP	Clustered Euclidean	1.0000	1.0000	+0.00	6.082	1.601	3.80×
TSP	Latent-metric	1.0000	1.0000	+0.00	6.113	4.312	1.42×
TSP	Paired-ribbon zigzag	1.0000	1.0000	+0.00	0.652	0.037	17.53×

Table 13. **Zero-sample versus sample-conditioned synthesis.** The zero-sample variant uses no public training instances. The sample-conditioned variant uses the default training size, $n = 64$. We report mean normalized quality Q and mean runtime T in milliseconds over 10 paired synthesis seeds. The speedup column is the ratio of mean runtimes, T_0/T_{64}; values above $1\times$ indicate that using samples produced a faster deployed solver.

The ambient comparison remains $O(n^2 2^n)$, since a generic exact TSP solver must remain valid for arbitrary instances. The gain is therefore distributional: for instances matching the ribbon structure, the generated solver avoids general tour search and most quadratic local-search work.

Summary. Across the benchmark families, the generated solvers win when the training samples reveal reusable low-dimensional, local, or template structure. Coloring can become template verification plus bounded repair. MaxSAT can become seeded local search. MIS and MDS can become greedy construction plus tiny residual enumeration or bounded local improvement. Packing LP can become density sorting over active resources. MDKP can become surrogate pricing, restricted one-dimensional DP, or bounded repair. TSP can become structured candidate generation plus bounded quadratic local search, and in the paired-ribbon case even a near-sorting computation on the fast path.

The limitation is equally important. These programs do not establish new worst-case algorithms for the ambient problem classes. Several rows are bounded heuristics, and some include fallback or repair. The faithful claim is distributional and mechanistic: on these benchmark distributions, the generated programs often replace generic exact optimization over a large ambient space by the smaller computations summarized in Table 12.

E.4. Zero-Sample versus Sample-Conditioned Synthesis

We use a focused ablation to test whether public samples help the synthesis procedure compile better deployment solvers. The zero-sample variant uses the same synthesis pipeline but provides no public training instances. The sample-conditioned variant uses $n = 64$ public training instances, matching the default setting in our main experiments rather than selecting the best sample size. Results are averaged over 10 paired synthesis seeds, so each row compares the two variants under matched randomness.

Table 13 shows that normalized quality is near-saturated in both variants. Thus, in these target distributions, the main observable effect of sample access is not feasibility or final quality, but deployment runtime. Sample-conditioned synthesis is at least as fast on $10/12$ targets and strictly faster on $9/12$, with large gains on Single-resource MDKP, Separator-palette Coloring, and Paired-ribbon zigzag TSP. The only substantial regression occurs on Decoy-complement MDKP, where the sample-conditioned solver is slower despite a small quality gain.

We interpret this experiment as evidence for sample-conditioned compilation: public samples often help the synthesis process discover faster deployment code, even when zero-sample synthesis already finds high-quality solutions. The effect is distribution-dependent, however, rather than a monotone consequence of providing more samples.

1430
1431
1432
1433
1434
1435
1436
1437
1438
1439

Problem family	Q_{LLM}	T_{LLM} (ms)	Shortcut rate	Fallback rate	Residual size	Repair iters.
Coloring	0.868	14.6	67.4%	0.0%	0.00	0.00
MAXSAT	1.000	55.9	88.1%	7.4%	26.24	0.00
MIS	0.992	32.1	66.7%	0.0%	235.92	0.33
MDS	0.973	75.6	86.4%	13.6%	441.72	0.00
Packing LP	0.994	32.0	100.0%	0.0%	346.78	0.00
MDKP	0.973	123.5	100.0%	0.0%	321.03	2.67
TSP	0.993	100.3	100.0%	0.0%	906.74	0.00
All 21 distributions	**0.971**	**62.0**	**86.9%**	**3.0%**	**325.49**	**0.43**

