# HILBERT: RECURSIVELY BUILDING FORMAL PROOFS WITH INFORMAL REASONING

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

## ABSTRACT

Large Language Models (LLMs) demonstrate impressive mathematical reasoning abilities, but their solutions frequently contain errors that cannot be automatically verified. Formal theorem proving systems such as Lean 4 offer automated verification with complete accuracy, motivating recent efforts to build specialized prover LLMs that generate verifiable proofs in formal languages. However, a significant gap remains: current prover LLMs solve substantially fewer problems than general-purpose LLMs operating in natural language. We introduce HILBERT, an agentic framework that bridges this gap by combining the complementary strengths of informal reasoning and formal verification. Our system orchestrates four components: an informal LLM that excels at mathematical reasoning, a specialized prover LLM optimized for Lean 4 tactics, a formal verifier, and a semantic theorem retriever. Given a problem that the prover is unable to solve, HILBERT employs recursive decomposition to split the problem into subgoals that it solves with the prover or reasoner LLM. It leverages verifier feedback to refine incorrect proofs as necessary. Experimental results demonstrate that HILBERT substantially outperforms existing approaches on key benchmarks, achieving 99.2% on miniF2F, 6.6% points above the best publicly available method. HILBERT achieves the **best known result** on PutnamBench. It solves 462/660 problems (70.0%), outperforming proprietary approaches like SeedProver (50.4%) and achieving a 422% improvement over the best publicly available baseline. Thus, HILBERT effectively narrows the gap between informal reasoning and formal proof generation.

## 1 INTRODUCTION

General-purpose Large Language Models (LLMs) have achieved dramatic improvements in mathematical understanding. Reasoning LLMs like GPT-5 and Gemini 2.5 Pro attain near-perfect performance on high-school olympiad exams such as AIME and can solve a significant proportion of competitive undergraduate-level problems from the Putnam exam (Dekoninck et al., 2025). These systems also show promise on research-level benchmarks like FrontierMath (Glazer et al., 2024; OpenAI, 2025).

However, several fundamental limitations severely constrain their practical utility. These systems frequently hallucinate, producing confident-sounding but ultimately incorrect solutions. Even when the final answers are correct, the underlying reasoning often contains serious flaws: "proving" by example, logical fallacies, unjustified assumptions, and calculation errors (Petrov et al., 2025; Guo et al., 2025; Mahdavi et al., 2025; Balunović et al., 2025). Manual verification of generated proofs is time-consuming, difficult, and error-prone. Although recent advances show LLM-based verifiers can approach human-level performance (Guo et al., 2025; Dekoninck et al., 2025), they remain fallible due to hallucinations and silent failures (Mahdavi et al., 2025; Petrov et al., 2025).

Formal theorem proving systems such as Lean 4 (Moura & Ullrich, 2021) offer a promising solution by enabling automated proof verification with complete accuracy, guaranteeing to prove or disprove the

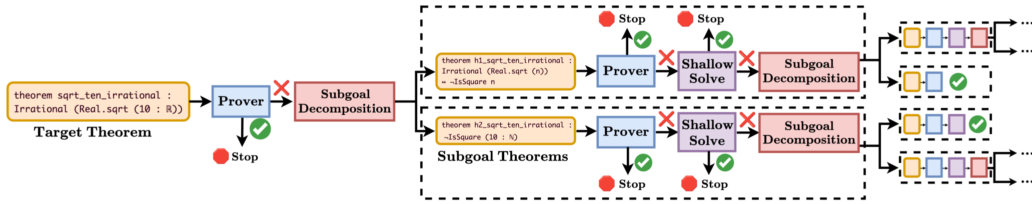


Figure 1: **The HILBERT algorithm.** Given a target theorem, HILBERT attempts formal proof generation with the prover. Upon failure, it decomposes the problem into subgoals and tries to solve them with the prover, followed by the reasoner (shallow solve). If both strategies fail, it resorts to recursive decomposition until all subgoals are resolved.

correctness of proofs in formal languages. This capability has spurred the development of purpose-built prover LLMs (Polu & Sutskever, 2020), with substantial research focused on developing specialized models for generating formal Lean 4 proofs (Yang et al., 2023; Xin et al., 2024a;b; 2025; Ren et al., 2025; Dong & Ma, 2025; Wang et al., 2025). The best open prover models achieve over 90% pass rate on miniF2F (Zheng et al., 2021) and solve 86 of 657 problems on the challenging PutnamBench (Tsoukalas et al., 2024). Proprietary systems such as AlphaProof (AlphaProof & AlphaGeometry, 2024) and SeedProver (Chen et al., 2025) demonstrate this paradigm's potential, achieving a silver-medal performance on problems from the International Mathematical Olympiad (IMO).

Despite this progress, a significant performance gap remains between specialized prover LLMs and general-purpose reasoning LLMs. For example, Dekoninck et al. (2025) found through human verification that reasoning LLMs can solve approximately 83% of PutnamBench problems informally, while the best publicly available prover LLMs achieve only 13% with formal proofs. General-purpose LLMs excel at informal mathematical reasoning and understand formal language syntax well enough to write effective proof sketches and short proofs (Ren et al., 2025; Liang et al., 2025). However, they struggle with full formal program synthesis, achieving only 49.1% pass rate (with 16384 attempts) on miniF2F (Zhou et al., 2025b). Conversely, specialized prover LLMs excel at producing syntactically correct formal proofs for standalone theorems, but are brittle at language-intensive tasks like leveraging existing theorems or error correction (Liang et al., 2025).

To address this gap, several works have explored incorporating informal reasoning from general-purpose LLMs to augment formal theorem-proving capabilities. Early approaches like DSP (Jiang et al., 2022) and LEGO-Prover (Wang et al., 2023) used general-purpose LLMs to propose proof sketches, with automated theorem provers (ATPs) filling formal components, but were limited by heuristics-based ATP capabilities. DSP+ (Cao et al., 2025) extended this approach using modern prover LLMs for intermediate steps. However, these methods struggle with complex subgoals due to shallow, single-layer decomposition. They break down the original problem but cannot further decompose subgoals that remain too difficult to solve directly. Recent agentic frameworks including COPRA (Thakur et al., 2024), Prover-Agent (Baba et al., 2025), and ProofCompass (Wischermann et al., 2025) iteratively construct proofs using informal reasoning with feedback from the formal verifier. Although these methods show promise, their performance still significantly lags behind general-purpose reasoning LLMs.

We introduce HILBERT, an agentic framework that bridges informal reasoning with formal verification (Figure 1). It orchestrates four key components: a general-purpose reasoning LLM, a prover LLM, a verifier, and a semantic theorem retriever. Given a mathematical problem, HILBERT first retrieves relevant theorems from Mathlib (mathlib Community, 2020) and generates a detailed informal proof using the reasoner. It then creates a Lean 4 proof sketch decomposing the problem into manageable subgoals. For each subgoal, HILBERT employs a two-stage approach: attempting formal proof generation with the prover, then falling back to the reasoner augmented with retrieved theorems. When both stages fail, the system recursively decomposes problematic subgoals into smaller problems. At every stage, HILBERT leverages the reasoner's superior in-context learning capabilities to interpret compilation errors, suggest corrections, and guide proof refinement. We summarize our main contributions below.

- We design HILBERT, a multi-turn agentic framework that systematically combines informal mathematical reasoning with formal proof verification, closing the performance gap between these two paradigms.
- We conduct comprehensive experiments on MiniF2F and PutnamBench, achieving state-of-the-art performance on both benchmarks. HILBERT reaches 99.2% pass rate on miniF2F (6.6 points above the best public method) and solves 462/660 PutnamBench problems (70.0%), outperforming proprietary systems like SeedProver (50.4%) and achieving over 4× improvement versus the best open-source baseline.
- Through extensive ablation studies, we validate the effectiveness of our key technical contributions: the recursive decomposition procedure for breaking down complex proofs and the retrieval-augmented generation mechanism for enhanced reasoning capabilities.

## 2 Related Work

Automated Theorem Provers (ATPs) are computational systems designed to automatically discover proofs of mathematical theorems. Traditional approaches have primarily relied on symbolic reasoning methods (Robinson, 1965; McCune, 2003; Schulz, 2002) and integration tools like Sledgehammer that connect ATPs with interactive proof assistants (Blanchette et al., 2013; Czajka & Kaliszyk, 2018). Recently, LLMs have emerged as a promising new tool for automated theorem proving (Polu & Sutskever, 2020; Yang et al., 2024).

**Prover LLMs.** The general principle is to train specialized prover LLMs on large datasets of formal proofs, most prominently for the Lean (Moura & Ullrich, 2021) theorem prover. Some prominent models include GPT-f (Polu & Sutskever, 2020), ReProver (Yang et al., 2023), DeepSeek Prover family of models (Xin et al., 2024a;b; Ren et al., 2025), ABEL (Gloeckle et al., 2024), Goedel Prover V1 and V2 (Lin et al., 2025a;b), BFS Prover (Xin et al., 2025), STP-Prover (Dong & Ma, 2025) and Kimina Prover (Wang et al., 2025). These models are trained by curating a substantial corpus of formal proofs and performing some combination of supervised finetuning and reinforcement learning. Several approaches have enhanced these models by incorporating subgoal decomposition into the training process (Zhao et al., 2023; 2024; Ren et al., 2025), while POETRY (Wang et al., 2024) and ProD-RL (Dong et al., 2024) employ recursive problem decomposition. Proprietary prover LLMs like AlphaProof (AlphaProof & AlphaGeometry, 2024) and SeedProver (Chen et al., 2025) have pushed the frontier further, achieving a silver-medal performance on problems from the International Mathematics Olympiad (IMO). Still, significant performance gaps remain between specialized prover models and general-purpose LLMs in mathematical reasoning capabilities (Dekoninck et al., 2025).

**Using Informal LLMs for Formal Theorem Proving.** Several previous works have attempted to incorporate informal reasoning from general-purpose LLMs to improve formal reasoning abilities. DSP (Jiang et al., 2022) used the Codex LLM to propose proof sketches in Isabelle, with intermediate steps filled in by Sledgehammer. LEGO-Prover (Wang et al., 2023) extended this framework to handle a growing skill library of intermediate theorems for retrieval-augmented proving. Liang et al. (2025) argue that general purpose reasoning LLMs are more effective at decomposing problems into simpler subgoals compared to prover LLMs. Our work extends upon this observation by using informal reasoners to recursively build proof sketches to break the problem down into simpler sub-problems that can be handled by a prover or reasoning LLM.

Several works have also proposed using an informal LLM in an agentic framework for automated theorem proving. COPRA (Thakur et al., 2024) queries an informal LLM to construct proofs tactic by tactic, incorporating execution feedback, search history, and retrieved lemmas into subsequent prompts. Prover-Agent (Baba et al., 2025) uses a small informal reasoning model to produce proof steps and lemmas, which are autoformalized and solved using a prover LLM. Feedback from Lean is used to iteratively refine incorrect proofs. ProofCompass (Wischermann et al., 2025) enhances prover LLMs by adding informal proof steps as comments in the input. When proof attempts fail, it analyzes these failures to extract intermediate lemmas that enable effective problem decomposition. DeltaProver (Zhou et al., 2025b) introduces a custom Domain-Specific Language to perform subgoal decomposition, and iteratively repair the generated proof using verifier feedback. Notably, it only uses an informal LLM and does not rely on prover LLMs. In contrast, our work demonstrates that prover LLMs become highly effective tools when orchestrated in an appropriately designed multi-agent framework.

# 3 HILBERT SYSTEM

In this section, we detail HILBERT, a multi-agent system that bridges informal mathematical reasoning and formal verification by orchestrating general-purpose reasoning LLMs with specialized prover LLMs. Our approach uses recursive subgoal decomposition to break complex theorems into simpler subgoals that can be proven and combined, achieving performance exceeding either approach in isolation.

## 3.1 COMPONENTS

Before we describe the inference algorithm, we first describe the components that HILBERT orchestrates.

**Reasoner.** A general-purpose reasoning LLM to write informal proofs, proof sketches in Lean, and in certain instances, a formal proof. In our work, we use Google Gemini 2.5 Flash and Pro (Comanici et al., 2025) due to their superior mathematical reasoning capabilities (Zhou et al., 2025b; Dekoninck et al., 2025).

**Prover.** A specialized prover LLM to write formal proofs given a formal theorem statement. In our work, we use DeepSeek-V2-7B (Ren et al., 2025) and Goedel-Prover-V2 32B (Lin et al., 2025b).

**Verifier.** A formal language verifier to check the correctness of the theorem statements and proofs. We use the Kimina Lean Server (Santos et al., 2025) with Lean v4.15.0 and Mathlib v4.15.0.

