nature

https://doi.org/10.1038/s41586-025-09833-y

Accelerated Article Preview

Olympiad-level formal mathematical reasoning with reinforcement learning

Received: 3 June 2025

Accepted: 30 October 2025

Accelerated Article Preview
Published online: 12 November 2025

Cite this article as: Hubert, T. et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature* https://doi.org/10.1038/s41586-025-09833-y (2025)

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli & David Silver

Olympiad-level formal mathematical reasoning with reinforcement learning

Thomas Hubert [*1], Rishi Mehta[1], Laurent Sartran[1], Miklós Z. Horváth[1], Goran Žužić[1], Eric Wieser[†1], Aja Huang[1], Julian Schrittwieser[1], Yannick Schroecker[1], Hussain Masoom[1], Ottavia Bertolli[1], Tom Zahavy[1], Amol Mandhane[1], Jessica Yung[1], Iuliya Beloshapka[1], Borja Ibarz[1], Vivek Veeriah[1], Lei Yu[1], Oliver Nash[1], Paul Lezeau[1], Salvatore Mercuri[1], Calle Sönne[1], Bhavik Mehta[1], Alex Davies[1], Daniel Zheng[1], Fabian Pedregosa[1], Yin Li[1], Ingrid von Glehn[1], Mark Rowland[1], Samuel Albanie[1], Ameya Velingker[1], Simon Schmitt[1], Edward Lockhart[1], Edward Hughes[1], Henryk Michalewski[1], Nicolas Sonnerat[1], Demis Hassabis[1], Pushmeet Kohli[1], and David Silver[1]

[1] *Google DeepMind, London, UK*

July 2025

Abstract

A long-standing goal of artificial intelligence is to build systems capable of complex reasoning in vast domains, a task epitomized by mathematics with its boundless concepts and demand for rigorous proof. Recent AI systems, often reliant on human data, typically lack the formal verification necessary to guarantee correctness. By contrast, formal languages such as Lean [1] offer an interactive environment that grounds reasoning, and reinforcement learning (RL) provides a mechanism for learning in such environments. We present AlphaProof, an AlphaZero-inspired [2] agent that learns to find formal proofs through RL by training on millions of auto-formalized problems. For the most difficult problems, it uses Test-Time RL, a method of generating and learning from millions of related problem variants at inference time to enable deep, problem-specific adaptation. AlphaProof substantially improves state-of-the-art results on historical mathematics competition problems. At the 2024 IMO competition, our AI system, with AlphaProof as its core reasoning engine, solved three out of the five non-geometry problems, including the competition's most difficult problem. Combined with AlphaGeometry 2 [3], this performance, achieved with multi-day computation, resulted in reaching a score equivalent to that of a silver medallist, marking the first time an AI system achieved any medal-level performance. Our work demonstrates that learning at scale from grounded experience produces agents with complex mathematical reasoning strategies, paving the way for a reliable AI tool in complex mathematical problem-solving.

One of the grand challenges in artificial intelligence is to develop agents that can reason effectively and discover solutions in complex, open-ended environments. Mathematics, with its role as a foundation for scientific understanding, serves as a profound and meaningful domain in which to develop these capabilities. As a natural step towards this goal, we focus on developing the necessary reasoning capabilities within the domain of elite mathematics competitions. While not open-ended themselves, these competitions are renowned for problems that demand a depth of creative and multi-step reasoning, thereby providing a crucial and standardized environment for measuring progress.

The historical arc of mathematics, from Euclid's foundational axiomatisation of geometry to the widespread adoption of symbolic algebraic notation, has been one of increasing formalization. Modern computer-verified systems like the Lean proof assistant [1] and collaborative libraries such as Mathlib [4] represent the logical continuation of this trajectory, enabling the expression of complex mathematics in a machine-understandable format. In these systems, a formal proof is not just a sequence of arguments, but a specific data structure called a "proof term" that encodes the entire logical argument from axioms to conclusion. While these terms can be constructed directly, a user typically builds them interactively by applying actions called tactics: small programs that manipulate the current proof state—the set of hypotheses and goals—to advance the proof one logical step at a time. The soundness of this process is guaranteed by Lean's kernel, which verifies that the generated proof term is a valid construction. These systems offer two transformative capabilities: first, the rigorous, automated verification of every logical step, guaranteeing proof correctness; and second,

[*] tkhubert@google.com
[†] efw@google.com

the transformation of mathematics into an interactive, verifiable domain, allowing mathematical reasoning to be treated as a process that can be simulated, experimented with, and learned.

Reinforcement learning (RL) offers a powerful paradigm for learning through interaction and experience, where agents optimize their behaviour via trial and error to achieve specified goals. This approach has proven particularly adept at mastering complex domains where optimal strategies are unknown. The AlphaZero family of agents, for instance, demonstrated the ability to achieve superhuman performance in challenging board games like Go, chess, and shogi [2], optimize quantum dynamics [5] and discover more efficient algorithms for fundamental computations such as sorting [6] and matrix multiplication [7]. The power of these systems stems from their ability to interact at scale with a verifiable environment and use grounded trial and error feedback to continually learn and refine their strategies. RL coupled with formal systems thus represents a particularly promising approach for tackling the challenge of automated mathematical reasoning.

While formal systems provide verifiable grounding, considerable progress in AI mathematical reasoning has also occurred using large language models (LLMs) trained on vast corpora of informal, natural-language mathematical text. These models have demonstrated impressive capabilities in solving a wide range of problems and generating human-like mathematical discourse [8, 9, 10], benefiting directly from the scale and breadth of existing human knowledge expressed in text. Rigorously verifying the correctness of their reasoning remains however an active research challenge, currently employing techniques such as checking final answers against known solutions or comparing, with systems that cannot be fully trusted, generated reasoning steps against reference proofs [11, 12]. This lack of guaranteed correctness limits their reliability for validating novel mathematical claims or tackling problems without pre-existing reference points. In contrast, the inherent verification capabilities of formal systems provide the necessary foundation for building AI agents whose reasoning process and outputs can be trusted, even when exploring beyond the boundaries of existing human proofs and training data.

AlphaProof combines the rigour of formal systems with the experiential learning of RL to find proofs within the Lean theorem prover environment and to develop powerful mathematical reasoning. AlphaProof dramatically improved state-of-the-art results on elite historical mathematics competition problems and, notably, proved 3 out of 5 problems at the 2024 IMO competition. While its solutions required computational time far exceeding that of human contestants, this success demonstrates the ability to tackle mathematical challenges previously considered beyond the reach of automated systems.

1 AlphaProof

AlphaProof is a RL agent designed to discover formal mathematical proofs by interacting with a verifiable environment based on the Lean theorem prover. Its architecture, training and inference integrate several key innovations.

1.1 The Lean RL Environment

We model the interactive proving process within Lean as a sequential decision-making problem, a standard formulation for RL tasks, in a similar way as [13, 14]. To distinguish our formal RL task from the Lean proof assistant itself, we term this specific formulation the "Lean Environment". Each mathematical statement to be proved constitutes a distinct problem instance. We now formally define this environment using the standard RL terminology of states, actions, rewards and returns. At any time step t, the state s_t is the logical state of the Lean prover, encompassing established hypotheses and remaining goals, observed by the agent as the Lean tactic state (fig. 1a left). The agent interacts by proposing an action a_t, a Lean tactic, as a text string. The environment attempts to execute these tactics, transitioning to a new state by updating hypotheses and goals (fig. 1a right). Each episode starts with a new problem statement and ends when a proof of that statement is successfully found, or a time-out occurs. The agent is incentivized to find short proofs by a reward signal $r_t = -1$ for each tactic applied. The return G_t from a state s_t is the sum of these rewards until termination. Crucially, for proof states that decompose into multiple independent subgoals that must all be solved, the return is defined as the minimum return over these subgoals (i.e., corresponding to the longest proof branch), rather than the more natural sum of returns from each subgoal (see Methods for full RL formulation and value function definition).

1.2 Prover Agent

The AlphaProof agent combines a deep neural network with a powerful search algorithm inspired by AlphaZero [2]. At its core is the proof network, a 3 billion parameters encoder-decoder transformer model [15, 16], that learns to interpret the observed Lean tactic state (fig. 1b) and generate two outputs: a policy, suggesting promising tactics to apply next, and a value function, estimating the expected return G_t (as defined in section 1.1). These outputs guide a specialized tree search that executes sequences of tactics and evaluates their consequences (fig. 1c). Key adaptations for formal

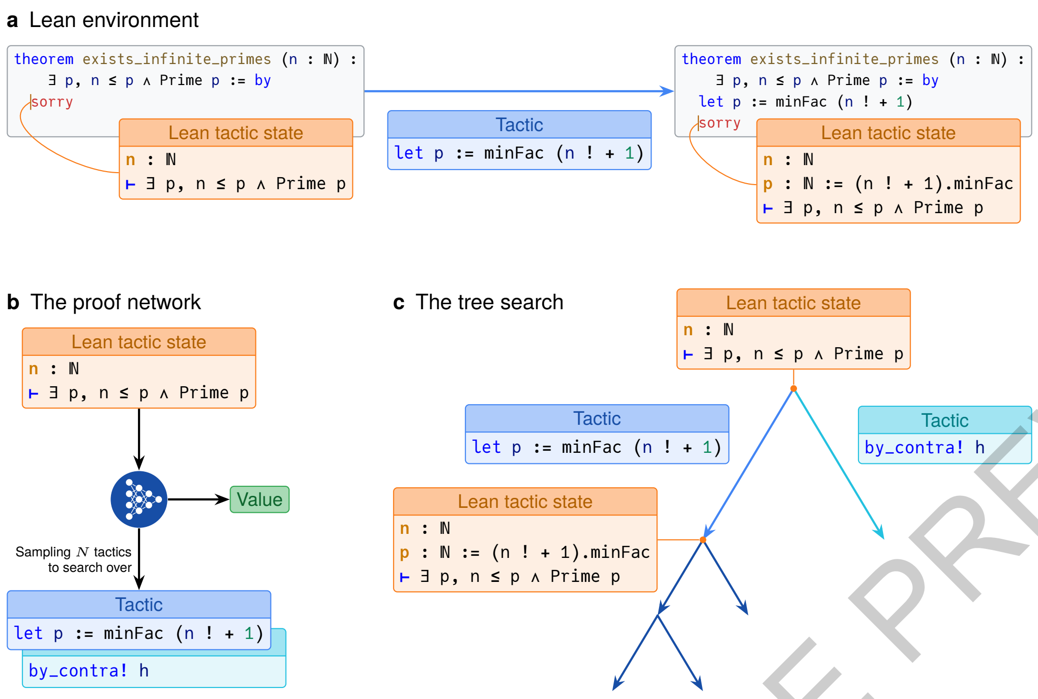


Figure 1: **AlphaProof Core Reasoning Components. a,** The Lean environment in which each step of the proving process takes place. An agent observes an initial Lean tactic state (left) which describes the theorem to be proved. The `sorry` keyword is a placeholder in Lean indicating that the proof is not yet complete. Applying a tactic to the environment (e.g., `let p := minFac (n ! + 1)`) results in a new state, potentially introducing new hypotheses or altering goals (right box). **b,** The proof network takes the current Lean tactic state as input and produces two outputs: a list of N promising tactics to try and a value estimating proof difficulty. **c,** The tree search uses the proof network's outputs to guide its exploration of potential proof paths. Starting from an initial state (root node), the search iteratively expands the tree. At each node, it calls the proof network (panel b) to generate promising tactics and a value estimate for the current state. These tactics (edges) are then applied within the Lean environment (panel a) which results in new proof states (child nodes). The network's value is used to intelligently guide the search, focusing computational effort on the most promising paths (e.g., by exploring the left branch more deeply).

theorem proving include an AND-OR tree structure to handle the decomposition of proofs into multiple independent subgoals, similar to [17], that must all be solved. Furthermore, to manage the large, open-ended space of possible tactics, AlphaProof samples actions [18] and incorporates progressive sampling [19] to explore a broader range of proof strategies along critical paths (see Methods for full details on the network architecture and tree search algorithm).

1.3 Training

AlphaProof's capabilities are primarily developed through a multi-stage training process. First, the proof network undergoes pre-training on a large corpus of approximately 300 billion tokens of code and mathematical text using a next-token prediction objective, a standard technique for language models. The goal of this stage is to imbue the network with a broad understanding of logical structures, programming syntax, and mathematical language—an essential foundation for the subsequent stage to effectively learn from much smaller formal datasets. Next, Supervised fine-tuning (SFT) is performed using approximately 300,000 state-tactic pairs extracted from human-written proofs in the Mathlib library [4]. This stage enables the proof network to understand Lean syntax and internal states, imitate expert Lean tactics, and provide initial estimates for proof difficulty.

The central learning phase, inspired by AlphaZero, is the main RL loop in which AlphaProof learns from self-generated experience. Unlike board games, however, the proving task lacks a single, universal initial state from which all other states can be derived through self-play. Consequently, learning to reason across diverse mathematical domains necessitates a vast and varied corpus of problem statements to serve as distinct starting points for the RL agent. While human-generated mathematical problems provide a natural source for such a corpus, the number of problems manually formalized in Lean is many orders of magnitude smaller than those available in natural language. To bridge this gap, we developed an auto-formalization process (fig. 2a, left and extended fig. 2). This process employs a Gemini-based LLM, fine-tuned and iteratively refined using human and synthetic data. Over the course of the project, this model auto-formalized approximately 1 million natural language problems into a dataset of around 80 million formal Lean problems

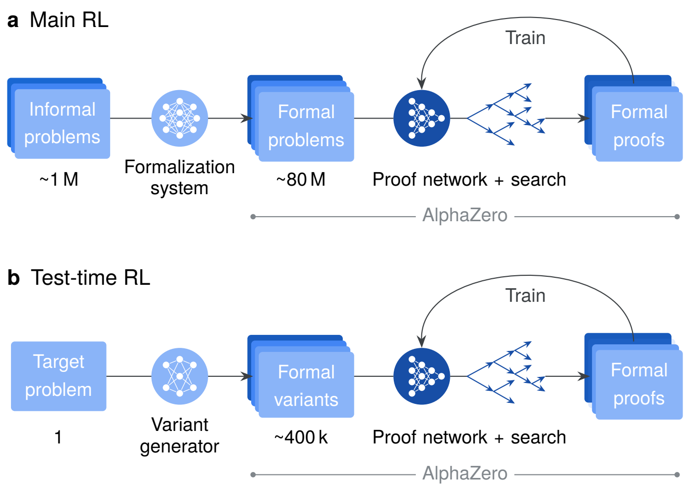


Figure 2: **AlphaProof Learning and Adaptation Processes. a,** The Main RL loop. Approx. 1 million informal mathematical problems are first translated into a large-scale (~80M) formal problem dataset by a formalization system. This dataset serves as the curriculum for an AlphaZero-like RL process. Here, the proof network, in conjunction with the tree search, interacts with the Lean Environment to generate formal proofs and disproofs. The experience gained from these attempts is used to iteratively train and improve the proof network, enhancing its ability to solve a broad range of mathematical problems. **b,** The Test-Time RL (TTRL) loop. For a specific, challenging formal problem, the variant generator generates a diverse set of relevant formal variants. A focused AlphaZero-like RL process, again utilizing the proof network and tree search, then trains on this bespoke curriculum of variants. This targeted learning adapts the proof network to the specific structures and difficulties of the original hard problem, often enabling it to find a formal proof that was previously intractable. This process can be run on multiple target problems at the same time.

vastly exceeding the scale of all other available datasets. Importantly, each auto-formalized statement, regardless of its fidelity to the original natural language problem, provides a valid formal problem that AlphaProof can attempt to prove or disprove, thus serving as a useful training instance.

A matchmaker system assigns auto-formalized problems and adaptive compute budgets to distributed actors, randomly tasking them to either prove or disprove each statement. Actors then generate attempts in the Lean Environment using a tree search algorithm that operates over proof trees and is guided by the current proof network. Lean-verified outcomes—whether a proof, a disproof, or a timeout—provide grounded feedback. The proof network is continually improved (fig. 2a, right) using experience from both successful proof and disproof attempts, refining its policy to predict effective tactics and adjusting its value to estimate the expected return. This continuous improvement based on self-generated experience allows AlphaProof to move far beyond imitating existing proofs, and discover novel, potentially complex reasoning strategies necessary to solve challenging problems (see Methods for full details on each training stage, auto-formalization, the matchmaker and a summary of the required computational budget).