1440
1441
1442
1443
1444
1445
1446
1447

Table 14. **Diagnostic traces for the synthesized solvers over held-out test instances.** Q_{LLM} is normalized quality and T_{LLM} is average wall-clock runtime per instance in milliseconds. The columns use a common logging interface, but their concrete interpretation is solver-dependent. Shortcut rate is the fraction of instances on which the solver used its learned fast path, which may correspond to different computations in different problem families. Fallback rate is the fraction routed to a generic safety routine when such a routine is present. Residual size is the solver-reported size of the remaining subproblem after specialization and is meaningful only within a problem family. Repair iters. is the average number of bounded local repair iterations. Values are first averaged within each target distribution and then averaged across the three targets in each problem family. The final row averages over all 21 target distributions.

1448
1449

E.5. Diagnostic Traces for Synthesized Solvers

1450
1451
1452
1453
1454

The preceding section summarized the distribution-specific computations used by the generated solvers. These computations are heterogeneous: depending on the problem family and selected program, a synthesized solver may verify a template, seed a local search, restrict a candidate set, solve a small residual subproblem, sort by an active-resource score, construct a structured tour, or invoke bounded repair or fallback. We therefore record diagnostic traces during held-out evaluation to understand which parts of the synthesized computation are actually used at test time.

1455
1456
1457
1458
1459
1460
1461
1462
1463
1464

The diagnostics use a common logging interface, but they should not be read as saying that every solver implements the same mechanisms. A *shortcut* denotes the solver's learned fast path, whose concrete meaning is problem-dependent. For example, it may correspond to template checking in Coloring, seeded assignment construction in MAXSAT, density sorting in Packing LP, surrogate scoring or restricted dynamic programming in MDKP, or structured candidate generation in TSP. A *fallback* records whether the solver routed the instance to a generic safety routine, when such a routine is present. *Residual size* records the solver-reported size of the remaining subproblem after applying the learned specialization, when the generated solver creates such a residual. Its scale depends on the native problem representation and should be interpreted within, not across, problem families. Finally, *repair iterations* records the amount of bounded local repair used after the initial construction or reduction.

1465
1466
1467
1468
1469

These traces are not intervention ablations: we do not disable shortcutting, fallback, residual solving, or repair and rerun the benchmark. Instead, they provide a behavioral diagnosis of the deployed synthesized solver under the same held-out evaluation protocol used for the headline quality and runtime results. For each target distribution, we average the diagnostic traces over held-out test instances. Family rows average over the three target distributions in that problem family, and the final row averages over all 21 target distributions.

1470
1471
1472
1473
1474
1475
1476
1477

Table 14 should therefore be read as a mechanism-use diagnostic, not as evidence that every synthesized solver uses the same algorithmic components. The dominant aggregate pattern is the use of a learned fast path: the average shortcut rate is 86.9%, and it reaches 100% for Packing LP, MDKP, and TSP. The concrete fast path differs across families. In Packing LP it corresponds to specialized density or active-resource rules; in MDKP, to surrogate scoring, candidate restriction, or restricted dynamic programming; and in TSP, to structured candidate generation or geometric tour construction. Thus, the high shortcut rate supports the interpretation from Table 12: the synthesized solvers often replace generic search by a distribution-specific computation.

1478
1479
1480
1481
1482
1483

Fallback is rare overall. The average fallback rate is only 3.0%, which indicates that the reported speedups are not primarily obtained by calling a generic backend on most instances. Instead, fallback acts as a safety mechanism for slices of the distribution not covered by the learned hint. The largest fallback rates occur in MDS and MAXSAT, matching the qualitative picture that these solvers often use useful but incomplete structure, such as seeded local search, hub or anchor rules, or bounded cleanup.

1484

Repair is also usually limited. The average number of repair iterations is 0.43, and most families have essentially zero repair.

