# REDUCING THE IRREDUCIBLE: LEARNING REDUCTIONS BEYOND EXACT KERNELIZATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Exact reductions simplify hard optimization problems while preserving optimality, but their structural requirements can leave difficult instances unchanged. We study this limitation for maximum independent set, a natural test case given the extensive exact-reduction literature for independent set and minimum vertex cover. We develop heuristic reductions that aim to retain optimal or near-optimal solutions while continuing to simplify instances on which exact rules stall. In reduce-then-solve, language-model-guided program search generates a fast reduction rule before an exact solver processes the remaining graph. Rules are selected jointly for graph shrinkage, solution quality, and runtime. These criteria reflect the practical purpose of reduction: remove computational work without losing too much solution value or spending more time than it saves. In reduce-as-solve, localized greedy trials guide successive decisions on the shrinking graph. Language-model-guided program search also generates a policy that uses cached trial solutions to decide which vertices to include or discard, requesting fresh trials only when the stored evidence is insufficient. The reduction rule and cache policy are generated offline on small synthetic graphs and applied to hard instances without further tuning or language-model calls. Experiments on the evaluated methods show that reduce-then-solve enables more reduced graphs to be solved to optimality within a fixed time budget while usually retaining most of the reference solution value. Reduce-as-solve achieves higher solution quality than the tested reduction-based and greedy baselines, while localization and caching reduce trial-evaluation costs. These results demonstrate the practical value of heuristic reduction when the tested exact rules provide no simplification.

## 1 INTRODUCTION

Preprocessing is a central tool in combinatorial optimization. Many hard instances are too large to solve directly, so practical solvers first simplify the input and then run an exact, approximate, or heuristic algorithm on the residual instance. The value of this preprocessing stage is operational: it should remove parts of the instance that do not need to be solved explicitly, reduce the burden on the downstream solver, and preserve enough information to construct a high-quality solution for the original instance.

Parameterized algorithms give the cleanest formal version of this idea through kernelization (Fomin et al., 2019). A kernelization algorithm reduces an instance to an equivalent smaller instance whose size is governed by a problem parameter, such as a target solution size, rather than by the full size of the original input. Figure 1 illustrates the principle workflow (Akiba & Iwata, 2016). In graph problems, a large graph may be replaced by a smaller residual graph, provided the reduction records how a solution of the residual should be converted into a solution of the original graph. In practical graph algorithms, this principle is implemented through exact reduction rules: a rule identifies a certifying structure, removes the corresponding part of the graph, and records the reconstruction step. When such certificates are present, exact reductions provide preprocessing with formal solution-preservation guarantees. Appendix C lists the full MIS rules used in our experiments.

The same exactness requirement, however, creates a practical limitation. A reduction rule can act only when its certificate is present, and the algorithm must still spend time checking for this certificate even when no useful reduction exists. Cheap local rules are fast to test, but on difficult residual graphs they



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

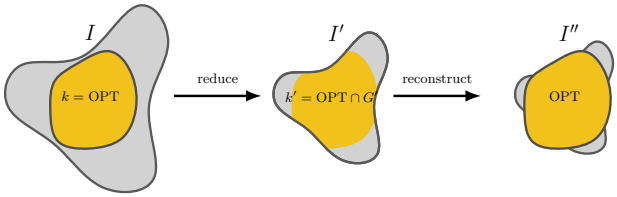



Figure 1: Kernelization shrinks the problem before search: the original instance $I$ is reduced to a smaller residual instance $I'$, while storing the information needed to reconstruct a solution for the original problem. After solving $I'$, reconstruction yields the original optimal solution.

may quickly exhaust all visible patterns; after that point, they are safe but no longer useful. Stronger rules can detect subtler structures, but their checks are more expensive and may still produce little shrinkage, or no reduction at all. Thus, on hard instances, exact preprocessing can enter a regime where it remains correct but loses its practical purpose: it either leaves the graph nearly unchanged or spends a substantial preprocessing budget verifying conditions that rarely lead to reductions.

This practical kernelization gap has direct runtime consequences. Exactness alone does not guarantee useful action on the residual instances where reduction is most needed. A reducer that produces little shrinkage still consumes time, and the downstream solver receives nearly the same hard instance. This is especially important for large-scale heuristic solving, where the reducer is not merely a source of formal guarantees but a mechanism for making the remaining instance computationally manageable.

We study this gap through the unweighted MIS problem. MIS is one of the canonical problems for reduction-based graph optimization: it is a central NP-hard problem, exact preprocessing has been studied extensively for MIS and its vertex-cover dual, and modern high-performance MIS solvers rely heavily on reductions before search (Cygan et al., 2015; Lamm et al., 2016). This makes MIS a demanding setting for evaluating the practical limits of exact preprocessing. When exact reductions become inactive or unproductive on hard MIS residuals, the issue is not merely the absence of a particular rule, but a more general scalability barrier: reductions based only on certifiable structure may no longer provide enough useful simplification relative to their cost (Akiba & Iwata, 2016; Großmann et al., 2024b;a).

This motivates heuristic reduction as a scalable complement to exact kernelization. A heuristic reduction (Chang et al., 2017) maps an input graph to a committed partial solution and a residual graph, but unlike exact kernelization it does not claim optimum preservation. We evaluate such reductions by the criteria that matter for practical preprocessing: residual size, lifted solution quality, and runtime. This broadens the design question from which transformations are always safe to which transformations can reliably simplify hard residuals while preserving enough solution quality to improve downstream solving.

We instantiate this idea in two complementary ways. The first is *reduce-then-solve*. This follows the classical preprocessing pipeline: reduce the graph first, then pass the residual to a downstream solver. Instead of hand-designing exact rules, we use LLM-guided program evolution to synthesize fast per-vertex heuristic reduction rules. Candidate programs are selected by an automatic objective that jointly rewards shrinkage, solution preservation, and runtime. A hard reduction requirement prevents the search from producing rules that merely predict good vertices without actually simplifying the instance. The resulting rules are designed to remain active in the regime where classical exact reductions have become silent. The second is *reduce-as-solve*. This treats heuristic computation itself as a reduction primitive. When a safe structural certificate is unavailable, repeated heuristic search can still provide stable evidence about which commitments simplify the graph with little quality loss. We run cheap rollout heuristics on the residual graph and use their repeated choices to identify vertices or regions that are consistently useful in high-quality independent sets. These signals are then converted into commitment decisions that reduce the remaining graph. To make this process scalable, we localize rollout evaluation around promising neighborhoods and reuse accumulated per-vertex evidence through LLM-evolved cache policies. This keeps rollout-based reduction independent of any particular downstream heuristic while directing reduction effort toward the parts of the graph where heuristic evidence is most informative.

Our experiments show that the practical kernelization gap appears across hard MIS benchmarks and diverse graph families. Standard exact reductions often either stop with limited shrinkage or spend significant preprocessing time without producing a substantially smaller residual. In contrast, our heuristic reductions continue to simplify these residuals. Reduce-then-solve provides fast preprocessing with small quality loss, while reduce-as-solve converts heuristic evidence into further scalable reductions. Together, they reduce graph size and runtime, enabling downstream solvers to return high-quality solutions in regimes where exact kernelization alone leaves the instance largely unresolved. We summarize our contributions below.

- **Empirical evidence of a practical kernelization gap.** We give a fine-grained study of when standard exact MIS reduction rules fire, how much they shrink the graph, and how much time their structural checks consume. On the hard DIMACS/BHOSLIB benchmark, the tested exact rules do not fire, while stronger checks can still spend substantial time certifying that no reduction is available. This identifies the regime where exact preprocessing remains correct but loses practical effect.
- **GENREDUCE: reduce-then-solve with an LLM-evolved reduction rule.** We introduce an LLM-guided program-evolution framework that synthesizes fast single-pass heuristic reduction rules. Candidate programs are selected by an executable objective that jointly rewards shrinkage, lifted solution quality, and runtime, with a hard shrinkage constraint to prevent the search from producing predictors that do not actually reduce the graph. The frozen rule is deployed out of distribution and evaluated by whether its residual makes exact solving easier under the same downstream budget.
- **LOCALIZEDREDUCE and CACHEREDUCE: reduce-as-solve beyond an exact residual call.** We introduce a reduce-as-solve framework for the regime where the residual remains too large or too hard for exact solving. The framework repeatedly converts heuristic evidence into reduction actions and recurses on the shrinking graph. LOCALIZEDREDUCE obtains this evidence from localized one-step rollouts restricted to the closed neighborhood $N[u]$ of a minimum-degree pivot, a restriction justified by the fact that every MIS must intersect $N[u]$. CACHEREDUCE uses the same outer loop but replaces explicit rollout consensus with an LLM-evolved cache policy that scores vertices directly from the cached residual state. Together, the two methods expose the quality–cost tradeoff evaluated in the experiments: explicit localized rollouts give stronger solution quality, while the cache policy reduces rollout cost.

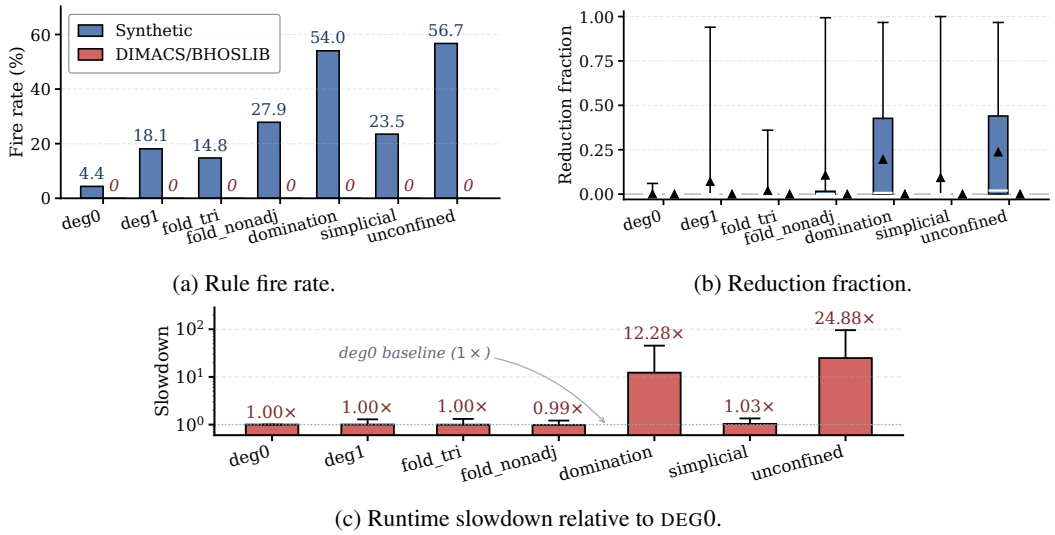
## 2 EXACT REDUCTIONS AND THE PRACTICAL KERNELIZATION GAP

**Exact reduction for MIS.** Let $G = (V, E)$ be an undirected graph. An independent set is a set $S \subseteq V$ with no internal edge, and MIS asks for one of maximum cardinality, denoted $\alpha(G)$. A reduction-based MIS solver first simplifies $G$, records the decisions made during preprocessing, solves a residual graph $G'$, and reconstructs a solution for the original graph from the residual solution. Kernelization is the formal version of this pipeline. A kernelization maps a parameterized instance $(x, k)$ in polynomial time to an equivalent instance $(x', k')$ such that $(x, k) \in L \iff (x', k') \in L, |x'| + k' \leq f(k)$ for some computable function $f$ depending only on $k$ (Fomin et al., 2019). In graph algorithms, this equivalence is implemented by exact reduction rules: a rule finds a certifying structure, removes the part of the graph justified by that certificate, and records how to lift a residual solution back to the original graph. When the certificate is present, the rule shrinks the instance without sacrificing optimality.