1.4 Inference

When presented with a new problem, AlphaProof leverages two complementary computational scaling mechanisms, both initialized from its main RL-trained agent. First, increasing the tree search budget allows for a more exhaustive exploration of proof paths, a technique proven effective in the AlphaZero lineage [2]. Second, for problems where extensive search may be insufficient, AlphaProof uses a novel Test-Time RL (TTRL) approach (fig. 2b). TTRL employs the same core AlphaZero-inspired RL algorithm as the main training phase (fig. 2a), but instead of learning from a broad curriculum of auto-formalized problems, TTRL focuses learning on a bespoke curriculum of synthetic problem variants (e.g., simplifications or generalizations) generated specifically around the target problem (fig. 2b). TTRL thus allows the agent to dedicate substantial resources to learn problem-specific strategies, often unlocking solutions intractable through scaling up the tree search alone (see Methods for full details).

2 Results

2.1 Benchmarks

We evaluated AlphaProof on a comprehensive suite of formal mathematics benchmarks, all manually formalized in Lean, spanning advanced high-school to elite Olympiad and university-level problems. Our evaluation suite comprises: (1) a corrected version of the publicly available miniF2F benchmark [20] (high-school mathematics competitions); (2) formal-imo, a benchmark of all non-geometry (because of specific Mathlib library limitations for Olympiad-style geometry, see section 6.3 in Methods) historical IMO problems internally formalized by experts; and (3) the public Putnam benchmark [21] (undergraduate Putnam Mathematical Competition problems). For the Putnam benchmark, problems from even-numbered years, 1990 onwards, were reserved as a dedicated held-out test set (PutnamBench-test). Rigorous data separation was maintained throughout all training phases to ensure evaluation of generalized reasoning capabilities rather than memorization (see Methods for details on data curation and separation). Together, these benchmarks form a standardized, diverse, and demanding testbed for assessing AlphaProof's capacity for advanced mathematical reasoning and problem-solving.

2.2 Main RL Progress

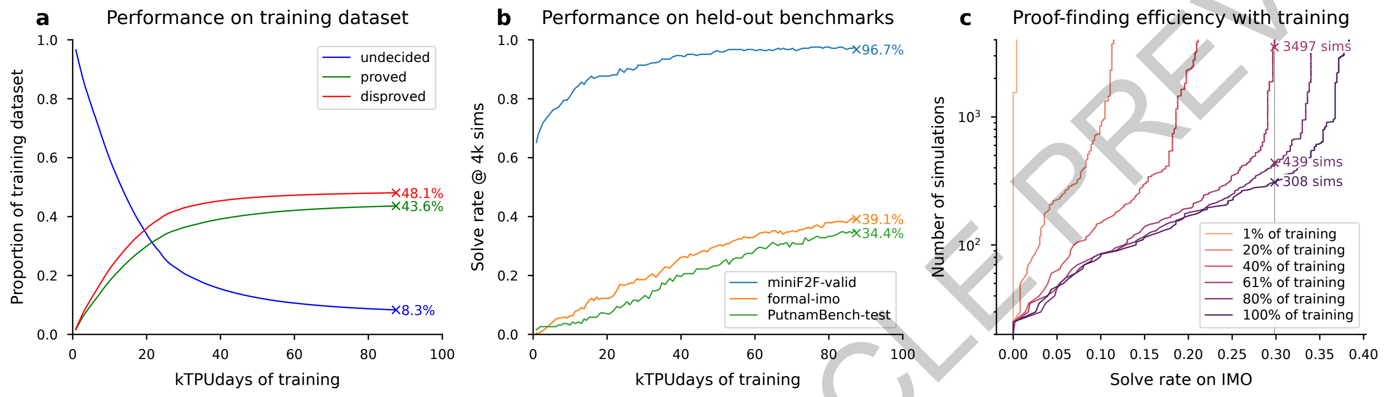


Figure 3: **AlphaProof's Learning Progression during Main RL.** **a**, Evolution of performance on the training dataset (auto-formalized problems). Lines show the proportion of starting problem instances that are proved (green), disproved (red), or remain undecided (blue) during Main RL, as a function of compute (on Google Cloud "TPU v6e"s). **b**, Solve rate (at 4000 simulations) on held-out benchmarks (miniF2F-valid, formal-imo, PutnamBench-test) throughout training. **c**, Number of tree search simulations (logscale) required to achieve a given solve rate on the historical IMO benchmark at different stages of main RL training. Each curve represents a checkpoint of the AlphaProof agent at a different stage of its main RL training. The plot shows that later-stage agents are not only stronger but also more efficient, requiring substantially less search to achieve the same solve rate as earlier agents. For instance, the final agent solves approximately 30% of problems with only 300 simulations, a level of performance that earlier agents could not reach even with vastly more search.

We analyse the performance of AlphaProof over the course of its main RL phase, which spanned approximately 80 kTPUdays of training (equivalent to, for instance, 4000 TPUs utilized for 20 days). In this stage, AlphaProof learns from self-generated experience by attempting to prove or disprove millions of problems drawn from its auto-formalized dataset. Throughout this training, the proportion of problems successfully proved or disproved steadily increases, leaving only a fraction of its dataset undecided, demonstrating mastery over its training curriculum (fig. 3a). This learned capability generalized robustly to unseen problems; AlphaProof's solve rate on held-out benchmarks (miniF2F-valid, formal-imo, and PutnamBench-test) consistently improved, starting from the performance of the initial supervised fine-tuned model (the 0 TPU-day point in fig. 3b) to a final performance that was far beyond other provers. Furthermore, the main RL phase significantly enhanced proof-finding efficiency. As training progressed, AlphaProof required substantially fewer tree search simulations to achieve a given solve rate (fig. 3c), indicating that the neural network became a much stronger guide, internalizing effective proof strategies. This demonstrates how the large computational cost of RL training is transformed into inference-time efficiency, producing a more powerful and more efficient agent.

2.3 Inference Time Scaling

Once main RL training is complete, AlphaProof's performance on unseen problems can be further enhanced by allocating additional computational resources at inference. We explore two complementary scaling mechanisms operating on different timescales (fig. 4). First, increasing the tree search budget yielded significant improvements in solve rates across all benchmarks (fig. 4a). The agent's learned search policy enabled a strong performance baseline even with low inference budgets (e.g., 2 minutes per problem on 1 TPU). Extending this search budget progressively to several TPUhours allowed for consistently increased solve rates. For instance, scaling the search from 2 TPUminutes to 12 TPUhours boosted solve

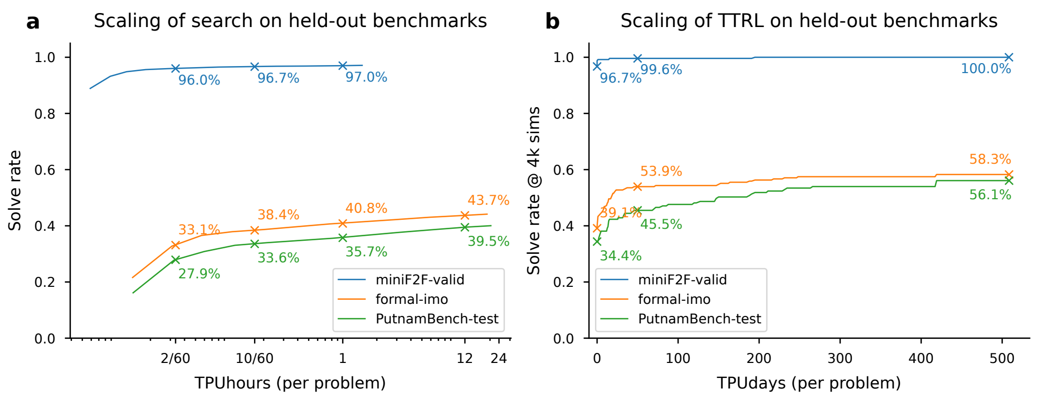


Figure 4: **AlphaProof Performance Scaling with Inference Compute per Problem.** Solve rates on held-out benchmarks (miniF2F-valid, formal-imo, and PutnamBench-test). In both panels, the "compute per problem" is an average, calculated as the total TPU compute consumed during the evaluation, divided by the total number of problems in the benchmark. **a**, Solve rates as a function of increasing tree search compute per problem, measured in v6e TPUhours (logarithmic scale). Solve rates are highlighted for low search budgets (e.g., 2/60 TPUhours per problem, corresponding to 2 minutes on 1 TPU) and more extensive search. **b**, Scaling with TTRL compute. Solve rates as a function of increasing TTRL training compute per target problem, measured in v6e TPUdays (linear scale). Solve rates are highlighted after an initial TTRL compute investment (e.g., 50 TPUdays or 1 day on 50 TPUs per problem) and at the end of the TTRL phase with performance evaluated using 4000 simulations. Note the different x-axis units and scales (logarithmic TPUhours vs. linear TPUdays) between panels.

rates by over 10 absolute percentage points on the formal-imo and PutnamBench-test benchmarks. We observe similar scaling properties on PutnamBench-train (extended table 2). For problems remaining unsolved even with extensive tree search, AlphaProof employs TTRL to enable deeper, problem-specific adaptation (fig. 4b; see Methods for the TTRL procedure). This approach yielded rapid initial gains, solving many new problems within the first 50 TPUdays, and continued to find new solutions as training progressed over longer durations. This sustained learning substantially elevated performance beyond tree search scaling alone, increasing solve rates by an additional 15 absolute percentage points on both formal-imo and PutnamBench-test compared to a 12 TPUhours search (fig. 4b, table 1). The efficacy of TTRL itself is also influenced by factors such as variant quality and quantity (extended fig. 3). The distinct x-axis scales in fig. 4 (TPUhours for search vs. hundreds of TPUdays for TTRL) highlight the different computational investments and capabilities of these two complementary scaling strategies, with TTRL providing a powerful mechanism for tackling the most challenging problems. A detailed breakdown of both tree search and TTRL scaling by mathematical subject for the formal-imo and PutnamBench-test benchmarks, illustrating the dynamic improvement with increasing compute, is provided in extended fig. 4.

2.4 Final Benchmark Evaluation

The comprehensive training and inference scaling strategies culminate in AlphaProof establishing new state-of-the-art (SOTA) performance across all evaluated formal mathematics benchmarks (table 1).

AlphaProof demonstrates strong performance even at modest compute budgets, achieving state-of-the-art results on several benchmarks even with just 2 TPUminutes of search budget per problem (fig. 4a, table 1). This efficiency is particularly notable on PutnamBench-test, where it substantially outperforms prior systems, underscoring its strong foundational reasoning capabilities and its effectiveness in resource-constrained scenarios. For peak performance on the most complex problems, TTRL is crucial, extending the state-of-the-art by a significant margin (fig. 4b). This approach yields comprehensive gains across all benchmarks (table 1), with perfect score on miniF2F-valid extended table 2 and near-perfect scores on the test split. On formal-imo, TTRL reached particularly strong performance in number theory (75.7%) and algebra (72.6%), while also making progress in combinatorics (20.3%). Similar comprehensive gains were observed on the PutnamBench-test. Detailed subject performance is presented in extended fig. 3 and Supplemental Data Table 1. A few proofs are also shown in the Supplementary Information.

2.5 Performance at the 2024 International Mathematical Olympiad

The International Mathematical Olympiad (IMO) is the world's most prestigious pre-collegiate mathematical competition. Held annually since 1959, each participating country sends a team of up to six elite students, selected through rigorous national competitions. Contestants face six exceptionally challenging problems across algebra, combinatorics,

Name	Compute Budget (per problem)	miniF2F-test [20]	formal-imo	PutnamBench-test [21]
Prior state-of-the-art (before IMO 2024)				
GPT-F Expert Iteration [22]	-	36.6%	-	-
Hypertree Proof Search [17]	-	41.0%	-	-
InternLM2-Math-Plus-7B [23]	-	43.4%	-	-
Prior state-of-the-art (after IMO 2024)				
Kimina-Prover-Preview [24]	-	80.7%	-	1.6%
DeepSeek-Prover-V2 [25]	-	88.9%	-	5.3%
This Work				
AlphaProof	2 TPUmins	96.3%	33.2%	27.9%
AlphaProof	12 TPUhours	97.7%	43.7%	39.4%
AlphaProof with TTRL	50 TPUdays	97.5%	53.9%	45.5%
AlphaProof with TTRL	500 TPUdays	99.6%	58.3%	56.1%

Table 1: **Performance of AlphaProof on formal mathematics benchmarks.** AlphaProof is compared against other methods, using the strongest reported result for each system, corresponding to their largest compute budgets. For AlphaProof, the reported "compute budget (per problem)" refers to the average computational cost as defined in fig. 4a and is an inference-time budget only that does not include the amortized cost of the main RL training loop. AlphaProof's miniF2F-test results are reported on a corrected version of the dataset (see Methods for details). Results for other methods on miniF2F-test are as reported in their respective publications, based on the dataset versions they utilized; direct comparison should therefore be made with caution, considering potential dataset differences. For PutnamBench-test, scores for prior work were recalculated based on the publicly available proofs reported by each system, for consistent comparison against our PutnamBench-test split; these may differ from figures reported by those systems for the full PutnamBench.

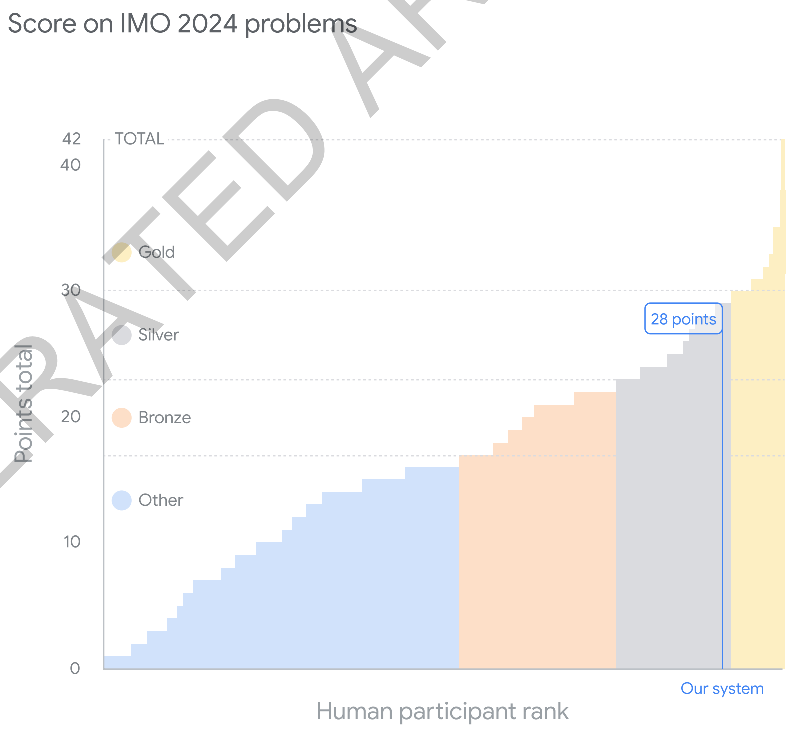


Figure 5: **Combined AI system performance at the IMO 2024.** Our combined system scored 28 points, with AlphaProof solving three problems (P1, P2, P6) and AlphaGeometry 2 [3] solving one (P4). This places the performance of our system relative to human competitors [26] within the silver medal threshold. Official medal boundaries (Gold, Silver, Bronze) are shown.

geometry, and number theory, designed to test deep conceptual understanding and creative problem-solving, pushing them far beyond standard curricula and often requiring hours of intense thought. Achieving an IMO medal is a significant honour, often marking the early careers of individuals who later become leading mathematicians.

