Discovering New Theorems via LLMs with In-Context Proof Learning in Lean

Kazumi Kasaura[1,5,3], Naoto Onda[1,2,3], Yuta Oriike[4,3], Masaya Taniguchi[5,3], Akiyoshi Sannai[6,7,8,9,10,3], Sho Sonoda[5,4,3]

[1]OMRON SINIC X Corporation, [2]NexaScience, [3]AutoRes, [4]CyberAgent, [5]RIKEN AIP, [6]Kyoto University, [7]RIKEN AGIS, [8]Shiga University, [9]NII, [10]NISTEP

Correspondence: kazumi.kasaura@sinicx.com

Abstract

Large Language Models (LLMs) have demonstrated significant promise in formal theorem proving. In this study, we investigate the ability of LLMs to discover novel theorems and produce verified proofs. We propose a pipeline called *Conjecturing-Proving Loop* (CPL), which iteratively generates mathematical conjectures and attempts to prove them in Lean 4. A key feature of CPL is that each iteration conditions the LLM on previously generated theorems and their formal proofs, enabling parameter-free improvement of proof strategies via in-context learning. We provide both theoretical and experimental evidence that CPL increases the discovery rate of hard-to-prove theorems compared to frameworks that generate statements and proofs simultaneously. Moreover, our experiments show that reusing the LLM's own formally verified outputs as context consistently improves subsequent proof success, demonstrating the effectiveness of self-generated in-context learning for neural theorem proving. The source code is available at https://github.com/auto-res/ConjecturingProvingLoop.

1 Introduction

Large Language Models (LLMs) have demonstrated significant promise in theorem proving. Since LLMs can hallucinate and it is difficult to detect such hallucinations in natural language, generating formal proofs using an LLM and verifying them using an interactive theorem prover (ITP), such as Lean[1], has been studied. In this paper, we focus on the ability of LLMs to discover novel theorems.

We propose the *Conjecturing-Proving Loop*, a pipeline for automatically generating mathematical conjectures and proving them in Lean 4 format. By separating the conjecturing and proving phases, we avoid the generation of identical theorems and encourage proving more difficult theorems. In other words, CPL employs *stratified sampling* over conjecture/proof candidates to allocate search resources according to proof difficulty, preventing the loop from collapsing to easy, short proofs. This stratification allows CPL to discover and verify longer proofs that are harder to discover in a simpler framework that samples statements and proofs simultaneously. In this paper, we present a more detailed theoretical discussion of this point.

Another feature of our approach is that we generate and prove further theorems using context that includes proven theorems and their proofs, which enables the generation of more difficult proofs by in-context learning of proof strategies without training of LLMs. Since the ability of reasoning and generation of Lean code by closed-source LLMs, such as GPT, has been improved recently, we use them as both conjecturer and prover. While a disadvantage of using closed-source LLMs is that models cannot be trained freely, in our framework, the proving ability of LLMs can be improved by in-context learning from previous verified proofs.

In our experiment, when mathematical notions were given as seeds, we verified whether important properties about them could be rediscovered in our framework. More specifically, we focused on a few topological notions which are defined only by notions in Mathlib[2], Lean's mathematical library, but not included in Mathlib. We generated theorems about these notions using our framework. As a result, we found that our framework rediscovered an important theorem about these notions that had been published in mathematical papers, which was not found in the simpler framework without separating the Conjecturer and Prover. Moreover, we verified that the in-context learning of proof strate-

[1] https://lean-lang.org/

[2] https://github.com/leanprover-community/mathlib4

Proceedings of the 6th Workshop on Natural Language Meets Logic and Machine Learning (NALOMA), pages 40–49

gies works within our framework: The important theorem, which cannot be proved without context by LLMs even in natural language, was proved with the generated context.

In summary, the contributions of this paper are as follows. First, we propose the Conjecturing-Proving Loop, a pipeline for automatically generating mathematical conjectures and proving them in Lean 4 format. Second, we demonstrated theoretically and experimentally that our framework enables automatic discovery of hard-to-prove theorems. Third, we verified that the proving ability of LLMs can be improved by in-context learning, when the verified proofs generated by LLMs themselves before the statement of a target theorem is provided are given as context.

