WHEN DOES STRUCTURED KNOWLEDGE HELP NEURAL THEOREM PROVING?

Anonymous authors
Paper under double-blind review

ABSTRACT

Does structured mathematical knowledge help LLMs prove theorems in Lean 4? If so, for which models, and does the answer vary by problem? Formal libraries such as MATHLIB encode 285,000+ verified theorems with syntactic dependencies, but the semantic layer mathematicians rely on for discovery, such as analogies, generalizations, and cross-domain bridges, remains implicit. We introduce MATH-AGENT, a system that builds this missing layer as a knowledge graph called MATHKG, and uses it to augment LLM theorem provers. MATHKG connects 364 MATHLIB theorems and definitions by 9,434 typed semantic edges, inferred via LLM-based relation extraction anchored to verified MATHLIB declarations. We run a controlled ablation across four augmentation modes (no external context, knowledge-graph context, MATHLIB library retrieval, and both combined) and five models: the general-purpose Qwen3-8B/32B, their Lean-specialized derivatives Goedel-Prover-V2-8B/32B, and the proprietary Claude Sonnet 4.6. We evaluate on miniF2F, adding PutnamBench and MathOlympiadBench for the proprietary model. Three findings emerge. (i) *Specialization dominates augmentation* by an order of magnitude: Lean fine-tuning adds 33–38 percentage points of solve rate in *every* augmentation mode, and a specialized 8B model even beats a $4\times$ larger general one by 29–35 points in every mode; by contrast, no single augmentation mode improves solve rate by more than 3 points, so external knowledge does not substitute for competence in the weights. (ii) Augmentation's aggregate effect is *capability-conditioned*: knowledge-graph context helps small models but hurts large ones, with the specialized model gaining more relative to its general base at every scale. (iii) Yet across every model the augmentation modes solve *different* problems, so an oracle that selects the best augmentation mode per problem solves 6% to 58% more problems than the unaugmented prover (least for the proprietary model, most for a weak general one), revealing a *complementarity effect* that strengthens on harder problems. Together these results motivate adaptive strategies that select augmentation by model capability and problem. Code, data, and artifacts will be released upon publication.

1 INTRODUCTION

Large language models are rapidly becoming capable theorem provers. Combined with search and formal verification, LLMs now produce IMO-level proofs (Achim et al., 2025; Hubert et al., 2026), and LLM-based systems increasingly contribute to resolving open problems (Romera-Paredes et al., 2024; Novikov et al., 2025; Georgiev et al., 2025; Tsoukalas et al., 2026; OpenAI, 2026; Alon et al., 2026; Ono, 2026). Specialized open provers are built on the Lean 4 proof assistant (Moura & Ullrich, 2021) and its MATHLIB library (The mathlib Community, 2020) of 285,000+ theorems,[1] and are fine-tuned on large corpora of machine-verified Lean proofs (Ren et al., 2025; Lin et al., 2026): everything they know about mathematics is stored in their weights.

Mathematicians work differently. Beyond what any individual knows, they draw on a shared semantic web of the field: which theorems are analogous across subfields, which results generalize others, which identities bridge number theory to combinatorics. As proofs become abundant, Tao (2026)

[1] MATHLIB spans algebra, analysis, combinatorics, number theory, topology, and beyond; it contains 285,000+ theorems and 135,000+ definitions (The mathlib Community, 2026).

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.

argues, emphasis should move from proof generation to proof digestion: exposition, refereeing, and connecting results into a shared body of knowledge. This paper asks: if we make this layer explicit and hand it to an LLM prover at inference time, does it prove more theorems? If so, for which models, and does the answer vary by problem?

Formal libraries such as MATHLIB encode *syntactic* structure: which theorems are used to prove others and which definitions they invoke. The *semantic* relationships that guide discovery live in textbooks, seminars, and expert intuition. We build this layer explicitly from a curated core. MATHKG is a knowledge graph (KG) of 364 MATHLIB-verified theorems and definitions connected by 9,434 typed semantic edges in eight categories (e.g., generalizes, analogous-to, cross-domain-bridge), inferred by LLM-based relation extraction over the nodes' statements and verified MATHLIB declaration names. MATHAGENT, four LLM agents coordinated by an Orchestrator over MATHKG as shared memory, injects knowledge-graph context, MATHLIB library retrieval, or both into an LLM prover with error-guided refinement; the ablation exercises only its Explorer and Prover.

Retrieval- and KG-augmented provers report gains for particular models (§2), but to our knowledge no controlled study isolates the effect of structured knowledge across matched general and Lean-specialized model pairs at multiple scales. We therefore evaluate every combination of four augmentation modes (none, knowledge-graph context, lexical MATHLIB library retrieval, both) and five models. Four form a matched 2×2 of general-purpose versus Lean-specialized provers at 8B and 32B: Qwen3-8B/32B (Yang et al., 2025) versus Goedel-Prover-V2-8B/32B, each fine-tuned from its Qwen3 base (Lin et al., 2026). The fifth, Claude Sonnet 4.6 (Anthropic, 2026), serves as a proprietary reference. All five are evaluated at pass@32 on miniF2F, and the proprietary model additionally on PutnamBench and MathOlympiadBench.

Contributions. We make four contributions:

1. **MATHKG**: A semantic knowledge graph of 364 MATHLIB-verified theorems and definitions across seven domains with 9,434 typed edges in eight relational categories, built by an LLM relation-extraction pipeline anchored to verified MATHLIB declarations (§3).
2. **MATHAGENT**: Four LLM agents (Explorer, Conjecturer, Prover, Integrator) coordinated by an Orchestrator, for targeted proving (evaluated here) and open-ended exploration, with MATHKG as persistent shared memory and error-guided Lean 4 verification (§4).
3. **Controlled cross-model ablation**: A systematic evaluation of four augmentation modes (KG context, MATHLIB retrieval, both, or neither) across matched general-purpose Qwen3 vs. Lean-specialized Goedel-Prover-V2 pairs at 8B and 32B, plus the proprietary Claude Sonnet 4.6, on miniF2F, extended to PutnamBench and MathOlympiadBench (§5).
4. **Three findings on when structured knowledge helps**: (i) *specialization dominates augmentation*: Lean fine-tuning adds 33–38 percentage points in every mode, a specialized 8B model beats a general model four times its size, and no augmentation mode improves any model's aggregate by more than 3 points; (ii) augmentation is *capability-conditioned*: knowledge-graph context helps small models but hurts large ones, with the specialized model gaining more relative to its general base at every scale; (iii) the aggregates hide *complementarity*: the modes solve different problems, so an oracle selecting the best mode per problem solves 6–58% more than the unaugmented prover, a margin that grows on harder benchmarks and motivates adaptive selection (§5.2–5.4).

2 RELATED WORK

AI for theorem proving. Since GPT-f brought language models to formal proving (Polu & Sutskever, 2020), the field has produced headline systems such as Aristotle (Achim et al., 2025) and AlphaProof (Hubert et al., 2026). Lean-specialized provers such as Goedel-Prover (Lin et al., 2025; 2026), DeepSeek-Prover-V2 (Ren et al., 2025), Kimina-Prover (Wang et al., 2025), BFS-Prover-V2 (Xin et al., 2025), and Pythagoras-Prover (Leang et al., 2026) push miniF2F upward through fine-tuning, expert iteration, RL, and tree search (Lample et al., 2022). Self-play (Dong & Ma, 2025; Bailey et al., 2026) and RL against Lean feedback (Chen et al., 2025a;b) extend training beyond fixed problem sets, and Leanstral 1.5 (Kabra et al., 2026) saturates miniF2F as a general code agent.

Minerva (Lewkowycz et al., 2022) and Llemma (Azerbayev et al., 2024) are trained on mathematical text and solve problems mainly in natural language. Draft, Sketch, and Prove (Jiang et al., 2022) guides formal provers with informal sketches, autoformalization has grown from competition statements (Wu et al., 2022) to verified libraries at scale (Rammal et al., 2026), and Pseudo-Formalization (Barkallah et al., 2026) verifies natural-language proofs in pseudo-formal blocks. Agentic frameworks such as Hilbert (Varambally et al., 2026), HERMES (Ospanov et al., 2026), and LEAP (Kung et al., 2026) interleave informal reasoning with iterative Lean verification, building on COPRA (Thakur et al., 2024) and Lemur (Wu et al., 2024); MATEK (Kerger, 2026) adds a persistent typed graph of claims, proofs, and audits as cross-session memory for research agents; and general multi-agent frameworks (Wu et al., 2023; Hong et al., 2023) supply generic orchestration, not mathematical reasoning. All draw their strength from training or from agentic scaffolding; none isolates the contribution of structured knowledge supplied at inference time, our central question.

Retrieval-augmented theorem proving. Premise selection (Irving et al., 2016; Wang et al., 2017; Mikuła et al., 2024) ranks lemmas by learned embeddings; LeanDojo (Yang et al., 2023) trains the retrieval-augmented ReProver on Lean premises, and Lean Copilot (Song et al., 2025) integrates such models natively into Lean. REAL-Prover (Shen et al., 2025) adds MATHLIB retrieval to a stepwise prover, LeanSearch v2 (Gao et al., 2026) retrieves whole premise sets, and DRIFT (Zhang et al., 2026) decomposes statements before retrieval, nearly doubling F1 on ProofNet (Azerbayev et al., 2023). Our Explorer also builds problem-level context, combining MATHLIB retrieval with MATHKG's typed relations and their reasoning. Khalov et al. (2026) and Ataeva et al. (2026) retrieve lemma hints for DeepSeek-Prover-V2-7B over an ontology-typed dependency graph of MATHLIB, with gains concentrated on problems the bare model rarely solves. KG-Prover (Li et al., 2025) augments provers with a text-mined knowledge graph, gaining most on smaller models; our graph is anchored to verified MATHLIB declarations under a typed ontology, and our controlled ablation isolates KG, retrieval, and both, showing when each helps or hurts and that the modes are complementary.

Mathematical knowledge graphs. NaturalProofs (Welleck et al., 2021) links ProofWiki theorems through their reference structure, without formal verification. TheoremGraph (Kurgan et al., 2026) unifies arXiv statements with Lean 4 declarations by embedding similarity, and Li et al. (2026) and Dasgupta (2026) map MATHLIB's dependency network; all three graphs encode dependency or citation links rather than semantic relations. AutoMathKG (Bian et al., 2025) builds an updatable KG from ProofWiki, textbooks, arXiv, and TheoremQA with LLM augmentation, aimed at informal reasoning. We apply LLM-based relation extraction (Ye et al., 2022; Wadhwa et al., 2023; Xu et al., 2024) over MATHLIB declarations. MATHKG differs in two ways: its nodes are verified MATHLIB declarations, and its edges are LLM-inferred semantic relations under an eight-type ontology.

3 MATHKG: A SEMANTIC KNOWLEDGE GRAPH

MATHKG organizes a curated core of MATHLIB, 364 nodes in seven domains, into a semantic graph.

Node selection. The seven domains cover the subject areas of the three evaluation benchmarks (§5.1). Within each domain we asked Claude Opus 4.7, Gemini 3.1 Pro, and GPT-5.4 for the fundamental theorems and definitions of a standard undergraduate mathematics curriculum, cross-checked the lists against one another, and kept only entries whose formalization exists in MATHLIB. The lists were also reviewed by a PhD mathematician, who judged each a reasonable set of fundamentals. This yields 229 theorems and 135 definitions, 49 to 55 nodes per domain (theorems/definitions): number theory 39/10, combinatorics 33/20, algebra 34/18, abstract algebra 30/25, linear algebra 28/24, analysis 40/14, and Euclidean geometry 25/24. The core is deliberately small: a graph over all of MATHLIB would have over 10^{11} ordered pairs, beyond exhaustive relation extraction, and would bury the fundamentals under specialized lemmas. A fundamentals-only graph keeps exhaustive pairwise extraction feasible (132,132 calls) and leaves specialized lemmas to MATHLIB retrieval (§4).