To assess AlphaProof's capabilities on an unseen competition, we applied it to the problems from the 2024 IMO, operating as the core reasoning engine within a complete problem-solving pipeline. This version of AlphaProof was developed until July 2024 using a similar training and inference methodology to that described above. Given specific Mathlib library limitations for Olympiad-style geometry (see Methods, IMO-style Geometry), the geometry problem (P4) was addressed using the specialized AlphaGeometry 2 system [3]. The remaining five non-geometry problems (algebra: P1, P6; number theory: P2; combinatorics: P3, P5) were manually formalized in Lean by experts immediately after the competition's release (see Methods, IMO 2024 Evaluation Protocol and Methods). Several problems (P1, P2, P5, P6) required identifying the answer (e.g., "Find all functions…") before constructing a proof. For these, hundreds of candidate answers were generated by querying the public Gemini 1.5 Pro model with Python tool-use enabled and a collection of hand-written examples given as a few-shot prompt [27], successfully guessing the correct answers for all applicable problems. AlphaProof then demonstrated high efficiency by rapidly refuting the vast majority of these incorrect candidates, isolating the correct ones for full proof attempts. Subsequently, using TTRL, AlphaProof successfully discovered formal proofs for all algebra and number theory problems: P1, P2, and P6 (proofs in extended figs. 7 to 9). Each of these solutions required two to three days of TTRL, demonstrating substantial problem-specific adaptation at inference. The two combinatorics problems (P3 and P5) remained unsolved by our systems.

The combined system, with AlphaProof solving P1, P2, and P6, and AlphaGeometry 2 solving P4, successfully solved four of the six IMO problems. This yielded a total score of 28 out of 42 points (section 2.5), placing our AI system's performance within the silver medal range of the official competition, one point below the gold medal threshold. While the multi-day computational effort for AlphaProof's solutions significantly exceeds the time constraints faced by human contestants, it is crucial to note that prior to this work, even the most advanced AI systems could typically only solve a small fraction of the easiest historical IMO problems [13, 17, 23]. AlphaProof's success in solving three problems from a live IMO competition, including 2024's most difficult P6 which was solved by only five human contestants, therefore demonstrates a profound advance in the potential of AI for mathematical reasoning, even when applied to problems renowned for their difficulty and requirement for novel insights.

3 Discussion and Conclusions

We have introduced AlphaProof, a RL agent capable of proving complex mathematical problems within the formal environment of the Lean theorem prover. By combining an AlphaZero-inspired learning framework with a curriculum of millions of auto-formalized problems and a novel TTRL approach based on variant generation, AlphaProof has demonstrated a powerful new capacity for automated mathematical reasoning. Our results show that AlphaProof significantly advances the state-of-the-art across a diverse suite of formal mathematics benchmarks, including those derived from historical IMO and Putnam competitions. The capabilities of this system were notably demonstrated at the 2024 IMO. As part of a complete problem-solving pipeline, AlphaProof successfully solved the three algebra and number theory problems, including the most difficult problem P6, while AlphaGeometry 2 solved the geometry problem. This combined performance resulted in solving four of the six competition problems, thereby reaching the same score as a 2024 IMO silver medallist. This marks a landmark achievement as the first time an AI system has attained any medal-level performance at the IMO, providing compelling evidence of the sophisticated reasoning abilities emerging from learning from grounded, verifiable experience at scale.

While AlphaProof's results represent a significant step, several limitations and avenues for future development remain. While open-source pre-trained foundation models are now available, the bespoke learning phase required by AlphaProof represents a scale of domain-specific training that is likely beyond the reach of most academic research groups. Furthermore, the multi-day inference time required by TTRL for solving the most difficult problems, while effective, highlights the need for more efficient inference-time strategies. Future progress will likely involve continued algorithmic and engineering refinements, further advancements in auto-formalization, and more optimised strategies for TTRL.

Furthermore, AlphaProof's current successes are primarily within the domain of advanced high-school and undergraduate competition mathematics. While exceptionally challenging, these problems operate within a known fixed library of mathematical concepts with a certain degree of thematic consistency. Our TTRL approach proved highly effective in this setting, and understanding its generalization is part of the considerable next step of extending these capabilities to the frontiers of research mathematics. This transition is particularly challenging as it requires moving beyond pure problem solving to theory building—the continuous expansion of this library with new concepts. Ultimately, imbuing AI with an understanding of mathematical taste, interestingness, or beauty, remains a fascinating open research question.

The significant computational requirements of this work raise important questions about accessibility and the future of an open research ecosystem. We believe this work is best viewed as foundational, pathfinding research; demonstrating for the first time that a certain capability is achievable often requires a scale of investment that can later be optimized and democratized. To that end, and to ensure our work serves as a community catalyst rather than a barrier, we are taking concrete steps such as providing an interactive tool for exploration. A key direction for our future research is to substantially improve algorithmic efficiency, with the explicit goal of lowering the computational barriers to entry and ensuring these techniques can become a collaborative tool for the entire mathematical community.

Despite the challenges mentioned above, we anticipate that continued development, particularly in addressing computational and scope limitations will pave the way for AlphaProof to become an increasingly valuable collaborator for human researchers in exploring the frontiers of mathematics and, by extension, other scientific disciplines.

References

[1] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 theorem prover and programming language". In: *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer. 2021, pp. 625–635.

[2] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[3] Yuri Chervonyi et al. *Gold-medalist Performance in Solving Olympiad Geometry with AlphaGeometry2*. 2025. arXiv: 2502.03544 [cs.AI].

[4] The mathlib Community. "The Lean mathematical library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2020, pp. 367–381.

[5] Mogens Dalgaard et al. "Global optimization of quantum dynamics with AlphaZero deep exploration". In: *NPJ quantum information* 6.1 (2020), p. 6.

[6] Daniel J Mankowitz et al. "Faster sorting algorithms discovered using deep reinforcement learning". In: *Nature* 618.7964 (2023), pp. 257–263.

[7] Alhussein Fawzi et al. "Discovering faster matrix multiplication algorithms with reinforcement learning". In: *Nature* 610.7930 (2022), pp. 47–53.

[8] Aaron Jaech et al. *OpenAI o1 system card*. 2024. arXiv: 2412.16720 [cs.AI].

[9] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning*. 2025. arXiv: 2501.12948 [cs.CL].

[10] Google. *Gemini 2.5: Our newest Gemini model with thinking*. Mar. 2025. URL: https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/.

[11] Ivo Petrov et al. "Proof or bluff? evaluating LLMs on 2025 USA Math Olympiad". In: *The second AI for MATH Workshop at the 42nd International Conference on Machine Learning* (2025).

[12] Hamed Mahdavi et al. *Brains vs. bytes: Evaluating LLM proficiency in olympiad mathematics*. 2025. arXiv: 2504.01995 [cs.AI].

[13] Stanislas Polu and Ilya Sutskever. *Generative language modeling for automated theorem proving*. 2020. arXiv: 2009.03393 [cs.LG].

[14] Kaiyu Yang et al. "LeanDojo: Theorem proving with retrieval-augmented language models". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 21573–21612.

[15] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[16] Yujia Li et al. "Competition-level code generation with alphacode". In: *Science* 378.6624 (2022), pp. 1092–1097.

[17] Guillaume Lample et al. "Hypertree proof search for neural theorem proving". In: *Advances in neural information processing systems* 35 (2022), pp. 26337–26349.

[18] Thomas Hubert et al. "Learning and planning in complex action spaces". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 4476–4486.

[19] Rémi Coulom. "Computing Elo ratings of move patterns in the game of Go". In: *ICGA journal* 30.4 (2007), pp. 198–208.

[20] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. *MiniF2F: A cross-system benchmark for formal olympiad-level mathematics*. 2021. arXiv: 2109.00110 [cs.AI].

[21] George Tsoukalas et al. "PutnamBench: Evaluating neural theorem-provers on the putnam mathematical competition". In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 11545–11569.

[22] Stanislas Polu et al. “Formal mathematics statement curriculum learning”. In: *The Eleventh International Conference on Learning Representations*.

[23] Huaiyuan Ying et al. *InternLM-Math: Open math large language models toward verifiable reasoning*. 2024. arXiv: `2402.06332 [cs.CL]`.

[24] Haiming Wang et al. *Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning*. 2025. arXiv: `2504.11354 [cs.AI]`.

[25] ZZ Ren et al. *Deepseek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition*. 2025. arXiv: `2504.21801 [cs.CL]`.

[26] International Mathematical Olympiad. *65th IMO 2024*. July 2024. URL: `https://www.imo-official.org/year_info.aspx?year=2024`.

[27] Gemini Team et al. *Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context*. 2024. arXiv: `2403.05530 [cs.CL]`.

Methods

4 Related Work

Interactive Theorem Proving: Our research is grounded in the field of interactive theorem proving (ITP), where humans formalize theorems and construct verifiable proofs using proof assistants like Isabelle [28], HOL Light [29], Rocq [30], and Lean [1]. These tools have enabled significant achievements, including the formal verification of the Four Color Theorem [31] and critical software like the CompCert C compiler [32]. However, such successes typically demand extensive work from domain experts. AlphaProof aims to mitigate this bottleneck by automating aspects of both theorem formalization and proving, thereby reducing the requisite labor and expertise.

Machine Learning for Theorem Proving: The intersection of machine learning and theorem proving has seen significant advancements, particularly with the advent of large language models (LLMs) [33, 34, 35] *inter alia*, and the availability of formalized mathematics benchmarks such as MiniF2F [20], ProofNet [36], PutnamBench [21], LISA, [37], or HOList [38]. Our work builds upon these developments.

An automated feedback signal from theorem provers for correct proofs has been pivotal for applying large-scale machine learning to theorem proving. Early research [13, 39, 22, 40] explored generative language modeling for automated theorem proving, learning from human-generated proofs and incorporating techniques like expert iteration or RL. Concurrently, several studies [41, 42, 43, 14] have focused on jointly training tactic prediction with premise selection. In contrast to these step-by-step and search-based methods, Baldur [44] pioneered whole-proof generation, using a large language model to produce a complete proof at once. Notably, [45] introduced an approach that guided formal proving with informal proofs ("draft, sketch, and prove" methodology). This line of research is further extended by work on subgoal decomposition [46]. More sophisticated search strategies have also been explored in neural theorem proving, for example HyperTree Proof Search [17] and variants of MCTS [47, 48]. Very recently, Kimina [24] and DeepSeek Prover [25] explored generating the interleave of natural language explanations and formal proofs via LLMs.

Auto-formalization of Mathematics: A critical component of bridging the gap between human-written mathematics and formal theorem provers is auto-formalization. This area focuses on automatically translating natural language mathematical statements into formal proofs or formalized statements. Significant progress includes early work [49, 50], and, more recently, LLM-based approaches for this challenging task [51, 52, 53]. Subsequent research used synthetically generated data to fine-tune LLMs and enhance their auto-formalization capabilities [54, 48, 55]. More recently, contributions have addressed the evaluation of auto-formalizations [56, 57]. Our work extends these research directions by developing an auto-formalization specialist that generates millions of formal statements for our prover agent.

Reasoning with LLMs: Beyond dedicated theorem provers, the broader field of reasoning with LLMs has seen substantial progress. Techniques like chain-of-thought prompting [58] and RL, often coupled with planning, have significantly boosted LLM reasoning capabilities, especially in mathematical and coding tasks [59]. More recently, OpenAI O1 [60] and DeepSeek R1 [9] have demonstrated the effectiveness of scaling RL for LLM reasoning, observing logarithmic scaling with compute at both training and test time. This approach has led to gold-equivalent performance at the International Olympiad in Informatics (IOI) [61] and strong performance in the American Invitational Mathematics Examination (AIME). However, the International Mathematical Olympiad (IMO) has so far remained out of reach for current LLMs, with recent studies highlighting their limited proving capabilities [11, 12].

TTRL: Our methodology extends on trends in test-time compute scaling [62, 60, 9, 61] and adaptation [63, 64, 65], including early TTRL in games [66, 67]. Distinct from these approaches, and concurrent TTRL methods employing majority-vote rewards for unlabeled informal mathematics [68] or numerical verification for recursively simplified integration variants [69], AlphaProof's TTRL methodology (first outlined in [70]) uniquely addresses elite-level formal mathematical problem-solving, particularly in the domain of competition mathematics. It achieves this by generating a diverse curriculum of relevant formal variants for challenging IMO/Putnam-level problems within Lean, and leveraging Lean's rigorous verification to guide learning, ultimately enabling the discovery of complex, validated proofs in this formal setting.

5 AlphaProof

AlphaProof is a RL agent designed to discover formal mathematical proofs by interacting with a verifiable environment based on the Lean theorem prover [1]. The system integrates several core technical components necessary for tackling this complex domain: an RL environment defining the theorem-proving task (the Lean Environment), a deep neural network architecture capable of representing proof states and suggesting proof continuations (proof network), a specialized tree search algorithm adapted for formal proofs, a translation process for generating a large-scale problem dataset for training (auto-formalization), a large-scale RL phase to discover general proof strategies (Main RL) and a focused learning procedure for difficult instances (TTRL). The specifics of each component are described below.

5.1 Lean Environment

5.1.1 Formal Mathematics as the RL Environment

Formal mathematics provides a rigorous framework for expressing mathematical statements and their proofs, enabling automated verification of correctness. We utilize the Lean interactive theorem prover [1], which is built upon a dependent type theory known as the calculus of inductive constructions [71]. In this paradigm, mathematical statements are represented as types, and a proof corresponds to constructing a term that is of (inhabits) that type. For instance, the theorem `∀ P Q, P ∧ Q → Q ∧ P` is proven by constructing a function term, such as `fun P Q ⟨hp, hq⟩ ↦ ⟨hq, hp⟩`, that inhabits the corresponding type. While it is ultimately this "term mode" that Lean's trusted kernel consumes to verify the correctness of a proof, Lean also offers a higher-level "tactic mode" which saves the user from constructing these terms by hand. Tactics are tools that construct the underlying proof term by manipulating the proof state, which consists of current hypotheses and goals. For example, the aforementioned theorem can also be proven using a sequence of tactics like `intro P Q h; simp [h]`. During tactic-mode proving, Lean provides contextual information via a "tactic state" display (fig. 1a, extended fig. 1a), showing current goals and their associated hypotheses. It is this tactic mode of Lean that we cast as an RL environment (extended fig. 1b).

States (s_t): A state represents the complete logical context within the Lean prover at a given step t of a proof attempt, encompassing all active hypotheses and remaining goals.

Observations: The agent observes the state as a pretty-printed string representation of the Lean tactic state.

Actions (a_t): An action is a Lean tactic, represented as a text string.

Transitions: Applying an action a_t to state s_t involves the Lean environment attempting to execute the tactic. If successful and valid (see 'Tactic Validation and Execution'), this results in a new state s_{t+1} with updated hypotheses and/or goals.

Episodes: Each episode commences with a new mathematical statement (the initial state) to be proven. An episode terminates when: (i) a complete, Lean-verified proof of the initial statement is found; or (ii) a predefined computational budget is exhausted.

Reward Signal & Objective: To incentivize the discovery of the shortest proof, a reward of $r_t = -1$ accompanies each tactic applied.

Return Definition: The return G_t from a state s_t is the sum of rewards until termination. For states that decompose into multiple independent subgoals (AND-nodes, see 'Goal Decomposition' in section 5.1.2), the return (and thus the value function target) is defined as the minimum return (i.e., corresponding to the longest/hardest proof branch) among its constituent subgoals. This definition recursively applies to ancestor states of such multi-goal states. While summing the returns from each subgoal, thereby minimizing the total number of tactics in the proof, is a natural alternative, our choice of the minimum return provides an interesting incentive as it encourages the agent to split goals into subgoals of balanced difficulty. This results in the value of a state to correspond to $-T_{steps}$, where T_{steps} is the number of tactics in the longest branch of the proof required to resolve all subgoals.

5.1.2 Interfacing with Lean

To enable RL, AlphaProof interacts with a custom environment built upon the Lean 4 interactive theorem prover [1]. It enables executing tactics, handles proof states, implements required additional operations and incorporates optimizations for large-scale tree search.