Problem	Targets	Q_{orig}	Q_{pert}	ΔQ	Qual. changed	Opt. changed	Feas. changed	Runtime ratio
Coloring	3	0.868	0.791	-0.077	0.342	0.342	0.000	$1.38\times$
MDS	3	0.973	0.819	-0.155	0.419	0.333	0.000	$1.39\times$
MIS	3	0.992	0.992	-0.001	0.258	0.191	0.000	$1.31\times$
All	9	0.945	0.867	-0.077	0.340	0.289	0.000	$1.36\times$

Table 15. **Aggregate graph-relabeling perturbation ablation.** The perturbation preserves each graph instance up to isomorphism, but changes the vertex identifiers and input order seen by the generated solver. The changed columns report the fraction of test instances whose quality, optimality, or feasibility value changes under relabeling. Runtime ratios are geometric means of $t_{\text{pert}}/t_{\text{orig}}$ across target distributions.

The main exception is MDKP, where repair is part of the synthesized packing strategy: the solver first constructs a candidate solution using surrogate scores or restricted dynamic programming, then performs bounded add/drop or feasibility repair. Thus, MDKP is more repair-mediated, whereas Packing LP and TSP are more directly fast-path-mediated.

The residual-size column gives a complementary but problem-dependent view. In some families, the solver does not solve the ambient instance directly; it first reduces the instance to a structured residual and then applies enumeration, repair, or a backend routine to that residual. This is the distributional computation pattern described in the previous section. However, because "residual size" has different native meanings across problem families, it should not be used for cross-family comparisons.

Overall, the diagnostic traces strengthen the mechanistic interpretation of the benchmark while clarifying its limits. The synthesized solvers are not merely faster implementations of the same generic algorithms, nor are they mostly wrappers around exact solvers. They usually execute a learned distribution-specific fast path, occasionally use fallback as a safety mechanism, and perform bounded repair only when the generated strategy leaves small violations or residual choices.

E.6. Perturbation Robustness Ablation

We include a graph-relabeling perturbation ablation as a diagnostic for presentation dependence. The perturbation randomly relabels the vertices of each graph instance. This preserves the underlying graph up to isomorphism, but changes the vertex identifiers and the input order seen by the generated solver. Thus, if a generated solver relies only on isomorphism-invariant structure, its quality and runtime should remain similar after relabeling. If it relies on incidental identifiers or ordering artifacts, its behavior may change.

For each selected generated solver, we evaluate the solver on the original held-out test instances and on relabeled copies of the same instances. This perturbation is applied only as a post-hoc diagnostic and is not used during solver selection. We report the original normalized quality Q_{orig}, the perturbed normalized quality Q_{pert}, and

$$\Delta Q = Q_{\text{pert}} - Q_{\text{orig}}.$$

We also report the fraction of test instances for which the quality value, optimality indicator, or feasibility indicator changes after relabeling. Finally, we report the runtime ratio $t_{\text{pert}}/t_{\text{orig}}$. Runtime ratios are aggregated by geometric mean, since runtime effects are multiplicative.

Table 15 shows that feasibility is fully stable under relabeling: the feasibility-changed fraction is zero for every problem class and every tested target distribution. Thus, the generated solvers continue to return feasible solutions under an isomorphic presentation of the same graph instances.

Quality is more mixed. Across the nine graph target distributions, mean normalized quality decreases from 0.945 to 0.867. This drop is concentrated in two target distributions: Ring-template in Coloring and Geometric-anchor in MDS. The remaining seven graph targets have nearly unchanged mean quality under relabeling. This suggests that many generated solvers capture reusable distribution-specific structure, but that some selected solvers also depend on presentation-specific regularities.

Runtime is also presentation-sensitive. Across the nine graph targets, relabeling increases runtime by a geometric mean factor of $1.36\times$. Most targets remain within a small constant factor, but there are notable outliers: Ring-template slows down by $9.65\times$, while Core-fringe trap slows down by $2.19\times$ despite unchanged quality. Thus, relabeling probes not only solution robustness but also computational robustness. A generated solver may preserve feasibility and quality while still taking a different, slower execution path under an isomorphic presentation.

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

Table 16. **Target-level graph-relabeling perturbation ablation.** The aggregate quality drop is concentrated in Ring-template and Geometric-anchor, while feasibility remains unchanged on all tested targets.