**Retriever.** A semantic search engine to retrieve relevant theorems from Mathlib (mathlib Community, 2020) built using sentence transformers (`all-mpnet-base-v2` (Song et al., 2020)) and FAISS (Douze et al., 2024) indexing. The system computes cosine similarity between query embeddings and pre-computed embeddings of informal theorem descriptions from the `mathlib_informal` (Gao et al., 2024) dataset, providing a simple yet effective alternative to custom retrieval models (Gao et al., 2024; Lu et al., 2025).

## 3.2 ALGORITHM

Given a formal statement in Lean 4, we first attempt direct proof using the Prover. It generates $K_{\text{initial proof}} = 4$ candidate proofs, which we verify using the Verifier. If any proof is valid, we return it immediately. When direct proof attempts fail, we use the Reasoner to decompose the problem into simpler subproblems and assemble them into a valid proof strategy. Figure 2 provides an overview of this stage.

### 3.2.1 SUBGOAL DECOMPOSITION

**Step 1** (Theorem Retrieval). Given the formal statement, we prompt the Reasoner to produce $s = 5$ search queries to look for theorems that might help simplify the proof strategy. For each search query, we use the Retriever to retrieve the top $m = 5$ most semantically similar theorems and tactics from Mathlib. We again query the Reasoner to select only the relevant theorems from the fetched search results.

**Step 2** (Formal Proof Sketch Generation). We prompt the Reasoner to produce a detailed informal proof using the retrieved theorems. With this proof supplied in-context, we ask the Reasoner to generate a Lean 4 proof sketch that decomposes the problem into simpler subproblems represented as `have` statements. All subgoals are initially filled with `sorry`, a placeholder keyword that Lean can temporarily treat as a proof of

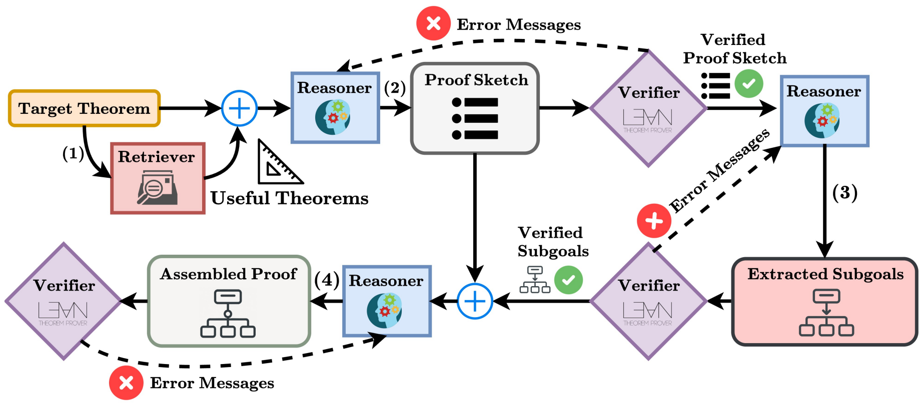


Figure 2: **Subgoal Decomposition:** Given a theorem statement, HILBERT: (1) retrieves relevant theorems from Mathlib using semantic search, (2) generates a formal proof sketch with subgoals marked as `have` statements with `sorry` placeholders, (3) extracts these subgoals as independent theorem statements, and (4) assembles the proof by replacing `sorry` placeholders with calls to the subgoal theorems. Verifiers ensure correctness at each stage. The error correction loops are indicated by dotted lines.

the subgoal. We verify that the proof sketch is valid using the Verifier and leverage its feedback to correct any errors. We generate a maximum of $K_{\text{sketch attempts}} = 4$ sketch attempts for each input theorem.

**Step 3** (Subgoal Extraction). The Reasoner extracts subgoals from the proof sketch, converting them into independent theorem statements with relevant context from the original problem and preceding subgoals. As before, we use `sorry` for the proof. We verify completeness by counting `have` statements in the proof sketch and ensuring that all of them are extracted. In case any of them are missing, we prompt the Reasoner to extract the missing subgoals. Each extracted theorem undergoes syntax verification using the Verifier. When errors occur, we provide error messages in-context to the Reasoner for correction. This approach proves more reliable than parsing source code directly or extracting subgoals from Lean 4's proof state data structure (InfoTree) (Liang et al., 2025).

**Step 4** (Proof Assembly from Subgoals). We provide the Reasoner with the extracted subgoal theorem statements (which contain `sorry` placeholders) and validated proof sketch. The Reasoner produces an assembled proof for the target theorem by replacing each `sorry` placeholder in the proof sketch with calls to the corresponding subgoal theorem. We then verify both the subgoal theorem statements and the assembled proof together using the Verifier to ensure the overall structure is sound. We check for errors using the Verifer and correct them through iterative feedback with the Reasoner. This guarantees that after all subgoals are proven, we will have a complete proof of the given theorem.

### 3.2.2 SUBGOAL VERIFICATION

At this stage, we have a valid theorem proof structure and a list of subgoals that, if proven, complete the original proof. However, the mathematical correctness and provability of these subgoals remain unverified. For each subgoal, we execute the following verification and proof process:

**Step 1** (Prover Attempts). We first attempt to prove each subgoal directly using the Prover, generating $K_{\text{formal proof}} = 4$ candidate proofs and verifying them with the Verifier. If any generated proof is valid, we accept it and proceed to the next subgoal.

**Step 2** (Correctness Verification). For subgoals that cannot be directly proven, we prompt the Reasoner to evaluate whether the subgoal is mathematically correct and whether the formal statement is formulated correctly and provable. If the Reasoner identifies the subgoal as mathematically incorrect, unprovable, or poorly formulated, we flag it for correction and return to refine the original proof sketch, repeating all steps from Section 3.2.1 onwards with the identified issues incorporated as feedback. Apart from mathematical errors, some common failure modes detected by the Reasoner at this stage include missing hypotheses or conditions in the subgoal theorem statement, and atypical behavior due to the Lean type system, such as truncation of natural numbers[1].

We prioritize direct Prover attempts over Reasoner verification because the Prover models are computationally cheaper, and a valid proof automatically confirms mathematical correctness. Empirically, we observe that a significant proportion of generated subgoals can be successfully proven by the Prover. Step 1 ensures that we save on the computational costs of the expensive Reasoner model for verification on the successful subgoals.

**Step 3** (Shallow Solve). After Step 1 fails and Step 2 confirms subgoal correctness, we employ a Reasoner model for a "shallow solve" approach that writes short proofs for subgoals the Prover could not directly solve. We retrieve relevant theorems from the Mathlib library and ask the Reasoner to write a formal proof for the subgoal. The Reasoner iteratively refines proofs based on Verifier feedback for up to $K_{\text{proof correction}} = 6$ passes. When compilation errors indicate missing or incorrect theorem references, we retrieve additional relevant theorems. To preserve computational resources, we terminate this step if an incorrect proof exceeds the length threshold $K_{\text{max shallow solve length}} = 30$ lines, as excessively long proofs indicate the need for further decomposition. This entire shallow solve process repeats for up to $K_{\text{informal passes}} = 6$ attempts until we obtain a successful proof or exhaust all attempts.

**Step 4** (Recursive Decomposition and Proof Assembly). If subgoals remain unproven after Steps 1-3, we recursively apply the subgoal decomposition process (Section 3.2.1) to break them down further. Each subgoal is subdivided until it is either successfully proven or we reach the maximum recursion depth $D$. Should all subgoals become proven, we proceed to create a complete proof for the given theorem by stitching together the proofs for all subgoals and the assembled proof outline from Step 4 of subgoal decomposition. This is done by concatenating the proofs of the subgoals with the assembled proof produced in Step 4 of subgoal decomposition (Section 3.2.1). Any remaining unsolved subgoals at this point trigger a failed proof attempt, prompting us to restart the subgoal decomposition process for the theorem.

The complete algorithm is presented in Algorithm 1. For implementation details, particularly parallelization strategies, refer to Section A.3.

# 4 Experimental Results

## 4.1 Main Results

**MiniF2F.** The MiniF2F dataset (Zheng et al., 2021) is a 488 problem dataset comprising of high-school mathematics competition problems. Some problems are particularly challenging, sourced from the AMC, AIME and IMO competitions. We benchmark on the 244 problems from the test split of MiniF2F. We use recursion depth $D = 5$ for all our experiments. For the Prover, we instantiate Hilbert with two LLMs: DeepSeek-Prover-V2-7B (Ren et al., 2025), representing a relatively weaker model, and Goedel-Prover-V2-32B (Lin et al., 2025b), representing a stronger one. This pairing allows us to compare performance across different capability levels. For the Reasoner, we analogously employ Google's Gemini 2.5 Flash and Gemini 2.5 Pro (Comanici et al., 2025). The results are presented in Table 1.

Hilbert, demonstrates strong performance across all model configurations. Our top-performing setup combines Gemini 2.5 Pro with Goedel-Prover-V2-32B, achieving a 99.2% pass rate and failing on only two problems (AMC 12A 2020 Problem 25 and IMO Shortlist 2007 Problem A6). Even with weaker formal

[1] `https://lean-lang.org/doc/reference/latest/Basic-Types/Natural-Numbers/`

| Method | Pass Rate |
|---|---|
| STP (Dong & Ma, 2025) (pass@3200) | 65.0% ± 0.5% |
| (pass@25600) | 67.6% |
| Kimina-Prover-8B (Wang et al., 2025) (pass@32) | 78.3% |
| Kimina-Prover-72B (pass@1024) | 87.7% |
| w/ TTRL | 92.2% |
| Gemini 2.5 Pro (pass@16384) | 49.1% |
| Delta Prover (Zhou et al., 2025b) (pass@16384) | 95.9% |
| Seed Prover (Chen et al., 2025) | 99.6% |
| Goedel-Prover-SFT (Lin et al., 2025a) (pass@3200) | 62.7% |
| Goedel-Prover-V2-8B (Lin et al., 2025b) (pass@8192) | 90.2% |
| w/ self-correction (pass@1024) | 89.3% |
| Goedel-Prover-V2-32B (pass@4) | 74.6% ± 1.2% |
| (pass@8192) | 92.2% |
| w/ self-correction (pass@1024) | 92.6% |
| HILBERT (Gemini 2.5 Flash) + Goedel-Prover-V2-32B | 94.7% [+20.1%] |
| HILBERT (Gemini 2.5 Pro) + Goedel-Prover-V2-32B | 99.2% [+24.6%] |
| DeepSeek-Prover-V2-7B (CoT) (Ren et al., 2025) (pass@8192) | 82.0% |
| DeepSeek-Prover-V2-7B (non CoT) (pass@4) | 61.3% ± 0.2% |
| (pass@8192) | 75.0% |
| DeepSeek-Prover-V2-671B (pass@8192) | 88.9% |
| HILBERT (Gemini 2.5 Flash) + DS Prover-V2-7B (non-CoT) | 96.7% [+35.4%] |
| HILBERT (Gemini 2.5 Pro) + DS Prover-V2-7B (non-CoT) | 98.4% [+37.1%] |

Table 1: **Results on the MiniF2F-Test dataset.** Improvements shown in brackets for HILBERT are calculated relative to the pass@4 baseline for each prover family. Note: Delta Prover and Seed Prover are proprietary methods and not publicly available to use. Gemini 2.5 Pro result obtained from Zhou et al. (2025b)

| Model | # Solved Problems | % Solved Problems |
|---|---|---|
| Goedel-Prover-SFT (Lin et al., 2025a) (pass@512) | 7/644 | 1.1% |
| ABEL (Gloeckle et al., 2024) (pass@596) | 7/644 | 1.1% |
| Self-play Theorem Prover (Dong & Ma, 2025) (pass@3200) | 8/644 | 1.2% |
| Kimina-Prover-7B-Distill (Wang et al., 2025) (pass@192) | 10/657 | 1.5% |
| DSP+ (Cao et al., 2025) (pass@128) | 23/644 | 3.6% |
| Bourbaki (Zimmer et al., 2025) (pass@512) | 26/658 | 4.0% |
| DeepSeek-Prover-V2 671B (Ren et al., 2025) (pass@1024) | 47/657 | 7.1% |
| SeedProver (Chen et al., 2025) | <u>331/657</u> | <u>50.4%</u> |
| Goedel-Prover-V2-32B (self-correction) (Lin et al., 2025b) (pass@184) | 86/644 | 13.4% |
| HILBERT (Gemini 2.5 Pro) + Goedel-Prover-V2-32B | 462/660 | 70.0% |

Table 2: **Results on the PutnamBench dataset.** We benchmark on the most recent version (as of September 2025) containing 660 problems.

provers, HILBERT maintains impressive results: pairing DeepSeek-Prover-V2-7B with Gemini 2.5 Pro yields 98.4%, while using Gemini 2.5 Flash achieves 96.7%. Notably, the choice of informal reasoner appears more critical than prover strength. Gemini 2.5 Pro consistently outperforms Flash variants by 3-4%, a larger gap than observed between different prover models. Compared to standalone base provers at pass@4, our approach delivers substantial improvements ranging from 20.1% to 37.1%.