Core Lean and Mathlib Integration: The environment utilizes Lean 4 as its foundational proof assistant. The extensive Mathlib library [4] is loaded, providing a wide array of mathematical definitions, established theorems, and high-level tactics (e.g., the linear arithmetic solver linarith) that the agent can leverage. To support the formalization of specific problem domains encountered (e.g., IMO-style problems) and to provide general utility tactics not yet present in the main library at the time of development, a focused set of custom additions was developed. This included a small set of custom definitions, approximately 100 elementary theorems and specialized, reusable tactics for common symbolic manipulations (e.g., for numeral and list simplification like `(3/4).num` to `3`). These additions were designed to be general-purpose, addressing common patterns or missing low-level infrastructure, and critically, many have since been accepted and integrated into the public Mathlib library, underscoring their broad utility and alignment with the library's development goals.

Environment Operations To support the demands of AlphaProof, our environment enables the following:

Parallel Execution and State Management: The environment is designed to manage and operate on multiple Lean proof states concurrently across multiple threads. This allows for parallel simulations within the tree search and efficient batching of queries to the proof network. Mechanisms are implemented to save, restore, and uniquely identify Lean tactic states, enabling the tree search to resume search from any previously visited state.

Tactic Validation and Execution: When the proof network proposes a tactic (as a text string), the environment attempts to execute it within the current Lean proof state. For a tactic application to be considered successful, it must not only execute without error, but the resulting proof term must also not employ the `sorry` placeholder used for incomplete proofs, and moreover must be type-correct. To evaluate the latter, goals not directly addressed by the tactic are temporarily "closed" using a private `internalSorry` axiom, in a way that also captures intermediate claims generated by tactics like `have`. The `private` keyword limits the axiom's visibility to discourage its use outside this specific tactic. Ultimate soundness is guaranteed by the resulting closed term being verified by the Lean kernel (see section 5.1.2).

Goal Decomposition (AND-node State Splitting): If a tactic application successfully decomposes the current goal into $N > 1$ independent subgoals (e.g., proving $P \wedge Q$ splits into proving P and proving Q), the environment splits the current Lean tactic state into N distinct child states. In each of these child states, all but one goal are assigned with `internalSorry` to leave a single open goal. This allows each subgoal to be treated as an independent problem by the tree search, forming the basis for AND-node handling in the search algorithm (see section 5.2.2).

Disproving Mechanism: To enable the agent to disprove statements (i.e., prove their negation), the environment provides an operation to transform a goal into its negation. This is achieved by applying a custom tactic that utilizes a private axiom `internal_true_if_false : (α → False) → α`. This tactic reverts all local hypotheses, applies the axiom, and performs cleanup of negated quantifiers, thereby establishing a new context for disproving the original statement, containing exactly the negation of the *fully quantified* goal we were previously in, and do so in a way that still lets our final proof be type-checked by the kernel (see section 5.1.2).

Performance and Stability Considerations Several modifications were implemented to ensure the Lean environment could operate efficiently and robustly under the high throughput demands of RL training. In particular:

- Key Mathlib tactics frequently used by the agent (e.g., `linarith`) were compiled to C code. This results in some tactics executing (i.e. generating proof terms) 6× faster.
- A wall-clock time limit was imposed on individual tactic executions, complementing Lean's internal "heartbeat" limit for computational steps.
- Lean's internal `checkSystem` function, which monitors resource consumption, was invoked more frequently within critical code paths to allow for earlier, safe abortion of overly long or resource-intensive tactic applications.
- The multi-precision integer library within Lean was modified to enforce a hard cap on the size of numerals, preventing runaway computations with extremely large numbers.

Proof Verification - Final Check: To provide absolute assurance of correctness (assuming the soundness of the Lean kernel and the declared axioms), every proof or disproof found by AlphaProof undergoes a final, independent verification step. This involves executing the standard Lean command-line tool on a `.lean` file containing the complete theorem statement and the agent-generated proof. A custom command also verifies that the proof only relies on the three commonly accepted axioms built-in to Lean itself (propositional extensionality, global choice, and soundness of quotient types).

5.2 Prover Agent

5.2.1 Proof Network

The proof network is a 3-billion parameter encoder-decoder transformer model [15], similar to [16]. The encoder takes the pretty-printed Lean tactic state as input and outputs a latent representation of the current proof state. This latent representation serves as input to two distinct heads: the decoder, which acts as the policy and generates K tactics in parallel at inference, and a value head situated atop the encoder. Consistent with prior work [72], the value head parameterizes the value as a categorical distribution, estimating the expected return. Key architectural hyperparameters are summarized in Supplemental Data Table 2.

5.2.2 Tree Search Algorithm

The proof network guides a tree search algorithm to explore the space of possible proof constructions. The tree search is adapted from AlphaZero [2] and Sampled MuZero [18], incorporating several key modifications for formal theorem proving. The search tree consists of nodes representing Lean tactic states and edges representing tactics applicable at those states (extended fig. 1c). Each search iteration involves three phases: selection, expansion, and backpropagation. Key hyperparameters for the search are provided in Supplemental Data Table 3.

Standard Tree Search Framework

Selection: Starting from the root state, a path is traversed to a leaf node by recursively selecting an action a at each state s that maximizes a probabilistic upper confidence tree (PUCT) bound [73, 74], adapted from [2]:

$$Q(s,a) + c(s)\pi^{1/\tau}(a|s)\frac{\sqrt{\sum_b N(s,b)}}{N(s,a)+1}$$

Here, $N(s,a)$ is the visit count for action a from state s. The prior $\pi^{1/\tau}(a|s)$ is derived from the proof network's policy output, modified by a temperature τ. The exploration factor $c(s)$ is given by:

$$c(s) = c_{\text{init}} + \log\left(\frac{N(s) + c_{\text{base}} + 1}{c_{\text{base}}}\right)$$

whereas in prior work [72], the values $Q(s,a)$ were normalised to [0,1] using the maximum and minimum estimated values of all children in state s, in AlphaProof we set

$$Q(s,a) = \gamma^{-V(s,a)-1}$$

where γ is a discounting factor and V(s,a) is the aggregated search value, which has the semantic of the negative number of steps left to resolve the current goal. For state-action pairs which have not been visited in the search already, we set V(s, a) to be equal to the network prediction for the parent node V(s) minus a fixed penalty.

Expansion: When a simulation reaches a leaf node s_L not previously expanded, K tactics are sampled from the proof network's policy $\pi(\cdot|s_L)$. Each tactic is validated in the Lean Environment; invalid tactics are discarded; tactics that lead to the same Lean state (up to renaming and reordering of syntactically identical hypotheses) are merged together, keeping the one that minimizes a cost function linearly depending on string length and execution time. The newly expanded node's value $V(s_L)$ is estimated by the proof network.

Backpropagation: The value estimate from the expanded leaf node is backed up along the selected path, updating the $N(s,a)$ and $V(s,a)$ statistics for all traversed state-action pairs.

Key Adaptations for AlphaProof

Progressive Sampling: To dynamically broaden the search in promising regions, progressive sampling is employed. If a node s encountered during a simulation path satisfies $n(s) \leq CN(s)^\alpha$ (where $n(s)$ is the number of times tactics have previously been sampled at s and $N(s)$ its total visit count), an additional K tactics are sampled from $\pi(\cdot|s)$ and added as new edges to s. This allows AlphaProof to explore a wider range of proof strategies along critical paths.

AND-nodes for Multi-Goal States: To manage proof states with multiple independent subgoals (a logical AND condition), the tree search incorporates AND-nodes, conceptually similar to [17]. An AND-node is introduced when a tactic application decomposes a goal into several independent subgoals (extended fig. 1b). Actions from an AND-node correspond to selecting one of its child subgoal to focus on. During selection at an AND-node, only unproven subgoals are considered for exploration. The PUCB formula is modified to prioritize the unproven subgoal perceived as most difficult (i.e., having the largest $V(s,a)$ or smallest $Q(s,a)$). This is achieved by replacing $Q(s,a)$ in the standard PUCB formula with $1 - Q(s,a)$. The modified selection for an AND-node s_{AND} acting on subgoal a is:

$$(1 - Q(s_{\text{AND}}, a)) + c_{\text{AND}} c(s_{\text{AND}}) \pi(a|s_{\text{AND}}) \frac{\sqrt{\sum_b N(s_{\text{AND}}, b)}}{N(s_{\text{AND}}, a) + 1}$$

The prior $\pi(a|s_{\text{AND}})$ is set to be uniform over all subgoals, and c_{AND} is an additional multiplier.

During backpropagation through an AND-node, the value passed up is the minimum $V(s_{\text{AND}}, a))$ of its children subgoals, consistent with the definition of the return (see section 5.1.1).

Single Search: For each proof attempt on an initial problem statement, AlphaProof executes a single search which terminates if a proof or disproof is found, or if the allocated simulation budget is exhausted. Unlike prior AlphaZero applications, AlphaProof does not commit to intermediate actions and does not restart the search from subsequent states within a single proof attempt. This is possible as theorem proving within a formal system is analogous to a single-player game in a perfectly simulated environment. The agent can retain and expand a single search tree over the entire proof attempt, allowing it to globally allocate its computational resources across all potential proof paths without needing to commit to any intermediate step.

5.3 Training

5.3.1 Pre-training and Supervised Fine-tuning

Prior to RL, the AlphaProof proof network is initialized through a multi-stage training process.

First, the network is **pre-trained** on large datasets consisting of approximately 300 billion tokens of publicly available code and mathematical texts. This phase utilizes a next-token prediction objective with dropout and masked span reconstruction for regularization. The encoder processed approximately 12 trillion tokens while the decoder reconstructed approximately 3 trillion tokens, totaling roughly 50 epochs over the dataset. This stage imbued the policy head (decoder) with a broad understanding of programming syntax, logical structures, and mathematical language.

Following pre-training, the model undergoes **supervised fine-tuning (SFT)**. This stage utilizes a much smaller but more specialized corpus, containing approximately 300,000 state-tactic pairs amounting to approximately 5 million tokens of tactics, extracted from the human-authored proofs within the Mathlib library [4]. This process refines the policy head for Lean-specific tactic generation and initializes the value head to estimate the number of remaining steps in a proof, based on the structure of the Mathlib proofs. Hyperparameters for both pre-training and SFT are detailed in Supplemental Data Table 4.

5.3.2 Auto-Formalization

To provide a sufficiently large and diverse curriculum for RL, AlphaProof relies on an auto-formalization process. This process translates mathematical statements from natural language into the formal language of Lean (see extended fig. 2), generating a dataset of approximately 80 million Lean problems from an initial set of roughly one million natural language statements.

Model and Training Data: The auto-formalization system is built upon a specialized Gemini 1.5 Pro language model finetuned on a set of (natural language problem statement, formalization chain-of-thought, Lean formal statement) triplets. Initial training data comprised approximately 2,500 problems, spanning pre-algebra to IMO-level difficulty, manually formalized in Lean by domain experts. For a subset of ~50 of these problems, experts also provided detailed chain-of-thought (CoT) explanations outlining the formalization steps. This initial seed set was expanded to roughly 7,000 triplets by prompting the public Gemini 1.5 Pro model with the expert-annotated examples to generate additional CoTs for the existing translations; low-quality synthetic CoTs, identified by ad-hoc formatting rule checks, were discarded.

Iterative Refinement: To further enhance the formalization model's accuracy, we employed an iterative refinement strategy analogous to Self-Taught Reasoner (STaR) [59]. This process focused on a subset of problems for which ground-truth ("golden") Lean formalizations were available. The current fine-tuned model repeatedly generated CoTs and Lean formalizations (see Supplementary Pseudocode for the generation process). These outputs were automatically assessed for equivalence against the golden formalizations by tasking a low-compute version of AlphaProof to prove equivalence between the generated and golden statement. Concretely, the Lean statement AlphaProof was certifying was `type_of% @generated = type_of% @golden`. Traces leading to equivalence were added to the training data for subsequent fine-tuning iterations, ultimately expanding the high-quality triplet dataset to approximately 70,000 examples.

Evaluation and Benchmarking: The performance of the auto-formalization model was regularly monitored through A/B studies. These studies compared a baseline model with a test model on two internal benchmarks: one comprising 50 representative IMO problems and another with 50 Putnam problems (Supplemental Data Table 5), both in natural language. For each problem, both models generated a single candidate formalization using a standardized procedure (Supplementary Pseudocode, incorporating majority voting, syntax checking, cycle consistency, and rapid disproof/proof attempts by AlphaProof). These paired outputs were then evaluated by an internal panel of approximately three Lean experts for pointwise correctness and side-by-side preference.

Prior to the IMO 2024 competition, the final auto-formalization process achieved a 60% pass@1 success rate on the 50 representative IMO problems, with notable strengths in algebra (81.3%) and number theory (76.9%), and a lower rate for combinatorics (33.3%). On the 50 representative Putnam problems, the pass@1 rate was 64%, with strong performance in number theory (72.7%), algebra (61.9%) and geometry (54.6%). See details in extended table 1.

Dataset Generation: Several key methods facilitated the generation of a training dataset suitable for formalizing a wide range of mathematical problems. To address problems requiring an answer ("Find all $n\dots$"), plausible answers were gathered alongside questions and injected during auto-formalization. We also continuously augmented the dataset by reapplying the improved formalization pipeline to the source problems. Finally, this sampling process involved producing multiple distinct formal translations for each natural language problem. This strategy ensured a higher likelihood of including a correct formalization, with incorrect ones often being quickly resolved during main RL or sometimes presenting interesting problems in their own right. This resulted in a large and diverse dataset of ~80 million formalized mathematical problems necessary for effective RL training.

5.3.3 Main RL

The main RL phase further trains the proof network, initialized from pre-training and supervised fine-tuning (SFT). The RL training architecture comprises three main components: a centralized matchmaker, distributed actors, and a centralized learner. The RL training was conducted for approximately 1 million training steps (Supplemental Data Table 6).

Start Positions Dataset The training curriculum for this RL phase primarily comprised the ~80 million problem statements generated by the auto-formalization process. This was supplemented by a small fraction (~3.5k problems) of manually human-formalized problems sourced from: (i) the miniF2F-curriculum benchmark [22]; (ii) the set of problems manually formalized for training the auto-formalization model; and (iii) our training split of the Putnam benchmark (PutnamBench-train, derived from [21] by excluding PutnamBench-test).

Matchmaker System The matchmaker orchestrates the training process by assigning tasks to available actors. For each task, it selects a formal problem statement from the start position datasets and randomly assigns the objective as either proving or disproving the statement. Problem selection is prioritized based on "interestingness", determined from the outcomes of the last N attempts recorded for that problem. A problem is deemed highly interesting if: (a) it has not been attempted previously; (b) it has been attempted fewer than a trust_count threshold; or (c) it has been attempted more than trust_count times but exhibits a mixed success rate (i.e., solved in some, but not all, of the last N attempts). Conversely, a problem's priority is reduced if it remains consistently unsolved after trust_count attempts, or if it has been proven for trust_count_proved consecutive attempts, indicating mastery. Statements that are successfully disproved are not retried. The simulation budget allocated to an actor for each attempt is adaptively determined: it starts with a base value and multiplicatively increases with the number of failures within the last N results for that specific problem up to a predefined cap value, thereby dedicating more computational resources to challenging statements. See Supplemental Data Table 7 for hyperparameters.

Actor Experience Generation Each actor, upon receiving a problem statement and simulation budget from the matchmaker, interacts with the Lean Environment. It performs a tree search (see sections 5.2 and 5.2.2) guided by the current iteration of the proof network. Unlike previous work [2], the agent never commits to one action and instead searches until it either finds a Lean-verified proof or disproof, or when its allocated simulation budget is exhausted. The verified outcome is reported back to the matchmaker, which updates its internal statistics for the problem. Proofs and disproofs discovered in the course of search are extracted and sent to the learner, failed attempts are filtered out and do not contribute to network updates.

Learner and Network Updates The learner continuously updates the proof network parameters by training on batches of (state, tactic) pairs drawn with a fixed ratio of 10% from the Mathlib SFT dataset and 90% from a replay buffer filled with proofs and disproofs collected from the actors. The policy head is trained to predict the tactic using a cross-entropy loss. The value head is trained to predict the return obtained at the current proof (sub)goal. This iterative process of experience generation and network updates progressively enhances the proof network's mathematical reasoning capabilities. See Supplemental Data Table 6 for hyperparameters.