3.1 GRAPH SCHEMA

MATHKG is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, built with NetworkX,[2] where:

[2]NetworkX is a standard Python library for in-memory graphs (details in Appendices D.3 and F).

Nodes. Each node is a mathematical object with an identifier, a human-readable name, a domain, a natural-language statement, a type (theorem or definition), keywords, its MATHLIB declaration name and import, and a verification status. Every declaration name is verified against the 404,440-entry MATHLIB declaration database, so each node names a theorem or definition formalized in MATHLIB.[3]

Edges. Each directed edge from node A to node B, encoded as (A, B, r, c, ρ), carries a relation type $r \in \mathcal{R}$, a confidence $c \in [0, 1]$, and a natural-language justification ρ. The eight edge types (Table 1) carry node-type restrictions (Appendix D.2) and range from logical relations (generalizes, specializes, equivalent-to) to discovery-oriented ones (analogous-to, cross-domain-bridge).

Table 1: The eight semantic edge types (directed edge $A \to B$) in MATHKG.

Edge Type	**Semantics**	A/B **Restriction**	**Example**
prerequisite	A is needed to state B	A: definition	Prime Number $\to$ Fermat's LT
generalizes	A is more general than B	both theorems	Binom. Thm $\to$ Sum of Binom. Coeff.
specializes	A is a special case of B	both theorems	Fermat's LT $\to$ Lagrange's Thm
equivalent-to	$A \leftrightarrow B$, same level	both theorems	Open Mapping $\leftrightarrow$ Closed Graph
applied-in	A's result used in B's proof	both theorems	FTA $\to$ Lifting the Exp. Lemma
analogous-to	Same structure, other context	both theorems	Hensel $\leftrightarrow$ Impl. Fn. Thm
cross-domain-bridge	Formal link across domains	different domains	Vector Space $\to$ Field
instance-of	Concrete A exemplifies B	both definitions	ZMod $n \to$ Ring

3.2 RELATION EXTRACTION PIPELINE

Given an ordered pair of nodes (A, B), we prompt Claude Sonnet 4.6 to infer the directed semantic relationship $(A \to B)$. The prompt provides both nodes' natural-language names and statements, keywords and domain labels, and MATHLIB declaration names and imports, together with definitions, usage restrictions, and an illustrative pair for each of the eight edge types, plus few-shot examples (the complete prompt is in Appendix D). Hard type restrictions (e.g., generalizes requires both endpoints to be theorems) are stated in the prompt and enforced again in post-processing, and a "none" fallback lets the model reject weak connections.

The model returns a JSON object $\{\text{relationship} : r,\ \text{confidence} : c,\ \text{reasoning} : \rho\}$; edges with $c < \tau$ (default $\tau = 0.75$) are discarded. We check both directions $(A \to B)$ and $(B \to A)$ for all pairs, as relationships are generally asymmetric. A post-processing pass then (i) removes edges that violate the type restrictions; (ii) removes contradictory bidirectional generalizes/specializes cycles; and (iii) fills in inverses for the symmetric types and the generalizes/specializes pair.

4 MATHAGENT: FOUR-AGENT DISCOVERY SYSTEM

MATHAGENT implements a four-agent architecture (Figure 1) where MATHKG serves as the shared persistent memory substrate. The four agents, *Explorer*, *Conjecturer*, *Prover*, and *Integrator*, are LLM-powered modules with specialized roles that share access to the graph.

The system supports two operation modes. In *open-ended exploration*, the Explorer identifies investigation targets (structural gaps, cross-domain analogies) in MATHKG, the Conjecturer proposes a candidate theorem, the Prover attempts a Lean 4 proof, and the Integrator evaluates novelty and writes verified results back into the graph. This mode is implemented but not evaluated in this paper.

Targeted problem-solving, which is the focus of our work, is shown in Algorithm 1: the Explorer searches MATHKG and MATHLIB for context relevant to a given problem and the Prover attempts a proof using that context. In both modes a deterministic *Orchestrator* sequences the agents; in targeted mode it also feeds failed approaches back as negative context.

The four augmentation conditions evaluated in §5 correspond to lines of Algorithm 1: `prover_only` skips lines 1–2 (no external context); `with_kg` executes only line 1 (MATHKG semantic context); `with_mathlib` executes only line 2 (MATHLIB library retrieval context); `full_system` executes both (MATHKG + MATHLIB context combined).

[3] Appendix B shows example nodes; Appendix A introduces Lean 4, MATHLIB, and the declaration database.

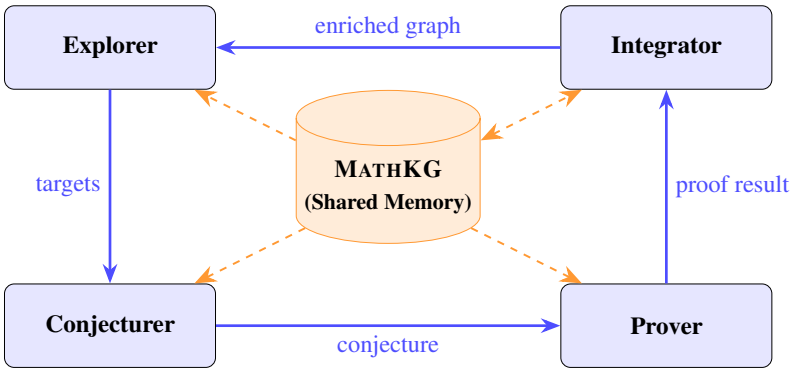

Figure 1: The MATHAGENT architecture: four LLM agents with MATHKG as persistent shared memory (the Orchestrator, which coordinates the cycle, is omitted).

Algorithm 1 MATHAGENT Targeted Problem-Solving Mode

Require: MATHKG $\mathcal{G}$, problem P, max attempts K (default 32)
context$_{\text{KG}} \leftarrow$ Explorer.search_kg$(P, \mathcal{G})$ ▷ Layer 1: KG semantic search
context$_{\text{Mathlib}} \leftarrow$ Explorer.search_mathlib(P) ▷ Layer 2: Mathlib declaration search
proof $\leftarrow$ Prover.generate$(P, \text{context}_{\text{KG}}, \text{context}_{\text{Mathlib}})$ ▷ Attempt 1: context injected once
for $k = 2, \ldots, K$ **do** ▷ Error-guided refinement
  if proof.verified **then return** proof
  proof $\leftarrow$ Prover.refine(proof, proof.error_log) ▷ Previous code + its Lean errors
end for
return proof if proof.verified else failed

4.1 EXPLORER AGENT

In targeted problem-solving, the Explorer performs a *two-layer search* to gather the Prover's context.

Layer 1: MATHKG semantic search. The Explorer (1) pre-filters MATHKG to nodes in the problem's domain plus their cross-domain-bridge neighbors, (2) selects s=5 seeds by LLM relevance scoring, (3) walks MATHKG edges one hop from each seed, and (4) attaches to each retrieved node the edges that connect it to the seeds. The output is a context block of the seeds and their one-hop neighbors, each with its statement and a relevance rationale, and each neighbor with its connecting edges (type, confidence, and reasoning); details are in Appendix C.

Layer 2: MATHLIB declaration search. The Explorer (1) extracts search terms (Lean 4 identifiers regex-extracted from the formal statement plus LLM-suggested MATHLIB namespace prefixes), (2) prefix-matches them against the declaration names of the MATHLIB documentation index (§3), (3) scores and ranks the matches by source priority, match precision, name specificity, declaration kind, and module proximity, and (4) enriches the top-ranked declarations with their Lean 4 type signatures and docstrings, so the Prover receives exact names and argument types (details in Appendix C).

Full system (Layers 1+2). When both layers run, their context blocks are concatenated, knowledge-graph context first.

4.2 PROVER AGENT

The Prover attempts a complete Lean 4 proof with error-guided refinement:

1. **Initial generation**: Prompt the prover LLM (the evaluated model, §5.1) with the natural-language and Lean 4 statements plus the Explorer's context (targeted problem-solving) or the Conjecturer's proof sketch (open-ended exploration); prompts are in Appendix E.2.
2. **Lean 4 verification**: Submit the generated code to the Lean 4 type-checker.

3. **Error-guided refinement**: On failure, feed the previous code and its error messages (up to ten) back to the model to fix. Repeat steps 2–3 until success or K attempts.[4]

5 EXPERIMENTS

We run a controlled ablation to determine when structured knowledge augmentation helps an LLM theorem prover, for which models, and by how much: five models under four augmentation modes, isolating the contributions of the MATHKG semantic graph, MATHLIB library retrieval, and their combination. §5.2 asks whether augmentation or Lean specialization matters more; §5.3 whether augmentation's effect depends on the model that receives it; and §5.4 whether different modes solve different problems, so that choosing the augmentation mode per problem would beat any fixed mode.

5.1 SETUP

MATHKG. MATHKG (§3) contributes 364 nodes and 9,434 edges: 8,026 LLM-inferred plus 1,408 auto-inverse. Of the 8,026 LLM-inferred edges, cross-domain-bridge (3,481) and prerequisite (2,293) account for 72%, applied-in and analogous-to for most of the rest (1,503 and 578), and the logical types are infrequent (instance-of 80, generalizes 48, specializes 30, equivalent-to 13); connectivity statistics are in Appendix D.3.

Benchmarks. We evaluate on three standard formal-mathematics benchmarks: **miniF2F** (Zheng et al., 2022) (488 olympiad-level problems: the full validation and test splits), **MathOlympiad-Bench** (Lin et al., 2026) (360 competition problems), and **PutnamBench** (Tsoukalas et al., 2024) (326 undergraduate/research-level problems, those stated without an answer slot). The cross-model comparison uses miniF2F; §5.4 adds the harder benchmarks for the proprietary model.

Augmentation modes. Four conditions isolate each component (Algorithm 1): `prover_only` (no external context), `with_kg` (MATHKG semantic search, Layer 1), `with_mathlib` (MATHLIB library retrieval, Layer 2), and `full_system` (both). Comparing each augmented mode against the baseline isolates each component's contribution; the combined mode tests for synergy.

Models. We compare five models along two axes, general-purpose vs. Lean-specialized and small (8B) vs. large (32B), forming a controlled 2×2: Qwen3-8B and Qwen3-32B (general) (Yang et al., 2025) versus Goedel-Prover-V2-8B and -32B (Lean-specialized) (Lin et al., 2026). Goedel-Prover-V2-8B is Qwen3-8B fine-tuned on Lean proofs through expert iteration and self-correction training, and -32B is fine-tuned from Qwen3-32B, so each pair is matched in scale and architecture. We add Claude Sonnet 4.6 (Anthropic, 2026) as a proprietary, larger reference.

Protocol. We report pass@32 with sequential, error-guided refinement (Algorithm 1) rather than 32 independent samples. Sonnet 4.6 runs on AWS Bedrock and the four open-weight models on local vLLM; only the model and serving backend change, with benchmark sets and augmentation modes held fixed. Every LLM call in a run, including the Explorer's relevance scoring and search-term suggestion, uses that run's model, so each result measures the end-to-end system a practitioner would deploy with that model. All five models use the same generic prompt (Appendix E.2); only the injected context varies across modes. We use temperature 0.3 for initial generation and 0.2 for refinement, and set each model's maximum output tokens above the length of its longest verified proof (per-model caps in Appendix F). Lean 4 compilation has a 1,200 s timeout per attempt. The Explorer uses the same default settings for every model (§4.1, Appendix C).

5.2 SPECIALIZATION DOMINATES KNOWLEDGE AUGMENTATION

Table 2 and Figure 2 report pass@32 on miniF2F for all five models under the four augmentation modes. The dominant factor is neither scale nor augmentation but *specialization*: Goedel-8B solves 205/488 (42.0%) versus 62/488 (12.7%) for Qwen3-32B without external context, a 29-point gap, even though Goedel-8B is four times smaller. Within each matched pair, specialization adds 35.2

[4] We use $K{=}32$ for benchmark evaluation (pass@32) in §5.

points at 8B (42.0% vs. 6.8%) and 35.9 points at 32B (48.6% vs. 12.7%) in `prover_only`. By contrast, no single augmentation mode improves solve rate by more than 3 percentage points.[5]

Table 2: Cross-model pass@32 solve rates on miniF2F (488 problems). Best mode in **bold**.