Overall, the perturbation ablation is best viewed as a diagnostic test of presentation dependence. The results are consistent with the synthesis agent often extracting reusable distribution-specific structure: feasibility is invariant, and quality is nearly unchanged on most graph targets. At the same time, the brittle cases show that some generated solvers depend on incidental presentation details such as vertex identifiers or ordering. Graph relabeling is therefore a useful stress test for separating invariant distributional structure from presentation-specific shortcuts.

F. Proofs for Section 5

Theorem 5.1 (Runtime-aware generalization for library selection)**.** *Assume $0 \le \mathrm{T}(c,x) \le \mathrm{T}_{\max}$ for every $c \in \mathcal{C}$ and $x \in \mathcal{X}$, and assume $\mathcal{C}$ contains at least one solver that is correct almost surely under D. Let π be a prior on $\mathcal{C}$ with $\sum_{c\in\mathcal{C}} \pi(c) \le 1$, and write $\Gamma(c) := \log(1/\pi(c))$. Let $\mathcal{C}^{\mathrm{feas}} = \{c \in \mathcal{C} : \mathrm{Err}_D(c) = 0\}$. For every $\delta \in (0,1)$, with probability at least $1-\delta$ over $S \sim D^n$, $\mathrm{Err}_D(\widehat{c}_S) \le \frac{\Gamma(\widehat{c}_S)+\log(\frac{2}{\delta})}{n}$ and $\mathrm{Run}_D(\widehat{c}_S) \le \inf_{c\in\mathcal{C}^{\mathrm{feas}}} \left\{ \mathrm{Run}_D(c) + 2\mathrm{T}_{\max}\sqrt{\frac{\max\{\Gamma(\widehat{c}_S),\Gamma(c)\}+\log(\frac{4}{\delta})}{2n}} \right\}$.*

Proof. For $c \in \mathcal{C}$, write $e(c) := \mathrm{Err}_D(c)$. If $\widehat{\mathrm{Err}}_S(c) = 0$, then $\Pr[\widehat{\mathrm{Err}}_S(c) = 0] = (1-e(c))^n \le e^{-ne(c)}$. Hence

$$\Pr\left(\widehat{\mathrm{Err}}_S(c) = 0 \text{ and } e(c) > \frac{\Gamma(c)+\log(2/\delta)}{n}\right) \le \pi(c)\frac{\delta}{2}.$$

A union bound gives, with probability at least $1-\delta/2$,

$$\mathrm{Err}_D(c) \le \frac{\Gamma(c)+\log(2/\delta)}{n} \quad \text{for all } c \in \mathcal{C} \text{ with } \widehat{\mathrm{Err}}_S(c) = 0.$$

Since $\widehat{c}_S$ is sample-consistent, this proves the error bound.

For runtime, set

$$r(c) := \mathrm{T}_{\max}\sqrt{\frac{\Gamma(c)+\log(4/\delta)}{2n}}.$$

By Hoeffding's inequality and another union bound, with probability at least $1-\delta/2$,

$$|\widehat{\mathrm{Run}}_S(c) - \mathrm{Run}_D(c)| \le r(c) \quad \text{for all } c \in \mathcal{C}.$$

On the intersection of the two events, fix any $c \in \mathcal{C}^{\mathrm{feas}}$. Since $\mathrm{Err}_D(c) = 0$, c is sample-consistent almost surely, so the selection rule gives $\widehat{\mathrm{Run}}_S(\widehat{c}_S) \le \widehat{\mathrm{Run}}_S(c)$. Therefore

$$\mathrm{Run}_D(\widehat{c}_S) \le \widehat{\mathrm{Run}}_S(\widehat{c}_S) + r(\widehat{c}_S) \le \widehat{\mathrm{Run}}_S(c) + r(\widehat{c}_S) \le \mathrm{Run}_D(c) + r(c) + r(\widehat{c}_S).$$