The pendant rule is a standard MIS example. If $v$ has degree one and unique neighbor $u$, then some MIS contains $v$. Indeed, any optimum containing $u$ can replace $u$ by $v$, and any independent set containing neither can add $v$. Hence the solver may delete $\{u, v\}$, solve $G - \{u, v\}$, and add $v$ back: $\alpha(G) = 1 + \alpha(G - \{u, v\})$. More details can be found in Appendix.

The example also exposes the operational limitation of exact reduction. Every exact rule needs two things: its certificate must appear in the current graph, and the solver must spend time checking whether that certificate exists. Cheap local rules have low verification cost, but they can quickly exhaust all visible patterns. Stronger rules can detect subtler certificates, but their checks are more expensive and may still return the same residual graph. Thus the practical value of exact reduction

depends not only on correctness, but also on whether useful certificates remain available at acceptable cost.




(a) Rule fire rate.

(b) Reduction fraction.

(c) Runtime slowdown relative to DEG0.

Figure 2: Applicability, shrinkage, and runtime cost of representative exact MIS reduction rules.

**Empirical kernelization gap.** We now examine this dependency on a representative exact-reduction toolkit. Figure 2 compares seven classical MIS rules, ordered from simple local checks to more structured certificates: DEG0, DEG1, FOLD-TRI, FOLD-NONADJ, DOMINATION, SIMPLICIAL, and UNCONFINED. We use a synthetic control collection, where local reducible structure is often present, and the DIMACS/BHOSLIB benchmark, a standard hard benchmark for MIS. The purpose of this experiment is not to exhaustively benchmark MIS solvers. It is to isolate the three quantities that determine whether exact preprocessing is practically useful: whether a rule fires, how much graph it removes, and how much time is spent checking it.

Fig. 2a shows applicability. On the synthetic collection, the rules fire on a nontrivial fraction of graphs, ranging from $4.36\%$ for DEG0 to $56.71\%$ for UNCONFINED. This confirms that the rules are active on graphs containing compatible structure. On DIMACS/BHOSLIB, however, all seven rules have fire rate $0.00\%$. This is the certificate-side failure mode: the exact rules are not merely less frequent on the hard benchmark; in this experiment their structural preconditions are absent across the entire benchmark.

Fig. 2b shows the corresponding shrinkage distribution. On the synthetic graphs, several rules have a heavy-tailed pattern: many graphs receive no reduction, but compatible graphs can lose a large fraction of their vertices. This is exactly the behavior one expects from local exact reductions: they may be inactive on most instances, but highly effective when the right structure appears. On DIMACS/BHOSLIB, the distribution collapses to zero for every rule. Thus the zero fire rate is not hiding rare large reductions; the tested exact rules remove no vertices on the hard benchmark.

Fig. 2c shows the runtime side of the same gap. On DIMACS/BHOSLIB, we measure each rule relative to the trivial isolated-vertex check DEG0. For graph $g$ and rule $r$, the slowdown is $s_g(r) = \mathrm{time}(r, g)/\mathrm{time}(\mathrm{DEG0}, g)$. Since none of the rules fire on these graphs, this runtime is pure verification overhead: time spent checking that the relevant certificate is absent. The cheap-rule cluster remains near the baseline, with mean slowdowns between $0.99\times$ and $1.03\times$. The more structured checks are substantially more expensive: DOMINATION has mean slowdown $12.28\times$ and maximum slowdown $45.33\times$, while UNCONFINED has mean slowdown $24.88\times$ and maximum slowdown $95.77\times$.

Together, these three panels give the empirical form of the practical kernelization gap used throughout this paper. On the hard benchmark, the representative exact rules do not fire, do not shrink the graph, and the stronger checks can still spend substantial time certifying that no reduction is available. This does not make exact reductions unimportant: when a certificate is present at acceptable cost,

an exact rule is still the preferred preprocessing step. The gap appears after that point, when the residual graph remains large but exact rules no longer provide useful simplification. In this regime, scalable preprocessing needs a different objective: continue reducing the instance while preserving enough solution quality to improve downstream solving. The next section formalizes this objective as heuristic reduction.

## 3 METHOD: LLM-ASSISTED HEURISTIC REDUCTION

The previous section identifies our target regime: exact reductions remain correct, but no longer simplify the residual enough. We address it with *heuristic reduction*. For MIS, a reduction step chooses vertices to include, $C$, and/or vertices to discard, $D$, producing the residual graph $R = G[V \setminus (N[C] \cup D)]$. Any independent set $S_R$ in $R$ lifts back as $C \cup S_R$. Unlike exact reductions, these decisions need not be certified as optimum-preserving; they are selected for the practical tradeoff among shrinkage, reconstructed solution quality, and runtime. **LLM-guided program evolution:** We use a FunSearch-style (Romera-Paredes et al., 2024) evaluator-driven loop: the LLM proposes candidate programs, executable evaluation scores them on training graphs, and feedback from previous candidates is returned to the next prompt. The LLM is only an offline program proposer. Unlike standalone solver discovery, our templates generate reduction policies, which are selected by shrinkage, reconstructed quality, and runtime, then frozen before test-time deployment.

**Algorithm 1** LLM-guided evolution of reduction policies

**Require:** Training graphs $\mathcal{D}_{\mathrm{tr}}$, validation graphs $\mathcal{D}_{\mathrm{val}}$; policy class $\mathcal{H}$; evaluator $\mathcal{E}$; LLM proposer $\mathcal{M}$; shrinkage threshold $\delta$; rounds $T$
**Ensure:** Frozen policy $h^\star$
1: Initialize candidate pool $\mathcal{P} \leftarrow \emptyset$
2: **for** $t = 1, \ldots, T$ **do**
3: Generate candidates $\mathcal{C}_t \leftarrow \mathcal{M}(\mathcal{P}, \mathrm{feedback})$
4: Evaluate each $h \in \mathcal{C}_t$ by

$$\mathcal{E}(h) = \lambda_\rho \cdot \mathrm{shrinkage}(h) + \lambda_q \cdot \mathrm{quality}(h) - \lambda_\tau \cdot \mathrm{runtime}(h).$$

5: Keep only candidates with average shrinkage at least $\delta$
6: Update $\mathcal{P}$ and return representative successes/failures to $\mathcal{M}$
7: **end for**
8: Select the best retained candidate on $\mathcal{D}_{\mathrm{val}}$
9: **return** frozen policy $h^\star$

This object separates heuristic reduction from ordinary heuristic solving. A heuristic solver returns an independent set and stops. A heuristic reduction returns a smaller residual instance together with the include/exclude decisions needed to lift a residual solution back to the original graph. The relevant quantities are therefore not only the final solution value, but also how much graph is removed and how much time the reduction spends. In the rest of the section, we instantiate this principle in two regimes. *Reduce-then-solve* reduces once and hands the residual to an exact downstream solver. *Reduce-as-solve* continues reducing the graph round by round and never invokes an exact solver on the residual.

**Reduce-then-solve: GENREDUCE.** The reduce-then-solve setting follows the classical preprocessing pipeline: reduce the graph first, then solve the residual. This regime is appropriate when exact solving is still possible after sufficient shrinkage. The question is how to obtain reduction rules that remain active after exact certificates become rare. GENREDUCE uses LLM-guided program evolution to synthesize a fast single-pass heuristic reduction rule. Each candidate program proposes include-or-discard decisions from local or cheaply computable graph information. The program is evaluated by applying it to training graphs, solving the residual, reconstructing the lifted independent set, and measuring three quantities: shrinkage, lifted solution quality, and runtime. The LLM is used only as a program proposer; selection is performed by executable feedback. A minimum-shrinkage constraint is essential. Without it, the search could discover programs that identify useful vertices but leave the instance essentially unchanged, which would make the method a predictor rather than a reducer. We therefore retain only candidates whose average shrinkage exceeds a threshold $\delta$.

Among those candidates, the selection objective favors high lifted quality and low runtime. After validation, the selected rule is frozen and deployed as GENREDUCE on unseen graphs. At test time, GENREDUCE is applied once. It applies the selected include-or-discard decisions, removes the implied closed neighborhoods, and passes the residual to the exact solver. The evaluation question for this regime is therefore not whether GENREDUCE alone solves MIS, but whether its residual makes exact solving easier under the same downstream budget.

**Reduce-as-solve.** Reduce-as-solve targets the complementary regime. Here, even after preprocessing, the residual may remain too large or too hard for an exact solver. The algorithm therefore does not wait for a final exact call. Instead, it repeatedly makes reduction decisions, applies the corresponding include-or-discard action, and continues on the smaller residual. The source of evidence is heuristic search behavior. A single heuristic completion is not trusted as a certificate of optimality. However, repeated completions can reveal stable signals: some vertices repeatedly appear in high-quality completions, while others are consistently avoided or blocked by stronger choices. Reduce-as-solve converts these signals into reduction decisions. The rollouts are not the final answer; their consensus determines how the residual is simplified.