Computational Budget Summary The AlphaProof-specific training pipeline, which follows a general pre-training phase, involves several computationally intensive stages. The SFT stage on the Mathlib dataset was comparatively inexpensive (approximately 10 TPUdays). In contrast, the subsequent stages represented a substantial investment. The auto-formalization process, which generated the ~80 million problem curriculum over the course of the project, required approximately 100,000 TPUdays in total (equivalent to, for instance, 10 runs of 2,000 TPUs utilized for 5 days). The main reinforcement learning (RL) training loop, which learned from this curriculum, was of a similar scale, spanning approximately 80,000 TPUdays (equivalent to, for instance, 4,000 TPUs utilized for 20 days).

5.4 Inference

Following main RL training, AlphaProof possesses general proof-finding capabilities for a wide range of mathematical problems. At inference time, when presented with new, unseen theorems to prove, several strategies can be employed to enhance its capabilities by allocating additional computational resources.

5.4.1 Scaling with Tree Search

The primary method for applying more computation at inference, common in the AlphaZero family of agents [2, 72], is scaling the tree search and running multiple independent search attempts. Increasing the number of simulations performed allows the agent to refine the initial tactic suggestions provided by the proof network, integrate information over more lines of deeper reasoning and explore other potential proof sequences. This enhanced search leads to stronger tactic selection and a higher probability of finding a proof. Attempting multiple independent search attempts also increases the probability of finding a proof and offers latency benefits via parallelization. The results presented (fig. 4a, table 1) demonstrate the efficacy of scaling tree search compute, typically measured in compute or wall-clock time, on held-out benchmark performance.

5.4.2 Scaling with TTRL

For problems that remain intractable even with extensive tree search, AlphaProof employs TTRL for deeper, problem-specific adaptation (fig. 2b). This process involves two main stages: the generation of a problem-specific curriculum of variants, and a focused RL phase on this curriculum.

Variant Generation for TTRL Curriculum: The TTRL phase commences with the generation of a problem-specific curriculum of synthetic variants for each target Lean instance T. This process utilizes the Gemini [27] large language model, prompted with few-shot examples from a curated dataset of 791 (problem, variant) Lean pairs. These exemplar pairs illustrate how the LLM can generate variants by emulating problem-solving heuristics that build on classical strategies [75] and their formal counterparts in proof engineering, such as simplification [76], generalization [77], lemma proposal [78], exploring analogies [79], and learning from existing proofs [80]. Based on a sampled prompting strategy, the LLM generates either individual candidate variants or sets of correlated variants (e.g., problem decompositions or simulated proof steps). In addition, programmatically created variants are generated to ensure systematic coverage of local variations (e.g., by systematically altering hypotheses or goals of T). All generated candidates are validated for syntactic correctness in Lean. To enrich this curriculum, an evolutionary refinement process iteratively expands the variant set: promising validated variants (e.g., those identified by high string similarity to T) recursively serve as seeds for further LLM-based generation over up to N_{evo} iterations (where $N_{evo} = 15$ for our reported results). This iterative cycle of LLM-based and programmatic generation, validation, and evolutionary refinement, followed by deduplication, yielded hundreds of thousands of unique, syntactically-valid Lean variants (V_T) for each target T. This comprehensive set of variants forms a localized and structured curriculum for the subsequent RL phase (see Supplementary Pseudocode for procedural details and extended fig. 5 for examples of generated variants).

Focused RL: Following variant generation, AlphaProof executes its core AlphaZero-inspired RL loop, initializing a specialist agent from the main RL generalist model. The key distinction from the main RL phase is the training curriculum: instead of general auto-formalized statements, the TTRL agent focuses on the problem-specific set comprising the original target problem T and its syntactically generated variants V_T. This framework can be concurrently applied to multiple distinct target problems by incorporating all their respective variants into a shared start position dataset for the matchmaker, allowing for simultaneous TTRL across different target problems as presented in fig. 4b. The subsequent learning process mirrors the main RL: distributed actors, coordinated by a matchmaker (see Matchmaker System and Supplemental Data Table 7 for TTRL-specific hyperparameters), repeatedly attempt to prove or disprove these problems. Experience, in the form of (state, tactic) pairs from successful proofs or disproofs, is used to update the specialist proof network's parameters (see Learner and Network Updates and Supplemental Data Table 6 for TTRL-specific hyperparameters). To optimize the computational budget, once a target problem T is successfully proven, the matchmaker ceases assigning new attempts for T and its associated variants V_T, as the primary objective of solving T has been achieved. This problem-specific adaptation allows the agent to fine-tune its learned strategies and potentially discover insights crucial for the target problem.

The TTRL phase thus enables AlphaProof to dedicate substantial, targeted computational resources at inference. By allowing the strategic reapplication of its full RL machinery for problem-specific adaptation, AlphaProof unlocks the ability to solve theorems that are intractable for the fixed, generalist model using standard inference alone. Indeed, this TTRL approach—systematically generating and solving related, often simpler, problem variants to build towards a solution for a complex target problem—is inspired by a core tenet of human mathematical problem-solving, as notably described by Polya [75]. The efficacy of TTRL is demonstrated by the performance gains on held-out benchmarks (fig. 4b, table 1).

6 Evaluation

We evaluated AlphaProof on a comprehensive suite of formal mathematics benchmarks, spanning advanced high-school to elite Olympiad and university-level problems.

6.1 Standard Benchmarks

We introduce **formal-imo**, a benchmark of all non geometry historical IMO problems which were internally formalized by experts (see section 6.3 for an explanation of geometry exclusion). It contains 258 problems: 107 algebra, 77 combinatorics, and 74 number theory problems (based on the official subject when available, or classification by domain experts). Detailed performance per problem of AlphaProof is shown in extended fig. 6. AlphaProof was also evaluated on our internally corrected version of the **miniF2F** benchmark [20], using its "test" and "valid" splits as held-out test sets. These corrections addressed various misformalized problems (e.g., disprovable statements, or statements with contradictions in its hypothesis) in the original dataset. AlphaProof was also evaluated on the Putnam benchmark [21] consisting of challenging undergraduate problems. Our held-out test set, **PutnamBench-test**, comprised 189 problems from even-numbered years (1990 onwards). PutnamBench assigns to each one or more subjects. The most frequent ones in PutnamBench-test were algebra (78), analysis (57), number theory (33), geometry (20), and linear algebra (18). During our work, several misformalizations in the original PutnamBench were identified and subsequently corrected upstream.

6.2 Data Separation and Generalization

To ensure a rigorous evaluation of AlphaProof's generalization capabilities, we carefully managed data separation across its training stages. While the initial pre-training of the underlying language model utilized a broad web corpus that might have incidentally contained natural language discussions or solutions to some historical problems, our pre-training pipeline explicitly excluded all Lean code—and critically, the formal proofs corresponding to our evaluation benchmarks—from this pretraining data. Furthermore, we only trained during the supervised fine-tuning (SFT) phase on Mathlib which does not contain the benchmark problems used for evaluation. The main RL training phase exclusively used the curriculum generated by our auto-formalization process and the supplementary human-formalized problems (see Start Positions Dataset). Crucially, we explicitly removed all documents from the auto-formalization pipeline that were deemed too similar to any problem in the evaluation benchmarks. This data separation protocol was designed so that AlphaProof's performance would reflect learned reasoning capabilities rather than memorization.

6.3 IMO-style Geometry

Handling IMO-style planar geometry within Lean for AlphaProof presented challenges due to specific gaps existing at the time of development in Mathlib's higher-level geometry library (e.g., incircles, congruence), making it impractical to even *state* many questions. Consequently, for dedicated Olympiad geometry problems (like IMO 2024 Problem 4), we employed AlphaGeometry 2 [3], a system specifically designed for modern olympiad geometry. AlphaProof tackled IMO problems with mixed geometric-combinatorial aspects and all PutnamBench problems, where purely planar geometry is rare, using Mathlib's available geometry.

6.4 IMO 2024 Evaluation Protocol and Methods

For the IMO 2024 competition, a specific, pre-defined evaluation protocol was followed to ensure a rigorous assessment of live, out-of-distribution generalization. As part of this, an agreement was established with Prof Sir Timothy Gowers and Dr Joseph Myers to serve as independent expert judges for any solutions produced by our AI system, evaluating them according to official IMO point-awarding rules.

Pre-Competition Freeze: The main RL training of AlphaProof was halted, and all system hyperparameters were frozen prior to the official release of the IMO 2024 problems.

Problem Formalization: Immediately upon release, the five non-geometry IMO 2024 problems were manually formalized into Lean by domain experts. This approach was adopted following consultation with the designated IMO judges, who indicated that the primary evaluation concern was the system's ability to solve the formally stated problems, rather than its capacity for natural language translation of the problem statements under competition conditions. This expert formalization also mirrors the scenario where human contestants may receive problems translated into their native language if needed.

Notably, P5 (combinatorics) proved exceptionally challenging to formalize, with initial attempts by experts being unsuccessful within the first day. Alongside this manual effort, we conducted a side experiment to test our auto-formalization system, which in fact successfully formalized P1, P2, and P6.

For problems requiring an answer (e.g., “Find all…”), placeholders (`sorry`) were used for the answer component in the formalization and the system was responsible for generating candidate answers (see Main Text).

Answer Guessing: Answer generation was specifically optimized to maximize the likelihood of including the correct answer within a large set of proposals (i.e., optimized for pass@k where k was large), even at the cost of generating many plausible but incorrect candidates. We generated $k = 500$ answers, successfully guessing the correct answers for all applicable problems. AlphaProof was then crucial for efficiently sifting through these numerous proposals; in a low-compute refutation mode (within a 10-minute tree search per candidate), it successfully disproved 99% of incorrect Gemini-generated candidates for P1, 98% for P2, and 7% for P6, thereby isolating promising candidates for the subsequent full proof attempts.

Geometry Problem: For reasons described in section 6.3, the geometry problem (IMO 2024 P4) was addressed by AlphaGeometry 2 [3].

Verification and Judging: Solutions generated by AlphaProof (P1, P2, and P6, shown in extended figs. 7 to 9) were first verified for correctness by the Lean kernel. Subsequently, these formal proofs were judged according to the official IMO point-awarding rules by Prof Sir Timothy Gowers and Dr Joseph Myers, who awarded full points for each solved problem.

7 Data availability

The human-authored formal proofs used for supervised fine-tuning were drawn from the publicly available Mathlib library [4]. Our corrected versions of the miniF2F benchmark and the formal-imo benchmark are available at `https://github.com/google-deepmind/miniF2F` and `https://github.com/google-deepmind/formal-imo`, respectively. The Putnam benchmark problems come from commit `02b2cff66d7ac8e9372c778c4ab9ec80082ecbd8` of the publicly available PutnamBench [21]. The formal problem statements that form the primary curriculum for the main RL phase, as well as the problem variants generated for TTRL, were synthetically produced by AlphaProof’s auto-formalization and variant generation components, respectively, as described in the Methods section (sections 5.3.2 and 5.4.2). Similarly, the proofs used to train AlphaProof during RL were generated by the system itself through interaction with the Lean environment as described in the Methods section (sections 5.1 and 5.3.3).

8 Code availability

The pseudocode for the AlphaProof algorithm can be found in the file `alphaproof_pseudocode.py` in the Supplementary Information. Detailed hyperparameters for the training and search procedures are described throughout the Methods section and summarized in Supplementary Data Tables. The Lean 4 proof assistant, which forms the interactive environment for AlphaProof, is open source and available at `https://lean-lang.org/`. The Mathlib library, used extensively by AlphaProof, is also an open-source community project, accessible at `https://github.com/leanprover-community/mathlib4`.

We are making AlphaProof’s capabilities available to members of the scientific community through an interactive tool communicating with Google servers. Researchers interested in gaining access will be able to apply at `https://deepmind.google/alphaproof`.

Methods References

[28] Lawrence C Paulson. *Isabelle: A generic theorem prover.* Springer, 1994.

[29] John Harrison. “HOL Light: A tutorial introduction”. In: *International Conference on Formal Methods in Computer-Aided Design.* Springer. 1996, pp. 265–269.

[30] Bruno Barras et al. “The Coq proof assistant reference manual”. In: *INRIA, version* 6.11 (1999), pp. 17–21.

[31] Georges Gonthier. “The four colour theorem: Engineering of a formal proof”. In: *Asian Symposium on Computer Mathematics.* Springer. 2007, pp. 333–333.

[32] Xavier Leroy et al. “CompCert-a formally verified optimizing compiler”. In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress.* 2016.

[33] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[34] AI Anthropic. “The Claude 3 model family: Opus, Sonnet, Haiku”. In: *Claude-3 Model Card* 1 (2024), p. 1.

[35] Gemini Team et al. *Gemini: A family of highly capable multimodal models.* 2023. arXiv: 2312.11805 [cs.CL].

[36] Zhangir Azerbayev et al. *Proofnet: Autoformalizing and formally proving undergraduate-level mathematics.* 2023. arXiv: 2302.12433 [cs.CL].

[37] Albert Qiaochu Jiang et al. “Lisa: Language models of isabelle proofs”. In: *6th Conference on Artificial Intelligence and Theorem Proving.* 2021, pp. 378–392.

[38] Kshitij Bansal et al. “HOList: An environment for machine learning of higher order logic theorem proving”. In: *International Conference on Machine Learning.* PMLR. 2019, pp. 454–463.

[39] Jesse Michael Han et al. “Proof Artifact Co-Training for Theorem Proving with Language Models”. In: *International Conference on Learning Representations.* 2022.

[40] Minchao Wu et al. “TacticZero: Learning to prove theorems from scratch with deep reinforcement learning”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 9330–9342.

[41] Christian Szegedy, Markus Rabe, and Henryk Michalewski. “Retrieval-augmented proof step synthesis”. In: *Conference on Artificial Intelligence and Theorem Proving (AITP).* Vol. 4. 2021.

[42] Sean Welleck et al. “NaturalProver: Grounded mathematical proof generation with language models”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 4913–4927.

[43] Albert Qiaochu Jiang et al. “Thor: Wielding hammers to integrate language models and automated theorem provers”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 8360–8373.

[44] Emily First et al. “Baldur: Whole-proof generation and repair with large language models”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2023, pp. 1229–1241.

[45] Albert Q Jiang et al. “Draft, sketch, and prove: Guiding formal theorem provers with informal proofs”. In: *Proceedings of the International Conference on Learning Representations* (2023).

[46] Xueliang Zhao, Wenda Li, and Lingpeng Kong. “Subgoal-based demonstration learning for formal theorem proving”. In: *International Conference on Machine Learning.* 2024.

[47] Haiming Wang et al. “DT-Solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* 2023, pp. 12632–12646.

[48] Huajian Xin et al. *DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search.* 2024. arXiv: 2408.08152 [cs.CL].

[49] Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. “First experiments with neural translation of informal to formal mathematics”. In: *Intelligent Computer Mathematics: 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings 11.* Springer. 2018, pp. 255–270.

[50] Qingxiang Wang et al. “Exploration of neural machine translation in autoformalization of mathematics in Mizar”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs.* 2020, pp. 85–98.

[51] Yuhuai Wu et al. “Autoformalization with Large Language Models”. In: *Advances in Neural Information Processing Systems.* Ed. by S. Koyejo et al. Vol. 35. 2022, pp. 32353–32368.

[52] Ayush Agrawal et al. *Towards a mathematics formalisation assistant using large language models.* 2022. arXiv: 2211.07524 [cs.CL].

[53] Siddhartha Gadgil et al. “Towards automating formalisation of theorem statements using large language models”. In: *36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on MATH-AI.* 2022.

[54] Albert Q Jiang, Wenda Li, and Mateja Jamnik. “Multi-language diversity benefits autoformalization”. In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 83600–83626.

[55] Huaiyuan Ying et al. “Lean workbook: A large-scale lean problem set formalized from natural language math problems”. In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 105848–105863.

[56] Zenan Li et al. “Autoformalize mathematical statements by symbolic equivalence and semantic consistency”. In: *Advances in Neural Information Processing Systems* (2024).

[57] Jianqiao Lu et al. "FormalAlign: Automated Alignment Evaluation for Autoformalization". In: *Proceedings of the International Conference on Learning Representations* (2025).