**PutnamBench.** PutnamBench is a challenging theorem-proving benchmark comprising 660 problems from the William Lowell Putnam Mathematical Competition from 1962 to 2024. It contains undergraduate-level

| Method | Retrieval | Pass Rate | # Reasoner Calls | # Prover Calls | # Reasoner Tokens | # Prover Tokens |
|---|---|---|---|---|---|---|
| HILBERT+ DeepSeek-Prover-V2-7B | ✓ | **98.4%** | **420** | **205** | **1.9M** | **0.3M** |
| HILBERT+ DeepSeek-Prover-V2-7B | ✗ | 97.1% | 426 | 290 | 2.1M | 0.4M |
| HILBERT+ Goedel-Prover-V2-32B | ✓ | **99.2%** | **548** | **391** | **2.3M** | 1.3M |
| HILBERT+ Goedel-Prover-V2-32B | ✗ | 97.9% | 862 | 449 | 4.0M | **1.2M** |

Table 3: **Ablation with/without retrieval.** HILBERT with retrieval achieves a higher pass rate while using less inference-time compute than without retrieval. Numbers show average calls and tokens per sample, computed over samples requiring subgoal decomposition.

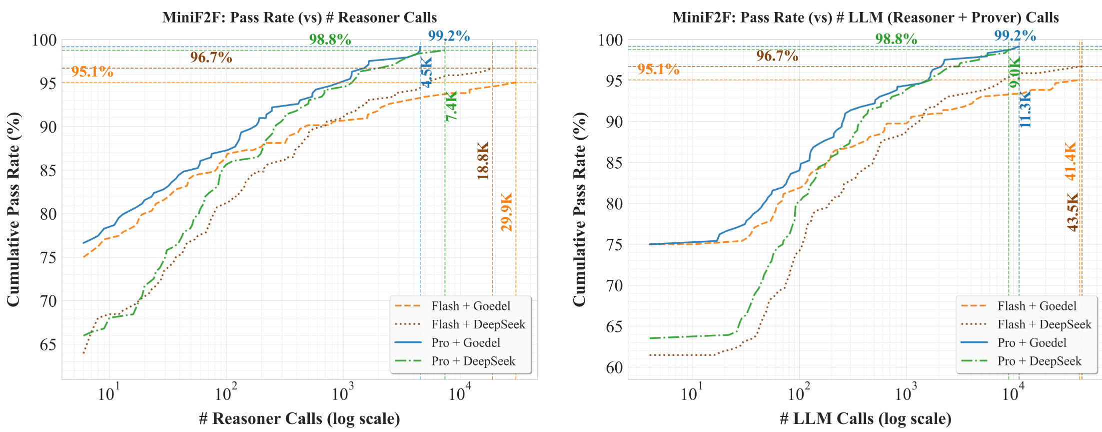


Figure 3: **Pass rate (vs) Inference-time Budget.** We plot the pass-rate for HILBERT on MiniF2F as a function of (left) the number of Reasoner calls (right) the total number of LLM (Reasoner + Prover) calls per sample.

problems across Algebra, Analysis, Number Theory, Geometry, Linear Algebra, Combinatorics, Abstract Algebra, Probability, and Set Theory. Given the high computational cost of evaluating on this dataset, we only experiment with the strongest configuration of HILBERT, (HILBERT with Gemini 2.5 Pro and Goedel-Prover-V2-32B). As before, we set $D = 5$. Our results are presented in Table 2.

HILBERT achieves state-of-the-art performance on PutnamBench, solving 462 out of 660 problems (70.0% pass rate). This surpasses the previous best method, the proprietary SeedProver (50.4%), by nearly 20 percentage points. HILBERT solves over 5 times more problems than the closest publicly available baseline, Goedel-Prover-V2-32B. We attribute this success to HILBERT's ability to compose long proofs (see Figure 9) without the long-context reasoning issues that plague traditional LLMs (Zhou et al., 2025a).

### 4.2 Scaling Behavior with Inference-Time Compute

Unlike traditional prover LLMs that distribute compute across many independent proof attempts from scratch, HILBERT allocates inference-time compute across multiple interconnected stages, from subgoal decomposition to subgoal proof generation. Since this compute allocation is adaptive, it cannot be captured by a simple count of independent attempts. To illustrate the compute-performance tradeoff, we plot HILBERT's pass rate against the per-sample number of calls to (1) the Reasoner and (2) the Reasoner + Prover combined (Figure 3. The results reveal a clear scaling relationship where pass rates increase with the number of calls per sample. Our best-performing configuration (Gemini 2.5 Pro with Goedel Prover) requires at most 4.5K reasoner calls and 11.3K total calls, significantly fewer than DeltaProver's 16,384 calls with Gemini 2.5 Pro. Interestingly, the weaker reasoner (Gemini 2.5 Flash) demands a substantially higher inference budget to achieve comparable performance with both prover variants. While HILBERT+ DeepSeek Prover starts with lower pass rates, it demonstrates faster improvement rates, particularly in low-budget settings, eventually matching HILBERT+Goedel-Prover performance. For additional analyses of pass rates versus prover/verifier calls and total token usage, refer to Section A.6.

### 4.3 Ablation Studies

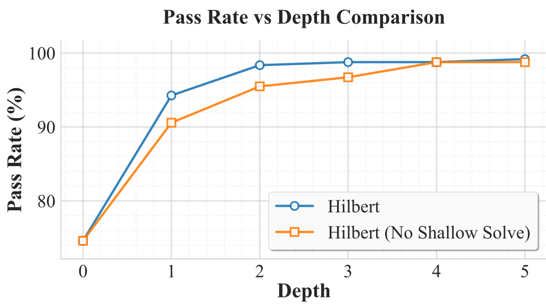


Figure 4: Pass rate (vs) recursive depth $D$ on MiniF2F for HILBERT (Gemini 2.5 Pro) + Goedel-Prover-V2-32B

**Performance (vs) depth.** To evaluate the effectiveness of subgoal decomposition, we analyze the pass rate of HILBERT using Gemini 2.5 Pro + Goedel-Prover-V2-32B on the MiniF2F dataset across different recursive depths $D$. The baseline ($D = 0$) corresponds to no decomposition, where we report the standalone Prover (pass@4) performance. We compare two configurations: the full HILBERT system, and a variant with shallow solving disabled ($K_{\text{informal passes}} = 0$). This variant relies solely on using the Prover for resolving subgoals. Figure 4 shows performance across different values of $D$, and demonstrates substantial gains from subgoal decomposition. Both configurations show monotonically increasing performance with depth, but exhibit different convergence patterns. The full HILBERT system achieves rapid performance gains, reaching 98.36% at $D = 2$ and 98.7% by $D = 3$. In contrast, the no-shallow-solve variant requires greater depth to achieve comparable performance, highlighting the importance of the shallow solving mechanism. The consistent improvement over the $D = 0$ baseline (75% pass rate) validates the efficacy of hierarchical subgoal decomposition, with the full system achieving near-optimal performance at relatively shallow depths.

**Retrieval Ablation.** To assess the impact of the Retriever on both performance and computational efficiency, we compare HILBERT to a variant that omits the retrieval step. We experiment on MiniF2F across two Prover configurations: DeepSeek-Prover-V2-7B and Goedel-Prover-V2-32B. Table 3 presents the results. With retrieval enabled, HILBERT achieves higher pass rates across both configurations: 98.4% vs 97.1% for DeepSeek Prover and 99.2% vs 97.9% for Goedel Prover. More importantly, retrieval significantly reduces inference-time compute utilitzation. For the DeepSeek model, retrieval decreases reasoner calls from 426 to 420, average prover calls from 290 to 205, and average reasoner tokens from 2.1M to 1.9M. The efficiency gains are even more pronounced with the Goedel Prover, where retrieval reduces average reasoner calls from 862 to 548 and average reasoner tokens from 4.0M to 2.3M. These results show that retrieval improves both performance and efficiency by surfacing useful theorems that simplify proofs and preventing failures from incorrect theorem names.

## 5 Conclusion

We present HILBERT, a hierarchical agentic framework that bridges formal theorem proving in Lean with the informal mathematical reasoning capabilities of general-purpose LLMs. Our approach recursively decomposes complex problems into manageable subgoals and orchestrates informal reasoners (Gemini 2.5 Pro/Flash) with formal provers (DeepSeek-Prover-V2-7B and Goedel-Prover-V2-32B) to solve theorems that neither component can handle alone. HILBERT achieves state-of-the-art performance on miniF2F with pass rates of 94.7% to 99.2%. On the challenging PutnamBench dataset, HILBERT achieves 70.0% pass rate, nearly 20 percentage points above previous methods and approaching the 82% informal proof rate reported in Dekoninck et al. (2025). In the future, we plan to leverage this framework to train increasingly capable models. Proofs and reasoning traces generated by HILBERT can be used to train better Prover and Reasoner models. These improved models should be able to solve more complex problems than before, resulting in a virtuous cycle that has the potential to continually advance formal reasoning capabilities.

**Reproducibility Statement**. We provide comprehensive implementation details to ensure reproducibility of our results. The proposed algorithm (HILBERT) is described in detail in Section 3 with complete pseudocode provided in Algorithm 1. All hyperparameters, model configurations, and experimental settings are specified in Section 3, while the complete set of prompts used for both reasoning and prover LLMs are provided in Appendix A.2. We plan to release the source code and other artifacts upon publication.

**LLM Usage.** We acknowledge using LLMs as writing assistants to help refine phrasing and improve the clarity of the presentation. LLMs were not used for any substantive aspects of this work, including ideation, conceptual development, or literature review.

## REFERENCES

Team AlphaProof and Team AlphaGeometry. Ai achieves silver-medal standard solving international 178 mathematical olympiad problems. *DeepMind blog*, 179:45, 2024.

Kaito Baba, Chaoran Liu, Shuhei Kurita, and Akiyoshi Sannai. Prover agent: An agent-based framework for formal mathematical proofs. *arXiv preprint arXiv:2506.19923*, 2025.

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions. *arXiv preprint arXiv:2505.23281*, 2025.

Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. Extending sledgehammer with smt solvers. *Journal of automated reasoning*, 51(1):109–128, 2013.

Chenrui Cao, Liangcheng Song, Zenan Li, Xinyi Le, Xian Zhang, Hui Xue, and Fan Yang. Reviving dsp for advanced theorem proving in the era of reasoning models. *arXiv preprint arXiv:2506.11487*, 2025.

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*, 2025.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of automated reasoning*, 61(1):423–453, 2018.

Jasper Dekoninck, Ivo Petrov, Kristian Minchev, Mislav Balunovic, Martin Vechev, Miroslav Marinov, Maria Drencheva, Lyuba Konova, Milen Shumanov, Kaloyan Tsvetkov, et al. The open proof corpus: A large-scale study of llm-generated mathematical proofs. *arXiv preprint arXiv:2506.21621*, 2025.

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

Kefan Dong, Arvind Mahankali, and Tengyu Ma. Formal theorem proving by rewarding llms to decompose proofs hierarchically. *arXiv preprint arXiv:2411.01829*, 2024.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

Guoxiong Gao, Haocheng Ju, Jiedong Jiang, Zihan Qin, and Bin Dong. A semantic search engine for mathlib4. *arXiv preprint arXiv:2403.13310*, 2024.

Elliot Glazer, Ege Erdil, Tamay Besiroglu, Diego Chicharro, Evan Chen, Alex Gunning, Caroline Falkman Olsson, Jean-Stanislas Denain, Anson Ho, Emily de Oliveira Santos, et al. Frontiermath: A benchmark for evaluating advanced mathematical reasoning in ai. *arXiv preprint arXiv:2411.04872*, 2024.

Fabian Gloeckle, Jannis Limperg, Gabriel Synnaeve, and Amaury Hayat. Abel: Sample efficient online reinforcement learning for neural theorem proving. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024.

Jiaxing Guo, Wenjie Yang, Shengzhong Zhang, Tongshan Xu, Lun Du, Da Zheng, and Zengfeng Huang. Right is not enough: The pitfalls of outcome supervision in training llms for math reasoning. *arXiv preprint arXiv:2506.06877*, 2025.

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.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Zhenwen Liang, Linfeng Song, Yang Li, Tao Yang, Feng Zhang, Haitao Mi, and Dong Yu. Towards solving more challenging imo problems via decoupled reasoning and proving. *arXiv preprint arXiv:2507.06804*, 2025.