Taking the infimum over $c \in \mathcal{C}^{\mathrm{feas}}$ and using

$$r(c) + r(\widehat{c}_S) \le 2\mathrm{T}_{\max}\sqrt{\frac{\max\{\Gamma(\widehat{c}_S),\Gamma(c)\}+\log(4/\delta)}{2n}}$$

gives the stated runtime bound. The two events together hold with probability at least $1-\delta$. □

Theorem 5.2 (Exact recovery under identifiable structure)**.** *If $|\mathcal{H}| = N$ and the margin is $\gamma > 0$, then $n \geq \frac{2}{\gamma^2} \log \frac{2N}{\delta}$ samples suffice for $\widehat{h} = h^\star$ with probability at least $1 - \delta$.*

Proof. Fix the true hint $h^\star \in \mathcal{H}$, and for each $h \in \mathcal{H}$ let $\mu(h) := \mathbb{E}_{X \sim D_{h^\star}}[\psi_h(X)]$. By assumption, $\mu(h^\star) \geq \mu(g) + \gamma$ for every $g \neq h^\star$. Since $\psi_h(X) \in [0,1]$, Hoeffding's inequality gives $\Pr(|\widehat{\mu}_S(h) - \mu(h)| > \gamma/2) \leq 2e^{-n\gamma^2/2}$ for each h, and a union bound over $\mathcal{H}$ yields

$$\Pr\big(\exists h \in \mathcal{H} : \ \big|\widehat{\mu}_S(h) - \mu(h)\big| > \tfrac{\gamma}{2}\big) \leq 2|\mathcal{H}|e^{-n\gamma^2/2}.$$

For $n \geq \frac{2}{\gamma^2} \log \frac{2|\mathcal{H}|}{\delta}$, the right-hand side is at most δ, so with probability $1 - \delta$ we have $|\widehat{\mu}_S(h) - \mu(h)| \leq \gamma/2$ for all $h \in \mathcal{H}$. On this event, for every $g \neq h^\star$,

$$\widehat{\mu}_S(h^\star) \geq \mu(h^\star) - \tfrac{\gamma}{2} \geq \mu(g) + \tfrac{\gamma}{2} \geq \widehat{\mu}_S(g),$$

so $\hat{h} = h^\star$. □

Theorem 5.3 (Learning a hidden SAT backdoor from samples)**.** *If $m \geq 8\,\gamma^{-2} \log \frac{2d}{\delta}$, then $\widehat{B} = B$ with probability at least $1 - \delta$.*

Proof. Set $\varepsilon := \gamma/4$. For each variable $i \in [d]$, the random variables $\sigma_i(F^{(1)}), \ldots, \sigma_i(F^{(m)})$ are IID and lie in $[0,1]$, with mean $\mu_i := \mathbb{E}_{F \sim D_B}[\sigma_i(F)]$. By Hoeffding's inequality,

$$\Pr(|\widehat{\sigma}_i - \mu_i| > \varepsilon) \leq 2e^{-2m\varepsilon^2}.$$

A union bound over all $i \in [d]$ yields

$$\Pr\left(\max_{1 \leq i \leq d} |\widehat{\sigma}_i - \mu_i| > \varepsilon\right) \leq 2d\, e^{-2m\varepsilon^2}.$$

Therefore, if $m \geq \frac{8}{\gamma^2} \log \frac{2d}{\delta}$, then with probability at least $1 - \delta$ we have $|\widehat{\sigma}_i - \mu_i| \leq \varepsilon$ for all $i \in [d]$.

On this event, if $i \in B$ then $\widehat{\sigma}_i \geq q_1 - \varepsilon$, while if $j \notin B$ then $\widehat{\sigma}_j \leq q_0 + \varepsilon$. Since $\gamma = q_1 - q_0$ and $\varepsilon = \gamma/4$, we get $q_1 - \varepsilon > q_0 + \varepsilon$. Hence every variable in B has strictly larger empirical score than every variable outside B, so the top k empirical scores are attained exactly on the variables in B. Therefore $\widehat{B} = B$. □