**Algorithm 2** Reduce-as-solve: for LOCALIZEDREDUCE, $\Pi$ uses localized min-degree rollouts to infer the next include-or-discard action. For CACHEREDUCE, $\Pi$ is the LLM-evolved cache policy: it scores candidates from cached residual state and emits action without running rollout simulations.

**Require:** Graph $G$; decision rule $\Pi$
**Ensure:** Independent set $S$
$R \leftarrow G$, $S \leftarrow \emptyset$, cache state $\Xi \leftarrow \emptyset$
**while** $V(R) \neq \emptyset$ **do**
  Choose a minimum-degree pivot $u$ in $R$
  Set localized candidate set $K \leftarrow N_R[u]$
  Choose disjoint include and discard sets $C, D$ using $\Pi(R, K, \Xi)$
  $S \leftarrow S \cup C$
  Update cache state $\Xi$
  $R \leftarrow R[V(R) \setminus (N_R[C] \cup D)]$
**end while**
**return** $S$

**LOCALIZEDREDUCE.** LOCALIZEDREDUCE instantiates the decision rule $\Pi$ with localized one-step rollouts. Let $u$ be a minimum-degree pivot in the current residual graph. Instead of evaluating every alive vertex, as in textbook one-step rollout, LOCALIZEDREDUCE evaluates only vertices in the closed neighborhood $N_R[u]$. This restriction is structurally justified: every MIS of $R$ intersects $N_R[u]$, since otherwise $u$ could be added. Thus the localized candidate set always contains at least one optimum-compatible next decision.

For each candidate $v \in N_R[u]$, the method temporarily includes $v$, removes $N_R[v]$, and completes the remaining residual with a cheap min-degree greedy rollout. The resulting rollout solutions are then aggregated into a consensus signal. Vertices repeatedly selected across strong localized rollouts provide positive evidence for inclusion, while vertices consistently absent from such rollouts provide evidence for discarding or deprioritization. The method applies the reduction action supported by this localized consensus and then recurses on the smaller graph. The restriction to $N_R[u]$ is the main contribution of LOCALIZEDREDUCE: it preserves the relevant one-step structural choice while avoiding the cost of exhaustive rollout over the full residual graph.

**CACHEREDUCE.** CACHEREDUCE uses the same reduce-as-solve outer loop, but replaces explicit rollout consensus with an LLM-evolved cache policy. The policy scores candidates directly from the cached state of the residual and emits the next include-or-discard decision without launching rollout simulations. Its purpose is to reuse the evidence accumulated during the shrinking process rather than recomputing similar lookahead information at every round.

Thus LOCALIZEDREDUCE and CACHEREDUCE differ only in how the per-round reduction decision is produced. LOCALIZEDREDUCE uses explicit localized rollouts and aggregates their consensus. CACHEREDUCE replaces those rollouts with a learned cache policy. The two methods therefore

cover different points on the same quality–cost axis: explicit rollout evidence gives higher quality, while the cache policy trades a small amount of quality for lower runtime.

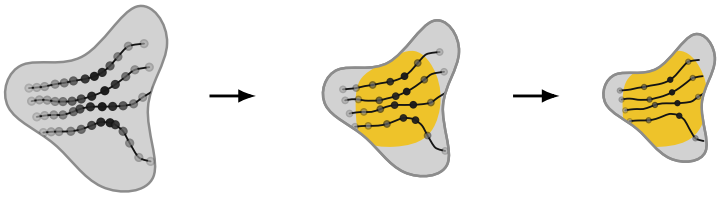


Figure 3: Reduce-as-solve turns repeated heuristic behavior into reduction evidence. Each curve represents a rollout trajectory, and dots mark the decisions made along that trajectory. Stable include/exclude signals across repeated rollouts form a consensus region, shown in yellow. The method does not treat any single rollout as the final solution; it uses the consensus to simplify the residual graph and continue on the smaller instance.

Together, the three methods instantiate heuristic reduction in the two operating regimes needed by the paper. GENREDUCE reduces once so that an exact solver can succeed on the residual. LOCALIZEDREDUCE and CACHEREDUCE reduce as they solve, for the regime where an exact residual solve is no longer the intended final step. All three methods produce smaller residual graphs together with include/exclude decisions for reconstruction; the experiments therefore evaluate them by shrinkage, lifted solution quality, and wall-clock time.

## 4 EXPERIMENTS

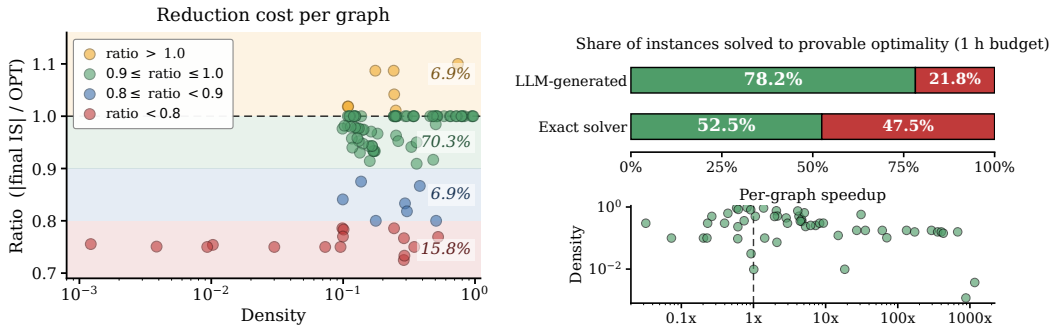
**Experimental setup.** We evaluate on the 101-instance DIMACS/BHOSLIB MIS benchmark, and deferred details in Appendix. The benchmark is large enough to expose the limits of exact preprocessing yet small enough to permit detailed per-instance comparisons across reduction quality, solution quality, and runtime.

The comparison covers exact reductions, reduce-then-solve, and reduce-as-solve. The exact toolkit includes standard local MIS rules: isolated-vertex, pendant, degree-two folding, domination, simplicial, and unconfined reductions. For reduce-then-solve, GENREDUCE applies an evolved single-pass heuristic rule before passing the residual to the exact solver. For reduce-as-solve, LOCALIZEDREDUCE uses one-step rollout (Bertsekas et al., 1997) only over $N[u]$, the closed neighborhood of a minimum-degree pivot, justified because every MIS intersects $N[u]$; it then commits the consensus decision from these localized rollouts. CACHEREDUCE replaces rollout consensus with an LLM-evolved cache policy that scores residual-state candidates directly. We compare the reduce-as-solve methods with Min-Degree Greedy, Reducing-Peeling (Chang et al., 2017), and the Scheduling Framework (Piao et al., 2020). All heuristic methods use the same isolated-vertex and pendant exact-reduction backbone, isolating the effect of the inexact reduction step.

We report shrinkage, solution quality relative to a common one-hour Gurobi reference, and wall-clock runtime. The reference is the proven optimum when Gurobi terminates and the best incumbent otherwise. All Gurobi and heuristic runs are single-threaded under the same one-hour budget.

**Reduce-then-solve: making residuals easier for exact solving.** The first experiment follows the classical reduce-then-solve pattern, but replaces the exact reduction step with the learned GENREDUCE rule. Each hard instance is reduced once, and the residual graph is handed to Gurobi. The two natural questions are how much solution quality the inexact reduction sacrifices, and whether the smaller residual is meaningfully easier for the downstream solver to certify.

Figure 4a shows that the quality cost of the inexact reduction is small on most graphs. The median ratio is $0.977$ and the mean is $0.939$; $77.2\%$ of the benchmark preserves at least $90\%$ of the reference value. Seven instances exceed ratio $1$: on these, the reduced residual let Gurobi find a larger independent set than the incumbent obtained by running Gurobi for the full budget on the original graph. The reduction is therefore not just removing vertices — on some hard instances it changes the search space enough for the downstream solver to find better solutions within the same budget.




(a) Reduction cost per graph.

(b) Downstream solver payoff.

Figure 4: Reduce-then-solve with GENREDUCE. Left: reconstructed solution quality after reduction, measured by the ratio between the final independent set size and the reference value. Right: downstream payoff, comparing Gurobi on the original graph with Gurobi on the reduced residual, together with per-graph speedup as a function of input density.

The downstream payoff is shown in Figure 4b. Gurobi proves optimality on $52.5\%$ of the benchmark when run on the original graphs, against $78.2\%$ when run on the reduced residuals. The speedup panel further shows that the gain is not a small shift in the solvability threshold: for many instances, the reduction changes downstream solving time by orders of magnitude.

GENREDUCE is not an exact kernelization rule, but on this benchmark it does what classical exact reduction stops doing on the hard residuals — it shrinks them in a way the downstream solver can exploit, while preserving most of the solution value in the typical case.

**Reduce-as-solve: heuristic evidence as scalable reduction.** The reduce-then-solve setting still relies on an exact downstream solver, so it inherits the cost of that final call. When the residual remains hard even after reduction, exact solving becomes the bottleneck and a single reduction pass is no longer enough. Reduce-as-solve addresses this regime: the algorithm commits one reduction decision per round and recurses on the shrinking graph, without invoking an exact solver on the residual. Both of our reduce-as-solve methods follow this outer loop and differ only in where the per-round decision comes from. LOCALIZEDREDUCE produces it by aggregating the outcomes of localized min-degree rollouts; CACHEREDUCE produces it directly from the LLM-evolved cache policy. Table 1 compares both of our methods against Gurobi and the three classical heuristic baselines.