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

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. *arXiv preprint arXiv:2508.03613*, 2025b.

Jialin Lu, Kye Emond, Weiran Sun, and Wuyang Chen. Lean finder: Semantic search for mathlib that understands user intents. In *2nd AI for Math Workshop @ ICML 2025*, 2025. URL `https://openreview.net/forum?id=5SF4fFRw7u`.

Hamed Mahdavi, Alireza Hashemi, Majid Daliri, Pegah Mohammadipour, Alireza Farhadi, Samira Malek, Yekta Yazdanifard, Amir Khasahmadi, and Vasant Honavar. Brains vs. bytes: Evaluating llm proficiency in olympiad mathematics. *arXiv preprint arXiv:2504.01995*, 2025.

The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20, pp. 367–381. ACM, January 2020. doi: 10.1145/3372885.3373824. URL `http://dx.doi.org/10.1145/3372885.3373824`.

William McCune. Otter 3.3 reference manual. *arXiv preprint cs/0310056*, 2003.

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.

OpenAI. Introducing gpt-5. 2025. URL `https://openai.com/index/introducing-gpt-5/`.

Ivo Petrov, Jasper Dekoninck, Lyuben Baltadzhiev, Maria Drencheva, Kristian Minchev, Mislav Balunović, Nikola Jovanović, and Martin Vechev. Proof or bluff? evaluating llms on 2025 usa math olympiad. *arXiv preprint arXiv:2503.21934*, 2025.

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

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.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965. ISSN 0004-5411. doi: 10.1145/321250.321253. URL `https://doi.org/10.1145/321250.321253`.

Marco Dos Santos, Haiming Wang, Hugues de Saxcé, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. Kimina lean server: Technical report. *arXiv preprint arXiv:2504.21230*, 2025.

Stephan Schulz. E–a brainiac theorem prover. *Ai Communications*, 15(2-3):111–126, 2002.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding. *Advances in neural information processing systems*, 33:16857–16867, 2020.

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

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.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Lin Li, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, et al. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*, 2023.

Haiming Wang, Huajian Xin, Zhengying Liu, Wenda Li, Yinya Huang, Jianqiao Lu, Zhicheng Yang, Jing Tang, Jian Yin, Zhenguo Li, et al. Proving theorems recursively. *Advances in Neural Information Processing Systems*, 37:86720–86748, 2024.

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.

Nicolas Wischermann, Claudio Mayrink Verdun, Gabriel Poesia, and Francesco Noseda. Proofcompass: Enhancing specialized provers with llm guidance. In *2nd AI for Math Workshop@ ICML 2025*, 2025.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024a.

Huajian Xin, ZZ Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, et al. Deepseek-prover-v1. 5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024b.

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Kai Shen. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving. *arXiv preprint arXiv:2502.03438*, 2025.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*, 2024.

Xueliang Zhao, Wenda Li, and Lingpeng Kong. Decomposing the enigma: Subgoal-based demonstration learning for formal theorem proving. *arXiv preprint arXiv:2305.16366*, 2023.

Xueliang Zhao, Lin Zheng, Haige Bo, Changran Hu, Urmish Thakker, and Lingpeng Kong. Subgoalxl: Subgoal-based expert learning for theorem proving. *arXiv preprint arXiv:2408.11172*, 2024.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

Yang Zhou, Hongyi Liu, Zhuoming Chen, Yuandong Tian, and Beidi Chen. Gsm-infinite: How do your llms behave over infinitely increasing context length and reasoning complexity?, 2025a. URL `https://arxiv.org/abs/2502.05252`.

Yichi Zhou, Jianqiu Zhao, Yongxin Zhang, Bohan Wang, Siran Wang, Luoxin Chen, Jiahui Wang, Haowei Chen, Allan Jie, Xinbo Zhang, et al. Solving formal math problems by decomposition and iterative reflection. *arXiv preprint arXiv:2507.15225*, 2025b.

Matthieu Zimmer, Xiaotong Ji, Rasul Tutunov, Anthony Bordg, Jun Wang, and Haitham Bou Ammar. Bourbaki: Self-generated and goal-conditioned mdps for theorem proving. *arXiv preprint arXiv:2507.02726*, 2025.

# A Appendix

## A.1 Algorithm

The complete algorithm is presented across multiple blocks for clarity and modularity. Algorithm 1 provides the main entry point and high-level control flow, while Algorithm 2 details the subgoal resolution strategies. Algorithms 3 and 4 focus on sketch generation, validation, and assembly processes. Algorithm 5 contains the core proof generation functions that interface with different LLM components, while Algorithm 6 specifies the prompt-based functions for various reasoning tasks. Algorithm 7 handles error correction and refinement procedures, and Algorithm 8 provides supporting functions for theorem retrieval and verification.

**Algorithm 1** HILBERT: Hierarchical Proof Generation System

```
function GENERATEPROOF(problem, header)
        ▷ Input: problem (formal statement), header (context)

    ▷ Phase 1: Direct Proof Attempt
    proof ← ATTEMPTPROVERLLMPROOF(problem, header)
    if proof ≠ ⊥ then
        return proof
    end if

    ▷ Phase 2: Subgoal Decomposition
    proof ← SUBGOALDECOMPOSITION(problem, header, depth=1)
    return proof
end function

function SUBGOALDECOMPOSITION(problem, header, depth)
        ▷ Decompose problem into subgoals and solve recursively
    if depth > D then
        return ⊥    ▷ Maximum recursion depth reached
    end if

    for attempt ← 1 to K_sketch attempts do
        relevant_theorems ← RETRIEVETHEOREMS(problem)
        sketch ← GENERATEPROOFSKETCH(problem, relevant_theorems)
        sketch_assembled, subgoals, proved_subgoals ←
      REFINEANDVALIDATESKETCH(sketch, header, relevant_theorems)

        if sketch_assembled ≠ ⊥ then
            final_proof ← SOLVEALLSUBGOALS(subgoals, proved_subgoals,
sketch_assembled, header, depth)
            if final_proof ≠ ⊥ then
                return final_proof
            end if
        end if
    end for
    return ⊥
end function
```

**Algorithm 2** HILBERT: Subgoal Resolution

```
function SOLVEALLSUBGOALS(subgoals, proved_subgoals, sketch_assembled, header, depth)
        ▷ Solve all remaining subgoals and assemble final proof
    subgoal_proofs ← ∅

    for all subgoal ∈ subgoals \ proved_subgoals do
        proof ← SOLVESUBGOAL(subgoal, header, depth)
        if proof = ⊥ then
            return ⊥    ▷ Failed to prove required subgoal
        end if
        subgoal_proofs[subgoal] ← proof
    end for

    final_proof ← CONCATENATE(header, subgoal_proofs, sketch)
    return final_proof
end function

function SOLVESUBGOAL(subgoal, header, depth)
        ▷ Solve individual subgoal with multiple strategies

    Strategy 1: Direct Prover Attempt
    proof ← ATTEMPTPROVERLLMPROOF(subgoal, header)
    if proof ≠ ⊥ then
        return proof
    end if

    Strategy 2: Shallow Solve with Reasoner
    relevant_theorems ← RETRIEVETHEOREMS(subgoal)
    proof ← SHALLOWSOLVE(subgoal, header, relevant_theorems)
    if proof ≠ ⊥ then
        return proof
    end if

    Strategy 3: Recursive Decomposition
    if depth < D then
        proof ← SUBGOALDECOMPOSITION(subgoal, header, depth + 1)
        if proof ≠ ⊥ then
            return proof
        end if
    end if
    return ⊥
end function
```

**Algorithm 3** HILBERT: Sketch Validation and Refinement

```
function REFINEANDVALIDATESKETCH(sketch, header, relevant_theorems)
        ▷ Iteratively refine sketch until all subgoals are valid
    for correction ← 1 to K_{sketch corrections} do
        sketch_syntactic ← COMPILEANDCORRECTSYNTAXERRORS(sketch, header, relevant_theorems)
        if sketch_syntactic == ⊥ then
            return ⊥, ∅, ∅
        end if
        subgoals ← EXTRACTSUBGOALS(sketch_syntactic, header)
        if subgoals == ⊥ then
            return ⊥, ∅, ∅
        end if
        sketch_assembled ← ASSEMBLEPROOFFROMSUBGOALS(sketch_syntactic, subgoals, header)
        if sketch_assembled == ⊥ then
            return ⊥, ∅, ∅
        end if
        valid, verified_subgoals, proved_subgoals, error_justification ← VALIDATESUBGOALS(subgoals, header)
        if valid then
            return sketch_assembled, verified_subgoals, proved_subgoals
        else
            sketch ← REFINESKETCHBASEDONERROR(sketch_syntactic, error_justification)
        end if
    end for
    return ⊥, ∅, ∅
end function

function VALIDATESUBGOALS(subgoals, header)
        ▷ Validate subgoals through formal proving and correctness checking
    verified_subgoals ← ∅
    proved_subgoals ← {}

    for all subgoal ∈ subgoals do
        proof ← ATTEMPTPROVERLLMPROOF(subgoal, header)
        if proof ≠ ⊥ then
            verified_subgoals ← verified_subgoals ∪ {subgoal}
            proved_subgoals[subgoal] ← proof
        else
            mathematically_correct, justification ← CHECKMATHEMATICALCORRECTNESS(subgoal)
            if mathematically_correct then
                verified_subgoals ← verified_subgoals ∪ {subgoal}
            else
                return false, ∅, ∅, justification
            end if
        end if
    end for
    return true, verified_subgoals, proved_subgoals, ⊥
end function
```

**Algorithm 4** HILBERT: Proof Sketch Refinement and Assembly

```
function COMPILEANDCORRECTSYNTAXERRORS(sketch, header, relevant_theorems)
        ▷ Compile sketch with sorry statements and correct errors
    verified, error_message ← VERIFYPROOF(header + sketch)
    if verified then
        return sketch
    end if

        ▷ Error correction loop for sketch
    for correction ← 1 to K_theorem corrections do
        augmented_theorems ← AUGMENTTHEOREMS(error_message, relevant_theorems)
        sketch ← CORRECTSKETCHERROR(sketch, error_message, augmented_theorems)
        verified, error_message ← VERIFYPROOF(header + sketch)
        if verified then
            return sketch
        end if
    end for
    return ⊥
end function

function ASSEMBLEPROOFFROMSUBGOALS(sketch, subgoals, header)
        ▷ Assemble complete proof outline with verification
    all_theorems ← CONCATENATETHEOREMS(subgoals)
    sketch_assembled ← REASONERLLM(USE_SKETCH_AND_THEOREMS_PROMPT, sketch,
all_theorems)
    corrected_proof ← VERIFYANDCORRECTPROOFWITHTHEOREMS(sketch_assembled,
all_theorems, header)
    return corrected_proof
end function

function VERIFYANDCORRECTPROOFWITHTHEOREMS(sketch_assembled, theorems, header)
        ▷ Verify assembled sketch and correct errors
    full_proof ← header + theorems + sketch_assembled
    verified, error ← VERIFYPROOF(full_proof)
    if verified then
        return sketch_assembled
    end if

    for correction ← 1 to K_theorem corrections do
        corrected_proof ← REASONERLLM(ASSEMBLY_CORRECTION_PROMPT, error)
        if sketch_assembled == ⊥ then
            continue
        end if
        full_proof ← header + theorems + sketch_assembled
        verified, error ← VERIFYPROOF(full_proof)
        if verified then
            return sketch_assembled
        end if
    end for
    return ⊥
end function
```

**Algorithm 5** HILBERT: Proof Generation

```
function ATTEMPTPROVERLLMPROOF(problem, header)
        ▷ Multiple attempts with formal prover LLM
    for attempt ← 1 to K_formal attempts do
        proof ← PROVERLLM(problem)
        verified, error ← VERIFYPROOF(header + proof)
        if verified then
            return proof
        end if
    end for
    return ⊥
end function

function GENERATEPROOFSKETCH(problem, relevant_theorems)
        ▷ Generate informal proof sketch using prompts
    informal_proof ← REASONERLLM(INFORMAL_PROOF_PROMPT, problem, relevant_theorems)
    sketch ← REASONERLLM(CREATE_LEAN_SKETCH_PROMPT, problem, relevant_theorems, informal_proof)
    return sketch
end function

function SHALLOWSOLVE(subgoal, header, relevant_theorems)
        ▷ Shallow solve with error correction loop
    proof ← ATTEMPTREASONERPROOF(subgoal, relevant_theorems)
    verified, error_message ← VERIFYPROOF(header + proof)
    if verified then
        return proof
    end if

        ▷ Error correction loop
    for correction ← 1 to K_subgoal corrections do
        augmented_theorems ← AUGMENTTHEOREMS(error_message, relevant_theorems)
        proof ← CORRECTPROOFERROR(proof, error_message, augmented_theorems)
        verified, error_message ← VERIFYPROOF(header + proof)
        if verified then
            return proof
        else
                ▷ Check proof length cutoff when verification fails
            if |proof| > K_max shallow solve length then
                return ⊥    ▷ Proof too long and still incorrect, abandon
            end if
        end if
    end for
    return ⊥
end function
```