Type	Model	prover_only	with_kg	with_mathlib	full_system
General	Qwen3-8B	33 (6.8%)	35 (7.2%)	31 (6.4%)	**36 (7.4%)**
	Qwen3-32B	**62 (12.7%)**	48 (9.8%)	**62 (12.7%)**	46 (9.4%)
Specialized	Goedel-8B	205 (42.0%)	**219 (44.9%)**	204 (41.8%)	205 (42.0%)
	Goedel-32B	**237 (48.6%)**	234 (48.0%)	235 (48.2%)	208 (42.6%)
Proprietary	Claude Sonnet 4.6	365 (74.8%)	358 (73.4%)	359 (73.6%)	**366 (75.0%)**

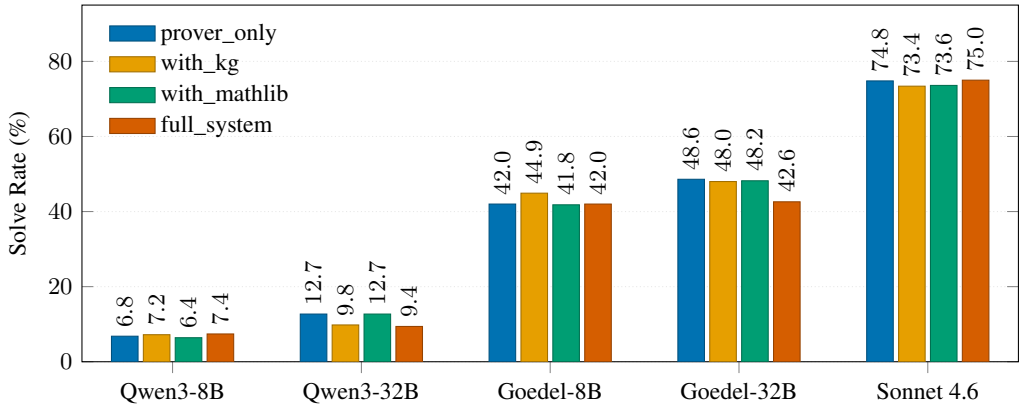

Figure 2: Cross-model pass@32 on miniF2F (Table 2).

This dominance holds across every mode: the specialization gap (the Lean-specialized model minus its general base at matched scale) never falls below ∼33 percentage points, and `with_kg` in fact *widens* it (to 37.7 points at 8B and 38.1 at 32B). KG context gains the specialized model more relative to its general base at both scales: +14 vs +2 at 8B, and at 32B −3 vs −14 as reported, or +1 vs −14 once overflow is excluded.[6] Even against a four-times-larger general model, a specialized 8B model (Goedel-8B) leads Qwen3-32B by 29–35 percentage points in every mode. No augmentation mode we tested lets a general model close the gap that Lean fine-tuning opens.

5.3 Augmentation is capability-conditioned

Whether augmentation helps *on aggregate* depends on the model. Table 3 decomposes each component's effect as solve-count deltas. KG augmentation helps the specialized small model most (Goedel-8B, +14), helps the weak general model marginally (Qwen3-8B, +2), is roughly neutral on the large specialized model (Goedel-32B, −3)[6], and *hurts* the large general model (Qwen3-32B, −14) and the proprietary model (Sonnet, −7). MATHLIB retrieval never helps on aggregate (−6 to 0). Combining both shows no consistent synergy: `full_system` improves on the best single augmentation for only two models (Qwen3-8B +1, Sonnet +7), by 1–3 problems over `prover_only`, and elsewhere falls below the best single mode (−14 to −27; p-values in Appendix J).[6]

KG augmentation's effect is governed jointly by scale and specialization: scale decides the sign (KG context helps at 8B and is neutral or harmful at 32B), while specialization decides how much better than the general baseline a model does (+14 vs +2 at 8B; −3 vs −14 at 32B). KG augmentation is thus *capability-conditioned*: its value depends on the model that receives it, so it does not help uniformly across models. This echoes retrieval-augmented question answering, where retrieval helps

[5] Appendix G shows two verified proofs of one miniF2F problem, by Sonnet 4.6 and by Goedel-8B.

[6] For Goedel-32B, 10 problems in `with_kg` and 28 in `full_system` had KG-augmented prompts that exceeded the served 40,960-token context window; we count them as failures. Excluding them, the rates would be 49.0% and 45.2% (Appendix I). Appendix J gives McNemar p-values for the deltas of Table 3.

most when the model's parametric knowledge is weakest and can hurt when the model already knows the answer (Mallen et al., 2023); we observe the same conditionality for structured mathematical knowledge in formal proving. TheoremBench (Pham et al., 2026) independently finds context gains model-dependent, and Khalov et al. (2026) and Ataeva et al. (2026) find that graph-retrieved hints lift DeepSeek-Prover-V2-7B only on problems it rarely solves unaided.

MATHLIB retrieval, by contrast, is *uniformly unhelpful*, so its effect is not conditioned on the model; this holds for our lexical retriever, which prefix-matches declaration names (§4.1), and does not speak to learned premise selection (Yang et al., 2023; Mikuła et al., 2024). The two sources also differ in kind: knowledge-graph context is natural-language statements and typed relations, while MATHLIB retrieval supplies formal signatures. No fixed mode helps every model, nor is this a miniF2F artifact: for Sonnet it holds across the three benchmarks, where every fixed mode lowers the aggregate by 0.5–2.9% (Appendix H).

Table 3: Solve-count deltas on miniF2F (positive: augmented mode solves more); *: $p < .05$.

Question	Qwen3-8B	Qwen3-32B	Goedel-8B	Goedel-32B	Sonnet
KG helps? (kg − PO)	+2	−14*	+14*	−3	−7
Mathlib helps? (mathlib − PO)	−2	0	−1	−2	−6
KG beyond Mathlib? (full − mathlib)	+5	−16*	+1	−27*	+7
Synergy? (full − max(kg,mathlib))	+1	−16*	−14*	−27*	+7
Best fixed mode	full	PO/mathlib	KG	PO	full

5.4 COMPLEMENTARITY: AUGMENTATION MODES SOLVE DIFFERENT PROBLEMS

Aggregate solve rates understate augmentation's value, because the modes solve *different* problems. Table 4 reports, per model on miniF2F, the number of problems solved (solve rate in parentheses) under `prover_only` (no context), the best single mode (the best fixed augmentation mode for that model), and the oracle (a selector that picks the best mode per problem, equivalent in solve count to the union of the four modes). For every model the oracle exceeds `prover_only`: +23 for Sonnet (365→388, 79.5%), +36 for Goedel-8B (205→241, 49.4%), +29 for Goedel-32B (237→266, 54.5%), and +19/+18 for the two general models. The oracle uses four runs' worth of attempts; at a matched budget, with 8 attempts per mode (32 in total, the same as `prover_only` at pass@32), the union still exceeds `prover_only` for every model: +5 for Sonnet, +27 for Goedel-8B, +22 for Goedel-32B, and +18/+17 for the two general models (Appendix K).

Table 4: Complementarity on miniF2F (488 problems). Best single mode: the best fixed augmentation mode per model. Oracle: union of problems solved by any of the four modes. Last column: oracle minus `prover_only` (no context), in problems and solve-rate points.

Model	prover_only	best single mode	oracle	oracle − prover_only
Qwen3-8B	33 (6.8%)	36 (7.4%)	52 (10.7%)	+19 (+3.9%)
Qwen3-32B	62 (12.7%)	62 (12.7%)	80 (16.4%)	+18 (+3.7%)
Goedel-8B	205 (42.0%)	219 (44.9%)	241 (49.4%)	+36 (+7.4%)
Goedel-32B	237 (48.6%)	237 (48.6%)	266 (54.5%)	+29 (+5.9%)
Claude Sonnet 4.6	365 (74.8%)	366 (75.0%)	388 (79.5%)	+23 (+4.7%)

Figure 3 plots these gains as *relative* increases over `prover_only`: +58% for Qwen3-8B, +29% for Qwen3-32B, +18% for Goedel-8B, +12% for Goedel-32B, and +6% for Sonnet. The oracle beats the no-context `prover_only` for every model, even Sonnet, where no fixed mode improves the aggregate by more than one problem; the gain shrinks as capability grows but never vanishes. In absolute terms (Table 4), per-problem selection adds 4–7 points of solve rate.

This complementarity is not specific to miniF2F, and it *strengthens* on harder problems. For Sonnet across all three benchmarks, each augmented mode recovers a different set of problems (Figure 4): the oracle exceeds the `prover_only` baseline by +6% on miniF2F, +28% on MathOlympiadBench, and +50% on PutnamBench, so an adaptive augmentation selector gains most on the two harder

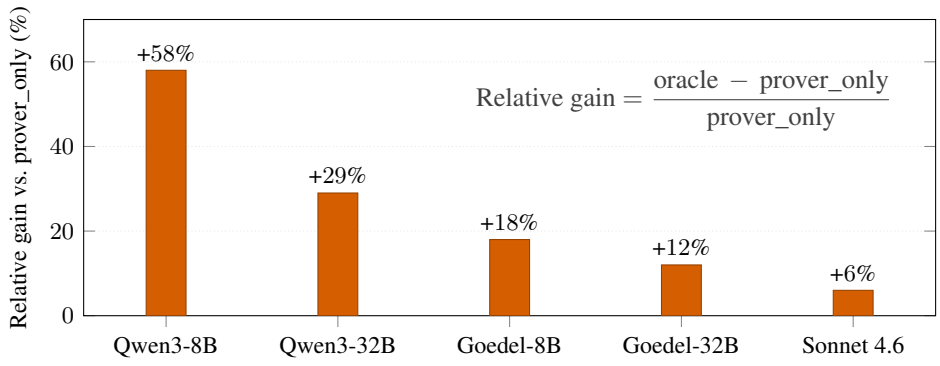

Figure 3: Relative gain in the number of problems solved by the oracle (union of the four modes) over prover_only (no augmentation) on miniF2F; the gain shrinks as model capability grows.

benchmarks (full four-mode results in Appendix H). Across the three benchmarks, augmentation recovers 41 problems that `prover_only` misses and loses 10, roughly a 4:1 ratio.

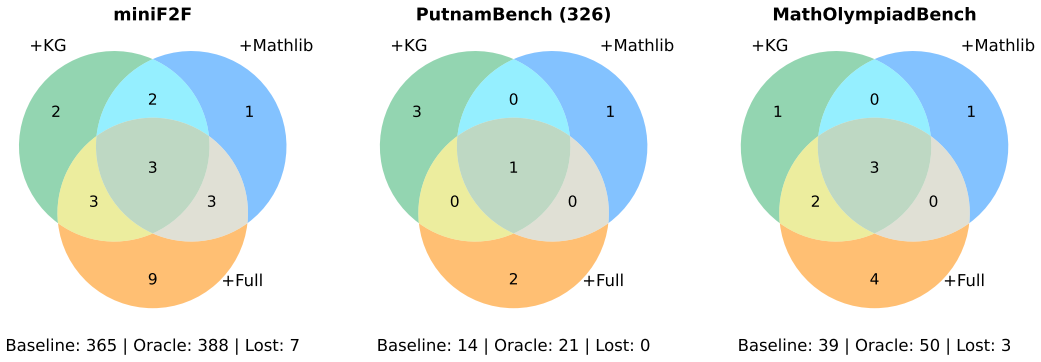

Figure 4: Complementarity for Sonnet 4.6: problems recovered beyond `prover_only` by each augmentation mode combination. Oracle: solved by any mode; Lost: solved only by `prover_only`.