Table 1: Reduce-as-solve comparison on hard MIS benchmark. Gurobi "# ratio $= 1.0$" reports the proven-optimal count; ratios above $1.0$ occur when $\mathrm{OPT}_{\mathrm{ref}}$ is a one-hour timeout incumbent and a heuristic finds a larger independent set. Green cells mark where our methods win the heuristic block.

| | MEAN RATIO | # RATIO $= 1.0$ | # RATIO $> 1.0$ | WALL-CLOCK TIME (S) | | |
|---|---|---|---|---|---|---|
| | | | | MEAN | MEDIAN | MAX |
| GUROBI ($\mathrm{OPT}_{\mathrm{ref}}$) | 1.000 | 53 / 101 (PROVEN) | — | 1945.25 | 2234.42 | — |
| MINDEG | 0.885 | 20 / 101 | 1 / 101 | 0.15 | 0.07 | 1.93 |
| RP (CHANG+'17) | 0.815 | 19 / 101 | 0 / 101 | 0.54 | 0.24 | 8.77 |
| SCHED. (PIAO+'20) | 0.878 | 19 / 101 | 1 / 101 | 4.46 | 0.95 | 123.37 |
| LOCALIZEDREDUCE (OURS) | 0.979 | 50 / 101 | 8 / 101 | 14.26 | 2.58 | 355.56 |
| CACHEREDUCE (OURS) | 0.973 | 44 / 101 | 7 / 101 | 11.40 | 1.56 | 350.40 |

Both LOCALIZEDREDUCE and CACHEREDUCE sit at the top of the heuristic block on every quality column. They reach mean ratio $0.979$ and $0.973$ respectively, against $0.885$ for Min-Degree Greedy, $0.815$ for Reducing-Peeling, and $0.878$ for the Scheduling Framework — a 9–10 percentage-point gap. The reference-match counts show the same pattern: LOCALIZEDREDUCE reaches the reference on $50/101$ instances and CACHEREDUCE on $44/101$, while the strongest classical baseline reaches it

on $20/101$. LOCALIZEDREDUCE also exceeds the reference on $8/101$ instances and CACHEREDUCE on $7/101$, against at most one for any classical baseline; these are graphs where the reference is a one-hour incumbent rather than a proved optimum, and the reduce-as-solve search reaches a larger independent set within seconds.

The quality gain does not come at exact-solver cost. LOCALIZEDREDUCE runs at $14.26$ s and $2.58$ s median; CACHEREDUCE at $11.40$ s and $1.56$ s. Both are two orders of magnitude below the $1945.25$ s of the Gurobi reference, and remain in the same range as the cheapest non-trivial classical baseline. The two methods trade off in the expected direction: LOCALIZEDREDUCE pays for explicit rollouts and recovers slightly higher quality, while CACHEREDUCE skips the rollouts in favour of the learned cache decision and is faster on every runtime aggregate at a small quality cost.

What this experiment describes is the regime reduce-as-solve was designed for. Once exact reduction stalls and a single inexact pass is not enough, repeated heuristic evaluation can itself act as the reduction mechanism: each round commits one structural decision to the residual, and the accumulated record of those decisions keeps the graph shrinking at heuristic cost while retaining most of the solution value.

**Ablation: localized candidate restriction.** We compare LOCALIZEDREDUCE with exhaustive one-step rollout, EXH-1L, which evaluates every alive vertex at each round (Bertsekas et al., 1997). LOCALIZEDREDUCE instead evaluates only $N[u]$, the closed neighborhood of a minimum-degree pivot, which is sufficient since every MIS intersects $N[u]$. This restriction makes rollout practical: on the $97$ shared completed graphs, LOCALIZEDREDUCE is faster on every graph, with $62.7\times$ median speedup, $46.3\times$ geometric-mean speedup, $446\times$ maximum speedup, and $98\%$ lower total wall time, while losing only $1.3$ percentage points in mean ratio and $7$ points in worst-case ratio; moreover, EXH-1L times out on four instances, whereas LOCALIZEDREDUCE completes all $101$ graphs.

Table 2: Localization ablation. Green cells mark rows where LOCALIZEDREDUCE wins.

| | EXH-1L | LOCALIZEDREDUCE (OURS) |
|---|---|---|
| *Quality* | | |
| MEAN RATIO | 0.99 | 0.98 |
| MIN RATIO | 0.90 | 0.83 |
| # RATIO $= 1.0$ (MATCHES OPT) | 57 / 97 | 50 / 101 |
| # RATIO $> 1.0$ (BEATS GUROBI-1 H) | 7 / 97 | 8 / 101 |
| *Wall-clock time per graph (s)* | | |
| MEAN | 424.52 | 14.26 |
| MEDIAN | 78.28 | 2.58 |
| MAX | 2847.72 | 355.56 |
| COMPLETED WITHIN 1 H BUDGET | 97 / 101 | 101 / 101 |
| *Head-to-head (97 shared graphs)* | | |
| MEAN RATIO GAP (LOCALIZEDREDUCE − EXH-1L) | — | -1.30 % |
| # GRAPHS FASTER | 0 / 97 | 97 / 97 |
| PER-GRAPH SPEEDUP (MEDIAN / GEOMEAN / MAX) | — | 64.4× / 47.3× / 446× |
| TOTAL WALL (S) | 41,178 | 846 (−98%) |

## 5 CONCLUSION

We studied the practical kernelization gap: on hard MIS instances, exact reductions remain correct yet offer little useful shrinkage at acceptable cost. We proposed heuristic reduction assisted by LLM as a scalable complement, replacing the requirement for certified safety with an explicit tradeoff among shrinkage, solution quality, and runtime. In the reduce-then-solve regime, GENREDUCE learns fast rules that produce residuals easier for an exact solver to solve. In the reduce-as-solve regime, LOCALIZEDREDUCE and CACHEREDUCE use repeated heuristic evidence to guide reductions without relying on a final exact solve. Across hard MIS benchmarks, these methods simplify residual graphs after classical exact rules stall, reduce runtime, and preserve high-quality solutions. Our results suggest that, when exact kernelization becomes practically inactive, learned and evidence-driven heuristic reductions can restore reduction as an effective tool for large-scale combinatorial solving.

## AI USE STATEMENT

In this work, we used generative AI tools for academic writing, plotting, literature review, and anonymizing submission materials. Our research investigates AI-generated code, and we have reviewed all AI-generated code used in this work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## REFERENCES

Faisal N Abu-Khzam, Michael R Fellows, Michael A Langston, and W Henry Suters. Crown structures for vertex cover kernelization. *Theory of Computing Systems*, 41(3):411–430, 2007.

Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *Theoretical Computer Science*, 609:211–225, 2016.

R Balasubramanian, Michael R Fellows, and Venkatesh Raman. An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters*, 65(3):163–168, 1998.

Dimitri P Bertsekas, John N Tsitsiklis, and Cynara Wu. Rollout algorithms for combinatorial optimization. *Journal of heuristics*, 3(3):245–262, 1997.

Lijun Chang, Wei Li, and Wenjie Zhang. Computing a near-maximum independent set in linear time by reducing-peeling. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1181–1196, 2017.

Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015.

Fedor V Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. *Kernelization: theory of parameterized preprocessing*. Cambridge University Press, 2019.

Ernestine Großmann, Kenneth Langedal, and Christian Schulz. Accelerating reductions using graph neural networks and a new concurrent local search for the maximum weight independent set problem. *arXiv preprint arXiv:2412.14198*, 2024a.

Ernestine Großmann, Kenneth Langedal, and Christian Schulz. A comprehensive survey of data reduction rules for the maximum weighted independent set problem. *arXiv preprint arXiv:2412.09303*, 2024b.

Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F Werneck. Finding near-optimal independent sets at scale. In *2016 Proceedings of the eighteenth workshop on algorithm engineering and experiments (ALENEX)*, pp. 138–150. SIAM, 2016.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*, 2024.

George L Nemhauser and Leslie E Trotter Jr. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Chengzhi Piao, Weiguo Zheng, Yu Rong, and Hong Cheng. Maximizing the reduction ability for near-maximum independent set computation. *Proceedings of the VLDB Endowment*, 13(12): 2466–2478, 2020.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Mingyu Xiao and Hiroshi Nagamochi. Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs. *Theoretical Computer Science*, 469: 92–104, 2013.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

# A Related Work

**Exact kernelization and data reduction for MIS.** Data reduction is a central tool in parameterized algorithms and practical graph optimization (Fomin et al., 2019). For MIS and its dual vertex-cover formulation, classical kernelization rules include LP-based reductions such as the Nemhauser–Trotter theorem (Nemhauser & Trotter Jr, 1975), crown decompositions (Abu-Khzam et al., 2007), and a large family of local and semiglobal graph reductions. Modern branch-and-reduce solvers (Akiba & Iwata, 2016) show that these rules are not only theoretically safe, but also highly effective on many practical instances: the solver repeatedly reduces the graph, solves or branches on the residual, and lifts the solution back to the original instance. Recent scalable kernelization work further improves this pipeline by accelerating expensive reductions, parallelizing matching-based routines, and tracking when reductions cease to be productive. These works define the exact-reduction baseline for our study: they share our solver-facing goal of making the residual instance smaller, but they remain constrained by exact certificates and therefore may stall on hard instances where such structures are rare.

**Reduction-then-solve frameworks for large independent sets.** Reduction is also a key component of large-scale MIS heuristics. ReduMIS and KaMIS-style solvers combine exact reductions with evolutionary or local-search mechanisms: after kernelization, an evolutionary solver identifies vertices that appear likely to belong to a large independent set, and a fixed rule commits a small number of low-degree vertices before kernelizing the residual again (Lamm et al., 2016). This line of work is important context, but it is not the direct comparison target for our reduce-then-solve contribution: its performance reflects a full solver stack, including exact reductions, evolutionary search, recursive kernelization, and implementation engineering. Moreover, its inexact reduction policy is manually fixed rather than learned. Our question is narrower: whether LLM-generated heuristic reduction rules can improve the shrinkage–quality–runtime tradeoff of the reduction stage itself when exact rules are inactive. The closest reduction-policy baseline is Reducing-Peeling (Chang et al., 2017), which uses a fixed hand-designed policy that alternates simple reductions with peeling decisions to obtain near-maximum independent sets at very large scale. We compare with it because it is scalable, explicitly reduction-centered, and not a full branch-and-reduce solver; the contrast is that our framework synthesizes reduction programs under explicit constraints on shrinkage, solution preservation, and runtime.

**Rollout and lookahead heuristics.** Rollout algorithms (Bertsekas et al., 1997) improve a base heuristic by simulating the consequences of candidate decisions and choosing the candidate with the best heuristic completion. This principle naturally yields lookahead heuristics for MIS: try a candidate vertex, complete the residual graph with a cheap heuristic, and commit the candidate whose rollout gives the best solution. Such methods are the right baseline for our reduce-as-solve component because they use the same computational primitive: repeated heuristic completions. The distinction is our objective. A standard rollout heuristic uses these completions only to select the next move. Our framework treats the repeated choices made during rollouts as reusable evidence for reduction: the signals identify which vertices or regions of the residual instance should be committed, prioritized, or cached. Thus the comparison with rollout lookahead is necessary, but the role of rollout computation is different in our method.