[58] Jason Wei et al. "Chain-of-thought prompting elicits reasoning in large language models". In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.

[59] Eric Zelikman et al. "Star: Bootstrapping reasoning with reasoning". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 15476–15488.

[60] OpenAI. *OpenAI o1 System Card.* 2024. arXiv: `2412.16720 [cs.AI]`.

[61] OpenAI et al. *Competitive programming with large reasoning models.* 2025. arXiv: `2502.06807 [cs.LG]`.

[62] Andy L Jones. *Scaling scaling laws with board games.* 2021. arXiv: `2104.03113 [cs.LG]`.

[63] Yu Sun et al. "Test-time training with self-supervision for generalization under distribution shifts". In: *International conference on machine learning.* PMLR. 2020, pp. 9229–9248.

[64] Moritz Hardt and Yu Sun. *Test-time training on nearest neighbors for large language models.* 2023. arXiv: `2305.18466 [cs.CL]`.

[65] Ekin Akyürek et al. "The surprising effectiveness of test-time training for abstract reasoning". In: *In International Conference on Machine Learning* (2025).

[66] David Silver. "Reinforcement learning and simulation-based search in computer Go". In: (2009).

[67] Tom Zahavy et al. *Diversifying AI: Towards creative chess with AlphaZero.* 2023. arXiv: `2308.09175 [cs.AI]`.

[68] Yuxin Zuo et al. *TTRL: Test-time reinforcement learning.* 2025. arXiv: `2504.16084 [cs.CL]`.

[69] Toby Simonds and Akira Yoshiyama. *LADDER: Self-improving LLMs through recursive problem decomposition.* 2025. arXiv: `2503.00735 [cs.LG]`.

[70] AlphaProof and AlphaGeometry teams. *AI achieves silver-medal standard solving International Mathematical Olympiad problems.* July 2024. URL: `https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/`.

[71] Thierry Coquand and Christine Paulin. "Inductively defined types". In: *International Conference on Computer Logic.* Springer. 1988, pp. 50–66.

[72] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609.

[73] David Silver et al. "Mastering the Game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[74] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), pp. 354–359.

[75] George Polya. "How to solve it: A new aspect of mathematical method". In: *How to solve it.* Princeton university press, 2014.

[76] Georges Gonthier and Assia Mahboubi. "An introduction to small scale reflection in Coq". In: *Journal of formalized reasoning* 3.2 (2010), pp. 95–152.

[77] Robert W Hasker and Uday S Reddy. "Generalization at higher types". In: *Proceedings of the Workshop on the λProlog Programming Language.* 1992, pp. 257–271.

[78] Jónathan Heras et al. "Proof-pattern recognition and lemma discovery in ACL2". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning.* Springer. 2013, pp. 389–406.

[79] Erica Melis and Jon Whittle. "Analogy in inductive theorem proving". In: *Journal of Automated Reasoning* 22.2 (1999), pp. 117–147.

[80] Thibault Gauthier et al. "TacticToe: learning to prove with tactics". In: *Journal of Automated Reasoning* 65.2 (2021), pp. 257–286.

9 Acknowledgements

We thank our colleagues Tom Eccles, Eser Aygün, Zhitao Gong, Richard Evans, Soňa Mokrá, Amin Barekatain, Wendy Shang, Hannah Openshaw, and Felix Gimeno for their contributions during earlier stages of the AlphaProof project. We are grateful to Andrew Yang, Lawrence Wu, Luigi Massacci, Thomas Murrills, Moritz Firsching, Dan Shved, and Bogdan Georgiev for their invaluable Lean expertise, formalization work, and reviews. We extend our sincere appreciation to the vibrant Lean and Mathlib open-source communities for developing and maintaining the foundational tools and libraries that made this research possible. Finally, we thank the entire Google DeepMind team for their continuous support, encouragement, and insightful discussions throughout this endeavor.

10 Author Contributions

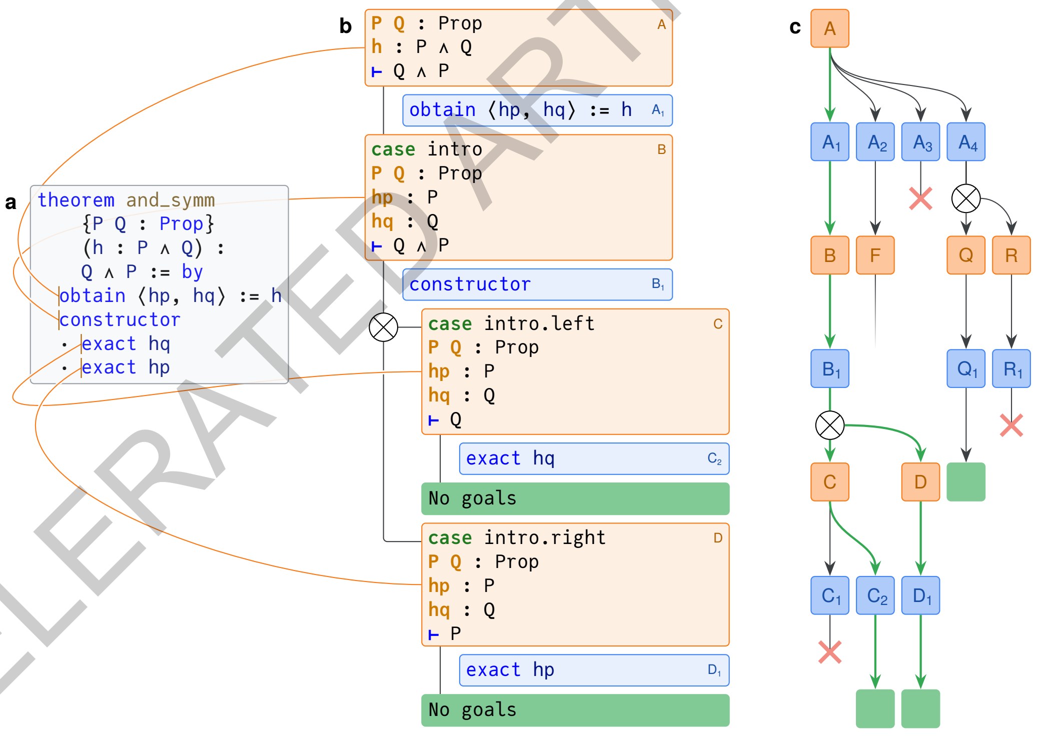
T.H. and J.S. conceived and initiated the AlphaProof project. T.H., R.M., and L.S. provided primary scientific, technical and strategic leadership over the course of the project. H.Ma. and O.B. led project management and operations. D.H, P.K. and D.S. provided overall project sponsorship and senior strategic advice.

The Lean Environment development and integration was led by L.S. and E.W., with significant contributions from A.M., V.V., and J.Y. Formal Benchmarks were curated and managed by E.W., O.N., P.L., S.M., C.S., B.M., M.Z.H., T.Z. and V.V. Informal datasets were curated by M.Z.H., R.M., G.Z., T.Z., V.V., A.V. and H.Mi. Proof Network design and pre-training were led by J.S. The tree search algorithm design and adaptations were developed by J.S., T.H. , A.H. and M.R. Auto-formalization research was led by G.Z. and R.M. with significant contributions made by J.Y., I.B., L.S, L.Y., T.Z., V.V., Y.S., A.D., D.Z., and E.W. Expert evaluation of auto-formalization outputs was conducted by O.N., P.L., S.M., C.S., B.M. The Main RL framework was led by T.H. and R.M., with significant contributions from Y.S., J.S., A.H., B.I., E.L. and E.H. TTRL was conceived and developed by M.Z.H. with significant contributions from T.Z., R.M., T.H., G.Z., I.B., N.S., S.A., and S.S. For the IMO 2024 effort, O.B. led organisation and stakeholder relations. Formalizations and reviews were led by E.W., O.N., P.L., S.M., C.S., B.M. Answer guessing was developed by A.D., D.Z., Y.L., F.P. and I.v.G. The overall system and coordination for the IMO 2024 effort was led by T.H., R.M., L.S. Paper Preparation was led by T.H., with significant contributions from M.Z.H., L.S., G.Z., E.W., M.R., L.Y., and P.L. All authors contributed to reviewing and editing the manuscript.

11 Competing interests

The authors declare no competing financial interests.

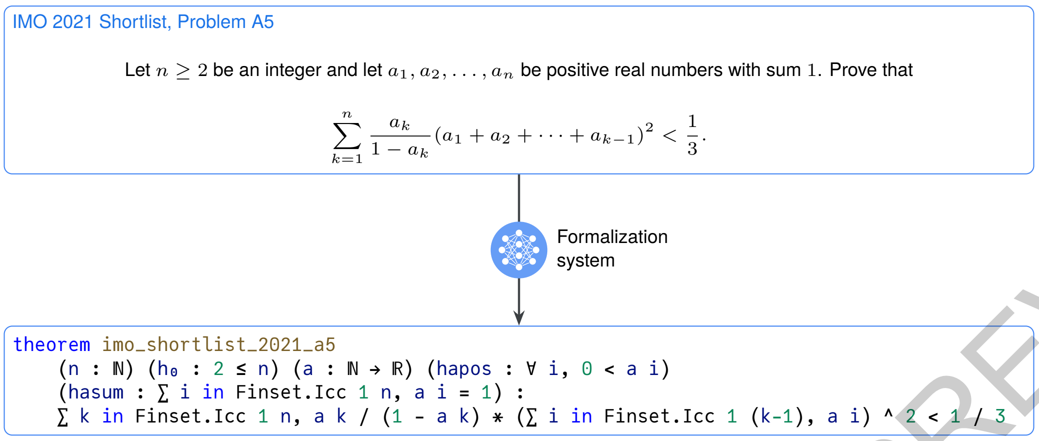
12 Extended Figures / DataTable



```
theorem and_symm
    {P Q : Prop}
    (h : P ∧ Q) :
    Q ∧ P := by
  obtain ⟨hp, hq⟩ := h
  constructor
  · exact hq
  · exact hp
```

```
P Q : Prop
h : P ∧ Q
⊢ Q ∧ P

obtain ⟨hp, hq⟩ := h

case intro
P Q : Prop
hp : P
hq : Q
⊢ Q ∧ P

constructor

case intro.left
P Q : Prop
hp : P
hq : Q
⊢ Q

exact hq

No goals

case intro.right
P Q : Prop
hp : P
hq : Q
⊢ P

exact hp

No goals
```

Extended Data Figure 1: **The Lean tactic environment and its framing for tree search. a**, The Lean editor environment, with tactic state information associated with cursor positions. **b**, A corresponding proof tree, linking observations (tactic states) through actions (tactics). The ⊗ symbol indicates where there are multiple subgoals which must all be solved. **c**, The search tree, with the successful proof found highlighted in thick green arrows.


IMO 2021 Shortlist, Problem A5

Let $n \geq 2$ be an integer and let $a_1, a_2, \ldots, a_n$ be positive real numbers with sum 1. Prove that

$$\sum_{k=1}^{n} \frac{a_k}{1-a_k}(a_1 + a_2 + \cdots + a_{k-1})^2 < \frac{1}{3}.$$

```
theorem imo_shortlist_2021_a5
    (n : ℕ) (h₀ : 2 ≤ n) (a : ℕ → ℝ) (hapos : ∀ i, 0 < a i)
    (hasum : ∑ i in Finset.Icc 1 n, a i = 1) :
    ∑ k in Finset.Icc 1 n, a k / (1 - a k) * (∑ i in Finset.Icc 1 (k-1), a i) ^ 2 < 1 / 3
```

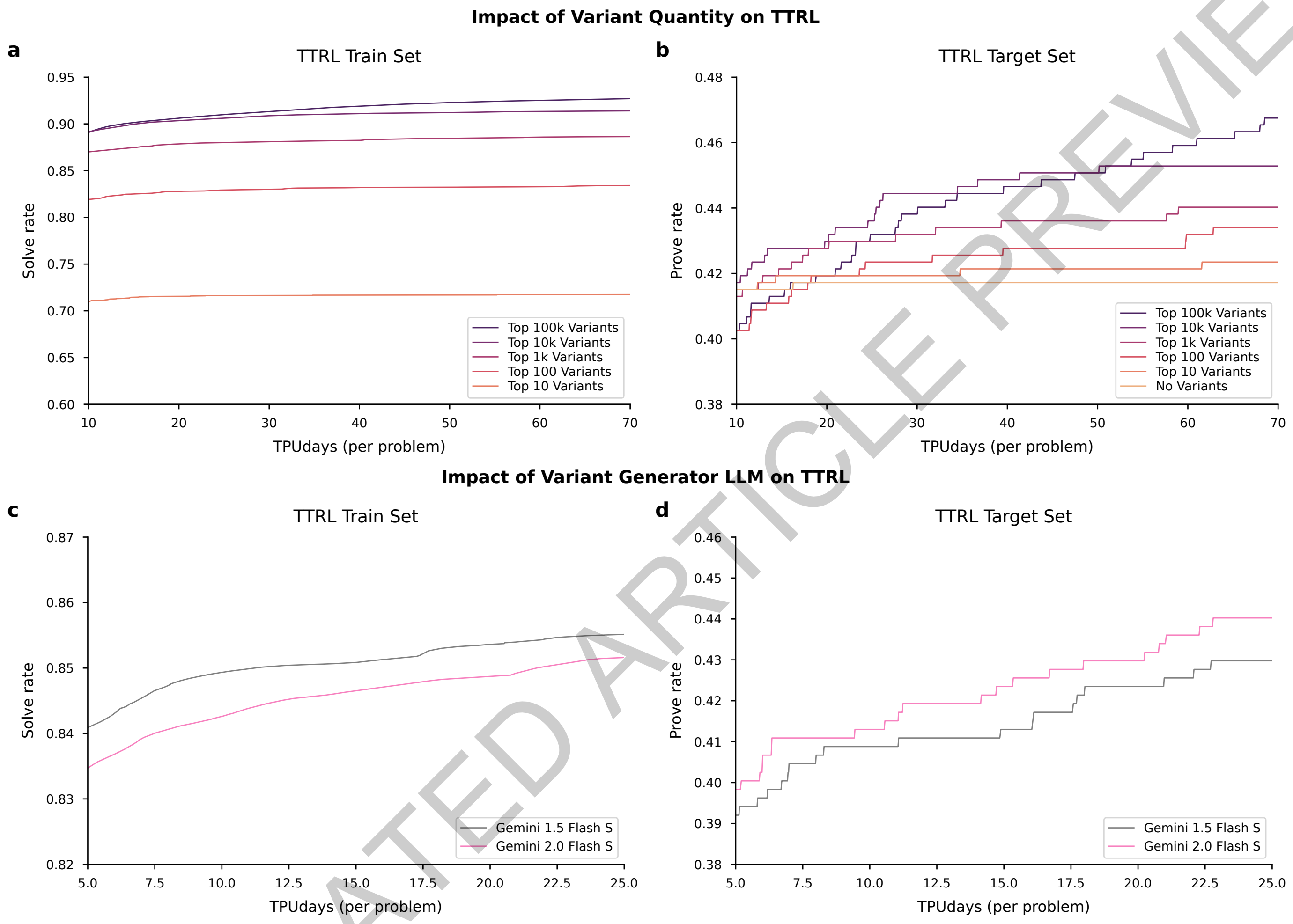
Extended Data Figure 2: **An example of the auto-formalization process.** The formalization system in fig. 2a receives the LaTeX-formatted natural language text of a problem statement, and outputs formal statements as valid Lean code.