6 Conclusion

We set out to measure whether, and when, structured mathematical knowledge helps LLMs prove theorems in Lean 4. In a controlled ablation of four augmentation modes (no context, knowledge-graph context, lexical Mathlib retrieval, and both) across matched general and Lean-specialized provers at 8B and 32B, plus a proprietary larger reference, three lessons emerge. The first concerns where a prover's knowledge lives. We find that what a model knows in its weights matters far more than what we supply at inference time: Lean fine-tuning is worth 33 to 38 percentage points of solve rate in every mode, enough that a specialized 8B model outperforms a general model four times its size, while no augmentation mode improves solve rate by more than three percentage points. Structured knowledge is no substitute for specialization.

Whether augmentation helps at all depends on the model: knowledge-graph context lifts small provers but hurts larger ones, with the specialized model gaining more relative to its general base at every scale. Yet this aggregate picture is incomplete. Beneath it, the augmentation modes are genuinely complementary: they solve different problems, so a selector that picks the best mode per problem would solve 6 to 58% more than the unaugmented prover, a margin that widens on harder problems (50% more on PutnamBench). The value of augmentation is real but latent, unlocked by selection rather than by uniform application. This complementarity points toward adaptive proving that chooses augmentation by model capability and problem. We release all code, graph data, and evaluation infrastructure for the AI-for-math community.

AI USE STATEMENT

In this work, we used generative AI tools (Claude via Claude Code) to assist in implementing the experiment pipeline and in setting up the vLLM serving stack and GPU instances on which the evaluated models ran. We have not used generative AI tools for research ideation, for formulating hypotheses or theoretical frameworks, or for formulating or proving mathematical claims; the Lean 4 proofs reported here are outputs of the evaluated prover systems, verified by the Lean 4 kernel. Generating synthetic datasets, translation, and qualitative data analysis are not applicable to this work; the use of LLMs inside the method itself (node curation and relation extraction for MATHKG) is described in §3. Additionally, we used generative AI tools to polish the writing. We have reviewed all AI-assisted work: two authors checked all AI-assisted code line by line and tested its correctness, and all text was reviewed and edited by the authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Section 5 and Appendix F specify the models, decoding parameters, per-model token caps, Lean 4 and MATHLIB toolchain versions, hardware, and the pass@32 protocol; Appendices D.1 and E reproduce the relation-extraction, Explorer, and Prover prompts verbatim; Appendix C describes the retrieval procedure. We release the complete implementation, the MATHKG graph file (364 nodes, 9,434 edges), the benchmark evaluation scripts, and the per-problem results of all 28 evaluation runs. Runs of the proprietary model require AWS Bedrock access; the open-weight runs reproduce with vLLM on a single multi-GPU machine.

REFERENCES

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, et al. Aristotle: IMO-level automated theorem proving. *arXiv preprint arXiv:2510.01346*, 2025.

Noga Alon, Thomas F. Bloom, W. Timothy Gowers, Daniel Litt, Will Sawin, Arul Shankar, Jacob Tsimerman, Victor Wang, and Melanie Matchett Wood. Remarks on the disproof of the unit distance conjecture. *arXiv preprint arXiv:2605.20695*, 2026.

Anthropic. System card: Claude Sonnet 4.6. https://www.anthropic.com/claude-sonnet-4-6-system-card, 2026.

Olga M. Ataeva, Andrey P. Khalov, and Natalia P. Tuchkova. Multi-level knowledge graph for formal mathematics. *Lobachevskii Journal of Mathematics*, 2026. Accepted April 22, 2026. https://openreview.net/forum?id=eZqzdol80P.

Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in Lean 4. https://lean-lang.org/theorem_proving_in_lean4/, 2026. Accessed September 18, 2026.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. In *International Conference on Learning Representations*, 2024.

Luke Bailey, Kaiyue Wen, Kefan Dong, Tatsunori Hashimoto, and Tengyu Ma. Scaling self-play with self-guidance. *arXiv preprint arXiv:2604.20209*, 2026.

Slim Barkallah, Luke Bailey, Kaiyue Wen, Mohammed Abouzaid, and Tengyu Ma. Pseudo-Formalization for automatic proof verification. *arXiv preprint arXiv:2605.20531*, 2026.

Rong Bian, Yu Geng, Zijian Yang, and Bing Cheng. AutoMathKG: The automated mathematical knowledge graph based on LLM and vector database. *Computational Intelligence*, 41(4):e70096, 2025.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, et al. Seed-Prover 1.5: Mastering undergraduate-level theorem proving via learning from experience. *arXiv preprint arXiv:2512.17260*, 2025a.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-Prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025b.

Haskell B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

Ashani Dasgupta. Geometric structure in the Lean 4 Mathlib knowledge graph. In *Sci-K 2026: 6th International Workshop on Scientific Knowledge Representation, Discovery, and Assessment (co-located with ISWC 2026)*, 2026. https://faculty.tcu.edu/gfriedman/gtw_archive/Problems/Dasgupta.pdf.

Kefan Dong and Tengyu Ma. STP: Self-play LLM theorem provers with iterative conjecturing and proving. *arXiv preprint arXiv:2502.00212*, 2025.

Guoxiong Gao, Zeming Sun, Jiedong Jiang, Yutong Wang, Jingda Xu, Peihao Wu, Bryan Dai, and Bin Dong. LeanSearch v2: Global premise retrieval for Lean 4 theorem proving. *arXiv preprint arXiv:2605.13137*, 2026.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490. Academic Press, 1980.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026.

Geoffrey Irving, Christian Szegedy, Alexander A Alemi, et al. Deepmath - deep sequence models for premise selection. *Advances in Neural Information Processing Systems*, 29, 2016.

Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022.

Aditi Kabra, Albert Q. Jiang, Andrew Zhao, Dhia Garbaya, Indraneel Mukherjee, Jason Rute, Mert Unsal, Roman Soletskyi, and Simon Sorg. Leanstral. alphaXiv preprint, https://www.alphaxiv.org/abs/2608.leanstral, 2026. Technical report for Leanstral 1.5 (Mistral AI).

Phillip Kerger. MATEK: An open orchestration framework for agentic mathematical research with persistent knowledge graphs. 2026. SSRN preprint, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=7163458.

Andrey P. Khalov, Olga M. Ataeva, and Natalia P. Tuchkova. Explainable AI for mathematics: Proofs as code with knowledge graph and domain ontology support. In *Proceedings of MathAI 2026*, 2026. https://openreview.net/pdf/775c1617ab35a9644ff0eb8310a9579427a3c88e.pdf.

Po-Nien Kung, Linfeng Song, Dawsen Hwang, Jinsung Yoon, Chun-Liang Li, Simone Severini, Mirek Olšák, Edward Lockhart, Quoc V Le, Burak Gokturk, et al. LEAP: Supercharging LLMs for formal mathematics with agentic frameworks. *arXiv preprint arXiv:2606.03303*, 2026.

Simon Kurgan, Evan Wang, Eric Leonen, Sophie Szeto, Luke Alexander, Artemii Remizov, Jarod Alper, Giovanni Inchiostro, and Vasily Ilin. TheoremGraph: Bridging formal and informal mathematics. *arXiv preprint arXiv:2606.25363*, 2026.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.

Joshua Ong Jun Leang, Zheng Zhao, Mihaela Cătălina Stoian, Qiyuan Xu, Haonan Li, Wenda Li, Shay B. Cohen, and Eleonora Giunchiglia. Pythagoras-Prover: Advancing efficient formal proving via augmented Lean formalisation. *arXiv preprint arXiv:2606.12594*, 2026.

Aitor Lewkowycz, Anders Andreassen, David Dohan, et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35, 2022.

Vincent Li, Tim Knappe, Yule Fu, Kevin Han, and Kevin Zhu. Scaling natural-language graph-based test time compute for automated theorem proving. *arXiv preprint arXiv:2503.11657*, 2025.

Xinze Li, Nanyun Peng, Simone Severini, and Patrick Shafto. The network structure of Mathlib. *arXiv preprint arXiv:2604.24797*, 2026.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, 2025.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, et al. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. In *International Conference on Learning Representations*, volume 2026, pp. 11793–11818, 2026.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 9802–9822, 2023.

Maciej Mikuła, Szymon Tworkowski, Szymon Antoniak, Bartosz Piotrowski, Qiaochu Jiang, Jin Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. Magnushammer: A transformer-based approach to premise selection. In *International Conference on Learning Representations*, volume 2024, pp. 39326–39350, 2024.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

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

Ken Ono. Integer values of $\tan(\arctan 1 + \arctan 2 + \cdots + \arctan n)$ are rare. *arXiv preprint arXiv:2607.05739*, 2026.

OpenAI. Ten advances in mathematics and theoretical computer science. OpenAI Technical Report, https://cdn.openai.com/pdf/ten-proofs-oai.pdf; blog: https://openai.com/index/ten-advances-in-mathematics/, 2026.

Azim Ospanov, Zijin Feng, Jiacheng Sun, Haoli Bai, Xin Shen, and Farzan Farnia. HERMES: Towards efficient and verifiable mathematical reasoning in LLMs. In *International Conference on Machine Learning*, 2026.

QuocViet Pham, Elvir Karimov, Andrey Galichin, and Ivan Oseledets. TheoremBench: Evaluating LLMs on theorem proving in formal mathematics. *arXiv preprint arXiv:2606.09450*, 2026.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Ahmad Rammal, Niket Patel, Fabian Gloeckle, Amaury Hayat, Julia Kempe, Remi Munos, Charles Arnal, and Vivien Cabannes. Formalizing mathematics at scale. *arXiv preprint arXiv:2605.29955*, 2026.

ZZ Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

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.

Ziju Shen, Naohao Huang, Fanyi Yang, Yutong Wang, Guoxiong Gao, Tianyi Xu, Jiedong Jiang, Wanyi He, Pu Yang, Mengzhou Sun, Haocheng Ju, Peihao Wu, Bryan Dai, and Bin Dong. REAL-Prover: Retrieval augmented Lean prover for mathematical reasoning. *arXiv preprint arXiv:2505.20613*, 2025.

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Lean Copilot: Large language models as copilots for theorem proving in Lean. In *Proceedings of the International Conference on Neuro-symbolic Systems*, volume 288 of *Proceedings of Machine Learning Research*, pp. 144–169. PMLR, 2025.

Terence Tao. Mathematics in the age of AI. *arXiv preprint arXiv:2608.16753*, 2026.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving. In *Proceedings of the Conference on Language Modeling (COLM)*, 2024.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '20, pp. 367–381. ACM, 2020. doi: 10.1145/3372885.3373824.

The mathlib Community. Mathlib statistics. https://leanprover-community.github.io/mathlib_stats.html, 2026. Accessed September 13, 2026.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. PutnamBench: Evaluating neural theorem-provers on the Putnam Mathematical Competition. *Advances in Neural Information Processing Systems*, 37: 11545–11569, 2024.

George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, et al. Advancing mathematics research with AI-driven formal proof search. *arXiv preprint arXiv:2605.22763*, 2026.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. In *International Conference on Learning Representations*, volume 2026, pp. 38998–39031, 2026.

Somin Wadhwa, Silvio Amir, and Byron C Wallace. Revisiting relation extraction in the era of large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, 2023.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-Prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. NaturalProofs: Mathematical theorem proving in natural language. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *International Conference on Learning Representations*, volume 2024, pp. 2968–2978, 2024.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in neural information processing systems*, 35:32353–32368, 2022.

Ran Xin, Zeyu Zheng, Yanchen Nie, Kun Yuan, and Xia Xiao. Scaling up multi-turn off-policy RL and multi-agent tree search for LLM step-provers. *arXiv preprint arXiv:2509.06493*, 2025.

Derong Xu, Wei Chen, Wenjun Peng, Chao Zhang, Tong Xu, Xiangyu Zhao, Xian Wu, Yefeng Zheng, Yang Wang, and Enhong Chen. Large language models for generative information extraction: A survey. *Frontiers of Computer Science*, 18(6):186357, 2024.

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

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Hongbin Ye, Ningyu Zhang, Hui Chen, and Huajun Chen. Generative knowledge graph construction: A review. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022.

Meiru Zhang, Philipp Borchert, Milan Gritta, and Gerasimos Lampouras. Drift: decompose, retrieve, illustrate, then formalize theorems. In *International Conference on Learning Representations*, volume 2026, pp. 6828–6857, 2026.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: A cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022.