**LLM-generated heuristics and algorithm design.** Recent LLM-based algorithm-design systems show that language models can generate useful programs or heuristics when paired with executable feedback. Yang et al. (2023) uses language models as derivative-free optimizers; FunSearch (Romera-Paredes et al., 2024) combines LLM-generated code with evolutionary search and an evaluator to discover programs with strong empirical or mathematical performance; Evolution of Heuristics (Liu

et al., 2024) uses LLMs and evolutionary computation to design combinatorial optimization heuristics; and AlphaEvolve (Novikov et al., 2025) extends this idea to a broader coding-agent pipeline for scientific and algorithmic discovery. Our work belongs to this evaluator-driven program-search line, but its target is more specific. We do not ask the LLM to produce a standalone MIS solver. Instead, we constrain the search around reduction: generated programs must shrink the graph, preserve solution quality, and run quickly. In the reduce-as-solve setting, the LLM further designs cache policies that decide how accumulated heuristic evidence should be reused. This makes the LLM component reduction-aware rather than merely heuristic-generating.

## B More details on experiments

**Benchmarks.** The primary test set is the 101-instance DIMACS/BHOSLIB hard MIS benchmark. It contains 66 DIMACS Clique benchmark instances, converted to MIS by taking graph complements, and 35 BHOSLIB forced-hard random instances. The graphs range from $n = 28$ to $n = 3361$ vertices, with edge counts ranging from 72 to approximately one million and densities spanning roughly $10^{-3}$ to $0.96$. This benchmark is used for all headline comparisons involving exact reductions, reduce-then-solve, and reduce-as-solve.

Table 3: Hard Maximum Independent Set benchmark. DIMACS is the standard DIMACS Clique benchmark, taken in its complement form so that the target becomes maximum independent set. BHOSLIB is a family of forced-hard random instances widely used to stress MIS solvers. Per-source per-graph minimum and maximum of vertex count $n$ and edge count $m$ are reported, alongside the average edge density. All graphs are labelled with Gurobi-exact OPT.

| Source | Origin | #Graphs | $n$ range | $m$ range | Avg. density |
|---|---|---|---|---|---|
| DIMACS | DIMACS Clique (complemented) | 66 | 28–3,361 | 72–1,026,582 | 0.370 |
| BHOSLIB | BHOSLIB forced-hard random | 35 | 450–1,534 | 17,794–127,011 | 0.137 |
| **Total / overall range** | | **101** | 28–3,361 | 72–1,026,582 | — |

For the empirical kernelization-gap analysis, we also use a synthetic control set of 298 graphs across standard graph families, including Erdős–Rényi, random regular, Barabási–Albert, geometric, Watts–Strogatz, power-law cluster, trees, bipartite graphs, grids, caveman graphs, complements of Erdős–Rényi graphs, hypercubes, and cycles. These graphs have sizes in the range $n \in [60, 300]$ and are used only to show how the same exact rules behave on instances where their local certificates are expected to occur.

The LLM-evolved components are trained on a separate set of 40 Erdős–Rényi graphs with $n \in \{80, 120, 160, 200\}$ and $p \in \{0.15, 0.25, 0.35, 0.45, 0.55\}$, using two seeds per parameter pair. These graphs are used only during evolution and do not appear in any reported test-set figure or table.

**Exact-reduction baselines.** The exact-reduction toolkit consists of seven standard local MIS rules: isolated-vertex reduction (DEG0), pendant reduction (DEG1), degree-two folding in the triangle case (FOLD-TRI), degree-two folding in the nonadjacent-neighbor case (FOLD-NONADJ), domination, simplicial reduction, and unconfined reduction. For the standalone kernelization-gap measurement, each rule is applied alone and exhaustively to a fixed point on every graph. We record whether the rule fires, how many vertices it removes, and its wall-clock time.

For head-to-head heuristic comparisons, all methods use the same exact backbone: isolated-vertex and pendant reductions applied to a fixed point at each outer round. The stronger exact rules are not mixed into these comparisons. This keeps the exact floor fixed, so the comparison isolates the effect of the inexact reduction or heuristic construction step.

**Compared methods.** The baseline heuristic constructor is MinDeg, which repeatedly commits a minimum-degree vertex and removes its closed neighborhood. Lookahead is exhaustive one-step lookahead: for each residual vertex $v$, it temporarily commits $v$, completes the remaining graph with MinDeg, and commits the candidate with the best resulting solution. NeighborLookahead uses the same scoring rule but restricts the candidate set to the current MinDeg pivot and its neighbors.

Table 4: Diverse synthetic dataset used to study the per-family behavior of classical kernelization rules. Each family is sampled at several sizes; $n$ and $m$ ranges report the per-graph minimum and maximum. All graphs are labelled with Gurobi-exact OPT.

| FAMILY | DESCRIPTION | #GRAPHS | $n$ RANGE | $m$ RANGE |
|---|---|---|---|---|
| BA | BARABÁSI–ALBERT | 30 | 100–300 | 196–2,336 |
| BIPARTITE | RANDOM BIPARTITE | 20 | 100–300 | 235–6,697 |
| CAVEMAN | CAVEMAN | 8 | 50–225 | 200–2,175 |
| COMPL-ER | COMPLEMENT OF ERDŐS–RÉNYI | 24 | 80–150 | 2,824–10,604 |
| CONN-CAVEMAN | CONNECTED CAVEMAN | 8 | 50–225 | 200–2,175 |
| CYCLE | CYCLE | 5 | 100–300 | 100–300 |
| ER | ERDŐS–RÉNYI | 60 | 100–300 | 156–22,446 |
| GEOMETRIC | RANDOM GEOMETRIC | 30 | 100–300 | 127–9,610 |
| GRID | 2D GRID | 8 | 64–250 | 112–465 |
| HYPERCUBE | HYPERCUBE | 5 | 16–256 | 32–1,024 |
| PLC | POWER-LAW CLUSTER | 20 | 100–300 | 196–1,181 |
| REGULAR | RANDOM REGULAR | 30 | 100–300 | 150–1,200 |
| TREE | RANDOM TREE | 20 | 100–300 | 99–299 |
| WS | WATTS–STROGATZ | 30 | 100–300 | 200–1,200 |
| **TOTAL / OVERALL RANGE** | | **298** | 16–300 | 32–22,446 |

Table 5: Per-rule activity of seven classical kernelization rules on the diverse synthetic dataset. Each rule is applied alone, exhaustively to fixed point. FIRE is the percentage of graphs where the rule applied at least once. AVG FIRES and AVG REMOVED are computed over the subset of graphs where the rule fired.

| RULE | FIRE (%) | AVG FIRES | AVG REMOVED | MAX REMOVED |
|---|---|---|---|---|
| DEG0 | 4.4 | 3.00 | 3.00 | 6 |
| DEG1 | 18.1 | 35.98 | 71.96 | 262 |
| FOLD-TRI | 14.8 | 8.05 | 24.14 | 81 |
| FOLD-NONADJ | 27.9 | 36.93 | 73.86 | 298 |
| DOMINATION | 54.0 | 65.11 | 65.11 | 212 |
| SIMPLICIAL | 23.5 | 10.96 | 60.60 | 225 |
| UNCONFINED | 56.7 | 74.79 | 74.79 | 255 |

Table 6: Per-rule activity of the same seven classical kernelization rules on the hard MIS benchmark. Same protocol as Table 5. No rule applies on any instance; the more powerful rules nonetheless incur substantial structural-check cost. MEAN TIME and MAX TIME are wall-clock seconds per (graph, rule) task.

| RULE | FIRE (%) | AVG REMOVED | MAX REMOVED | MEAN TIME (S) | MAX TIME (S) |
|---|---|---|---|---|---|
| DEG0 | 0.0 | 0.00 | 0 | 0.12 | 1.68 |
| DEG1 | 0.0 | 0.00 | 0 | 0.12 | 1.69 |
| FOLD-TRI | 0.0 | 0.00 | 0 | 0.12 | 1.68 |
| FOLD-NONADJ | 0.0 | 0.00 | 0 | 0.12 | 1.68 |
| DOMINATION | 0.0 | 0.00 | 0 | 3.05 | 60.63 |
| SIMPLICIAL | 0.0 | 0.00 | 0 | 0.13 | 1.87 |
| UNCONFINED | 0.0 | 0.00 | 0 | 6.52 | 138.08 |

PEELING follows the reducing-peeling template: after exact reductions stall, it discards a high-degree vertex and later extends the constructed independent set greedily when possible.

The reduce-then-solve method is STATICREDUCE. It is an LLM-evolved per-vertex heuristic reducer applied once to the input graph. The produced residual graph is then solved by the downstream exact

solver. The reduce-as-solve method is CACHEREDUCE. It is an LLM-evolved cache policy that consumes evidence from repeated local lookahead rollouts and decides whether to commit or discard vertices in the current residual graph. The policy can either request additional partial-lookahead evidence or act using evidence already stored in the cache.

**Exact solver and reference labels.** We use Gurobi as the exact downstream solver and as the source of reference labels. MIS is encoded as the binary integer program

$$\max \sum_{v \in V} x_v \quad \text{s.t.} \quad x_u + x_v \leq 1 \;\; \forall (u,v) \in E, \qquad x_v \in \{0,1\}.$$

All Gurobi calls are deterministic and single-threaded. On the DIMACS/BHOSLIB benchmark, the reference run uses a one-hour time limit. Gurobi proves optimality on 53 of the 101 instances; on the remaining 48 instances, the reference value is the best incumbent found within the time limit. All methods are evaluated against this same reference value, so ratios are mutually comparable across methods. Values above one can occur only on instances where the reference is a timeout incumbent rather than a proved optimum.

**Metrics.** Shrinkage is measured per graph as

$$\text{shrinkage}(G) = \frac{n_{\text{removed}}}{n}.$$

For exact-rule activity, we also report fire rate, defined as the fraction of graphs on which a rule applies at least once during exhaustive application.

Lifted solution quality is measured as

$$\text{ratio}(G) = \frac{|S_{\text{method}}(G)|}{\text{OPT}_{\text{ref}}(G)},$$

where $\text{OPT}_{\text{ref}}(G)$ is the common Gurobi reference value for the graph. We report mean ratio, worst-case ratio, and counts of instances matching or exceeding the reference value when relevant. Every produced solution is validated as an independent set before being included in any aggregate.

Runtime is measured as wall-clock time around the actual reduction or solve call, excluding scheduling and graph-loading overhead. For exact rules, runtime is measured for the rule's fixed-point application. For reduce-then-solve, we separately record reducer time and downstream solver time on the residual. For heuristic constructors and reduce-as-solve, we report end-to-end wall-clock method time on the graph.