**Algorithm 6** HILBERT: LLM Prompt Functions

```
function AttemptReasonerProof(subgoal, relevant_theorems)
    ▷ Shallow solve using informal reasoning
    proof ← ReasonerLLM(SOLVE_SUBGOAL_PROMPT, subgoal, relevant_theorems)
    return proof
end function

function CheckMathematicalCorrectness(subgoal)
    ▷ Verify mathematical correctness of subgoal
    correct, justification ← ReasonerLLM(DETERMINE_IF_CORRECT_SUBGOAL_PROMPT,
subgoal)
    return correct, justification
end function

function ExtractSubgoals(sketch, header)
    ▷ Extract have statements as independent subgoals
    subgoals ← ReasonerLLM(EXTRACT_SUBGOALS_FROM_SKETCH_PROMPT, sketch)

    ▷ Syntax check and correction for each subgoal
    corrected_subgoals ← ∅
    for all subgoal ∈ subgoals do
        verified, error ← VerifyProof(header + subgoal)
        if verified then
            corrected_subgoals ← corrected_subgoals ∪ {subgoal}
        else
            ▷ Error correction loop
            corrected ← false
            for attempt ← 1 to K_subgoal error corrections do
                subgoal ← CorrectTheoremError(subgoal, error)
                verified, error ← VerifyProof(header + subgoal)
                if verified then
                    corrected_subgoals ← corrected_subgoals ∪ {subgoal}
                    corrected ← true
                    break    ▷ Successfully corrected
                end if
            end for
            if ¬corrected then
                return ⊥    ▷ Failed to correct subgoal, return failure
            end if
        end if
    end for

    return corrected_subgoals
end function
```

**Algorithm 7** HILBERT: Error Correction

```
function REFINESKETCHBASEDONERROR(sketch, error_justification)
        ▷ Refine proof sketch based on subgoal validation errors
    refined ← REASONERLLM(CORRECT_SKETCH_BASED_ON_INCORRECT_SUBGOAL_PROMPT, sketch, error_justification)
    return refined
end function

function CORRECTSKETCHERROR(sketch, error_message, relevant_theorems)
        ▷ Correct syntax and compilation errors
    corrected ← REASONERLLM(PROOF_SKETCH_CORRECTION_PROMPT, error_message, sketch, relevant_theorems)
    return corrected
end function

function CORRECTPROOFERROR(proof, error_message, augmented_theorems)
        ▷ Correct proof errors using error feedback
    corrected ← REASONERLLM(PROOF_CORRECTION_PROMPT, error_message, proof, augmented_theorems)
    return corrected
end function

function CORRECTTHEOREMERROR(subgoal, error_message)
        ▷ Correct syntax errors in extracted subgoals
    corrected ← REASONERLLM(SUBGOAL_SYNTAX_CORRECTION_PROMPT, error_message, subgoal)
    return corrected
end function
```

**Algorithm 8** HILBERT: Retrieval and Helper Functions

```
function RETRIEVETHEOREMS(problem, error_message = None)
        ▷ Theorem retrieval from Mathlib with optional parameter for error message
    if retrieval_enabled then
        search_queries ← GENERATESEARCHQUERIES(problem, error_message)
        candidate_theorems ← SEMANTICSEARCHENGINE(search_queries)
        relevant_theorems ← SELECTRELEVANTTHEOREMS(candidate_theorems, problem)
        return relevant_theorems
    else
        return ∅
    end if
end function

function GENERATESEARCHQUERIES(problem)
        ▷ Generate search queries for theorem retrieval
    queries ← REASONERLLM(SEARCH_QUERY_PROMPT, problem)
    return queries
end function

function SELECTRELEVANTTHEOREMS(candidate_theorems, problem)
        ▷ Select most relevant theorems from candidates
    selected ← REASONERLLM(SEARCH_ANSWER_PROMPT, problem, candidate_theorems)
    return selected
end function

function VERIFYPROOF(full_proof)
        ▷ Verify proof using Lean verifier
    result, error_message ← LEANVERIFIER(full_proof)
    return result, error_message
end function

function AUGMENTTHEOREMS(error_message, existing_theorems)
        ▷ Add theorems for missing identifiers
    missing_ids ← EXTRACTMISSINGIDENTIFIERS(error_message)
    if missing_ids ≠ ∅ then
        additional_theorems ← RETRIEVETHEOREMS(problem, error_message)
        return existing_theorems + additional_theorems
    end if
    return existing_theorems
end function
```

### A.2 PROMPTS

**Search Query Generation (`SEARCH_QUERY_PROMPT`)**

```
You are helping solve a Lean theorem proving problem using the mathlib library.
Before attempting to write the proof, you must first search for relevant theorems and tactics.

Search Process:
1. Identify key concepts: Break down the problem into mathematical concepts, operations, and
↪  structures involved.
2. Generate search queries: For each concept, create informal search strings that describe:
   - Relevant theorems or results (e.g., "associativity of addition", "existence of inverse
   ↪  elements")
   - Useful tactics (e.g., "simplify arithmetic expressions", "split conjunctions")
   - Properties (e.g., "group structure on integers", "metric space properties")
   - Relevant definitions useful for the proof or any used theorem (e.g. "definition of a group",
   ↪  "definition of a metric space")

Search Query Format:
Enclose each search query in <search> tags with your informal description. Limit yourself to a
↪  maximum of 5 search queries. Make the search queries simple, concise, and clear.

Guidelines:
- You can either search by theorem name or natural language description
- Search for theorems that might automate parts of the proof
- Consider edge cases and special conditions mentioned in the problem

Problem to Solve:
{problem}
```

**Theorem Selection (`SEARCH_ANSWER_PROMPT`)**

```
You are helping to solve a Lean theorem proving problem using the mathlib library. The problem is:
{problem}

Here are some potentially relevant theorems and definitions:
{theorems}

Instructions:
1. Select important theorems and definitions necessary to solve the problem.
2. IMPORTANT: ONLY SELECT theorems from the GIVEN list.
3. Enclose each of them in separate <theorem> tags.
4. Only state the full names of the theorems. Do NOT include the module name.
5. Select all theorems that could be useful in the intermediate steps of the proof.
```

**Informal Proof Generation (`INFORMAL_PROOF_PROMPT`)**

```
You are a mathematical expert whose goal is to solve problems with rigorous
mathematical reasoning.

{useful_theorems_section}
Instructions:
1. Provide a natural language, step-by-step proof for the given problem.
2. Start from the given premises and reason step-by-step to reach the conclusion.
3. Number each step of the proof as 1, 2, and so on.
4. Be as pedantic and thorough as possible.
5. Keep each step precise, increase the number of steps if needed.
6. Do NOT gloss over any step. Make sure to be as thorough as possible.
7. Show the explicit calculations/simplifications, theorem applications and case
   analysis.
8. Enclose the informal proof in <informal_proof> tags.

Problem Statement: {problem}
```

### Lean Sketch Creation (`CREATE_LEAN_SKETCH_PROMPT`)

```
You are a Lean 4 expert who is trying to help write a proof in Lean 4.

Problem Statement: {problem}

{useful_theorems_section}
Informal Proof:
{informal_proof}

Instructions:

Use the informal proof to write a proof sketch for the problem in Lean 4 following
these guidelines:
- Break complex reasoning into logical sub-goals using `have` statements.
- The subgoals should build up to prove the main theorem.
- Make sure to include all the steps and calculations from the given proof in the
  proof sketch.
- Each subgoal should ideally require applying just one key theorem or lemma, or a
  few tactic applications.
- Base subgoals around:
  - Useful theorems mentioned in the problem context
  - Standard library theorems (like arithmetic properties, set operations, etc.)
  - The supplied premises in the theorem statement
- Do NOT create subgoals identical to any of the given hypotheses
- Do NOT create subgoals that are more complex than the original problems. The
  subgoals should be SIMPLER than the given problem.
- Do NOT skip over any steps. Do NOT make any mathematical leaps.

**Subgoal Structure Requirements:**
- **Simplicity**: Each subgoal proof should be achievable with 1-3 basic tactics
- **Atomic reasoning**: Avoid combining multiple logical steps in one subgoal
- **Clear progression**: Show logical flow: `premises → intermediate steps → final result`
- **Theorem-focused**: Design each subgoal to directly apply a specific theorem when possible

NOTE: Only add sub-goals that simplify the proof of the main goal.

When writing Lean proofs, maintain consistent indentation levels.

Rules:
1. Same proof level = same indentation: All tactics at the same logical level must
   use identical indentation
2. Consistent characters: Use either tabs OR spaces consistently (don't mix)
3. Proper nesting: Indent sub-proofs one level deeper than their parent
4. Do NOT nest `have` statements in each other. Use distinct sub-goals as much as
   possible. Ensure all sub goals are named. Do NOT create anonymous have statements.
5. Do NOT include any imports or open statements in your code.
6. One line = One `have` subgoal. Do NOT split subgoals across different lines.
7. Use proper Lean 4 syntax and conventions. Ensure the proof sketch is enclosed in
   triple backticks ```lean```
8. Use `sorry` for all subgoal proofs - focus on structure, not implementation
9. **Do NOT use `sorry` for the main goal proof** - use your subgoals to prove it
10. NEVER use `sorry` IN the theorem statement itself
11. Ensure subgoals collectively provide everything needed for the main proof
12. Make the logical dependencies between subgoals explicit. Ensure that the subgoals
    are valid and provable in Lean 4.
13. Do NOT change anything in the original theorem statement.

Lean Hints:
{lean_hints}
```

### Lean Sketch Creation (`CREATE_LEAN_SKETCH_PROMPT`) (continued)

```
IMPORTANT INSTRUCTION: Do NOT, under ANY circumstances, allow division and
subtraction operations on natural number literals with UNDEFINED types, unless
REQUIRED by the theorem statement. For example, do NOT allow literals like `1 / 3`
or `2 / 5` or `1 - 3` ANYWHERE in ANY of the subgoals. ALWAYS specify the types.
AVOID natural number arithmetic UNLESS NEEDED by the theorem statement.
ALWAYS specify types when describing fractions. For example, ((2 : ℝ) / 3) or
((2 : ℚ) / 3) instead of (2 / 3). Do this everywhere EXCEPT the given theorem statement.
IMPORTANT INSTRUCTION: Do NOT, under ANY circumstances, allow division and
subtraction operations on variables of type natural numbers (Nat or ℕ), unless
REQUIRED by the theorem statement. For example, do NOT allow expressions like (a-b)
or (a/b) where a, b are of type ℕ. ALWAYS cast the variables to a suitable type
(ℤ, ℚ or ℝ) when performing arithmetic operations. AVOID natural number arithmetic
UNLESS NEEDED by the theorem statement.
```

### Subgoal Extraction (`EXTRACT_SUBGOALS_FROM_SKETCH_PROMPT`)

````
From this proof sketch, extract any missing proofs (specified with `sorry`) as
independent subgoals (theorems).
Instructions:
1. Use the same name as the have statements for the theorems.
2. Each subgoal should have the relevant context from the previous subgoals needed
   to simplify the proof as much as possible.
3. There should be as many extracted theorems as `sorry`s in the given theorem.
4. Do NOT include any imports or open statements. Do NOT add any definitions. ONLY
   include the theorem statement.
5. Use a separate Lean 4 ``lean`` block for each subgoal.
6. Use sorry for the proof. Do NOT prove any theorem.
7. Do NOT change the conclusion of the theorems from the extracted subgoals. Keep
   them AS IT IS.
8. Do NOT change the conclusions of the preceding theorems when presenting them as
   hypotheses for the next subgoals. Keep them AS IT IS.
9. Do NOT duplicate theorem names. Use distinct theorem names for the different theorems.
10. Make sure the names and types of the premises/arguments in the extracted theorems
    MATCH the subgoals from which they are extracted.

IMPORTANT INSTRUCTION: Do NOT, under ANY circumstances, allow division and
subtraction operations on natural number literals with UNDEFINED types, unless
REQUIRED by the theorem statement. For example, do NOT allow literals like `1 / 3`
or `2 / 5` or `1 - 3` ANYWHERE in the theorem statement. ALWAYS specify the types.
AVOID natural number arithmetic UNLESS NEEDED by the theorem statement.
ALWAYS specify types when describing fractions. For example, ((2 : ℝ) / 3) or
((2 : ℚ) / 3) instead of (2 / 3)
IMPORTANT INSTRUCTION: Do NOT, under ANY circumstances, allow division and
subtraction operations on variables of type natural numbers (Nat or ℕ), unless
REQUIRED by the theorem statement. For example, do NOT allow expressions like (a-b)
or (a/b) where a, b are of type ℕ. ALWAYS cast the variables to a suitable type
(ℤ, ℚ or ℝ) when performing arithmetic operations. AVOID natural number arithmetic
UNLESS NEEDED by the theorem statement.

Lean Hints:
{lean_hints}

Proof Sketch:
```lean4
{proof_sketch}
```
````