Our work also suggests the potential for automatic expansion of formal mathematics libraries using AI. Formalized mathematics is only a part of mathematics expressed in natural language, and expanding formal libraries, such as Mathlib, is crucial for verifying and automating mathematics. On the other hand, the set of propositions that should be included in the library may not always be obtainable in natural language. Our framework can generate propositions about given notions while learning them.

2 Related Work

There are several works that use LLMs for mathematical reasoning, both in natural language (DeepSeek-AI et al., 2025) and formal language for ITP (Ren et al., 2025; Lin et al., 2025a). They focused mainly on solving existing problems and used Supervised Fine-Tuning (SFT) and/or Reinforcement Learning with Verified Rewards (RLVR) to improve problem-solving ability of LLMs. To overcome the limited dataset for training, approaches to generate problems to solve by AI are also proposed in some previous works (Dong and Ma, 2025; Huang et al., 2025; Zhan et al., 2025). These works are different from our approach in the following two points: First, our approach is focused on generating and proving meaningful theorems, while these works are focused on training the LLM prover. Second, while these works are based on reinforcement learning, our approach is based on in-context learning, which can be applied to closed-source LLMs.

Several studies have reported that including appropriate contexts in prompts improves the mathematical reasoning ability of LLMs (Wei et al., 2022; Zhou et al., 2022; Drori et al., 2022; Hu et al., 2024; Poiroux et al., 2025). While these studies use handmade examples or data extracted from a database as in-context learning sources, our framework uses outputs from the LLM itself, as in In-Context Reinforcement Learning (Moeini et al., 2025). It has also been proposed to conjecture and prove propositions to be used as lemmas for proving more difficult theorems (Thakur et al., 2023; Wang et al., 2023; Chen et al., 2025; Baba et al., 2025). Unlike these studies, we do not provide the theorem to prove to the LLM; instead, we focus on the LLM's ability to discover the theorem and the libraries used as context during proof generation are produced before the target theorem statement is given.

The technique to leverage feedback from ITP for formal proof generation was proposed (First et al., 2023; Thakur et al., 2023; Lin et al., 2025b) and we also adopted it (See § 3.3). However, what we emphasize here is learning strategies from verified proofs of other propositions.

Minimo (Poesia et al., 2024) shares a similar framework with ours: it jointly trains the conjecturer and prover agents to find theorems. However, while it aims to rediscover mathematics without using existing knowledge, our research has a more practical purpose: attempting to discover theorems by using existing large language models.

A survey (Zhang and Tan, 2026) provides a comprehensive and up-to-date summary of theorem generation, including methods that use LLMs.

3 Method

In this section, we first present an overview of the framework, and then explain the architectures of the conjecturer and the prover.

3.1 Pipeline Overview

Figure 1 illustrates our framework. Conjecturing-Proving Loop (CPL) consists of four main parts: conjecturer (LLM agent), prover (LLM agent), Lean server, and library (Lean code data). First, the library is initialized by the user.

1. The conjecturer generates novel mathematical conjectures in valid Lean 4 format based on the library, while accessing the Lean server.