**Compute and reproducibility.** All experiments are run under a single-thread discipline. Gurobi is called with one thread, and all heuristic methods are single-threaded. LLM-evolution loops evaluate candidate programs in parallel across training graphs, but each individual candidate evaluation is single-threaded. Synthetic graph generation and solver calls use fixed seeds. The evolved reducers used in the reported experiments are frozen before deployment on the hard benchmark.

## C EXACT REDUCTION RULES

We list the seven exact reduction rules used as the classical preprocessing baseline in this paper, in the form they are applied to the input graph. Each rule preserves the optimum of the Maximum Independent Set problem: applying it to a graph $G$ produces a smaller graph $G'$ together with a reconstruction step such that any maximum independent set of $G'$ extends to a maximum independent set of $G$.

**Notation.** Let $G = (V, E)$ be an undirected simple graph. For $v \in V$, write $N(v) = \{u \in V : uv \in E\}$ for the open neighborhood and $N[v] = N(v) \cup \{v\}$ for the closed neighborhood. For a set $S \subseteq V$, write $N(S) = (\bigcup_{v \in S} N(v)) \setminus S$ and $N[S] = N(S) \cup S$. Write $\alpha(G)$ for the size of a maximum independent set of $G$.

**Rule 1 — Isolated vertex (`deg0`).** Let $v \in V$ with $\deg(v) = 0$. Then $v$ is in every maximum independent set of $G$. Add $v$ to the independent set and remove $v$ from $G$. After the rule fires, $\alpha(G) = \alpha(G - v) + 1$.

**Rule 2 — Pendant (`deg1`).** Let $v \in V$ with $\deg(v) = 1$ and $N(v) = \{u\}$. Then there exists a maximum independent set of $G$ that contains $v$. Add $v$ to the independent set and remove $\{v, u\}$ from $G$. After the rule fires, $\alpha(G) = \alpha(G - \{v, u\}) + 1$.

*Justification.* If a maximum independent set $I$ contains $u$, replace $u$ by $v$ to obtain an independent set of the same size that contains $v$ instead.

Please refer to the original work (Balasubramanian et al., 1998) introducing the following rules.

**Rule 3 — Degree-two vertex folding, triangle case (`fold_tri`).** Let $v \in V$ with $\deg(v) = 2$ and $N(v) = \{u_1, u_2\}$ such that $u_1u_2 \in E$ (so $\{v, u_1, u_2\}$ induces a triangle). Then there exists a maximum independent set of $G$ that contains $v$. Add $v$ to the independent set and remove $\{v, u_1, u_2\}$ from $G$. After the rule fires, $\alpha(G) = \alpha(G - \{v, u_1, u_2\}) + 1$.

*Justification.* The triangle is a 3-clique, so any independent set contains at most one of $\{v, u_1, u_2\}$. Choosing $v$ excludes only $u_1$ and $u_2$, both of which would already be mutually exclusive with each other; choosing $u_1$ or $u_2$ may exclude additional vertices outside the triangle.

**Rule 4 — Degree-two vertex folding, non-adjacent case (`fold_nonadj`).** Let $v \in V$ with $\deg(v) = 2$ and $N(v) = \{u_1, u_2\}$ such that $u_1u_2 \notin E$. Construct the folded graph $G'$ from $G$ by

1. removing $v$, $u_1$, and $u_2$ from $V$;
2. adding a new vertex $w$ with neighborhood $N_{G'}(w) = (N(u_1) \cup N(u_2)) \setminus \{v\}$.

Then $\alpha(G) = \alpha(G') + 1$. A maximum independent set $I'$ of $G'$ extends to a maximum independent set $I$ of $G$ as

$$I = \begin{cases} (I' \setminus \{w\}) \cup \{u_1, u_2\}, & \text{if } w \in I', \\ I' \cup \{v\}, & \text{if } w \notin I'. \end{cases}$$

*Justification.* Either $v$ is in the optimum (and neither $u_1$ nor $u_2$ is), or both $u_1$ and $u_2$ are (since they are non-adjacent and any vertex outside $N(v)$ can be paired with both). Folding represents the second alternative by the merged vertex $w$, whose presence in $I'$ encodes "include both $u_1$ and $u_2$".

**Rule 5 — Domination.** Let $u, v \in V$ with $u \neq v$ and $N[u] \subseteq N[v]$ (equivalently, $uv \in E$ and $N(u) \setminus \{v\} \subseteq N(v)$). Then there exists a maximum independent set of $G$ that does not contain $v$. Remove $v$ from $G$. After the rule fires, $\alpha(G) = \alpha(G - v)$.

*Justification.* If a maximum independent set $I$ contains $v$, replace $v$ by $u$: any neighbor of $u$ in $I$ would already be a neighbor of $v$, so the swap preserves independence and the cardinality.

**Rule 6 — Simplicial vertex.** Let $v \in V$ such that $N(v)$ induces a clique in $G$ (i.e., $v$ is simplicial). Then there exists a maximum independent set of $G$ that contains $v$. Add $v$ to the independent set and remove $N[v]$ from $G$. After the rule fires, $\alpha(G) = \alpha(G - N[v]) + 1$.

*Justification.* Any independent set contains at most one vertex of the clique $N(v)$. If a maximum independent set $I$ contains some neighbor $u \in N(v)$, replace $u$ by $v$: $v$ is non-adjacent to all other vertices of $G$ that were in $I$ (since those vertices were already non-adjacent to $u$ and to each other). Rules 1, 2, and 3 are special cases of this rule for $\deg(v) = 0$, $\deg(v) = 1$, and the triangle subcase of $\deg(v) = 2$.

**Rule 7 — Unconfined vertex Akiba & Iwata (2016); Xiao & Nagamochi (2013).** A vertex $v \in V$ is *unconfined* if the procedure CONFINE($v$) below returns UNCONFINED:

1. Initialize $S \leftarrow \{v\}$.
2. Repeat:
   - If there exists $u \in N(S)$ with $|N(u) \cap S| = 1$ and $|N(u) \setminus N[S]| = 0$, return UNCONFINED.
   - Else if there exists $u \in N(S)$ with $|N(u) \cap S| = 1$ and $|N(u) \setminus N[S]| = 1$, let $w$ be the unique vertex of $N(u) \setminus N[S]$, set $S \leftarrow S \cup \{w\}$, and continue.

- Otherwise, return CONFINED.

If $v$ is unconfined, then no maximum independent set of $G$ contains $v$. Remove $v$ from $G$. After the rule fires, $\alpha(G) = \alpha(G - v)$.

*Justification (sketch).* The set $S$ is constructed so that any independent set containing $v$ can be transformed into one containing some other vertex of $S$ without loss; the UNCONFINED exit witnesses a vertex $u$ that forces $v$ to be strictly suboptimal. A full proof appears in (Akiba & Iwata, 2016, Lemma 2.3).

# D LLM-AUGMENTED RULE SYNTHESIS

## D.1 EVOLUTIONARY LOOP

Both GENREDUCE and CACHEREDUCE are produced by the same LLM-driven program-evolution loop. Each round, the loop samples a small set of parent rules from the archive, presents them to the LLM with the task prompt, and asks for a structurally distinct candidate. The candidate is validated for syntactic correctness, run on a held-out training set of 40 small Erdős–Rényi graphs, scored on a quality measure (mean OPT-ratio plus reduction fraction for GENREDUCE; mean OPT-ratio with a wall-clock penalty for CACHEREDUCE), and added to the archive. At the end of training, the highest-scoring rule is frozen and deployed out-of-distribution on the DIMACS+BHOSLIB hard MIS benchmark.

## D.2 LLMS USED

For each method, the rule search was run with two LLMs; the rule reported in this paper is the best across all runs by mean held-out OPT-ratio.

| Search | LLMs |
|---|---|
| GENREDUCE (reduce-then-solve) | `gemini-2.5-flash`, `gpt-4.1-mini-2025-04-14` |
| CACHEREDUCE (reduce-as-solve) | `gemini-2.5-flash`, `gpt-4.1-mini-2025-04-14` |

## D.3 PROMPTS

The prompt template is shared between the two LLMs and differs across methods only in the task definition (single-pass reduction vs cache-driven IN/OUT decision) and the scoring formula. Three parents sampled from the archive are interpolated into the user prompt each round, with their descriptions and code shown verbatim. The base prompts are reproduced below as sent to the API.

The runtime user prompt is assembled by appending an archive tail to a static base `PROMPT_BASE`: a header line, three sampled parent rules with their stats and code, and a closing task instruction. The two methods share this assembly logic and differ only in the per-method strings, as shown below.

**Prompt assembly — GENREDUCE**

```
function build_prompt(parents):
if parents is empty:
return PROMPT_BASE
tail = "## Previously proposed reduction heuristics\n"
tail += "Propose a STRUCTURALLY DIFFERENT mechanism from every "
tail += "one below -- a different signal, selection strategy, "
tail += "or drop fraction. Not a parameter tweak.\n"
for i, p in enumerate(parents, start=1):
tail += "### Candidate {i} (ratio_mean=..., ratio_min=..., "
tail +=     "reduction_mean=..., frac_valid=..., "
tail +=     "time_mean=...s)\n"
tail += "Theorem:\n" + p.theorem + "\n"
tail += "Code:\n```python\n" + p.code + "\n```\n"
tail += "## Your task\n"
tail += "Propose a NEW reduction that (a) uses a structurally "
tail += "distinct mechanism, (b) drops a meaningful fraction, "
tail += "(c) keeps both ratio_mean and ratio_min high, "
tail += "(d) runs in O(n log n) or O(n+m).\n"
```

```
    return PROMPT_BASE + tail
```

**Prompt assembly — CACHEREDUCE**

```
function build_prompt(parents):
if parents is empty:
return PROMPT_BASE
tail = "## Previously proposed candidate cache reducers\n"
tail += "Propose a STRUCTURALLY DIFFERENT mechanism from every "
tail += "one below -- different scoring formula, different "
tail += "confidence thresholds, different IN/OUT policy. "
tail += "Not a parameter tweak.\n"
for i, p in enumerate(parents, start=1):
tail += "### Candidate {i} (ratio_mean=..., time_mean=...s, "
tail +=     "frac_valid=...)\n"
tail += "Theorem:\n" + p.theorem + "\n"
tail += "Code:\n```python\n" + p.code + "\n```\n"
tail += "## Your task\n"
tail += "Propose a NEW pick_and_update that (a) uses a "
tail += "structurally distinct mechanism, (b) achieves higher "
tail += "OPT-ratio AND/OR lower wall-clock time by exploiting "
tail += "cache-driven IN or OUT reductions when confident.\n"
return PROMPT_BASE + tail
```