### Subgoal Solving (SOLVE_SUBGOAL_PROMPT)

```
Think step-by-step to complete the following Lean 4 proof.

{problem}

Lean Hints:
{lean_hints}

Tactic Hints:
{tactic_hints}

Rules:
1. Same proof level = same indentation: All tactics at the same logical level must
   use identical indentation
2. Consistent characters: Use either tabs OR spaces consistently (don't mix)
3. Proper nesting: Indent sub-proofs one level deeper than their parent
4. Do NOT include any imports or open statements.
5. Use proper Lean 4 syntax and conventions. Ensure the proof sketch is enclosed in
   triple backticks ```lean```.
6. Only include a single Lean 4 code block, corresponding to the proof along with
   the theorem statement.
7. When dealing with large numerical quantities, avoid explicit computation as much
   as possible. Use tactics like rw to perform symbolic manipulation rather than
   numerical computation.
8. Do NOT use sorry.
9. Do NOT change anything in the original theorem statement.
{useful_theorems_section}
```

### Mathematical Correctness Check (DETERMINE_IF_CORRECT_SUBGOAL_PROMPT)

```
You are an expert in mathematics.

Your task is to evaluate whether the given mathematical theorem statement is
mathematically correct. You do NOT have to provide a proof for the theorem in Lean.

Evaluation criteria:
1. Mathematical validity: Check for logical errors, incorrect assumptions, or
   calculation mistakes.
2. Do NOT flag general results or helper lemmas that are true independent of the
   given premises. ONLY flag inaccuracies or mistakes.
5. Provability: Determine if the statement can be proven given the provided premises,
   or otherwise.

Assumptions:
1. The given premises are mathematically correct. Do NOT check this.
2. The syntax is guaranteed to be correct (do not assess syntax)

Theorem Statement:
{problem}

Report your answer as either:
• YES - if the statement is mathematically correct
• NO - if the statement has mathematical errors that prevent proof

Also provide a brief justification for your decision in <justification></justification>
tags, adding details about why the statement is correct or incorrect.
If it is incorrect, also provide a description of how the error can be corrected.
If there are missing arguments, make sure to add the relevant missing proof steps.
```

### Sketch Assembly (USE_SKETCH_AND_THEOREMS_PROMPT)

```
You are a Lean 4 expert. Your goal is to write a proof in Lean 4, according to the
given proof sketch, using the supplied theorems.

Proof sketch:
{proof_sketch}

Theorems:
{theorems_string}

Instructions:
1. You can assume that the theorems are correct and use them directly in your proof.
2. Do NOT modify the given theorems.
3. Do NOT prove the given theorems.
4. Do NOT modify the given proof sketch steps. Simply apply the given theorems to
   complete the missing `sorry` steps.
5. Do NOT use `sorry` in your proof.
6. Do NOT include any imports or definitions or open statements.
7. Do NOT re-define the given theorems in your response.
8. Do NOT write a proof for any subgoal from scratch. ALWAYS use the supplied theorems.

IMPORTANT INSTRUCTION: Do NOT, under ANY circumstances, allow division and
subtraction operations on natural number literals with UNDEFINED types, unless
REQUIRED by the theorem statement. For example, do NOT allow literals like `1 / 3`
or `2 / 5` or `1 - 3`. ALWAYS specify the types. AVOID natural number arithmetic
UNLESS NEEDED by the theorem statement.
ALWAYS specify types when describing fractions. For example, ((2 : ℝ) / 3) or
((2 : ℚ) / 3) instead of (2 / 3). Do this everywhere EXCEPT the given theorem statement.
IMPORTANT INSTRUCTION: Do NOT, under ANY circumstances, allow division and
subtraction operations on variables of type natural numbers (Nat or ℕ), unless
REQUIRED by the theorem statement. For example, do NOT allow expressions like (a-b)
or (a/b) where a, b are of type ℕ. ALWAYS cast the variables to a suitable type
(ℤ, ℚ or ℝ) when performing arithmetic operations. AVOID natural number arithmetic
UNLESS NEEDED by the theorem statement.

Your answer should be a single Lean 4 block containing the completed proof for the
given theorem.
```

### Assembly Correction (ASSEMBLY_CORRECTION_PROMPT)

```
The following Lean 4 code has compilation errors. Please fix the errors while
maintaining the mathematical meaning.

{error_message}

Lean Hints:
{lean_hints}

Instructions:
1. Analyze what the theorem is trying to prove. Then, analyze why the error is
   happening, step-by-step. Add a brief explanation.
2. Then, provide a corrected version of the Lean 4 code that addresses these
   specific errors.
3. You should ONLY correct the main theorem that appears at the end. Do NOT
   change any of the helper theorems.
3. Do NOT include any other Lean code blocks except for the proof. Do NOT
   include any imports or open statements.
4. Do NOT use `sorry` in any part of the proof.
5. Do NOT change anything in the original theorem statement.
6. Do NOT include the helper theorem definitions in your response.
7. Do NOT write a proof for any subgoal from scratch. ALWAYS use the supplied
   theorems.
```

**Sketch Refinement Based on Incorrect Subgoal (CORRECT_SKETCH_BASED_ON_INCORRECT_SUBGOAL_PROMPT)**

```
You are an expert in writing Lean 4 proofs. You are given a Lean 4 proof sketch
where one of the subgoals has some issues.
Your task is to fix the issues and write a new proof sketch.

Proof Sketch:
{proof_sketch}

Issues:
{issues}

Lean Hints:
{lean_hints}

Rules:
1. Same proof level = same indentation: All tactics at the same logical level
   must use identical indentation
2. Consistent characters: Use either tabs OR spaces consistently (don't mix)
3. Proper nesting: Indent sub-proofs one level deeper than their parent
4. Do NOT nest `have` statements in each other. Write different have statements
   for different sub goals.
5. Ensure all sub goals are named. Do NOT create anonymous have statements.
6. Do NOT include any imports or open statements.
7. One line = One `have` subgoal. Do NOT split subgoals across different lines.
8. Use proper Lean 4 syntax and conventions. Ensure the proof sketch is enclosed
   in triple backticks ```lean```
9. Use `sorry` for all subgoal proofs - focus on structure, not implementation
10. **Do NOT use `sorry` for the main goal proof** - use your subgoals to prove it
11. NEVER use `sorry` IN the theorem statement itself
12. Ensure subgoals collectively provide everything needed for the main proof
13. Make the logical dependencies between subgoals explicit. Ensure that the
    subgoals are valid and provable in Lean 4.
14. Modify only the incorrect subgoal and everything that follows it in the proof
    sketch. Leave all preceding portions unchanged.
15. Either modify the problematic subgoals to fix the errors, or add additional
    subgoals to fill in the missing mathematical arguments.
```

### Proof Sketch Correction (PROOF_SKETCH_CORRECTION_PROMPT)

```
The following Lean 4 code has compilation errors. Please fix the errors while
maintaining the mathematical meaning.

Original statement: {informal_statement}

{error_message}

Lean Hints:
{lean_hints}

Instructions:
1. Analyze what the theorem is trying to prove. Then, analyze why the error is
   happening, step-by-step. Add a brief explanation.
2. Then, provide a corrected version of the Lean 4 code that addresses these
   specific errors.
3. Do NOT include any other Lean code blocks except for the proof. Do NOT
   include any imports or open statements.
4. Use sorry for the proof of all `have` statements.
5. Ensure there are no use of `sorry` statements outside of `have` statements.
   Do NOT use `sorry` while proving the main theorem.
6. Do NOT change anything in the original theorem statement.
7. Do NOT nest `have` statements in each other. Use distinct sub-goals as much
   as possible. Ensure all sub goals are named. Do NOT create anonymous have
   statements.

{useful_theorems_section}
```

### Proof Correction (PROOF_CORRECTION_PROMPT)

```
The following Lean 4 code has compilation errors. Please fix the errors while
maintaining the mathematical meaning.

{error_message}

Instructions:
1. Analyze what the theorem is trying to prove. Then, analyze why the error is
   happening, step-by-step. Add a brief explanation.
2. Then, provide a corrected version of the Lean 4 code that addresses these
   specific errors.
3. Do NOT include any other Lean code blocks except for the proof.
4. Do NOT use sorry.
5. Do NOT include any imports or open statements.
6. Do NOT change anything in the original theorem statement.

{useful_theorems_section}
```

### Subgoal Syntax Correction (SUBGOAL_SYNTAX_CORRECTION_PROMPT)

```
The following Lean 4 theorem has compilation errors. Please fix the errors while
maintaining the mathematical meaning.

{error_message}
Instructions:
1. Analyze why the error is happening, step-by-step. Add a brief explanation.
2. Then, provide a corrected version of the Lean 4 code that addresses these
   specific errors.
3. Do NOT include any other Lean code blocks except for the theorem.
4. Use sorry for the proof.
5. Do NOT include any imports or open statements.
{potentially_useful_theorems}
```

### A.3 Implementation Details

We improve HILBERT's efficiency through several runtime optimizations focused on parallelization. The Prover LLM is served using vLLM (Kwon et al., 2023) and the Lean Verifier using Kimina Lean Server (Santos et al., 2025) to handle multiple requests in parallel.

We implement `AsyncJobPool`, a mechanism built around Python's `asyncio` library, to orchestrate parallel requests across our framework's multiple steps. Submitted jobs run concurrently until specific completion criteria are met based on the algorithm step. Concurrency is controlled using Semaphores. We implement three completion criteria:

- **Wait for All**. The execution terminates when all jobs in the pool have finished execution. This criterion is used to parallelize across examples, and across subgoals (Section 3.2.2).
- **First-Success Termination**. Execution terminates as soon as one successful job is found, and pending jobs are terminated. This criterion is used to parallelize across proof attempts (the initial Prover attempts, and Steps 1 and 3 in Section 3.2.2).
- **First Failure**. Execution halts upon the first job failure, immediately canceling remaining jobs. This criterion is applied during subgoal correctness verification (Step 2 in Section 3.2.2). Since verification failures often indicate fundamental issues with the proof sketch that affect multiple subgoals, early termination prevents wasted computation on dependent subgoals, which may change after correcting the problematic subgoal.

### A.4 SUBGOAL DECOMPOSITION EXAMPLE

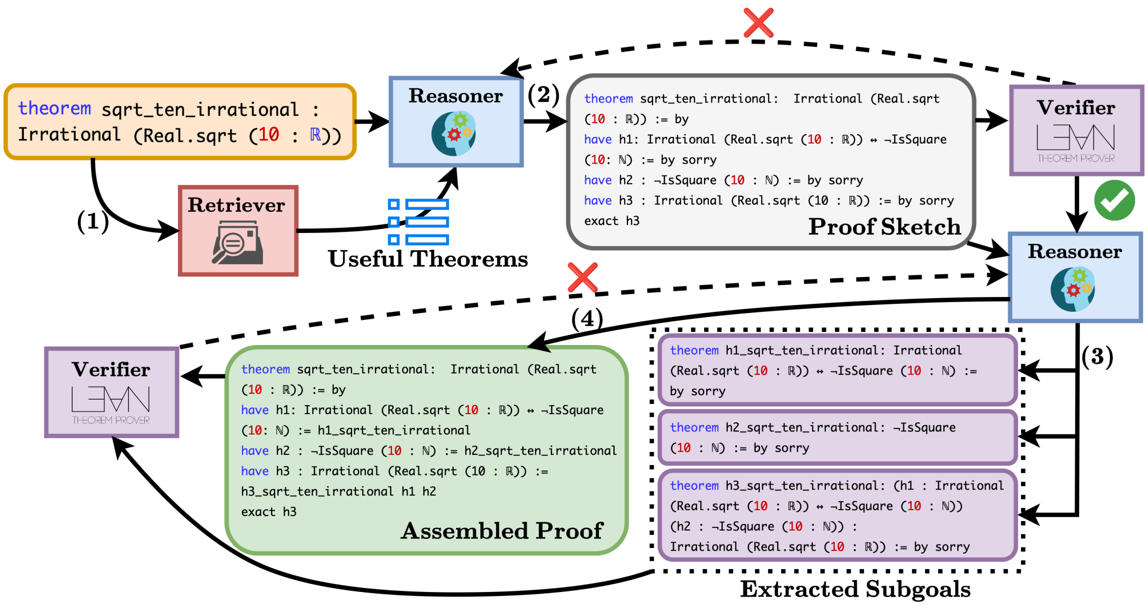


Figure 5: **Subgoal Decomposition Example.** We illustrate the subgoal decomposition process using the input theorem `sqrt_ten_irrational`. The process consists of four main steps: (1) We retrieve relevant theorems from Mathlib to inform the proof strategy. (2) The Reasoner generates a proof sketch, which is verified by the Lean Verifier for validity. If verification fails, error messages guide the Reasoner to make corrections. (3) The Reasoner extracts subgoals from the validated sketch and verifies their correctness, refining them as needed. (4) The Reasoner assembles a complete proof by incorporating the extracted subgoals into the original sketch. Since the subgoals lack proofs at this stage, they are denoted by `sorry`. This assembled proof undergoes final verification. The process outputs both the complete assembled proof and the verified subgoals (without their proofs). Note that while Steps (3) and (4) are shown together in this figure for simplicity, they represent distinct operations as detailed in Figure 2.