2. For each generated conjecture, the prover tries to generate valid proofs, while accessing the

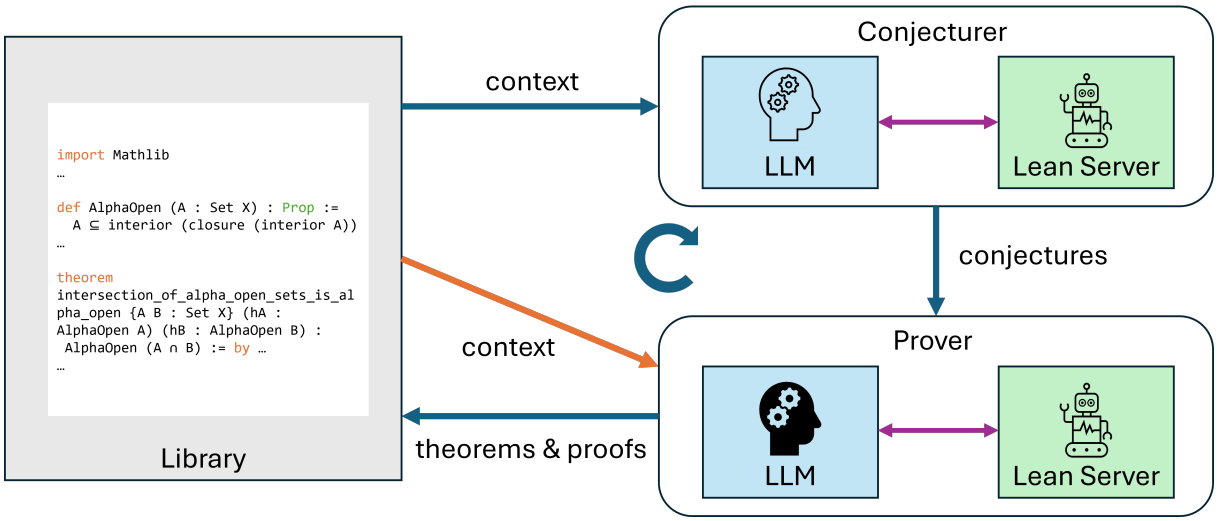

Figure 1: Overview of Conjecturing-Proving Loop. Conjecturer generates conjectures using the library as context, Prover attempts to prove them, and proven conjectures and their proofs are stored in the library as theorems. The library also provides context to Prover. Both Conjecturer and Prover processes consist of interactions between LLMs and Lean Server.

Lean server. The library is used as the context also in this step.

3. The verified pairs of conjectures and their proofs are added to the library. We return to the first step.

For the details of the conjecturing and proving steps, see the following subsections.

By separating the conjecturing and proving phases, we avoid the generation of identical theorems and encourage proving more difficult theorems. For a more detailed discussion, please refer to § 4.

The purpose of feeding the library to the conjecturer as the context is to prevent the generation of duplicate conjectures and to generate conjectures by analogy from already proven theorems.

The purpose of feeding the library to the prover as the context is to make already proven theorems available during proof and to learn proof strategies by in-context learning.

3.2 Conjecture Loop

To generate varied conjectures, for each conjecturing step, we use the following process.

I. The conjecturer LLM generates conjectures following the current library.

II. For each generated conjecture, the Lean server checks whether the conjecture is syntactically valid and novel. Verified conjectures are sent to the prover.

The novelty of conjectures is checked using the `exact?` command in Lean, which checks whether the conjecture can be proved by existing theorems in the context. Note that this command is done with context importing the whole Mathlib4 (Lean4 standard library) and including the library and the verified conjectures. Thus, the checked novelty means that the conjecture is not already in Mathlib4, the already generated library, and the verified conjectures.

The system prompt given to the conjecturer LLM is as follows:

> You are a contributor to the mathlib4 library. Based on a given library, please generate conjectural new theorems in Lean 4 format; they do not need to be true. Do not generate statements that already appear in the list. Do not include proofs, annotations, or imports. Each new statement should begin with 'theorem' (with no annotations) and end with ':= sorry'. Additionally, use standard mathematical symbols (e.g., ∀, ∃, √) rather than Unicode escape sequences (e.g., \u2200).

3.3 Prover Loop

For each generated conjecture, the prover tries to prove it in the following process.

1. The prover LLM produces the proof code of the conjecture. If the LLM judges that the

conjecture is not provable, the prover exits the loop as failure.

2. The Lean server verifies the generated proof. If the proof is verified, the prover exits the loop as success.

3. If the maximum number of trials has been reached, the prover exits the loop with failure. Otherwise, the error message from the Lean server is returned to the LLM and we return to step 1.

The context is given to both the prover and the Lean server. Thus, not only the prover can learn the proof strategies from the context, but also it can use the theorems in the context as lemmas.

The system prompt given to the prover LLM is as follows:

> You are a contributor to the mathlib4 library. Please prove the final theorem in the given content using Lean 4. Write the Lean 4 code that directly follows ':=' in the final theorem. The code should either begin with 'by' or be a term expression. You may use the theorems in the given content as lemmas. Do not use 'sorry' in the proof. If you determine that the theorem is not provable, return an empty string instead of a proof. Do not include any additional text.