A Background: Mathlib and Lean 4

We briefly review the Lean 4 and Mathlib concepts used in this paper.

Lean 4. Lean 4 (Moura & Ullrich, 2021) is an interactive theorem prover and programming language; Avigad et al. (2026) give a tutorial introduction. Mathematical statements are expressed as *types*, and proofs are *terms* of those types (the Curry–Howard correspondence; Curry, 1934; Howard, 1980). For example, a proof of $\forall n : \mathbb{N},\ 0 \leq n$ is a function that takes any natural number and returns evidence of the inequality.

Mathlib. Mathlib (The mathlib Community, 2020) is the community-maintained mathematical library for Lean 4, containing 285,000+ theorems and 135,000+ definitions as of September 2026 (The mathlib Community, 2026), spanning undergraduate through research-level mathematics. Each declaration is machine-verified: the Lean kernel checks every proof step.

Declaration kinds. The Mathlib library contains several kinds of declarations:

- `theorem`: A proved mathematical statement (e.g., Fermat's Little Theorem)
- `def`: A mathematical definition (e.g., `Nat.Prime`, `Nat.gcd`)
- `class`: A typeclass (e.g., `Group`, `Ring`, `Field`)
- `instance`: A typeclass instance (e.g., `Int` is a `Ring`)
- `structure`: A data structure definition
- `inductive`: An inductive type definition

Type signatures. Every declaration has a *type signature* that formally specifies what it states or computes. For example, Fermat's Little Theorem has the type:

Type signature: `ZMod.pow_card_sub_one_eq_one` (Fermat's Little Theorem)

```
theorem ZMod.pow_card_sub_one_eq_one
  {p : ℕ} [Fact (Nat.Prime p)] {a : ZMod p} (ha : a ≠ 0) :
  a ^ (p - 1) = 1
```

Reading this signature: `p` is a natural number assumed to be prime (via the `[Fact (Nat.Prime p)]` typeclass constraint); `a` is an element of $\mathbb{Z}/p\mathbb{Z}$; `ha` is a proof that $a \neq 0$; and the conclusion is $a^{p-1} = 1$. Lean's `exact?` tactic searches the imported environment by *type unification*: given a proof goal (a type), it finds theorems whose conclusion type matches after substituting variables.

Declaration metadata. The Mathlib documentation index (404,440 entries, covering Mathlib and its dependencies such as Lean core and Batteries) provides, for each declaration: (1) the declaration name (e.g., `ZMod.pow_card_sub_one_eq_one`), (2) its kind (`theorem`/`def`/etc.), (3) its full type signature, and (4) its docstring; proofs themselves are in the source files. Our Layer 2 search (§4.1) prefix-matches the extracted search terms against these declaration names and attaches the matching signatures and docstrings (Appendix C).

B MathKG Example Nodes

We show two MathKG node entries, Sperner's Theorem and the Cayley–Hamilton Theorem, each followed by the Lean 4 type signature that the Mathlib declaration database returns for its `lean4_mathlib_name`; the node file itself stores the nine fields shown.

Node record: Sperner's Theorem (Combinatorics)

```
id: "thm_comb_004"
name: "Sperner's Theorem"
domain: "Combinatorics"
statement: "Max antichain in P([n]) has size
  C(n, ⌊n/2⌋)."
type: "theorem"
keywords: ["Sperner", "antichain", "extremal"]
lean4_mathlib_name: "IsAntichain.sperner"
lean4_mathlib_import: "Mathlib.Combinatorics.SetFamily.LYM"
lean4_status: "verified"
```

Retrieved Lean 4 type signature: `IsAntichain.sperner`

```
theorem IsAntichain.sperner {α : Type u_2} [Fintype α]
  {𝒜 : Finset (Finset α)}
  (h𝒜 : IsAntichain (fun (x1 x2 : Finset α) => x1 ⊆ x2) ↑𝒜) :
  𝒜.card ≤ (Fintype.card α).choose (Fintype.card α / 2)
```

Node record: Cayley–Hamilton Theorem (Linear Algebra)

```
id: "thm_la_002"
name: "Cayley-Hamilton Theorem"
domain: "Linear Algebra"
statement: "Every square matrix satisfies its own
  characteristic polynomial."
type: "theorem"
keywords: ["Cayley-Hamilton", "characteristic polynomial"]
lean4_mathlib_name: "Matrix.aeval_self_charpoly"
lean4_mathlib_import: "Mathlib.LinearAlgebra.Matrix.Charpoly.Basic"
lean4_status: "verified"
```

Retrieved Lean 4 type signature: `Matrix.aeval_self_charpoly`

```
theorem Matrix.aeval_self_charpoly {R : Type u_1} [CommRing R]
  {n : Type u_4} [DecidableEq n] [Fintype n] (M : Matrix n n R) :
  (Polynomial.aeval M) M.charpoly = 0
```

C Explorer Search Procedure

This appendix gives the full detail of the two-layer Explorer search summarized in §4.1. In all reported runs the Explorer's LLM calls are answered by the evaluated model itself, so each model's context reflects its own selection quality; a stronger selector might raise the smaller models' gains.

Layer 1: MathKG semantic search. The Explorer searches MathKG in four steps:

1. **Domain pre-filter.** Restrict candidates to nodes whose domain label contains the problem's domain, plus their cross-domain-bridge neighbors. A number theory problem keeps 172 of the 364 nodes (49 number theory nodes plus 123 bridge neighbors); an algebra problem matches three domains (Algebra, Abstract Algebra, and Linear Algebra) and keeps 324. When no node domain matches the label, seed selection runs over the whole graph. On miniF2F, labels are inferred from problem identifiers; this applies to the 175 problems whose identifiers name a competition or a proof technique rather than a topic ("competition": 120 AMC/AIME problems, "imo": 39, "other": 16 induction problems). For these problems, all 364 nodes are candidates for seed selection.
2. **Seed selection.** Send each candidate's name, statement, and keywords to the evaluated model, which scores their relevance to the problem's formal statement and returns s seeds (default s=5), each with a one-line rationale.

3. **Edge walk.** From each seed, perform breadth-first traversal of the MATHKG edges in both directions up to a configurable depth (default 1 hop). For each discovered node, the edge *reasoning text* (generated during edge inference) is preserved, explaining *why* the connection exists.
4. **Context assembly.** For every node in the result set, emit its name and statement, tagged `SEED` or `CONNECTED`; seeds carry the scorer's relevance rationale, and neighbors carry a connection note (edge type and originating seed) plus one line per connecting edge with the edge type, confidence, and the leading 100 characters of the edge reasoning, to bound the block's size. The block itself is uncapped: every one-hop neighbor of every seed is included.

The output is a structured context block where seed nodes appear first (highest relevance), followed by edge-connected neighbors sorted by the highest confidence among their connecting edges.

Layer 2: MATHLIB declaration search. The Explorer searches the declaration names of the MATHLIB documentation index (§3) through a four-step pipeline:

1. **Search term extraction.** Two complementary sources provide search terms: (a) regex extraction of Lean 4 identifiers directly from the formal statement (e.g., `Nat.Prime`, `ZMod`); (b) an LLM call asks the evaluated model to suggest 10–15 MATHLIB namespace prefixes tailored to the problem (e.g., `Function.Injective`, `Nat.strongRecOn`), each classified as *essential*, *helpful*, or *speculative*; the two sources are merged into one set of terms. The regex receives the full formal statement, and the LLM call receives the natural-language statement plus the first 500 characters of the formal statement.
2. **Prefix matching.** Each term is matched, as an exact name or a namespace prefix, against the theorem and definition names of the MATHLIB documentation index (Appendix A). For example, `Nat.Prime` matches 110 declarations.
3. **Scoring and ranking.** Each matched declaration receives a composite score from five signals: *source priority* (formal statement identifiers score highest; LLM essential $>$ helpful $>$ speculative), *match precision* (exact name match $>$ prefix match), *name depth* (fewer namespace levels $=$ more fundamental), *declaration kind* (theorem $>$ definition), and *module proximity* (declarations in modules mapped to the problem's topic label receive a $1.2\times$ boost; problems without a topic label receive no boost). Results are capped per search term (8 for essential, 5 for helpful, 2 for speculative).
4. **Type-signature enrichment.** Each matched declaration is enriched with its full Lean 4 type signature and the first 200 characters of its docstring from the pre-processed database of 404,440 declarations. For example, instead of returning only the name `ZMod.pow_card_sub_one_eq_one`, the Prover receives (docstring wrapped here):

   Enriched Layer 2 result: `ZMod.pow_card_sub_one_eq_one`

   ```
   theorem ZMod.pow_card_sub_one_eq_one
     {p : ℕ} [Fact (Nat.Prime p)] {a : ZMod p}
     (ha : a ≠ 0) : a ^ (p - 1) = 1
     -- **Fermat's Little Theorem**: for all nonzero
     -- `a : ZMod p`, we have `a ^ (p - 1) = 1`.
   ```

   This enables the Prover to (a) use exact names without hallucination, (b) understand what arguments each lemma requires, and (c) judge whether a lemma is tactically relevant to the current goal. Type strings are stored as rendered by the documentation site, so about 14% of MATHLIB theorem and definition entries carry HTML-escaped operators (`<` and `>` for <, >, and =>). The same database and code served every model and both MATHLIB modes (`with_mathlib` and `full_system`), so all runs received these strings identically and the cross-model comparison is unaffected.

Full system (Layers 1+2). When both layers run together, the Layer 1 block is emitted first under the heading `RELEVANT THEOREMS FROM KNOWLEDGE GRAPH`, followed by the Layer 2 block under `MATHLIB LEMMAS (from library search)`.

D MATHKG EDGE CONSTRUCTION

D.1 RELATION EXTRACTION PROMPT

Below is the complete, verbatim prompt sent to the edge-inference LLM (Claude Sonnet 4.6 via AWS Bedrock) for each directed pair (A, B). The pair shown is a concrete instantiation for `thm_comb_003` (Binomial Theorem) as node A and `thm_comb_027` (Sum of Binomial Coefficients) as node B; for every other call, only the two `NODE` blocks change: the ontology, few-shot examples, and output schema are identical across all 132,132 directed pairs. Its valid outputs are the eight types of Table 1 plus `none`, matching the relation extraction pipeline (§3.2).