	Algebra	Number theory	Combinatorics	Geometry	All areas	Pass@k (k studies)
IMO	81.3%	76.9%	33.3%	(Not included)	60.0%	96% (k=29)
Putnam	61.9%	72.7%	60.0%	54.6%	64.0%	98% (k=16)

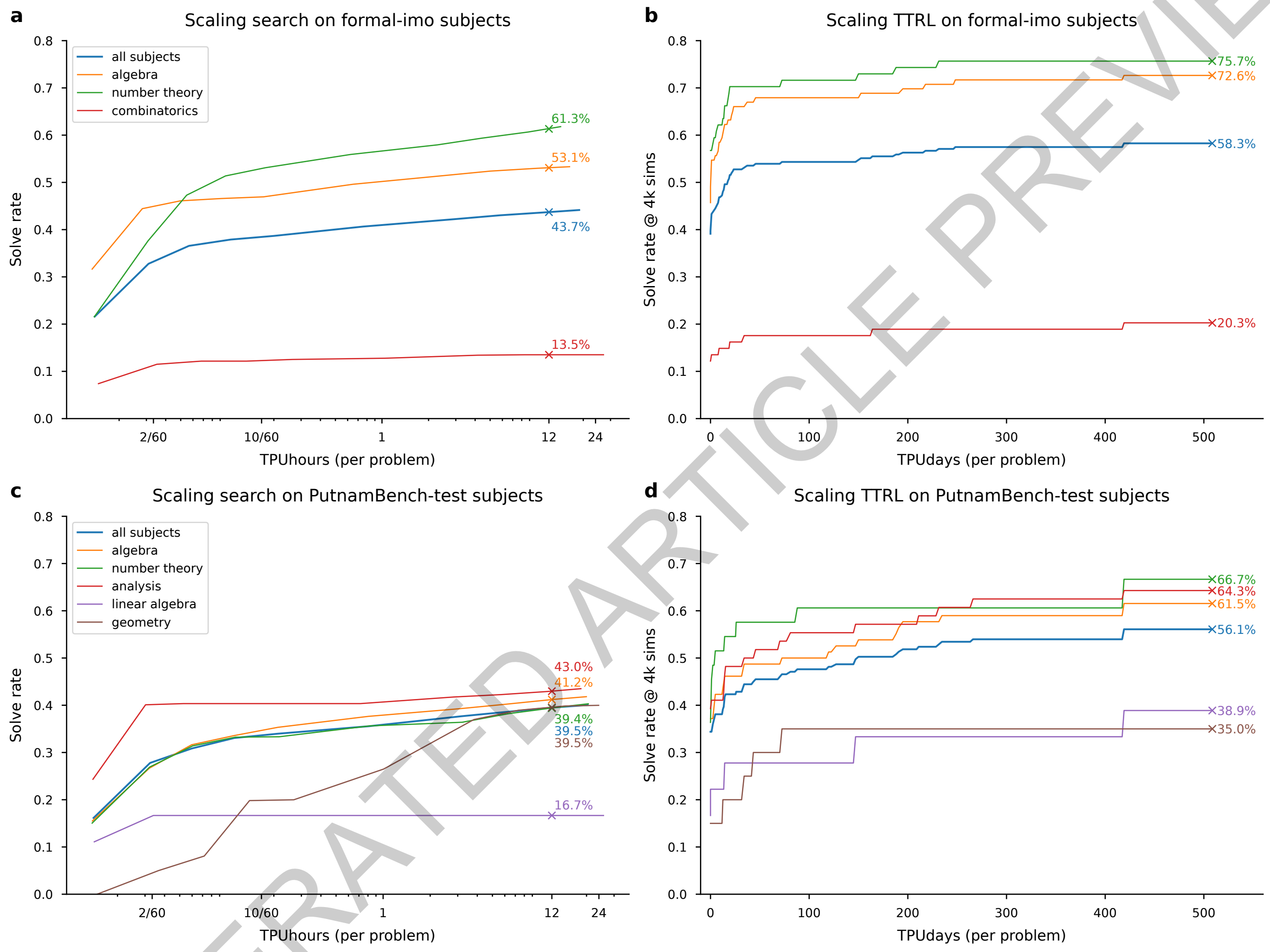
Extended Data Table 1: **Performance of the auto-formalization process on internal benchmarks of fifty representative IMO and Putnam problems**. The exact problems constituting these benchmarks are listed in Supplemental Data Table 5. All numbers are pass@1, except the last column which gives the proportion of problems correctly auto-formalized in any one of our k studies, as rated by Lean experts.

	Compute Budget (per problem)	miniF2F-valid	miniF2F-test	formal-imo	PutnamBench-train	PutnamBench-test
AlphaProof	2 TPUmins	96.0%	96.3%	33.2%	35.4%	27.9%
	12 TPUhours	97.1%	97.7%	43.7%	48.7%	39.5%
AlphaProof with TTRL	50 TPUdays	99.6%	97.5%	53.9%	–	45.5%
	500 TPUdays	100.0%	99.6%	58.3%	–	56.1%

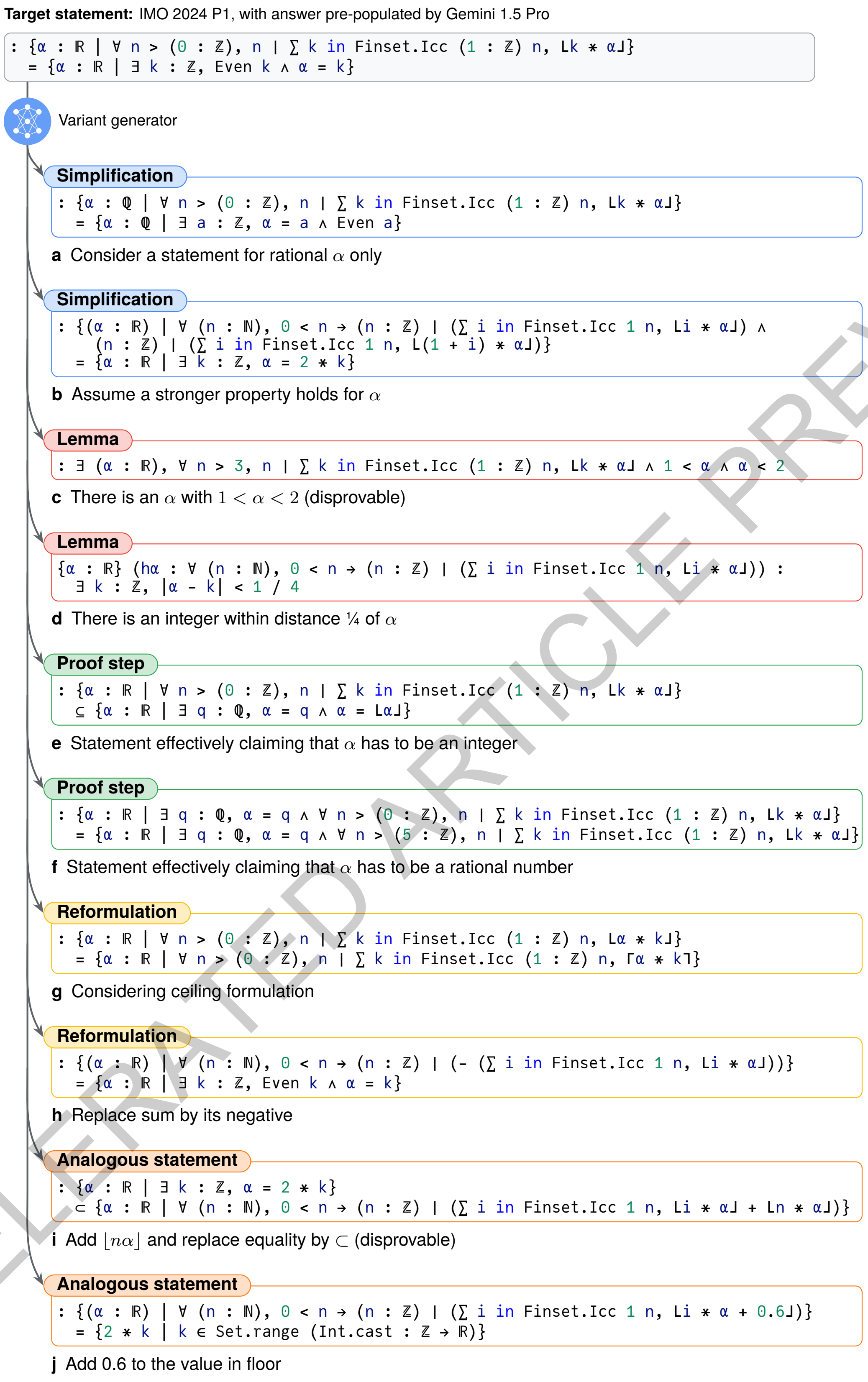
Extended Data Table 2: **Detailed performance of AlphaProof configurations across all benchmarks.** The “compute budget (per problem)” refers to the average computational cost as defined in fig. 4a. 100% of the miniF2F [20] valid split was proved. We observe similar scaling properties of tree search on PutnamBench-train and PutnamBench-test [21]. TTRL was not run on the PutnamBench-train set.



Extended Data Figure 3: **TTRL ablations when varying the quantity of synthetic variants and with different variant generator LLMs. a** Increasing the number of variants in the curriculum (from "Top 10" to "Top 100k") leads to a higher proportion of solved variants int the TTRL training set. **b** More synthetic variants consistently improves the final prove rate on the target problems, demonstrating the significant benefit of a larger set of problem-specific variants. **c** The variants in the TTRL train set from Gemini 1.5 Flash S (grey) show a higher solve rate than those from the stronger Gemini 2.0 Flash S (pink), suggesting Gemini 2.0 Flash S's variants may form a more challenging learning curriculum. **d** Using Gemini 2.0 Flash S for variant generation results in a notably higher prove rate on the target problems, indicating that the learning curriculum is indeed more effective.



Extended Data Figure 4: **AlphaProof inference scaling dynamics by mathematical subject on formal-imo and PutnamBench-test benchmarks**. Scaling of solve rates with tree search compute and TTRL on formal-imo and PutnamBench-test, broken down by subject. The thick blue line represents the overall benchmark performance, while thinner lines show performance on specific subjects.



Target statement: IMO 2024 P1, with answer pre-populated by Gemini 1.5 Pro

```
: {α : ℝ | ∀ n > (0 : ℤ), n ∣ ∑ k in Finset.Icc (1 : ℤ) n, ⌊k * α⌋}
  = {α : ℝ | ∃ k : ℤ, Even k ∧ α = k}
```

Variant generator

Simplification

```
: {α : ℚ | ∀ n > (0 : ℤ), n ∣ ∑ k in Finset.Icc (1 : ℤ) n, ⌊k * α⌋}
  = {α : ℚ | ∃ a : ℤ, α = a ∧ Even a}
```

a Consider a statement for rational α only

Simplification

```
: {(α : ℝ) | ∀ (n : ℕ), 0 < n → (n : ℤ) ∣ (∑ i in Finset.Icc 1 n, ⌊i * α⌋) ∧
    (n : ℤ) ∣ (∑ i in Finset.Icc 1 n, ⌊(1 + i) * α⌋)}
  = {α : ℝ | ∃ k : ℤ, α = 2 * k}
```

b Assume a stronger property holds for α

Lemma

```
: ∃ (α : ℝ), ∀ n > 3, n ∣ ∑ k in Finset.Icc (1 : ℤ) n, ⌊k * α⌋ ∧ 1 < α ∧ α < 2
```

c There is an α with $1 < \alpha < 2$ (disprovable)

Lemma

```
{α : ℝ} (hα : ∀ (n : ℕ), 0 < n → (n : ℤ) ∣ (∑ i in Finset.Icc 1 n, ⌊i * α⌋)) :
  ∃ k : ℤ, |α - k| < 1 / 4
```

d There is an integer within distance ¼ of α

Proof step

```
: {α : ℝ | ∀ n > (0 : ℤ), n ∣ ∑ k in Finset.Icc (1 : ℤ) n, ⌊k * α⌋}
  ⊆ {α : ℝ | ∃ q : ℚ, α = q ∧ α = ⌊α⌋}
```

e Statement effectively claiming that α has to be an integer

Proof step

```
: {α : ℝ | ∃ q : ℚ, α = q ∧ ∀ n > (0 : ℤ), n ∣ ∑ k in Finset.Icc (1 : ℤ) n, ⌊k * α⌋}
  = {α : ℝ | ∃ q : ℚ, α = q ∧ ∀ n > (5 : ℤ), n ∣ ∑ k in Finset.Icc (1 : ℤ) n, ⌊k * α⌋}
```

f Statement effectively claiming that α has to be a rational number

Reformulation

```
: {α : ℝ | ∀ n > (0 : ℤ), n ∣ ∑ k in Finset.Icc (1 : ℤ) n, ⌊α * k⌋}
  = {α : ℝ | ∀ n > (0 : ℤ), n ∣ ∑ k in Finset.Icc (1 : ℤ) n, ⌈α * k⌉}
```

g Considering ceiling formulation

Reformulation

```
: {(α : ℝ) | ∀ (n : ℕ), 0 < n → (n : ℤ) ∣ (- (∑ i in Finset.Icc 1 n, ⌊i * α⌋))}
  = {α : ℝ | ∃ k : ℤ, Even k ∧ α = k}
```

h Replace sum by its negative

Analogous statement

```
: {α : ℝ | ∃ k : ℤ, α = 2 * k}
  ⊂ {α : ℝ | ∀ (n : ℕ), 0 < n → (n : ℤ) ∣ (∑ i in Finset.Icc 1 n, ⌊i * α⌋ + ⌊n * α⌋)}
```

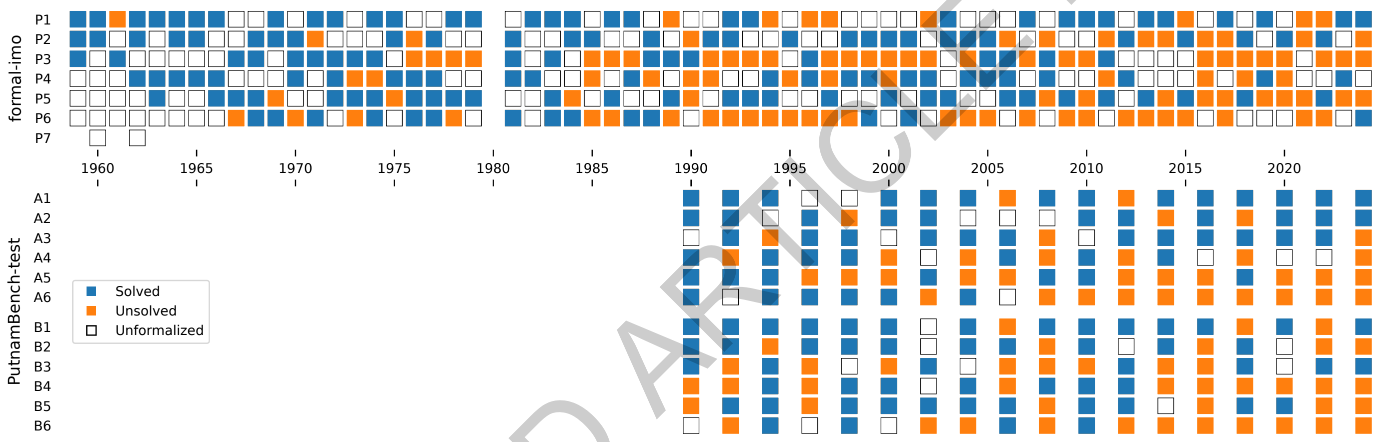
i Add $\lfloor n\alpha \rfloor$ and replace equality by $\subset$ (disprovable)

Analogous statement

```
: {(α : ℝ) | ∀ (n : ℕ), 0 < n → (n : ℤ) ∣ (∑ i in Finset.Icc 1 n, ⌊i * α + 0.6⌋)}
  = {2 * k | k ∈ Set.range (Int.cast : ℤ → ℝ)}
```

j Add 0.6 to the value in floor

Extended Data Figure 5: **Example synthetic variant problems generated by AlphaProof automatically for the IMO 2024 P1 problem.** These Lean variants, created automatically during the TTRL phase, are presented here with labels and descriptions by the authors to illustrate diverse problem-solving heuristics such as simplification, lemma proposal, proof steps, reformulation, and exploring analogies. Engaging with such variants provides insights into the target statement and facilitates problem-specific adaptation of the proof agent.