In our experiment, the maximum number of trials is set to 16.

3.4 Baseline

For comparison, we also generated theorems for these notions by a simple loop (SL) framework in which an LLM generates theorems and their proofs at once. First, the library is initialized by the user. Unlike CPL, in this simple loop baseline, the conjecturer and the prover are not separated, and the single loop is as follows:

I The LLM generates a statement and its proof in Lean 4 format based on the library, while accessing the Lean server to verify it.

II If the previous step succeeded, the generated pair of statement and proof is stored in the library. We return to step I.

Step I is similar to the prover loop. Its details are as follows:

1. The LLM produces a statement and its proof in Lean.

2. The Lean server checks the generated content. If it is verified, we exit the loop as success.

3. If the maximum number of trials has been reached, we exit the loop with failure. Otherwise, the error message from the Lean server is returned to the LLM and we return to step 1.

The system prompt given to LLM is as follows:

> You are a contributor to the mathlib4 library. Based on a given library, please generate a new theorem together with its proof in Lean 4 format. Do not output anything other than the Lean 4 code. The generated code must follow the given library and contain only the theorem statement and its proof. Do not output declarations other than theorem, such as variable, section, or namespace. Do not generate a theorem that already exists in the library. The new theorem should begin with 'theorem' (with no annotations). You may use the theorems in the given library as lemmas in the proof. Do not use 'sorry' in the proof. Additionally, use standard mathematical symbols (e.g., $\forall$, $\exists$, $\surd$) rather than Unicode escape sequences (e.g., \u2200).

As in the prover loop, the maximum number of trials is set to 16.

4 Theory

For the following reasons, it is expected that the distribution of generated theorems differs between SL and CPL. When a statement and its proof are generated at once, the distribution of generated theorems depends on both the distribution of statements and success rates of proofs. On the other hand, when multiple proofs are attempted after a statement is generated, the distribution of theorems shifts closer to the distribution of provable statements, and the influence of proof success rates diminishes.

More formally, let $s(T)$ be the probability distribution of statements T generated by the LLM, and let $r(T)$ be the probability that the LLM generates a successful proof of T. We model SL as a simplified process that generates a statement and

its proof sequentially, and outputs them if the proof is correct. CPL is also simplified and modeled as a process that attempts to generate a valid proof, after the generation of a statement, until it succeeds or the number of attempts reaches N. For simplicity, we ignore the influence of contexts.

The probability of generating a theorem T is proportional to $s(T)r(T)$ in SL, and is proportional to $s(T)\left(1-(1-r(T))^N\right)$ in CPL. Therefore, as N increases, the distribution of theorems in CPL approaches the distribution of provable statements (T such that $r(T) > 0$), and even theorems that are difficult to prove tend to be generated more frequently.

On the other hand, while the expected number of proof trials required to discover one theorem in SL is $E_{\mathrm{SL}} := (\mathbb{E}_{T\sim s}[r(T)])^{-1}$, it is

$$E_{\mathrm{CPL}} := \frac{\mathbb{E}_{T\sim s}\left[(1-(1-r(T))^N)r(T)^{-1}\right]}{\mathbb{E}_{T\sim s}\left[1-(1-r(T))^N\right]}$$

in CPL, because, when the statement T is generated, the probability of success in proving T is $1-(1-r(T))^N$ and the expected number of trials is $(1-(1-r(T))^N)r(T)^{-1}$.[3]

Since $\left(1-(1-r)^N\right)r^{-1}$ is decreasing for r, from Chebyshev's sum inequality,

$$\begin{aligned}&\mathbb{E}_{T\sim s}\left[(1-(1-r(T))^N)\right]\\ \leq&\mathbb{E}_{T\sim s}\left[(1-(1-r(T))^N)r(T)^{-1}\right]\mathbb{E}_{T\sim s}[r(T)].\end{aligned}$$

Thus, $E_{\mathrm{SL}} \leq E_{\mathrm{CPL}}$, which explains why CPL generates fewer theorems than SL.