Relation extraction prompt (Claude Sonnet 4.6, one directed pair)

```
You are a mathematical reasoning expert building a knowledge graph of
   fundamental mathematics. You must classify the relationship from
   NODE A to NODE B (directional) using a precise ontology. Be
   conservative: only identify strong, clear relationships.

NODE A (THEOREM):
Name: Binomial Theorem
Domain: Combinatorics
Statement: (x+y)^n = Σ C(n,k) x^{n-k} y^k.
Keywords: binomial, expansion
Mathlib name: Commute.add_pow
Mathlib import: Mathlib.Data.Nat.Choose.Sum

NODE B (THEOREM):
Name: Sum of Binomial Coefficients
Domain: Combinatorics
Statement: Σ_{k=0}^n C(n,k) = 2^n.
Keywords: binomial, sum
Mathlib name: Nat.sum_range_choose
Mathlib import: Mathlib.Data.Nat.Choose.Sum

==== ONTOLOGY (8 edge types) ====

1. prerequisite
   A (typically a DEFINITION) is required to state or prove B.
   Example: "Prime Number" --prerequisite--> "Fermat's Little Theorem"
   Restriction: A must be a DEFINITION. If A is a theorem used inside
   B's proof, use "applied-in" instead.

2. generalizes
   A is strictly more general; B is recoverable from A by fixing
   parameters/structure.
   Example: "Binomial Theorem" --generalizes--> "Sum of Binomial
   Coefficients = 2^n"
   Restriction: Both A and B must be THEOREMS (not definitions).

3. specializes
   Inverse of generalizes. A is a strict special case of B.
   Example: "Fermat's Little Theorem" --specializes--> "Lagrange's
   Theorem (group theory)"
   Restriction: Both A and B must be THEOREMS.

4. equivalent-to
   A ↔ B at the same level of abstraction. Different formulations of
   the same fact.
   Example: "Heine-Borel (compact iff closed and bounded in R^n)"
   --equivalent-to--> "Bolzano-Weierstrass (bounded sequence has
   convergent subsequence in R^n)"
```

```
   Restriction: Both A and B must be THEOREMS. Pick only when both
   directions genuinely hold. Do NOT use for "related" or "both about
   X".

5. applied-in
   A's RESULT is used as a step in B's proof.
   Example: "Fundamental Theorem of Arithmetic" --applied-in-->
   "Lifting the Exponent Lemma"
   Restriction: Both A and B must be THEOREMS. A is a tool; B is what
   uses the tool. Definitions do not have proofs, so they cannot
   appear as B. If A is a definition, use "prerequisite" instead.

6. analogous-to
   A and B have the same formal structure (same theorem schema or
   proof pattern) in different domains or contexts.
   Example: "Hensel's Lemma (lifting roots p-adically)"
   --analogous-to--> "Implicit Function Theorem (lifting solutions
   smoothly)"
   Restriction: Both A and B must be THEOREMS. Must be a real
   structural correspondence, not just "both involve X". If you cannot
   state the shared schema in one sentence, it is not analogous.

7. cross-domain-bridge
   A pair where A's domain and B's domain meet: use this when ANY of
   the following hold:
      (a) one side's object is defined using the other's structure, OR
      (b) a canonical/standard theorem establishes a formal
   correspondence between them
         (e.g., gcd(a,b)·Z = (a,b) as an ideal — the formal link
   between GCD and Ideal), OR
      (c) translation between the two domains routinely passes through
   this pair.
   Example: "Vector Space" --cross-domain-bridge--> "Field" (vector
   spaces are modules over fields)
   Example: "Greatest Common Divisor" --cross-domain-bridge-->
   "Ideal" (gcd(a,b) generates the ideal (a,b) in Z)
   Restriction: Nodes must be in DIFFERENT domains. The connection
   must be a recognized correspondence,
   not mere surface similarity.

8. instance-of
   A is a concrete object that exemplifies the abstract structure B.
   Example: "Ring ZMod n" --instance-of--> "Ring" / "Commutative Ring"
   Restriction: Both A and B must be DEFINITIONS (not theorems). A
   must be a concrete/named object; B must be an abstract structure or
   class that A belongs to.

==== WHEN TO RETURN "none" ====

Return "none" if:
- The connection is only "both are about X" (e.g., both about primes,
   both about groups) without a structural relationship.
- The analogy is vague ("both establish invariants", "both are
   foundational").
- You would rate confidence below 0.75.

It is perfectly acceptable — and preferred — to return "none" often.
   A sparse, high-quality graph beats a dense noisy one.

==== FEW-SHOT EXAMPLES ====

Example 1 — strong generalizes:
```

```
  NODE A: "Binomial Theorem" (theorem, Combinatorics): (x+y)^n = Σ
  C(n,k) x^(n-k) y^k
  NODE B: "Sum of Binomial Coefficients" (theorem, Combinatorics): Σ
  C(n,k) = 2^n
  → {"relationship": "generalizes", "confidence": 0.97, "reasoning":
  "Setting x=y=1 in the Binomial Theorem yields Σ C(n,k) = 2^n. B is
  literally a substitution instance of A."}

Example 2 — strong prerequisite:
  NODE A: "Prime Number" (definition, Number Theory): natural number
  p ≥ 2 divisible only by 1 and p
  NODE B: "Fermat's Little Theorem" (theorem, Number Theory): if p
  prime, a^p ≡ a (mod p)
  → {"relationship": "prerequisite", "confidence": 0.98, "reasoning":
  "The definition of prime number is required to state B: the
  hypothesis 'p prime' is essential and primality is used in the
  proof."}

Example 3 — "none" case (both about primes but no structural link):
  NODE A: "Wilson's Theorem" (theorem, Number Theory): (p-1)! ≡ -1
  (mod p) for p prime
  NODE B: "Bertrand's Postulate" (theorem, Number Theory): for n ≥ 1
  there is a prime in (n, 2n]
  → {"relationship": "none", "confidence": 1.0, "reasoning": "Both
  concern primes but share no structural relationship. Neither is
  used in the other's proof, neither generalizes the other, and they
  address unrelated questions (factorial modular identity vs. prime
  gap bound)."}

Example 4 — analogous-to (different domains, shared schema):
  NODE A: "Hensel's Lemma" (theorem, Number Theory): lifts a simple
  root of f mod p to a root in Z_p
  NODE B: "Implicit Function Theorem" (theorem, Analysis): lifts an
  approximate solution to an exact smooth solution under
  non-degeneracy
  → {"relationship": "analogous-to", "confidence": 0.85, "reasoning":
  "Both are lifting theorems with the same schema: 'given an
  approximate solution and a non-degeneracy condition (nonzero
  derivative / Jacobian), a true solution exists and is unique
  locally'. Direct structural parallel across different domains."}

Example 5 — cross-domain-bridge:
  NODE A: "Vector Space" (definition, Linear Algebra): module over a
  field
  NODE B: "Field" (definition, Abstract Algebra): commutative ring
  where every nonzero element is invertible
  → {"relationship": "cross-domain-bridge", "confidence": 0.92,
  "reasoning": "A vector space is defined as a module over a field;
  B's structure is the scalar domain of A. This is the foundational
  bridge between linear algebra and field theory."}

Example 6 — instance-of:
  NODE A: "Ring ZMod n" (definition, Number Theory): Z/nZ with
  addition and multiplication mod n
  NODE B: "Commutative Ring" (definition, Abstract Algebra): ring
  where multiplication is commutative
  → {"relationship": "instance-of", "confidence": 0.95, "reasoning":
  "ZMod n is a concrete example of a commutative ring. The abstract
  structure B is instantiated by the concrete object A."}

Example 7 — applied-in:
  NODE A: "Fundamental Theorem of Arithmetic" (theorem, Number
  Theory): unique prime factorization
```

```
  NODE B: "Lifting the Exponent Lemma" (theorem, Number Theory):
   v_p(a^n - b^n) = v_p(a-b) + v_p(n) under conditions
   → {"relationship": "applied-in", "confidence": 0.88, "reasoning":
   "LTE relies on p-adic valuations, which are well-defined only
   because FTA guarantees unique prime factorizations. FTA's result is
   invoked in B's proof."}

==== TASK ====

Classify the relationship from NODE A to NODE B (directional). Use
  Mathlib names/imports to assess structural proximity when useful.

Respond with EXACTLY this JSON (no other text):
{
  "relationship": "<one of: prerequisite | generalizes | specializes
   | equivalent-to | applied-in | analogous-to | cross-domain-bridge |
   instance-of | none>",
  "confidence": <0.0 to 1.0>,
  "reasoning": "<1-4 sentences. State the specific structural
   correspondence, not just 'both relate to X'.>"
}

If relationship is "none", set confidence to 1.0 (confident no
  relationship exists).
```

D.2 Edge Type Restrictions and Design Rationale

Each of the eight edge types has node-type restrictions (Table 5). These restrictions are stated in the LLM prompt (see the ontology block above) and serve two purposes: (1) they reduce misclassification by excluding structurally impossible combinations (e.g., specializes is excluded between two definitions, because specialization is a relationship between theorem statements); and (2) they sharpen the distinction between similar types, particularly prerequisite vs. applied-in.

Table 5: Node-type restrictions for each edge type. "Defn" = definition, "Thm" = theorem. These constraints are stated verbatim in the LLM prompt and enforced during post-processing.

Edge Type	A **must be**	B **must be**	**Rationale**
prerequisite	defn	any	A concept needed to *state* B
generalizes	thm	thm	Strict containment between statements
specializes	thm	thm	Inverse of generalizes
equivalent-to	thm	thm	Mutual implication between statements
applied-in	thm	thm	A's *result* used in B's *proof*
analogous-to	thm	thm	Shared proof schema across domains or contexts
cross-domain-bridge	any	any	Domains must differ
instance-of	defn	defn	Concrete object exemplifies abstract class

Prerequisite vs. applied-in. These two types both capture "A supports B" but differ in the nature of the dependency:

- prerequisite: A is a *definition* whose concept is required to *formulate* B. Without A, the statement of B cannot be written. Example: the definition of "prime number" is a prerequisite for Fermat's Little Theorem, since the hypothesis "p is prime" is part of the theorem statement.
- applied-in: A is a *theorem* whose result is invoked during the *proof* of B. The statement of B can be written independently, but proving it requires A. Example: Rolle's Theorem is applied in the standard proof of the Mean Value Theorem.

This separation ensures that definitional dependencies (which expand a reader's vocabulary) are distinguished from proof-level dependencies (which suggest proof strategies to the Prover agent).

D.3 Post-Processing Edge Cleanup

Although the node-type restrictions are stated verbatim in the prompt (Table 5), 11.4% of inferred edges (1,041 of 9,141) violate them. A programmatic post-processing pass enforces the restrictions and removes the edges that violate them (Table 6).

Table 6: Post-processing edge cleanup. "Before" counts are the raw LLM output before restriction enforcement; "Dropped" counts are the edges violating the restrictions in Table 5. The last two rows add the auto-inverse edges generated in post-processing.

Edge Type	Before	Dropped	After	Violation
applied-in	2,277	750	1,527	B (rarely A) is a definition
analogous-to	701	118	583	at least one endpoint is a definition
cross-domain-bridge	3,631	139	3,492	same domain
generalizes	69	19	50	B is a definition
specializes	37	6	31	B is a definition
equivalent-to	17	4	13	one endpoint is a definition
prerequisite	2,329	5	2,324	A is a theorem
instance-of	80	0	80	no violations
Total (type violations)	**9,141**	**1,041**	**8,100**	
Duplicate-ID edges (raw)	74	74	0	edges related to two duplicate node IDs
Final raw total			**8,026**	
Auto-inverse edges (generated)	1,417	9	1,408	edges related to two duplicate node IDs
Final graph			**9,434**	

The largest source of violations is applied-in (750 edges, 72% of all violations), where an endpoint, usually B, is a definition. A definition has no proof, so such an edge records no proof dependency; the reasoning texts typically describe the definition's role in stating or proving the theorem (e.g., "Pascal's Rule $\rightarrow$ Binomial Coefficient"), which is the prerequisite relation in the opposite direction. The 139 cross-domain-bridge violations are same-domain pairs.

Of the 132,132 calls, 276 (0.2%) failed after nine retries against a Bedrock daily token quota and were treated as no relation (the affected pairs were not recorded). An additional 74 raw edges were dropped because, at inference time, two node IDs were each assigned to two theorems (`thm_geom_007` to Parallelogram Law and Thales' Theorem; `thm_geom_008` to Orthocenter on Altitude and Riesz Representation), so an edge on either ID could not be attributed to one theorem. The IDs were corrected in the source data after inference. Removing these edges yields the final 8,026 raw edges (per-type counts in §5.1). The auto-inverse step generates 1,417 edges, of which 9 touch the duplicate IDs and are dropped, leaving 1,408 and a final graph of **9,434 edges**.

Table 7 reports connectivity in two ways: on average each node has 51.8 directed edges but only 31.5 distinct neighbors (median 25), because the symmetric types and the generalizes/specializes pair are stored as one edge in each direction, except where the reverse direction already carried an edge of another type (351 edges: 319 cross-domain-bridge, 25 analogous-to, 7 equivalent-to), which the inverse step leaves as returned. Of the 5,740 connected node pairs, 3,694 (64%) are linked in both directions, from the model returning both orderings or from auto-inversion; cross-domain-bridge, at 4,683 of the 9,434 edges, accounts for most of them. The 2,046 one-directional pairs are all prerequisite, applied-in, or instance-of edges (1,400, 636, and 10), types without an auto-inverse. Mutual pairs also occur: 390 carry applied-in in both directions and 288 carry prerequisite in both; post-processing removes contradictory pairs for generalizes and specializes, and these edges remain as returned. Four nodes are isolated but remain in the graph and the Explorer's candidate set.[7]

[7] All four stem from the duplicate-ID fix: Parallelogram Law and Orthocenter on Altitude kept the shared IDs and lost their ambiguous edges, while Thales' Theorem and Riesz Representation on Hilbert spaces received new IDs after inference and so have no edges in this build.