### A.5 MINIF2F PROOF EXAMPLE

We include an example to demonstrate how our hierarchical algorithm can help discover proofs that are too complex for either the Prover or informal Reasoner alone. We present the proof for Problem 4 from AMC 12B 2002, a problem which was unsolved even by DeepSeek Prover V2-671B after 8196 passes. We find that HILBERT is able to solve the problem with one layer of decomposition using Goedel Prover V2-32B model as the Prover.

```
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat
theorem k_one_amc12b_2002_p4 (n : ℕ) (h₀ : 0 < n) (k : ℕ) (hk : 41 * n + 42 = k * (42 * n))
↪   (h_k_is_one : ∀ (k : ℕ), 41 * n + 42 = k * (42 * n) → k = 1) : k = 1 := by
  have h_main : k = 1 := by
    have h₁ : k = 1 := h_k_is_one k hk
```

```
    exact h₁

  exact h_main

theorem h_sum_is_frac_amc12b_2002_p4 (n : ℕ) (h₀ : 0 < n) : (1 / 2 + 1 / 3 + 1 / 7 + 1 / ↑n : ℚ) =
↪   (↑(41 * n + 42)) / ↑(42 * n) := by
  have h₁ : (1 / 2 + 1 / 3 + 1 / 7 + 1 / ↑n : ℚ) = (↑(41 * n + 42)) / ↑(42 * n) := by
    have h₂ : (n : ℚ) ≠ 0 := by
      norm_cast
      <;> linarith
    -- Use field_simp to find a common denominator and simplify the expression
    field_simp [h₂]
    -- Normalize the expression by expanding and simplifying it
    <;> ring_nf
    <;> norm_cast
    <;> field_simp [h₂]
    <;> ring_nf
    <;> norm_cast
    <;> simp [mul_assoc]
    <;> norm_num
    <;> linarith

  apply h₁

theorem h_k_is_one_amc12b_2002_p4 (n : ℕ) (h₀ : 0 < n) : ∀ (k : ℕ), 41 * n + 42 = k * (42 * n) → k =
↪   1 := by
  intro k hk
  have h₁ : k ≠ 0 := by
    by_contra h
    -- Assume k = 0 and derive a contradiction.
    have h₂ : k = 0 := by simpa using h
    rw [h₂] at hk
    have h₃ : 41 * n + 42 = 0 := by
      ring_nf at hk ⊢
      <;> nlinarith
    have h₄ : 41 * n + 42 > 0 := by
      nlinarith
    linarith

  have h₂ : k < 2 := by
    by_contra h
    -- Assume k ≥ 2 and derive a contradiction.
    have h₃ : k ≥ 2 := by
      omega
    have h₄ : k * (42 * n) ≥ 2 * (42 * n) := by
      have h₅ : k * (42 * n) ≥ 2 * (42 * n) := by
        have h₆ : k ≥ 2 := h₃
        have h₇ : (42 : ℕ) * n > 0 := by positivity
        nlinarith
      exact h₅
    have h₅ : 2 * (42 * n) = 84 * n := by
      ring
    have h₆ : k * (42 * n) ≥ 84 * n := by
      linarith
    have h₇ : 41 * n + 42 < 84 * n := by
      have h₈ : n ≥ 1 := by linarith
      have h₉ : 43 * n ≥ 43 := by
        nlinarith
      have h₁₀ : 43 * n - 42 ≥ 1 := by
        have h₁₁ : 43 * n ≥ 43 := by nlinarith
        omega
      have h₁₁ : 84 * n > 41 * n + 42 := by
        cases n with
        | zero => contradiction
        | succ n =>
          simp [Nat.mul_add, Nat.add_mul, Nat.add_assoc] at h₆ ⊢
          <;> ring_nf at h₆ ⊢ <;>
            (try omega) <;>
            (try nlinarith)
```

```
        omega
      have h₈ : 41 * n + 42 < k * (42 * n) := by
        linarith
      linarith

    have h₃ : k = 1 := by
      have h₄ : k ≤ 1 := by
        omega
      have h₅ : k ≥ 1 := by
        by_contra h₆
        -- If k is not ≥ 1, then k = 0, which contradicts h₁.
        have h₇ : k = 0 := by
          omega
        contradiction
      omega

    exact h₃

theorem h_divides_amc12b_2002_p4 (n : ℕ) (h₀ : 0 < n) (h₁ : (1 / 2 + 1 / 3 + 1 / 7 + 1 / ↑n : ℚ).den
↪  = 1) (h_sum_is_frac : (1 / 2 + 1 / 3 + 1 / 7 + 1 / ↑n : ℚ) = (↑(41 * n + 42)) / ↑(42 * n)) : 42 *
↪  n ∣ 41 * n + 42 := by
  have h₂ : ((↑(41 * n + 42) : ℚ) / ↑(42 * n)).den = 1 := by
    have h₂₁ : ((↑(41 * n + 42) : ℚ) / ↑(42 * n)) = (1 / 2 + 1 / 3 + 1 / 7 + 1 / ↑n : ℚ) := by
      rw [h_sum_is_frac]
      <;> field_simp <;> ring_nf <;> norm_cast
      <;> field_simp <;> ring_nf <;> norm_cast
    rw [h₂₁]
    exact h₁

  have h₃ : (42 * n : ℤ) ∣ (41 * n + 42 : ℤ) := by
    have h₃₁ : (((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ)).den = 1 := by
      norm_cast at h₂ ⊢
      <;> simpa [div_eq_mul_inv] using h₂
    have h₃₂ : ∃ (k : ℤ), ((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ) = (k : ℚ) := by
      have h₃₃ : (((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ)).den = 1 := h₃₁
      have h₃₄ : ∃ (k : ℤ), ((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ) = (k : ℚ) := by
        -- Use the fact that the denominator is 1 to find an integer k
        have h₃₅ : ∃ (k : ℤ), ((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ) = (k : ℚ) := by
          -- Use the property of rational numbers with denominator 1
          have h₃₆ : (((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ)).den = 1 := h₃₁
          have h₃₇ : ∃ (k : ℤ), ((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ) = (k : ℚ) := by
            -- Use the fact that the denominator is 1 to find an integer k
            refine' ⟨(((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ)).num, _⟩
            have h₃₈ : (((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ)) = ((((41 * n + 42 : ℤ) : ℚ) / (42 * n
            ↪  : ℤ)).num : ℚ) := by
              have h₃₉ : (((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ)).den = 1 := h₃₁
              have h₄₀ : (((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ)) = ((((41 * n + 42 : ℤ) : ℚ) / (42 *
              ↪  n : ℤ)).num : ℚ) := by
                rw [← Rat.num_div_den (((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ))]
                <;> field_simp [h₃₉]
                <;> norm_cast
                <;> simp_all [Rat.den_nz]
              exact h₄₀
            exact h₃₈
          exact h₃₇
        exact h₃₅
      exact h₃₄
    obtain ⟨k, h₃₃⟩ := h₃₂
    have h₃₄ : (42 * n : ℤ) ∣ (41 * n + 42 : ℤ) := by
      have h₃₅ : ((41 * n + 42 : ℤ) : ℚ) / (42 * n : ℤ) = (k : ℚ) := h₃₃
      have h₃₆ : (42 * n : ℤ) ≠ 0 := by
        have h₃₇ : (n : ℕ) > 0 := h₀
        have h₃₈ : (42 * n : ℤ) > 0 := by
          norm_cast
          <;> nlinarith
        linarith
      have h₃₇ : ((41 * n + 42 : ℤ) : ℚ) = (k : ℚ) * (42 * n : ℤ) := by
        field_simp [h₃₆] at h₃₅ ⊢
        <;> ring_nf at h₃₅ ⊢ <;> norm_cast at h₃₅ ⊢ <;>
```

```
          (try norm_num at h₃₅ ⊢) <;>
          (try linarith) <;>
          (try simp_all [mul_comm]) <;>
          (try ring_nf at h₃₅ ⊢ <;> norm_cast at h₃₅ ⊢ <;> linarith)
        <;>
          (try
            {
              nlinarith
            })
        <;>
          (try
            {
              field_simp at h₃₅ ⊢ <;>
                ring_nf at h₃₅ ⊢ <;>
                norm_cast at h₃₅ ⊢ <;>
                linarith
            })
      have h₃₈ : (41 * n + 42 : ℤ) = k * (42 * n : ℤ) := by
        norm_cast at h₃₇ ⊢
        <;>
          (try ring_nf at h₃₇ ⊢) <;>
          (try norm_num at h₃₇ ⊢) <;>
          (try linarith)
        <;>
          (try
            {
              field_simp at h₃₇ ⊢ <;>
                ring_nf at h₃₇ ⊢ <;>
                norm_cast at h₃₇ ⊢ <;>
                linarith
            })
      have h₃₉ : (42 * n : ℤ) ∣ (41 * n + 42 : ℤ) := by
        use k
        <;>
          linarith
      exact h₃₉
    exact h₃₄

  have h₄ : 42 * n ∣ 41 * n + 42 := by
    have h₄₁ : (42 * n : ℤ) ∣ (41 * n + 42 : ℤ) := h₃
    have h₄₂ : (42 * n : ℕ) ∣ (41 * n + 42 : ℕ) := by
      -- Convert the divisibility from integers to natural numbers
      norm_cast at h₄₁ ⊢
      <;>
      (try omega) <;>
      (try
        {
          -- Use the fact that the integers are positive to convert to natural numbers
          cases' h₄₁ with k hk
          use k
          <;>
          (try ring_nf at hk ⊢) <;>
          (try norm_cast at hk ⊢) <;>
          (try omega)
        })
      <;>
      (try
        {
          -- Handle any remaining cases or simplifications
          simp_all [mul_comm]
          <;>
          ring_nf at *
          <;>
          norm_cast at *
          <;>
          omega
        })
    -- Use the natural number divisibility to conclude the proof
    simpa [mul_comm] using h₄₂
```

```
  exact h₄

theorem amc12b_2002_p4 (n : ℕ) (h₀ : 0 < n) (h₁ : (1 / 2 + 1 / 3 + 1 / 7 + 1 / ↑n : ℚ).den = 1) : n =
↪  42 := by
  -- Step 1: Combine the rational numbers into a single fraction.
  have h_sum_is_frac : (1 / 2 + 1 / 3 + 1 / 7 + 1 / ↑n : ℚ) = (↑(41 * n + 42)) / ↑(42 * n) := by
    exact h_sum_is_frac_amc12b_2002_p4 n h₀
  -- Step 2: Use the denominator condition (h₁) to establish a divisibility relation.
  -- According to `Rat.den_div_natCast_eq_one_iff`, for `m, d : ℕ` with `d ≠ 0`,
  -- `((m : ℚ) / d).den = 1` iff `d | m`.
  have h_divides : 42 * n ∣ 41 * n + 42 := by
    exact h_divides_amc12b_2002_p4 n h₀ h₁ h_sum_is_frac
  -- Step 3: By the definition of divisibility, `h_divides` implies there exists a natural number `k`
  -- such that `41 * n + 42 = k * (42 * n)`. This step proves that `k` must be 1.
  have h_k_is_one : ∀ k : ℕ, 41 * n + 42 = k * (42 * n) → k = 1 := by
    exact h_k_is_one_amc12b_2002_p4 n h₀
  -- From h_divides, we obtain the existence of such a `k` and its corresponding equation.
  rcases h_divides with ⟨k, hk⟩

  -- We use commutativity of multiplication to match the form expected by the helper theorem.
  rw [mul_comm (42 * n)] at hk

  -- We use our proof from h_k_is_one to show that this specific `k` must be 1.
  have k_one : k = 1 := by
    exact k_one_amc12b_2002_p4 n h₀ k hk h_k_is_one
  -- Substituting k = 1 back into the equation.
  rw [k_one, one_mul] at hk

  -- The equation is now `41 * n + 42 = 42 * n`. We solve for `n`.
  -- We can rewrite `42 * n` as `41 * n + n`.
  rw [show 42 * n = 41 * n + n by ring] at hk

  -- By cancelling `41 * n` from both sides, we get `42 = n`.
  exact (Nat.add_left_cancel hk).symm
```

### A.6 Inference Time Compute

Beyond inference-time scaling with the number of Reasoner calls (Figure 3), we demonstrate how HILBERT scales with additional metrics: the number of tokens consumed by the Reasoner and Prover (Figure 6), and the number of Prover and Verifier calls (Figure 7). Consistent with our previous findings, we observe a continuous increase in pass rate as token usage increases. Notably, the most challenging problems required 22.8M and 27.0M tokens for the Gemini 2.5 Pro variants with Goedel-Prover-V2 and DeepSeek-Prover-V2, respectively. These token counts far exceed the context length of most LLMs, demonstrating that our agentic framework enables models to go beyond their inherent context limitations when solving complex mathematical problems, at the cost of increased inference-time computation.