The condition under which a theorem T_0 is more likely to be generated in CPL than in SL under the fixed number of proof trials is as follows. In SL, at one generation of a statement, the probability to find T_0 is $s(T_0)r(T_0)$ and the number of proof trials is always 1. In CPL, at one generation of a statement, the probability to find T_0 is $s(T_0)(1-(1-r(T_0))^N)$ and the expected number of proof trials is $\mathbb{E}_{T\sim s}\left[(1-(1-r(T))^N)r(T)^{-1}\right]$. Since $s(T_0) \ll 1$, the desired condition can be approximated as

$$\frac{1-(1-r(T_0))^N}{\mathbb{E}_{T\sim s}\left[(1-(1-r(T))^N)r(T)^{-1}\right]} > r(T_0),$$

which is independent of $s(T_0)$. If $r(T_0) > 0$, this can also be written as

$$\begin{aligned}&(1-(1-r(T_0))^N)r(T_0)^{-1}\\ >&\mathbb{E}_{T\sim s}\left[(1-(1-r(T))^N)r(T)^{-1}\right].\end{aligned}$$

[3] Because $(1-(1-r)^N)r^{-1}$ is actually a polynomial, we consider its value for $r=0$ as N.

Since $\left(1-(1-r)^N\right)r^{-1}$ is decreasing for r, provable theorems with sufficiently low proof success rates are more likely to be generated in CPL.

5 Experiments

We demonstrated that research-level theorems can be rediscovered by our framework and verified that in-context learning worked effectively in our framework.

5.1 Setting

In our experiments, we focus on the minor notions in general topology: semi-openness, α-openness, and preopenness. We used the following file including the definitions of these notions in Lean 4 format as the initial library.

```
import Mathlib
import Aesop

namespace Topology

variable {X : Type*} [TopologicalSpace X]

def P1 (A : Set X) : Prop :=
  A ⊆ closure (interior A)

def P2 (A : Set X) : Prop :=
  A ⊆ interior (closure (interior A))

def P3 (A : Set X) : Prop :=
  A ⊆ interior (closure A)
```

The notions P1, P2, and P3 are 'semi-open', 'α-open', and 'preopen', respectively, and are anonymized to prevent LLMs from using existing knowledge. The reason for choosing these notions is that they can be defined using only notions already present in Mathlib, while they themselves are not yet included in Mathlib, and while their mathematical importance has already been recognized and researched, they are not yet well known enough for LLM to have knowledge of their properties.

We set the following theorem as a target and focused on whether it could be generated or not:

> *The intersection of two P2 (α-open) sets is P2 (α-open)*

This theorem is important because it is the most difficult part of the proof that α-open sets form another topology (Proposition 2 in (Njåstad, 1965)). We have confirmed that this theorem cannot, at least, be naively derived from the knowledge of the LLM used in the experiment. See § 5.3.3. Whether a generated library contains the desired theorem or not is checked as follows: place the statement

of the theorem after the generated library and see if the proof is completed by using `exact?` command. If the library contains a proposition that is trivially equivalent to or stronger than the theorem, the completion succeeds. By doing so, we can accommodate variations in the formulation of the theorem within the range that the Lean server can recognize.

We used GPT-o3[4] both in CPL and in SL. For both CPL and SL, we generated libraries 20 times as follows: we generated theorems until the API usage reached 14000000 tokens.

5.2 Results

In CPL, 106 theorems were generated on average, and **the target theorem was discovered 5 times out of 20**. In SL, 328 theorems were generated on average, but **the target theorem was never generated in any of the 20 runs**. According to Fisher's exact test ($p = 0.024$), CPL is more likely to generate the desired theorem.

An example of generated proofs of this theorem is shown in Appendix A. This proof differs from the original proof by Njåstad, which suggests that the LLM found this proof independently.