Table 7: MATHKG connectivity statistics (final graph, 364 nodes, 9,434 directed edges).

Statistic	Value
Directed edges per node (in + out), mean	51.8
Distinct neighbors per node, mean	31.5
Distinct neighbors per node, median	25
Unique connected node pairs	5,740
Isolated nodes	4
Highest-degree nodes (directed edges)	Complex Numbers (254), Ring (253), Field (241)

E EXPLORER AND PROVER PROMPTS

E.1 EXPLORER PROMPTS

The Explorer makes two LLM calls per problem, both answered by the evaluated model (Appendix C): seed scoring for Layer 1 (`with_kg` and `full_system`) and search-term suggestion for Layer 2 (`with_mathlib` and `full_system`). Each call sends the Explorer system prompt (first box) together with its own user prompt (second and third boxes). Placeholders in braces are filled per call; in the seed-scoring prompt, the candidate list holds one four-line entry per candidate node.

Explorer system prompt

```
You are an expert mathematician specializing in number theory and
   combinatorics. You excel at identifying structural patterns,
   analogies between mathematical domains, and gaps in mathematical
   knowledge. You reason carefully about formal mathematics.
```

Seed-scoring prompt (Layer 1, one call per problem)

```
Given this mathematical problem, score and select the {s} most
   relevant theorems from the candidates below.

PROBLEM:
{formal_statement}

DOMAIN: {domain}

CANDIDATE THEOREMS ({n} total):
- [{id}] {name}
  Statement: {statement}
  Keywords: {keywords}
  Mathlib: {lean4_mathlib_name}

Select exactly {s} theorems most relevant for solving this problem.
For each, explain WHY it's relevant (proof technique, key lemma,
   analogous structure, etc.).

Respond in JSON:
{
  "seeds": [
    {
      "theorem_id": "...",
      "name": "...",
      "relevance_score": 0.0-1.0,
      "rationale": "why this theorem is relevant"
    }
  ]
}
```

Search-term suggestion prompt (Layer 2, one call per problem)

```
Given this mathematical problem, suggest 10-15 Lean 4 Mathlib
   namespaces or identifier prefixes that would be useful to search
   for when proving it.

PROBLEM (natural language):
{informal_statement}

PROBLEM (formal Lean 4):
{formal_statement}

For each suggestion, classify its importance:
- "essential": directly needed for the proof
- "helpful": likely useful as a supporting lemma
- "speculative": might be relevant, worth checking

Focus on specific Mathlib namespaces and declaration families, e.g.:
- "Nat.Prime" (not just "prime")
- "Function.Injective" (not just "injective")
- "Finset.card" (not just "card")

Respond in JSON:
{
  "search_terms": [
    {"term": "Nat.Prime", "priority": "essential"},
    {"term": "Function.Injective", "priority": "helpful"},
    {"term": "Nat.strongRecOn", "priority": "speculative"}
  ]
}
```

E.2 Prover Prompts

The Prover uses three prompt components. On **attempt 1**, the LLM receives the system prompt plus the initial-generation prompt; on **attempts 2–K**, the system prompt plus the refinement prompt, which carries the previous Lean code and up to ten Lean error messages. When Lean reports unknown identifiers, a note listing them and forbidding their reuse (`{unknown_id_note}` below) is appended to the errors. Placeholders in braces are filled per call. The Explorer's context block (Appendix C) fills `{proof_sketch}` and so appears only at attempt 1; `{context}` is reserved for per-node formal statements, which the released MathKG does not include, so it carried no retrieval content in the reported runs.

Prover system prompt

```
You are an expert in Lean 4 and Mathlib. You write correct, idiomatic
   Lean 4 proofs. You are familiar with Mathlib tactics: exact, apply,
   simp, ring, omega, norm_num, decide, have, obtain, rw, induction,
   rcases, constructor, push_neg, by_contra, linarith, nlinarith,
   polyrith, positivity, field_simp, gcongr, aesop, tauto,
   native_decide, and others.
IMPORTANT Lean 4 syntax rules:
1) Function application uses SPACES not parentheses: write `f x` not
   `f(x)`.
2) `import Mathlib` must be the VERY FIRST line — nothing before it.
3) Stop immediately once all goals are closed — do NOT add extra
   tactics after proof is done.
4) Use Lean 4 syntax ONLY, never Lean 3.
5) NEVER use `sorry` — it is not a valid proof and will be rejected.
   You must complete every proof with real tactics.
When given error messages, you diagnose them precisely and fix the
   proof.
```

Initial-generation prompt (attempt 1)

```
Write a complete Lean 4 proof for the following theorem.

NATURAL LANGUAGE STATEMENT:
{statement}

LEAN 4 TYPE SIGNATURE:
{lean4_sketch}

INFORMAL PROOF SKETCH:
{proof_sketch}

RELEVANT MATHLIB CONTEXT:
{context}

CRITICAL RULES:
1. The FIRST line MUST be `import Mathlib` — nothing before it.
2. Do NOT include any `import` statement other than the first line.
3. Use Lean 4 syntax ONLY. Function application uses spaces: `f x`
   not `f(x)`.
4. Stop immediately once all goals are solved — do NOT add extra
   tactics.
5. For inequality goals: try `linarith` first, then `nlinarith`,
   `polyrith`, or `positivity`.
6. For arithmetic: try `omega`, `norm_num`, or `ring`.
7. For decidable propositions on finite types: try `decide` or
   `native_decide`.
8. Only use Mathlib names you are CERTAIN exist. If a name does not
   exist, try a different proof strategy entirely.
9. Use Lean 4 syntax ONLY. Do NOT use Lean 3 syntax (no begin/end
   blocks). Use by followed by tactic lines.

Return ONLY the Lean 4 code, no explanation.
```

Refinement prompt (attempts 2 to K, after Lean returns errors)

```
The following Lean 4 proof has errors. Fix it.

LEAN 4 CODE:
```lean
{previous_code}
```

ERRORS FROM LEAN:
{error_text}{unknown_id_note}

THEOREM TYPE SIGNATURE (must be preserved):
{lean4_sketch}

MATHLIB CONTEXT:
{context}

CRITICAL RULES:
- The FIRST line MUST be `import Mathlib` — nothing before it, no
  duplicate imports.
- Use Lean 4 syntax ONLY: function application uses spaces `f x` not
  `f(x)`.
- Stop immediately once all goals are solved — do NOT add extra
  tactics.

Common fixes:
```

```
- If 'unknown identifier' / 'unknown constant': the name does not
   exist in Mathlib.
  Try a different lemma or a different proof strategy entirely.
- If 'type mismatch': add explicit type annotations or use `show`
   tactic.
- If 'No goals to be solved' / 'goals accomplished': you have EXTRA
   tactics after
  the proof is done. REMOVE all tactics after the point where goals
   are closed.
- If 'unexpected token' / 'unexpected identifier': you have a SYNTAX
   error.
  Check: no Lean 3 syntax, correct indentation, correct keyword usage.
- If 'Function expected': you are applying something that is not a
   function.
  Check parentheses and application syntax.
- If 'failed to synthesize': add missing typeclass instances with
   `@[instance]` or `haveI`.
- If 'linarith failed': try `nlinarith`, `polyrith`, or `positivity`
   instead.
- For arithmetic goals: try `omega`, `norm_num`, `ring`, or
   `field_simp`.
- For decidable propositions: try `decide` or `native_decide`.
- Only use Mathlib names you are CERTAIN exist. If a name doesn't
   exist, try a different lemma or a different proof strategy entirely.
9. Use Lean 4 syntax ONLY. Do NOT use Lean 3 syntax (no begin/end
   blocks). Use by followed by tactic lines.

Return ONLY the corrected Lean 4 code, no explanation.
```

F Implementation and Compute Setup

Agent Architecture. The MathAgent system is implemented from scratch without external agentic frameworks (e.g., AutoGen, LangChain, CrewAI), using standard Python clients for the three LLM backends and NetworkX for the graph. All agents share one LLM interface that sends a system prompt and a user prompt to Claude via AWS Bedrock or the Anthropic API, or to a local vLLM endpoint for the open-weight models, and a JSON parser that accepts a JSON object whether it is the entire reply, inside a Markdown code fence, or embedded in surrounding text.

Communication Pattern. Agents communicate through:

- **Shared graph**: all agents work on the same MathKG graph object (NetworkX).
- **Structured output**: each agent returns a dictionary with a fixed set of fields.
- **Sequential execution**: for each problem the Explorer builds the context and the Prover then runs its attempt loop; parallelism is across problems only.

API Configuration. Claude Sonnet 4.6 via an AWS Bedrock inference profile. Temperature: 0.3 for initial proof generation, 0.2 for refinement attempts. Thinking mode is disabled for the Qwen3 and Goedel-Prover-V2 models. Maximum output tokens are configured per model: Claude Sonnet 4.6, Qwen3-8B, and Qwen3-32B use 2,000; Goedel-Prover-V2-8B uses 16,384; and Goedel-Prover-V2-32B uses 4,096. Each cap exceeds the length of that model's longest verified proof. LLM calls are retried with exponential backoff (5 retries, 10–160 s delays) to recover from transient API failures.

Graph Representation. MathKG is held in memory as a NetworkX `DiGraph` with one node per theorem or definition. Each node carries six attributes (`id`, `name`, `domain`, `statement`, `type`, `keywords`); each edge carries its type, confidence, and reasoning, and generated inverse edges are marked as such. The released graph file (364 nodes, 9,434 edges, 7.0 MB) is plain JSON with a `nodes` list and an `edges` list; it is the graph the `with_kg` and `full_system` runs searched. The node source file additionally stores each node's Mathlib declaration name, import, and verification status; the released graph carries the six attributes above (Appendix B).

Compute Infrastructure. *Sonnet 4.6 ablation:* AWS EC2 `r7i.48xlarge` (192 vCPU, 1,536 GB RAM), 20 parallel workers. Lean 4 v4.30.0-rc2 with MATHLIB v4.30.0-rc2. Each Lean compilation is allowed up to 1,200 s (20 minutes) of wall-clock time and 200 GB of virtual memory; a compilation exceeding either limit is terminated and counted as a failed attempt.

Qwen3-8B ablation: AWS EC2 `g6e.24xlarge` (96 vCPU, 768 GB RAM, 4× NVIDIA L40S GPUs), 15 parallel workers. Model served via vLLM in BF16, the precision of the released weights. Same Lean 4 timeout and memory limit as in all runs (1,200 s, 200 GB).

Goedel-Prover-V2 (32B, 8B) and Qwen3-32B ablations: AWS EC2 `g7e.48xlarge` (192 vCPU, 2,048 GB RAM, 8× NVIDIA RTX PRO 6000 Blackwell, 96 GB each), with 64 parallel workers for the Goedel-Prover-V2 runs and 96 for Qwen3-32B. All three models were served via vLLM 0.25.0 (PyTorch 2.11, CUDA 13) in BF16, the precision of the released weights, with one full-model replica per GPU and the models' native 40,960-token context window. Verification uses Lean 4 v4.32.0-rc1 with MATHLIB v4.32.0-rc1 under the same Lean timeout and memory limit (1,200 s, 200 GB).

Code Extraction and Verification. The same rule applies to every model: if a reply contains a fenced `lean4` code block, the last such block is taken as the proof; otherwise the reply is cut from its first `import`, `theorem`, or `def` line onward. An attempt then counts as solved only if all three checks pass: (1) *no sorry*: a Lean warning containing "sorry" fails the attempt; (2) *statement fidelity*: the file must contain the assigned theorem with its statement, and every declaration the benchmark supplies, identical to the benchmark text up to whitespace; a file that compiles but alters, omits, or replaces them is not a solve; (3) *clean compilation*: a non-zero exit code or any error diagnostic fails the attempt. PutnamBench problems that ask for a closed-form answer carry an answer slot (an `abbrev` ending in `:= sorry`) with the benchmark's answer in an adjacent comment; the loader fills the slot with that answer, the benchmark's standard setting, so check (2) requires the filled answer to be reproduced as well. This paper reports the 326 PutnamBench problems that carry no answer slot.