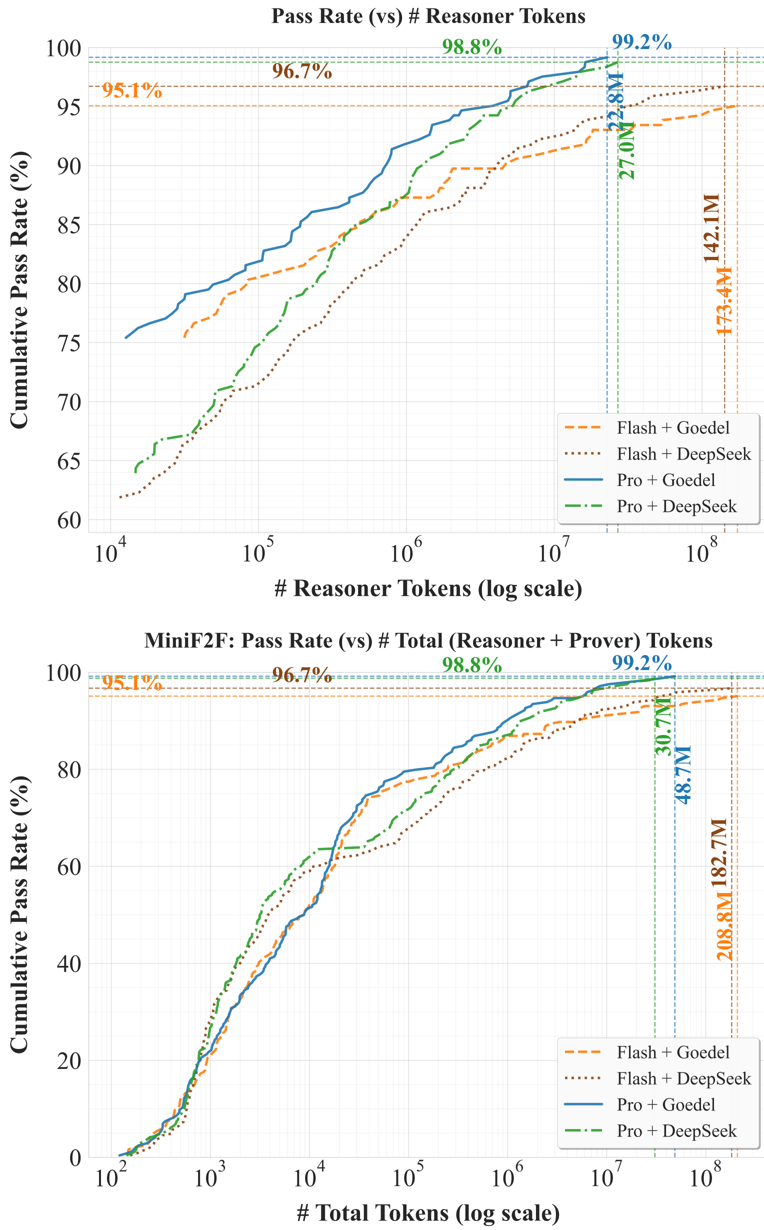


Figure 6: **Pass rate (vs) Reasoner and Total Tokens.** We plot the pass-rate for HILBERT on MiniF2F as a function of (top) the number of tokens used by the Reasoner (bottom) the total number of tokens used (Reasoner + Prover), per sample.

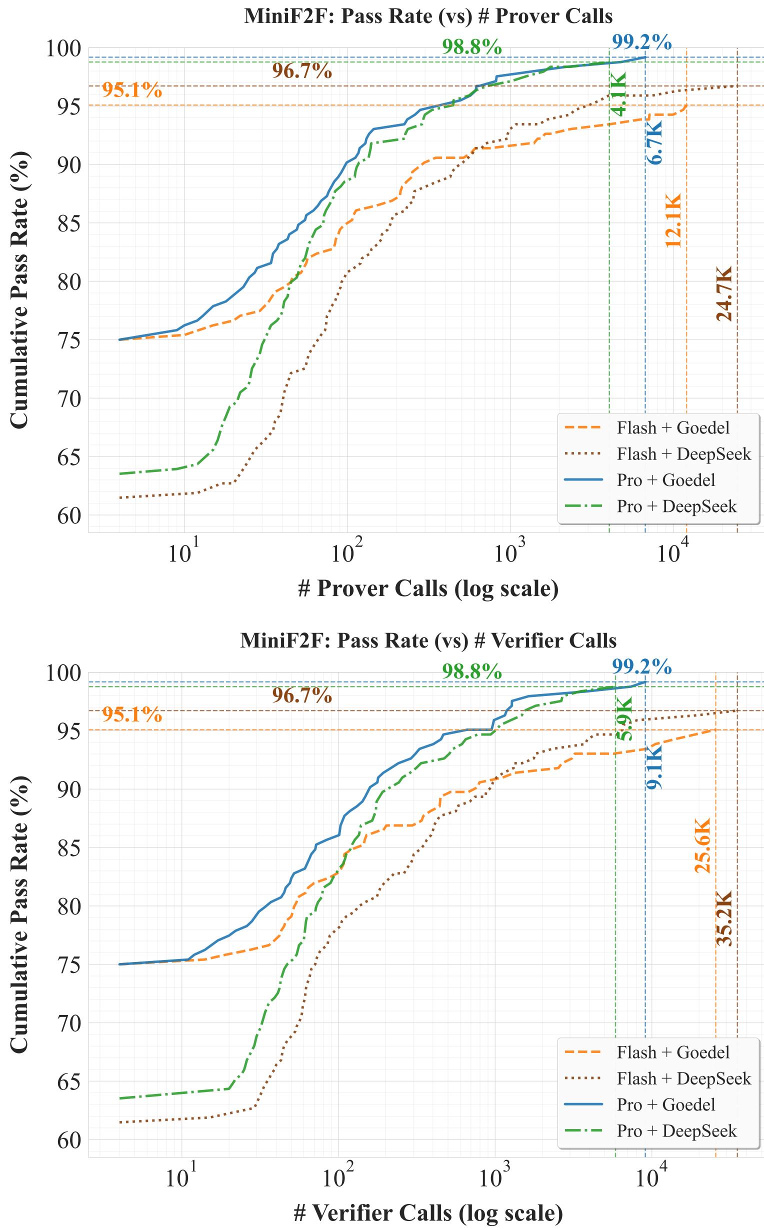


Figure 7: **Pass rate (vs) Prover and Verifier Calls.** We plot the pass-rate for HILBERT on MiniF2F as a function of (top) the number of calls to the Prover (bottom) the number of calls to the Verifier, per sample.

## A.7 PROOF LENGTHS

Figures 8 and 9 show the distribution of proof lengths generated by HILBERT on the MiniF2F and PutnamBench datasets, respectively. For comparison, Figure 8 also includes proof lengths from DeepSeek-Prover-V2-671B on MiniF2F problems.

On MiniF2F, HILBERT generates substantially longer proofs than DeepSeek-Prover-V2-671B, with an average length of 247 lines compared to 86.7 lines. Notably, HILBERT produces one proof spanning 8,313 lines, demonstrating its capacity for tackling hard problems.

This trend toward longer proofs is even more pronounced on PutnamBench, where HILBERT achieves an average proof length of 1,454 lines. The longest proof on this dataset exceeds 15,000 lines of code. The ability to consistently generate such extensive proofs likely contributes to HILBERT's superior performance on PutnamBench compared to baseline methods, as longer proofs may reflect more thorough exploration of intermediate steps necessary for a complete Lean proof.

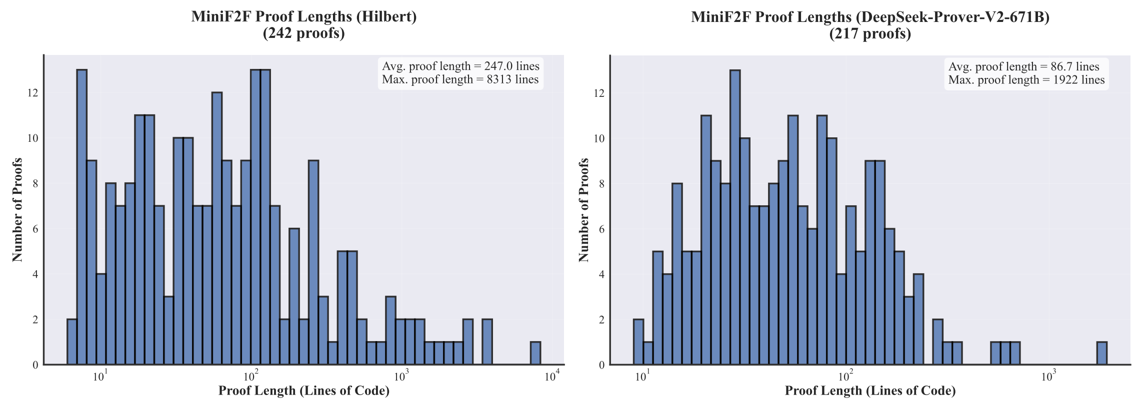


Figure 8: Lengths of proofs generated by (left) HILBERT (Gemini 2.5 Pro + Goedel-Prover-V2) (right) DeepSeek-Prover-V2 671B for problems from MiniF2F.

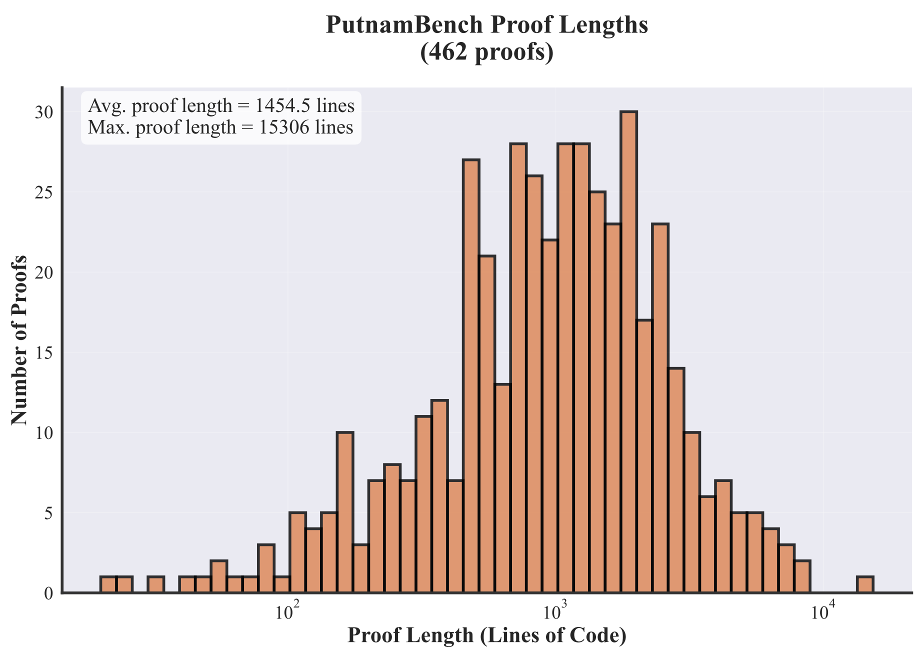


Figure 9: Lengths of proofs generated by HILBERT (Gemini 2.5 Pro + Goedel-Prover-V2) for problems from PutnamBench.