The results showing that while SL generates more theorems, CPL is more likely to generate theorems that are difficult to prove are consistent with the discussion in § 4. To further verify this, we measured the proof length of the generated theorems. Figure 2 shows the distribution of proof lengths (the numbers of characters) of the theorems generated by CPL and SL. It can be observed that CPL can generate theorems with longer proofs than SL. It is known that there is a positive relationship between the length and difficulty of proofs [5] (Wu et al., 2025; Sonoda et al., 2026). Thus, this result is consistent with the theoretical analysis.

5.3 Additional Experiments

To verify the aforementioned effect of CPL independently, we conducted an additional experiment (§ 5.3.1).

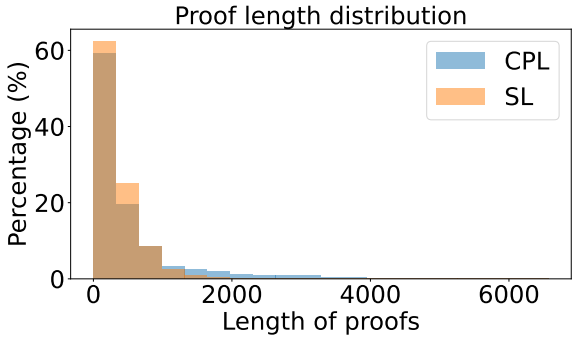

Figure 2: Distribution of proof lengths (the numbers of characters) of theorems generated by our framework and the simple loop framework.

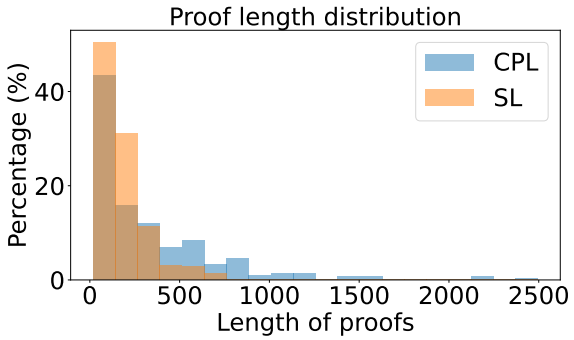

Figure 3: Distribution of proof lengths (the numbers of characters) of theorems generated by our framework and the simple loop framework without in-context learning.

We also verified that feeding the generated library as a context to the prover improves the proof ability (§ 5.3.2 and § 5.3.3).

5.3.1 Generation Without In-Context Learning

To observe the difference between CPL and SL without effects of in-context learning, we generated theorems with only the seed file as a context. In other words, for both CPL and SL, we independently conducted the first single loops several times, until the API usage reached 3000000 tokens.

Including duplicates, 309 theorems were generated in CPL and 941 theorems were generated in SL. The distributions of generated proofs are shown in Figure 3. The shift of distribution is observed. According to the Kolmogorov–Smirnov test, CPL tends to generate longer proofs than SL with a p-value of 1×10^{-13}.

The target theorem was not generated by either CPL or SL. See also the results of the following experiments.

[4] https://platform.openai.com/docs/models/o3. While GPT is currently released up to version 5.2, o3 was the latest version when this research began. Since the notions and theorems used in our experiments were designed assuming o3, GPT-5 was unsuitable because its performance was higher from the start, and thus it was not adopted.

[5] Note that in this context, "difficulty" refers to the difficulty of generating a valid proof in Lean, and "proof length" refers to the length of the Lean code; these are not necessarily the same as the difficulty or length in natural language.

5.3.2 Reproving Generated Theorems

First, we attempted to re-prove all theorems generated in CPL with two settings: one where the context includes the library generated before the theorem to be proved was generated, and one where the context includes only the definitions of the notions. As a result, in the setting with contexts, **99% of the theorems (2106/2123 theorems)** were proved, while in the setting without contexts, only **91% of the theorems (1935/2123 theorems)** were proved. According to the McNemar test, this difference is statistically significant at a p-value of 4×10^{-35}. Thus, the context improves LLMs' proof ability.