Extended Data Figure 6: **Performance of AlphaProof at the end of the TTRL phase on the formal-imo and PutnamBench-test benchmarks.** In blue, problems solved, in orange problems not solved and in white problems not formalized. The IMO competition was not held in 1980 for political reasons.

```
import Mathlib

/--
Determine all real numbers α such that, for every positive integer n, the integer

                              ⌊α⌋ + ⌊2α⌋ + ⋯ + ⌊nα⌋

is a multiple of n.

(Note that ⌊z⌋ denotes the greatest integer less than or equal to z. For example, ⌊−π⌋ = −4 and ⌊2⌋ = ⌊2.9⌋ = 2.)
-/
theorem imo_2024_p1 :
   {(α : ℝ) | ∀ (n : ℕ), 0 < n → (n : ℤ) ∣ (∑ i in Finset.Icc 1 n, ⌊i * α⌋)}
   = {α : ℝ | ∃ k : ℤ, Even k ∧ α = k} := by
  rw [(Set.Subset.antisymm_iff ), (Set.subset_def), ]
  existsλx L=>(L 2 two_pos).rec λl Y=>?_
  useλy . x=>y.rec λS p=>?_
  · simp_all[λL:ℕ=>(by norm_num[Int.floor_eq_iff]:⌊(L:ℝ)*S⌋=L* S )]
    rw[p.2,Int.dvd_iff_emod_eq_zero,Nat.lt_iff_add_one_le,<-Finset.sum_mul,←Nat.cast_sum, S.even_iff,
      ↪ ←Nat.Ico_succ_right,@ .((( Finset.sum_Ico_eq_sum_range))),Finset.sum_add_distrib ]at*
    simp_all[Finset.sum_range_id]
    exact dvd_trans ⟨2+((_:ℕ)-1),by linarith[((‹ℕ›:Int)*(‹Nat›-1)).ediv_mul_cancel$ Int.prime_two.dvd_mul.2<|by ·omega]⟩
      ↪ ↑↑(mul_dvd_mul_left @_ (p))
  suffices : ∀ (n : ℕ),⌊(n+1)*x⌋ =⌊ x⌋+2 * ↑ (n : ℕ) * (l-(⌊(x)⌋))
  · zify[mul_comm,Int.floor_eq_iff] at this
    use(l-⌊x⌋)*2
    norm_num
    apply@le_antisymm
    use not_lt.1 (by cases exists_nat_ge (1/(x-_)) with|_ N =>nlinarith[ one_div_mul_cancel $ sub_ne_zero.2
      ↪ ·.ne',9,Int.floor_le x, this N])
    use not_lt.1 (by cases exists_nat_ge (1/_:ℝ)with|_ A=>nlinarith[Int.lt_floor_add_one x,one_div_mul_cancel$
      ↪ sub_ne_zero.2 ·.ne',this A])
  intro
  induction‹_› using@Nat.strongInductionOn
  specialize L$ ‹_›+1
  simp_all[add_comm,mul_assoc,Int.floor_eq_iff,<-Nat.Ico_succ_right, add_mul,(Finset.range_succ),
    ↪ Finset.sum_Ico_eq_sum_range]
  revert‹ℕ›
  rintro A B@c
  simp_all[ Finset.mem_range.mp _,←eq_sub_iff_add_eq',Int.floor_eq_iff]
  suffices:∑d in .range A,⌊x+d*x⌋=∑Q in .range A,(⌊x⌋+2*(Q * (l-.floor x)))
  · suffices:∑d in ( .range A),(((d)):ℤ) =A * ( A-1)/2
    · have:(A : ℤ) * (A-1)%2=0
      · cases@Int.emod_two_eq A with|_ B=>norm_num[B,Int.sub_emod,Int.mul_emod]
      norm_num at*
      norm_num[ Finset.sum_add_distrib,<-Finset.sum_mul, ←Finset.mul_sum _ _] at*
      rw[eq_sub_iff_add_eq]at*
      zify[←mul_assoc, this,←eq_sub_iff_add_eq',‹_ =(@ _) /@_›,Int.floor_eq_iff] at *
      zify[*]at*
      cases S5:lt_or_ge c (A * (l-.floor ↑x)+⌊x⌋ + 1)
      · simp_all
        have:(c+1:ℝ)<=A*(l-⌊x⌋)+⌊x⌋+1:=by norm_cast
        simp_all
        cases this.eq_or_lt
        · repeat use by nlinarith
        nlinarith[(by norm_cast at* :(A*(l -⌊x⌋):ℝ)+⌊(x)⌋ >=(c)+01),9,Int.add_emod ↑5,Int.floor_le (@x :
          ↪ ℝ),Int.lt_floor_add_one (x:)]
      simp_all
      nlinarith[(by norm_cast:(c:ℝ)>=A*(l-⌊_⌋)+⌊_⌋+1),Int.floor_le x,Int.lt_floor_add_one x]
    rw [←Nat.cast_sum, mul_sub, Finset.sum_range_id]
    cases A with|_=>norm_num[mul_add]
  use Finset.sum_congr rfl<|by simp_all[add_comm,Int.floor_eq_iff]
```

Extended Data Figure 7: **A complete proof of IMO 2024 P1 found by AlphaProof.** This was found following the protocol in section 6.4. An interactive version of this figure (along with extended figs. 8 and 9), which allows inspecting all the intermediate Lean goal states is available in [70]. This algebra problem was fully solved by 413 of the 609 student contestants [26].

```
import Mathlib

open scoped Nat

/--
Determine all pairs (a, b) of positive integers for which there exist positive integers g and N such that

                              gcd(a^n + b, b^n + a) = g

holds for all integers n ≥ N. (Note that gcd(x, y) denotes the greatest common divisor of integers x and y.)
-/
theorem imo_2024_p2 : {(a, b) | 0 < a ∧ 0 < b ∧ ∃ g N, 0 < g ∧ 0 < N ∧ ∀ n ≥ N, Nat.gcd (a ^ n + b) (b ^ n + a) = g} =
  ↪ {(1, 1)} := by
  induction(10)+2
  · use Set.eq_singleton_iff_unique_mem.2 ⟨?_,λb g=>by_contra$ g.2.2.rec λY S i=>S.rec λL D=>?_⟩
    · exact⟨by left,by left,2,3,by simp_all⟩
    have:b.1+b.2|Y:=?_
    · suffices: b.1= b.2
      · norm_num[b.ext_iff,<-D.2.2 L,this]at*
        use(pow_lt_pow (g.1.nat_succ_le.lt_of_ne' i) (by left)).ne' (D.2.2 _ L.le_succ)
      suffices:b.1+b.2|b.fst^ (2 *L) +b.2 ∧(b).fst +(b).snd | b.snd^ (2 *L)+b.1
      · suffices:b.1^2%(b.1+b.2)=b.2^2%(b.1+b.snd)
        · norm_num[Nat.add_mod,pow_mul,this,Nat.dvd_iff_mod_eq_zero,Nat.pow_mod]at*
          norm_num[add_comm,b.ext_iff,sq _,←Nat.pow_mod,←Nat.dvd_iff_mod_eq_zero]at*
          zify at*
          cases this.1.sub this.2with|_ Z=> nlinarith [ (by (nlinarith): Z=0 )]
        apply@Nat.modEq_of_dvd
        use(b.snd)-b.fst , (by·ring: ( (b.snd) : ℤ)^2-b.fst^2=(b.fst+(b).2) * _)
      norm_num[(2).le_mul_of_pos_left,Nat.gcd_dvd,← D.2.2 (2 *L), this.trans, (D.right.1 :_)]
    suffices:b.1+b.2|b.1^(2*L)+b.2 ∧b.1+b.2 |b.snd^ (2 *L) +b.1
    · exact D.2.2 (2 *(L )) (le_mul_of_one_le_left' (by decide ) )▸dvd_gcd (this.left) (this).2
    exfalso
    suffices:b.1*b.2+1|Y
    · suffices:b.1^φ (b.1*b.2+1)%(b.1*b.2+1)=1%(b.1*b.2+1) ∧b.2^ φ (b.1* b.snd+1)%((b).1 * ↑(b.snd)+1)= 1% (b.1*b.snd +
      ↪ 1)
      · absurd D.2.2 (φ (b.1*b.2+1)*L) (by nlinarith [((b.fst *b.2+1).totient_pos).2 ↑ Fin.size_pos'])
        apply mt (.▸Nat.gcd_dvd _ _)
        useλH=>absurd (‹_|Y›.trans H.1) (λv=>absurd (‹_|Y›.trans H.2) ? _)
        norm_num[pow_mul,b.ext_iff,(1).mod_eq_of_lt,g.symm,this,Nat.add_mod,Nat.dvd_iff_mod_eq_zero,Nat.pow_mod]at(i)v⊢
        norm_num[add_comm,pow_mul,<-Nat.dvd_iff_mod_eq_zero]at*
        contrapose! i
        zify at*
        repeat use by nlinarith[Int.le_of_dvd (by linarith) v,Int.le_of_dvd (by linarith) i]
      repeat use↑(Nat.ModEq.pow_totient (by norm_num))
    by_contra! H
    suffices:b.1^φ (b.1*b.2+1)%(b.1*b.2+1)=1%(b.1*b.2+1) ∧b.2^φ (b.1*b.2+1)%(b.1*b.2+1)=1%( b.fst * ↑ (b.snd)+1)
    · simp_all
      suffices:b.1*b.2+1|b.1^(φ (b.1*b.2+1)*(L+1)-1)+b.2 ∧b.1*b.2+1|b.2^(φ (b.1* b.2+1)* (L+1)-1)+(b.fst)
      · use H$ D.2.2 (φ _ *(L+1)-1) (L.le_sub_of_add_le (by nlinarith[((b.1* b.2+1).totient_pos).2
        ↪ Nat.succ_pos'']))▸(((Nat.dvd_gcd) ( this).1)) this.right
      cases B:Nat.exists_eq_add_of_lt$ ((b.1*b.2+1).totient_pos).2 (by continuity)
      norm_num[*, g, ‹φ _ = _›,
        ↪ mul_add,Nat.pow_mod,(1).mod_eq_of_lt,pow_add,Nat.add_mod,pow_mul,Nat.dvd_iff_mod_eq_zero,Nat.mul_mod] at this⊢
      simp_all
      suffices:b.1*b.2+1|b.1*( (b.1%((b).1 * ( b.snd) + 1) : _)^‹Nat› +b.snd) ∧(b.fst * ↑(b.snd) + 1)|(b).snd*(
        ↪ (b.snd%((b).fst * b.snd + 1))^ ‹Nat›+b.fst)
      · norm_num[<-Nat.dvd_iff_mod_eq_zero,g,(1).mod_eq_of_lt,Nat.dvd_mul] at this⊢
        exists@?_
        · cases this.1 with|_ Q r=>simp_all[(Q.dvd_gcd r.1 ⟨_,.symm r.right.choose_spec.2⟩).antisymm]
        cases@this.2with|_ F X=>simp_all[(F.dvd_gcd X.1 ⟨_,symm X.2.choose_spec.2⟩).antisymm]
      simp_all[mul_comm, mul_add,add_comm,Nat.add_mod,Nat.dvd_iff_mod_eq_zero]
    repeat use(Nat.ModEq.pow_totient (by . . .norm_num) )
  congr 26
```

Extended Data Figure 8: **A complete proof of IMO 2024 P2 found by AlphaProof.** This was found following the protocol in section 6.4. The crux of this proof is considering the properties of $ab + 1$ (e.g. that it divides g), which can be seen on the highlighted line. This number theory problem was fully solved by 156 of the 609 student contestants [26].

```
import Mathlib

set_option maxHeartbeats 2000000
open Polynomial
/--
Let ℚ be the set of rational numbers. A function f : ℚ → ℚ is called aquaesulian if the following property holds: for every x, y ∈ ℚ,

                    f(x + f(y)) = f(x) + y      or      f(f(x) + y) = x + f(y).

Show that there exists an integer c such that for any aquaesulian function f there are at most c different rational numbers of the form f(r)+f(−r)
for some rational number r.
-/
theorem imo_2024_p6
    (IsAquaesulian : (ℚ → ℚ) → Prop)
    (IsAquaesulian_def : ∀ f, IsAquaesulian f ↔ ∀ x y, f (x + f y) = f x + y ∨ f (f x + y) = x + f y) :
    IsLeast {c : ℤ | ∀ f, IsAquaesulian f → {f r + f (-r) | (r : ℚ)}.Finite ∧ {f r + f (-r) | (r : ℚ)}.ncard ≤ c} 2 := by
  exists@?_
  · useλu b=>if j:u 0=0then by_contra λc=>?_ else ?_
    · suffices:({J|∃k,u k+u (-k)= J}) ⊆{0}
      · simp_all[this.antisymm]
      rintro - ⟨a, rfl⟩
      contrapose! c
      simp_all
      suffices:{U|∃examples6, (u) ‹ℚ› +u ( -‹_›)= U} ⊆{0,(u (a : Rat)+ (u<|@@↑(( (-a ))))) } ..
      · use ( Set.toFinite ( _) ).subset ↑@@this , (Set.ncard_le_ncard$ (((this )) ) ).trans (Set.ncard_pair$ Ne.symm (↑
        ↪ ( (c)) ) ).le
      rintro-⟨hz, rfl⟩
      induction b @hz a
      · have:=b (-a)$ hz+u a
        have:=b hz hz
        simp_all[add_comm]
        have:=b (-hz) (hz+u ↑(hz))
        simp_all[ add_assoc, C]
        induction this
        · simp_all
          have:=b hz (hz+(u a+u (-a)))
          have:=b (hz+(u a+u (-a)))$ hz+(u a+u (-a))
          use .inr$ by_contra$ by hint
        have:=b hz$ hz+(u hz+u (-hz))
        cases b (hz+(u hz+u (-hz)))$ hz+(u hz+u (-hz))with|_=>hint
      have:=b (-hz) (u hz+a)
      have:=b$ -a
      specialize this (u hz+a)
      simp_all[ ←add_assoc]
      have:=b 0
      have:=b
      specialize b a a
      simp_all[add_comm]
      have:=(this<| -a) (↑a + (((u a))): (↑_ :((( _) ) ) )) ..
      simp_all[add_assoc]
      cases this
      · simp_all
        contrapose! IsAquaesulian_def
        simp_all
        exfalso
        have:=this a (a+(u hz+u ( -hz)))
        simp_all[Ne.symm,Bool]
        have:=‹∀congr_arg G,_› (a+(u hz+u (-hz)))$ a+(u ↑hz+u ↑( -hz) )
        simp_all
      have:=this a (a +(u a+u (-a)))
      cases‹forall Jd S,_› (a+(u a+u (-a))) ( a + (u a +u ↑(-a)))with| _ =>hint
    simp_all
    cases b 0 0with|_=>exact absurd (b 0$ (0+(1 ∗(@(u ↑.((0) )))))^ 01: ↑ ((_)) ) (id$ (by(cases ( b (u 0) ( (u
      ↪ 0)))with|_ => continuity)))
  rintro K V
  specialize V $ λ N=>-N+2 ∗Int.ceil N
  specialize( V $ (IsAquaesulian_def _).mpr _)
  · simp_rw [ ←eq_sub_iff_add_eq']
    ring
    use mod_cast@?_
    norm_num[<-add_mul,Int.ceil_eq_iff]
    useλc K=>(em _).imp (⟨by linarith[Int.ceil_lt_add_one c,Int.le_ceil K],.⟩) (by repeat use by linarith[.,Int.le_ceil
      ↪ c,or,Int.ceil_lt_add_one$ K])
  simp_all[Int.ceil_neg, ←add_assoc]
  suffices:2<=V.1.toFinset.card
  · let M:=V.1.toFinset
    norm_num[this,V.2.trans',(Set.ext$ by simp_all[M]: {x :Rat|∃t:Rat, (↑2 ) ∗ ( ⌈ t ⌉:(ℚ ) ) .. + (- (2 ∗⌊(t)⌋)) = ↑x} =
      ↪ M)]
  use Finset.one_lt_card.2$ by exists@0,V.1.mem_toFinset.2 (by exists-1),2,V.1.mem_toFinset.2 (by exists 1/2)
```

Extended Data Figure 9: **A complete proof of IMO 2024 P6 found by AlphaProof.** This was found following the protocol in section 6.4. The highlighted line shows the key insight of constructing the function $f(x) = -x + 2\lceil x \rceil$. This algebra problem was fully solved by only 5 of the 609 student contestants [26].