**Prompt — GENREDUCE**

```
You are evolving a single Python function for the Maximum Independent
Set (MIS) problem:

def reduction(g) -> set[int]

## What the function must do

Given a graph `g` (a networkx Graph), return a set of node IDs that
your rule predicts are NOT in any maximum independent set of `g`.

**You decide how many nodes to drop.** The harness drops every valid
unique node ID you return -- no fill, no trim. So the drop fraction
is part of your design: a rule that returns 5% of n is making a
different bet from a rule that returns 25% of n.

You should write one line of code that picks the drop fraction. Pick
a fraction that you can defend -- rules that drop more get rewarded
more by the score, but only if the drops stay safe.

## How the harness uses your function (one call per graph)

1. Call your `reduction(g)` once on the original graph.
2. Take the set you returned.
3. Remove those nodes from `g`.
4. Solve the residual exactly with Gurobi (TimeLimit = 180 s, accept
best incumbent).
5. Compare OPT(residual) vs OPT(original). OPT(original) was labelled
by Gurobi with the same 180 s budget so the two numbers are directly
comparable.

## How safety is checked -- empirically, not by proof

You do NOT need to prove your rule is safe. You do NOT need a structural
correctness argument. The harness verifies safety empirically:

ratio = OPT(residual) / OPT(original) in [0, 1]

If your rule drops only "non-OPT" nodes (or nodes for which an
alternative maximum IS still exists in the residual), `ratio = 1.0`.
If your rule drops a node that strictly shrinks the achievable MIS,
`ratio < 1.0`.

## How you are scored

ratio_mean       mean(OPT(residual) / OPT(original))    -- average safety
ratio_min        worst-case OPT(residual) / OPT(original)  -- robustness
reduction_mean   mean(|dropped| / |V(G)|)               -- aggression
frac_valid       fraction of graphs Gurobi solved successfully
time_mean        mean wall-clock seconds per graph (rule + Gurobi)

score = 0.1 * frac_valid
+ 0.7 * ratio_mean + 0.3 * ratio_min
```

```
+ reduction_mean
- 1
- 0.05 * min(time_mean, 180)/180
+ 2 * max(0, ratio_mean - 0.9)

The score weights ratio in two ways:
- the BULK weight (0.7) on ratio_mean rewards average safety;
- a smaller weight (0.3) on ratio_min punishes rules that drop the
worst graph in the training set badly. So a rule that averages
0.95 but has one graph at 0.6 is worse than a rule that averages
0.94 with min 0.85.
- the +2 * max(0, ratio_mean - 0.9) bonus rewards near-perfect
average ratio: above 0.9, every percentage point of ratio_mean
earns 0.02 extra score, on top of the 0.7 weight already there.
The time penalty (-0.05 * time_mean/180) is moderate and uses
time_mean (stable) rather than time_max (which gets pegged when the
Gurobi residual solve hits its 180 s TimeLimit).

Read this carefully (assuming frac_valid = 1.0 and small time_mean):

- dropping nothing (ratio=1, reduction=0)              -> score ~= 0.30
- dropping 25% safely (ratio=1, reduction=0.25)        -> score ~= 0.55
- dropping 25%, ratio_mean=0.95, ratio_min=0.85        -> score ~= 0.34
- dropping 25%, ratio_mean=0.95, ratio_min=0.65        -> score ~= 0.28
- dropping 25% safely AND fast                         -> adds nothing extra
- dropping 25% safely BUT slow (time_mean = 60 s)      -> loses ~0.017 score

The right move is: drop a meaningful fraction while keeping BOTH
the average ratio AND the worst-case ratio high, and stay reasonably
fast.

## Your task -- rank, then pick a defensible top-k

Think of your function as ranking the nodes of `g` from "most likely
not in OPT" to "least likely not in OPT", then returning the top
fraction of that ranking. You decide the cut-off.

Suggested signals you could use (mix and match, invent your own):

- structural: pendant, domination, twin, mirror, crown, vertex-folding
- degree-based: extremes of degree, density of N(v), neighbour-degree
statistics
- randomness / sampling: build several quick greedy ISes via random
tie-breaking, count how often each node appears, drop nodes with
zero or low appearance
- hybrid: combine multiple signals additively, by veto, or by cutoff

Pure structural rules (textbook reductions like pendant or domination)
fire rarely on dense graphs -- they will return tiny sets and score
near 0. Sampling-based or scoring-based rules tend to fire on every
graph, so they can drop a meaningful fraction.

## CRITICAL: efficiency requirement (read this twice)

Your `reduction(g)` MUST run in **O(n log n)** or **O(n + m)** time.
The score's time term penalizes the WORST-CASE per-graph wall-clock
HEAVILY (coefficient 0.10), so a rule that is asymptotically slow on
even one dense graph in the dataset will lose meaningful score.

**HARD BAN -- do NOT use any of these patterns** (each is O(deg^2) per
vertex, which is O(n * deg^2) overall -- far too slow on dense graphs):

- `for v in g.nodes: for u in g.neighbors(v): for w in g.neighbors(v): ...`
(counting non-edges, P3 paths, induced paths, etc. per vertex)
- `for v in g.nodes: for pair in itertools.combinations(g.neighbors(v), 2): ...`
- per-vertex enumeration of all neighbour pairs for any structural check
- exhaustive triangle / 4-cycle enumeration through nested neighbour loops
- per-node `O(n)` neighbourhood comparisons against every other node
- any data-structure rebuild that is `Omega(n^2)`

**ALLOWED patterns** (linear or near-linear total work):

- degree-based scoring: per vertex, score is a function of `deg(v)` alone
- single-pass formulas: per vertex, constant-time scoring from local features
- neighbourhood-hash signatures: hash `frozenset(g.neighbors(v))` once per vertex
- sampled subsets: pick a small random subset of edges/vertices and score
- randomised greedy passes: build a greedy IS once and score by membership
- sorted-degree scans, low-degree pruning, single-edge iteration

If the per-vertex check requires looking at neighbour pairs, you must
either (a) sample a small constant number of pairs, or (b) reformulate
```

````
to avoid the nested loop.

## What you have to work with

```
g.nodes              residual node ids (int)
g.degree(v)          degree of v in g
g.neighbors(v)       iterator over N(v) in g
g.has_edge(u, v)     O(1) edge lookup
```

`g` is a networkx Graph. **Do NOT mutate `g` inside `reduction`** --
the harness owns the graph state.

## Output schema

- `theorem`: 2 to 5 short sentences. State (a) what signal your rule
uses to predict non-OPT nodes, (b) why you expect it to be predictive
on these graphs, (c) the drop fraction you chose and why, (d) the
time complexity of your test (must be O(n log n) or better). The
signal does NOT need a formal proof -- the harness checks safety
empirically.

- `code`: exactly one top-level `def reduction(g):` function, plus any
helper top-level functions you need. Polynomial time STRICTLY faster
than O(n^2). No docstrings inside the function body. Short variable
names.

## Skeleton

def reduction(g):
n = g.number_of_nodes()
# 1. score every node by predicted "not in OPT"
# 2. pick a drop fraction (your design choice -- defend it in theorem)
k = max(1, int(n * 0.15))     # example: 15% -- tune for your rule
# 3. take the top-k by your score
# 4. return as a set of node IDs
return set()  # replace with your top-k selection
````

**Prompt — CACHEREDUCE**

````
You are evolving a smart **cache-driven reducer** for the Maximum
Independent Set (MIS) framework below. The framework already has a
working algorithm; you only design ONE function: `pick_and_update`.

## The whole point of the cache (read this first)

The cache is a list of **past 1-lookahead snapshots** -- complete ISes that
prior rounds' rollouts produced. We keep it because every fresh
`partial_lookahead` call is expensive (proportional to deg(u)*O(n+m))
and the snapshots from past rounds hold useful evidence about which
vertices belong in good ISes and which don't.

Your job is to use that evidence to **REDUCE the residual graph** when
you are CONFIDENT, by one of two actions:

IN   -- vertex v is very likely IN the optimum. Commit it to the IS.
Remove v and N(v) from the residual.    (shrinks by deg(v)+1)

OUT  -- vertex v is very likely NOT in the optimum. Drop it.
Remove just {v} from the residual.       (shrinks by 1)

When you are NOT confident enough to make either reduction, FALL BACK to
the normal pick approach: call `helpers.partial_lookahead(rg)` and commit
the argmax of |S_v| (a force-IN). The fallback is what plain 1-step
lookahead does every round -- it is correct and produces a high-quality
IS, but every call is expensive. **The whole point of having a cache is
to skip the expensive fallback when confident reductions are available.**

## The framework (you do NOT change this)

Outer loop, fixed by the harness:

```python
def solve(G):
rg = ReducibleGraph(G.copy())
cache = []                              # list[frozenset] of past IS snapshots
while rg.g.number_of_nodes() > 0:
reduce_pendant(rg)                # exact pendant rule, fixed
if rg.g.number_of_nodes() == 0:
````
````

````
    break
    action, v, cache = pick_and_update(rg, cache, helpers)
    if action == "IN":
    rg.stack.append(ForceInRecord(v))
    rg.g.remove_nodes_from([v] + list(rg.g.neighbors(v)))
    elif action == "OUT":
    rg.stack.append(ForceOutRecord(v))
    rg.g.remove_node(v)
    return rg.reconstruct(set())
    ```

    So your function is called once per round. Returns `(action, vertex, cache)`
    where `action in {"IN", "OUT"}`.

    ## What you have to work with

    ```
    helpers.partial_lookahead(rg) -> list[(seed_vertex, frozenset)]
    Runs 1-step lookahead at the mindeg pivot u of rg.g; returns
    (deg(u)+1) (seed, snapshot) pairs -- one per candidate in
      {u} U N(u), in candidate order. The seed is the candidate vertex
    v itself; the snapshot is the IS that mindeg produced after
    committing v. Use the seed when you want to commit a vertex
    consistent with its snapshot (typical for the bootstrap and the
    fallback paths). Use only the snapshots (the second element) when
    you score residual vertices for cache-driven reductions.
    EXPENSIVE -- proportional to deg(u)*O(n+m). This is the call you
    want to AVOID when the cache already gives confident reductions.

    The cache itself stores ONLY snapshots: `cache: list[frozenset]`. To
    extend the cache from a fresh lookahead use:
    cache.extend(s for _, s in pairs)

    helpers.mindeg(g) -> set
    Plain min-degree greedy IS on g. Cheap.

    rg.g                          networkx Graph (the residual; do not mutate inside
pick_and_update)
    list(rg.g.nodes), rg.g.degree(v), rg.g.neighbors(v)
    ```

    ## Two directions of cache evidence

    The cache is `list[frozenset]`. Each frozenset is a complete IS that
    some past rollout produced. Per residual vertex v:

    - **IN evidence** -- v frequently appears in cached snapshots, especially
    in large ones. Suggests v belongs in some maximum IS. Strong -> "IN".
    - **OUT evidence** -- v rarely or never appears in any snapshot, OR only
    in small / shrinking ones. Suggests v is unlikely to be in any
    large IS. Strong -> "OUT".

    The exact aggregate (mean / max / count / weighted-by-size / etc) and
    the confidence threshold are YOUR design.

    ## A snapshot is a vote, not a usable IS

    A past snapshot S is informative about a vertex v as long as v in S and
    v is currently in the residual. You do NOT need every vertex of S to
    still exist -- the snapshot just contributes a vote.

    ## Fitness -- you are scored on OPT-ratio AND wall-clock time

    For each test graph G with Gurobi-certified OPT, the harness measures:

    ratio = |solve(G)| / OPT          (in [0, 1] for valid solutions)
    time  = wall-clock seconds of solve(G)
    valid = reconstructed IS is independent in the original graph

    Per-rule aggregate score:

    score = 0.1*frac_valid + 1.0*ratio_mean - 0.5*time_mean

    So: HIGHER ratio AND LOWER time. The time weight is INTENTIONALLY
    strong (0.5, not 0.05): a rule that always falls back to
    partial_lookahead achieves high ratio but pays full lookahead cost on
    every round, and the time penalty makes that strictly inferior to a
    rule that skips even half the fallbacks while preserving ratio. A rule
    that fires reductions when it shouldn't loses ratio. The whole point
    of the cache is to find the calibration where confident cache-driven
    IN/OUT reductions skip the expensive fallback without ratio loss.
````
````

```
Anti-degeneracy: a rule that falls back to partial_lookahead every
round is structurally equivalent to method_reuse (= column C). To
beat C, your cache-driven IN/OUT decisions MUST fire on a meaningful
fraction of rounds while keeping ratio close to C's.

The per-graph eval timeout is 10 minutes. A graph that times out
contributes 0 to ratio and is_size.

## Cold start -- IMPORTANT

If `cache` is empty (typical at the very first round, or if you have
never extended it before), you have **no evidence at all** and CANNOT
make a confident IN or OUT decision. In that case you MUST call
`helpers.partial_lookahead(rg)` to bootstrap the cache, and commit the
seed vertex of the LARGEST snapshot (a force-IN). The skeleton below
shows the exact code -- copy it.

Do NOT short-circuit the empty-cache case with a trivial vertex commit
  (e.g. `list(g.nodes())[0]`, or `nodes[i]` for some i derived from
snapshot ordering -- `nodes` is NOT aligned with the lookahead seed
order). That bypasses the lookahead, never populates the cache, and
leaves the rule running on an empty cache forever.

The same rule applies later in the run: any time you reach a state
where you have NO basis for either IN or OUT, the correct action is
`helpers.partial_lookahead(rg)` + extend cache + commit the largest
snapshot's seed.

## Output schema

- `theorem`: 2 to 5 short sentences. State (a) what evidence in the
cache makes you call IN vs OUT vs fall back, (b) the scoring formula,
(c) the confidence thresholds. Commit to ONE specific design.
- `code`: exactly one top-level `def pick_and_update(rg, cache, helpers):`
function, plus any helper top-level functions you need. Polynomial
time. No docstrings inside the function body. Short variable names.

## Skeleton

def pick_and_update(rg, cache, helpers):
g = rg.g

# 1. BOOTSTRAP -- if cache is empty, lookahead first. Do NOT
#                 commit a free vertex without evidence.
if not cache:
pairs = helpers.partial_lookahead(rg)        # list[(seed, snap)]
cache.extend(s for _, s in pairs)            # cache stores snaps only
best_v, _ = max(pairs, key=lambda p: len(p[1]))
return ("IN", best_v, cache)

# 2. score each residual vertex from the cache (your formula).
# 3. if some vertex has STRONG IN evidence  -> return ("IN",  v, cache)
# 4. else if some vertex has STRONG OUT evidence -> return ("OUT", v, cache)
# 5. else FALL BACK (no candidate qualifies):
#        pairs = helpers.partial_lookahead(rg)
#        cache.extend(s for _, s in pairs)
#        best_v, _ = max(pairs, key=lambda p: len(p[1]))
#        return ("IN", best_v, cache)
return action, vertex, cache
```

### D.4 SELECTED RULES

The rules below are the ones used to produce the empirical results in the main paper. Each rule is shown with the LLM's own description of its mechanism (its "theorem" field), followed by the verbatim Python code as it was evaluated.

**Rule — GENREDUCE**

*We rank vertices by a 2-hop local density signal: high-degree vertices whose neighbors have high degree and whose immediate neighborhood has many edges are more likely to sit in the crowded part of the graph and be absent from some maximum independent set. This is structurally different from greedy-witness counting because it never simulates independent sets; it only uses degree, a one-pass edge accumulation, and a two-step neighbor-degree summary. We drop the top* $14\%$ *for moderate, safe aggression that still fires on dense and clustered graphs*

*without overcommitting. The test is $O(n+m)$ to accumulate edge-based neighbor-degree scores, plus $O(n \log n)$ for selecting the top vertices.*

```
def reduction(g):
    n = g.number_of_nodes()
    if n == 0:
        return set()
    nodes = list(g.nodes())
    deg = dict(g.degree())
    m = g.number_of_edges()
    if n == 1:
        return set()
    s1 = {v: 0.0 for v in nodes}
    for u, v in g.edges():
        du = deg[u] + 1.0
        dv = deg[v] + 1.0
        s1[u] += 1.0 / dv
        s1[v] += 1.0 / du
    avg = (2.0 * m / n) + 1.0
    sc = {}
    for v in nodes:
        dv = deg[v]
        sc[v] = 1.8 * dv / avg + 2.6 * s1[v] + 0.000001 * ((v * 2654435761) & 1023)
    k = max(1, int(n * 0.14))
    return set(v for v, _ in sorted(sc.items(), key=lambda x: (-x[1], x[0]))[:k])
```

Held-out training (40 Erdős–Rényi graphs): mean OPT-ratio $0.991$, valid on $40/40$, mean reduction $13.7\%$ per graph.

## Rule — CACHEREDUCE

*We use a scoring scheme based on a weighted frequency-difference measure: for each vertex $v$, we compute the weighted count of snapshots containing $v$, weighting each snapshot by its size and normalizing by total snapshot weight; similarly, we compute the weighted average frequency of $v$'s neighbors. The confidence score is the difference between $v$'s weighted frequency and that of its neighbors. A high positive score is strong IN evidence ($v$ appears more often, and in larger snapshots, than its neighbors); a high negative score is strong OUT evidence. We commit IN to the vertex with highest positive score above $0.5$, OUT to the vertex with lowest negative score below $-0.4$ and degree $\leq 5$, and otherwise fall back to `partial_lookahead`. The mechanism is structurally distinct from frequency-ratio, median-snapshot, or set-operation scoring: weighted frequency differences balance snapshot size against vertex exclusivity with tunable confidence thresholds.*

```
def pick_and_update(rg, cache, helpers):
    g = rg.g
    nodes = list(g.nodes)

    if not cache:
        pairs = helpers.partial_lookahead(rg)
        cache.extend(s for _, s in pairs)
        best_v, _ = max(pairs, key=lambda p: len(p[1]))
        return ("IN", best_v, cache)

    total_weight = sum(len(s) for s in cache)
    if total_weight == 0 or len(nodes) == 0:
        pairs = helpers.partial_lookahead(rg)
        cache.extend(s for _, s in pairs)
        best_v, _ = max(pairs, key=lambda p: len(p[1]))
        return ("IN", best_v, cache)

    from collections import defaultdict

    # Weighted frequency per vertex: sum of snapshot sizes containing v / total_weight
    wfreq = defaultdict(float)
    for s in cache:
        w = len(s)
        for v in s:
            wfreq[v] += w

    # Normalize
    for v in wfreq:
        wfreq[v] /= total_weight

    scores = {}
    for v in nodes:
```

```
        v_wfreq = wfreq.get(v, 0.0)
        nbrs = list(g.neighbors(v))
        if nbrs:
            nbr_wfreq_avg = sum(wfreq.get(nbr, 0.0) for nbr in nbrs) / len(nbrs)
        else:
            nbr_wfreq_avg = 0.0
        # Score: difference between vertex wfreq and neighbors average wfreq
        scores[v] = v_wfreq - nbr_wfreq_avg

    # Thresholds
    IN_THRESHOLD = 0.5    # significantly more included than neighbors
    OUT_THRESHOLD = -0.4  # significantly less included than neighbors
    MAX_OUT_DEGREE = 5

    in_candidates = [(v, sc) for v, sc in scores.items() if sc >= IN_THRESHOLD]
    if in_candidates:
        v_in = max(in_candidates, key=lambda x: x[1])[0]
        return ("IN", v_in, cache)

    out_candidates = [v for v, sc in scores.items() if sc <= OUT_THRESHOLD and
     g.degree(v) <= MAX_OUT_DEGREE]
    if out_candidates:
        v_out = min(out_candidates, key=lambda v: scores[v])  # pick vertex with lowest
     score
        return ("OUT", v_out, cache)

    pairs = helpers.partial_lookahead(rg)
    cache.extend(s for _, s in pairs)
    best_v, _ = max(pairs, key=lambda p: len(p[1]))
    return ("IN", best_v, cache)
```

Held-out training (40 Erdős–Rényi graphs): mean OPT-ratio 0.983, valid on $40/40$.

### D.5 Closing remark

Neither rule is an exact kernelization rule. Both were selected by held-out evaluation on a separate Erdős–Rényi training set and frozen for out-of-distribution deployment on the hard MIS benchmark. Their value lies in playing the operational role of reduction on hard residuals where exact rules are silent — empirically closing the practical kernelization gap rather than improving the kernelization theory itself.