Code Availability. The complete implementation (agents, Orchestrator, graph construction, benchmark evaluation), the MATHKG graph and node files, and the per-problem results of all 28 evaluation runs will be released under the MIT license upon publication.

G EXAMPLE GENERATED PROOFS

We show two verified proofs of the same miniF2F problem, `mathd_numbertheory_335` (validation split): if $n \bmod 7 = 5$, then $5n \bmod 7 = 4$, produced in `prover_only` on the first attempt by Claude Sonnet 4.6 and by Goedel-Prover-V2-8B.

Claude Sonnet 4.6, `prover_only`, attempt 1

```
import Mathlib

theorem mathd_numbertheory_335
  (n : ℕ)
  (h₀ : n % 7 = 5) :
  (5 * n) % 7 = 4 := by
  omega
```

Sonnet closes the goal with a single decision procedure. Goedel-Prover-V2-8B, below, reaches the same goal through nested `have` steps, with the arithmetic done by one `simp` rewrite followed by `norm_num`. Both proofs pass the checks of Appendix F, and both models solved this problem in all four augmentation modes.

Goedel-Prover-V2-8B, `prover_only`, attempt 1

```
import Mathlib

theorem mathd_numbertheory_335
  (n : ℕ)
  (h₀ : n % 7 = 5) :
  (5 * n) % 7 = 4 := by
  have h₁ : (5 * n) % 7 = 4 := by
    have h₂ : n % 7 = 5 := h₀
    have h₃ : (5 * n) % 7 = 4 := by
      have h₄ : n % 7 = 5 := h₂
      have h₅ : (5 * n) % 7 = 4 := by
        -- Use the fact that n ≡ 5 mod 7 to simplify the expression
        have h₆ : n % 7 = 5 := h₄
        have h₇ : (5 * n) % 7 = 4 := by
          -- Use the property of modular arithmetic
          -- to simplify the expression
          have h₈ : n % 7 = 5 := h₆
          have h₉ : (5 * n) % 7 = (5 * (n % 7)) % 7 := by
            simp [Nat.mul_mod, Nat.add_mod, Nat.mod_mod]
          rw [h₉, h₈]
          <;> norm_num
        exact h₇
      exact h₅
    exact h₃
  exact h₁
```

H Sonnet Ablation Across Benchmarks

This appendix gives the full three-benchmark ablation behind §5.4, using Sonnet 4.6. Table 8 reports the four-mode results with the per-problem oracle (the union of problems solved by any of the four modes): each augmented mode lowers the three-benchmark aggregate by 0.5–2.9% relative to `prover_only`, while the oracle exceeds `prover_only` by 6% to 50%, most on PutnamBench.

Table 8: Sonnet 4.6 full four-mode ablation across three benchmarks (pass@32), with the per-problem *oracle* (union of all four modes). Best fixed mode per row in **bold**; the oracle exceeds `prover_only` by $+6\%$ to $+50\%$, most on PutnamBench.

Benchmark	prover_only	with_kg	with_mathlib	full_system	oracle	Δ vs. prover_only
MathOlympiadBench (360)	**39 (10.8%)**	33 (9.2%)	36 (10.0%)	38 (10.6%)	50 (13.9%)	+11 (+28%)
miniF2F (488)	365 (74.8%)	358 (73.4%)	359 (73.6%)	**366 (75.0%)**	388 (79.5%)	+23 (+6%)
PutnamBench (326)	14 (4.3%)	**15 (4.6%)**	13 (4.0%)	12 (3.7%)	21 (6.4%)	+7 (+50%)
Total (1,174)	**418 (35.6%)**	406 (34.6%)	408 (34.8%)	416 (35.4%)	**459 (39.1%)**	+41 (+10%)

Table 9 reports, per benchmark, how many problems each augmented mode solves *beyond* `prover_only`, together with the oracle and the problems lost (solved by `prover_only` but by no augmented mode); Figure 4 shows the disjoint overlap structure.

Table 9: Sonnet 4.6 problem-level complementarity. `+KG`/`+Mathlib`/`+Full`: problems each mode solves beyond `prover_only` (the modes overlap; disjoint regions are in Figure 4). Oracle: solved by any of the four modes. Lost: solved only by `prover_only`.

Benchmark	prover_only	+KG	+Mathlib	+Full	Oracle / Lost
MathOlympiadBench	39	+6	+4	+9	50 / −3
miniF2F	365	+10	+9	+18	388 / −7
PutnamBench	14	+4	+2	+3	21 / 0

Aggregated across the three benchmarks, 41 problems are solved by at least one augmented mode but not by `prover_only`, versus 10 solved only by `prover_only` and lost by every augmented

mode, roughly 4:1 in favor of augmentation at the problem level, even though aggregate counts favor the baseline. The oracle exceeds `prover_only` by 6% on miniF2F, 28% on MathOlympiadBench, and 50% on PutnamBench: complementarity strengthens on harder problems.

Pass@k analysis. Table 10 breaks down the Sonnet ablation by attempt budget. On miniF2F, `with_mathlib` leads at pass@1 (+12 over `prover_only`), but `prover_only` leads from pass@3 to pass@20, with the augmented modes trailing by 9 to 15 problems at pass@3 and pass@5; at pass@32 `with_kg` and `with_mathlib` trail by 7 and 6 while `full_system` edges ahead by one. On PutnamBench the counts are small and the lead alternates: `prover_only` is best or tied at pass@1, 3, 5, and 20, while `with_kg` leads by 2 at pass@10 and by 1 at pass@32. On MathOlympiadBench the counts are small and the modes trade places (`with_mathlib` leads at pass@5, `full_system` at pass@10 and pass@20), with `prover_only` ahead by 1 to 6 at pass@32.

Table 10: Sonnet 4.6 pass@k solve counts (solve rate); best mode per row in **bold**.

pass@k	prover_only	with_kg	with_mathlib	full_system
miniF2F (488 problems)				
pass@1	198 (40.6%)	201 (41.2%)	**210 (43.0%)**	206 (42.2%)
pass@3	**299 (61.3%)**	284 (58.2%)	290 (59.4%)	290 (59.4%)
pass@5	**326 (66.8%)**	311 (63.7%)	311 (63.7%)	316 (64.8%)
pass@10	**349 (71.5%)**	339 (69.5%)	339 (69.5%)	341 (69.9%)
pass@20	**358 (73.4%)**	352 (72.1%)	352 (72.1%)	352 (72.1%)
pass@32	365 (74.8%)	358 (73.4%)	359 (73.6%)	**366 (75.0%)**
PutnamBench (326 problems)				
pass@1	**3 (0.9%)**	2 (0.6%)	**3 (0.9%)**	2 (0.6%)
pass@3	**5 (1.5%)**	**5 (1.5%)**	4 (1.2%)	**5 (1.5%)**
pass@5	**7 (2.1%)**	6 (1.8%)	4 (1.2%)	**7 (2.1%)**
pass@10	9 (2.8%)	**11 (3.4%)**	7 (2.1%)	10 (3.1%)
pass@20	**12 (3.7%)**	11 (3.4%)	9 (2.8%)	**12 (3.7%)**
pass@32	14 (4.3%)	**15 (4.6%)**	13 (4.0%)	12 (3.7%)
MathOlympiadBench (360 problems)				
pass@1	**6 (1.7%)**	3 (0.8%)	**6 (1.7%)**	4 (1.1%)
pass@3	**11 (3.1%)**	5 (1.4%)	**11 (3.1%)**	8 (2.2%)
pass@5	16 (4.4%)	13 (3.6%)	**18 (5.0%)**	14 (3.9%)
pass@10	25 (6.9%)	22 (6.1%)	25 (6.9%)	**26 (7.2%)**
pass@20	33 (9.2%)	28 (7.8%)	33 (9.2%)	**34 (9.4%)**
pass@32	**39 (10.8%)**	33 (9.2%)	36 (10.0%)	38 (10.6%)

I Context Overflow (Goedel-Prover-V2-32B)

Context overflow appeared only in the Goedel-Prover-V2-32B runs, and only with KG augmentation. The KG context block is uncapped (Appendix C), and for some problems the KG-augmented prompt plus the 4,096-token generation budget exceeded the served 40,960-token context window, so the vLLM server rejected the request. The client then lowers the generation budget to the room the prompt leaves and resends the same prompt, so the model still sees the full augmentation context. When the prompt alone filled the window, every retry failed, and after five retries the evaluator recorded the problem as an error, counted as a non-solve. This affected only the two KG modes: 10 problems under `with_kg` and 28 under `full_system` (one problem in both), while `prover_only` and `with_mathlib` saw none. No other model's runs recorded an overflow.

In Table 2 the overflowed problems are counted as failures against the full denominator of 488, giving Goedel-32B 234/488 (48.0%) for `with_kg` and 208/488 (42.6%) for `full_system`. Excluding them, the rates are 234/478 (49.0%) and 208/460 (45.2%), while `prover_only`

and `with_mathlib` are unchanged at 237/488 (48.6%) and 235/488 (48.2%). Restricting `prover_only` to the same problems gives 233/478 and 214/460, so the `with_kg` gap of Table 3 moves from -3 to $+1$, and the `full_system` gap closes from -29 to -6 against `prover_only` and, in Table 3's KG-beyond-Mathlib row, from -27 to -3 against `with_mathlib`.

J Significance Tests for Table 3

Table 11 reports, for every delta in Table 3, the p-value of a two-sided exact McNemar test on the discordant problems, those solved under one mode and not the other. Six deltas are significant at $p < 0.05$; every Sonnet delta and every Mathlib $-$ `prover_only` delta is within noise.

Table 11: Solve-count deltas of Table 3 with McNemar p-values (miniF2F, 488 problems).

Question	Qwen3-8B	Qwen3-32B	Goedel-8B	Goedel-32B	Sonnet
KG helps? (kg − PO)	+2 (.79)	−14 (.007)	+14 (.029)	−3 (.76)	−7 (.25)
Mathlib helps? (mathlib − PO)	−2 (.82)	0 (1.0)	−1 (1.0)	−2 (.86)	−6 (.31)
KG beyond Mathlib? (full − mathlib)	+5 (.30)	−16 (.002)	+1 (1.0)	−27 (<.001)	+7 (.28)
Synergy? (full − max(kg,mathlib))	+1 (1.0)	−16 (.002)	−14 (.024)	−27 (<.001)	+7 (.28)

K Oracle at a Matched Attempt Budget

Table 12 compares the oracle of Table 4 with a matched-budget variant that gives each of the four augmentation modes 8 attempts, 32 in total, the same number `prover_only` receives at pass@32; a problem counts for the matched-budget union if any mode solves it within its first 8 attempts.

Table 12: Oracle at a matched attempt budget on miniF2F (488 problems); total attempts in parentheses in the header. Matched oracle: 8 attempts per mode (4×8=32). Full oracle: 32 attempts per mode (4×32=128), as in Table 4. Gains: oracle minus `prover_only`, in problems (solve-rate points).

Model	prover_only (32)	Matched oracle (32)	Matched gain	Full oracle (128)	Full gain
Qwen3-8B	33 (6.8%)	51 (10.5%)	+18 (+3.7)	52 (10.7%)	+19 (+3.9)
Qwen3-32B	62 (12.7%)	79 (16.2%)	+17 (+3.5)	80 (16.4%)	+18 (+3.7)
Goedel-8B	205 (42.0%)	232 (47.5%)	+27 (+5.5)	241 (49.4%)	+36 (+7.4)
Goedel-32B	237 (48.6%)	259 (53.1%)	+22 (+4.5)	266 (54.5%)	+29 (+5.9)
Sonnet 4.6	365 (74.8%)	370 (75.8%)	+5 (+1.0)	388 (79.5%)	+23 (+4.7)