5.3.3 Proof Ability for Alpha-Open Intersection

Moreover, we attempted to re-prove the target theorem 16 times for each of the 5 contexts in which the target theorem was generated. (The average number of theorems generated until this theorem was generated is 49.) For comparison, we also attempted to re-prove it 80 times without any generated library. The procedure is the same as the prover loop, except the system prompt was changed to the following:

> You are a contributor to the mathlib4 library. Please prove the final theorem in the given content using Lean 4. Write the Lean 4 code that directly follows ':=' in the final theorem. The code should either begin with 'by' or be a term expression. You may use the theorems in the given content as lemmas. Do not use 'sorry' in the proof. If you determine that the theorem is false, return an empty string instead of a proof. Do not include any additional text.

Note that the condition not to return a proof has changed from "not provable" to "false".

As a result, **in the setup that included the generated library as context, the re-proof succeeded** 7 **times, whereas, in the setup without the library, it failed in all** 80 **attempts.** This suggests that, through in-context learning, the prover acquires the ability to prove theorems that could not be proven without it.

The generated proof of this theorem, shown in Appendix A, does not use other generated theorems as lemmas. Thus, the generated library was used for in-context learning of proof strategies rather than as a collection of lemmas for the proof.

In addition, we also asked LLMs (GPT-4o[6] and GPT-o3) to prove this theorem in natural language (English) with context including the definitions of the notions 16 times and checked the responses by hand. In the natural language experiment, we used the following system prompt:

> Please prove the following theorem. If you judge that the theorem is false, please return "False" instead of the proof.

The statement to prove given to LLMs is as follows:

> In a topological space, a set is alpha-open if it is a subset of the interior of the closure of its interior. The intersection of any two alpha-open sets is alpha-open.

As a result, GPT-4o incorrectly stated that the proposition is false 10 times and generated incorrect proofs 6 times. GPT-o3 never generated an incorrect proof, but it always judged incorrectly that the theorem is false. The fact that the majority of judgments in GPT-4o identify the theorem as false implies that the theorem was not included in GPT's knowledge. An example of a proof with gaps generated by GPT-4o is shown in Appendix B.

6 Conclusion

We presented the Conjecturing-Proving Loop, a pipeline for automatically generating mathematical conjectures and proving them in Lean 4 format. We demonstrated that our framework can rediscover a research-level theorem. We also verified that in-context learning of proof strategies works effectively in our framework.

Limitations

In our experiment, we focused on only a single target theorem and did not investigate the ability to generate broader theorems. In particular, since the target theorem was relatively easy to conjecture, refining the conjecture generation process may be necessary to produce more profound and insightful mathematical statements.

Acknowledgements

This work was supported by JST Moonshot R&D Program JPMJMS2236, JST BOOST JPMJBY24E2, JST CREST JPMJCR2015, JSPS

[6] https://platform.openai.com/docs/models/gpt-4o

KAKENHI 24K21316, 24K16077, and Advanced General Intelligence for Science Program (AGIS), the RIKEN TRIP initiative.

References

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

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, and 1 others. 2025. Seed-prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 181 others. 2025. DeepSeek-V3 technical report. *Preprint*, arXiv:2412.19437.

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

Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, and 1 others. 2022. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32):e2123433119.

Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. 2023. 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*, pages 1229–1241.

Jiewen Hu, Thomas Zhu, and Sean Welleck. 2024. minictx: Neural theorem proving with (long-) contexts. *arXiv preprint arXiv:2408.03350*.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. 2025. R-zero: Self-evolving reasoning LLM from zero data. *arXiv preprint arXiv:2508.05004*.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia LI, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. 2025a. Goedel-prover: A frontier model for open-source automated theorem proving. In *Second Conference on Language Modeling*.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, and 1 others. 2025b. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*.

Amir Moeini, Jiuqi Wang, Jacob Beck, Ethan Blaser, Shimon Whiteson, Rohan Chandra, and Shangtong Zhang. 2025. A survey of in-context reinforcement learning. *arXiv preprint arXiv:2502.07978*.

Olav Njåstad. 1965. On some classes of nearly open sets. *Pacific journal of mathematics*, 15(3):961–970.

Gabriel Poesia, David Broman, Nick Haber, and Noah Goodman. 2024. Learning formal mathematics from intrinsic motivation. *Advances in Neural Information Processing Systems*, 37:43032–43057.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. 2025. Reliable evaluation and benchmarks for statement autoformalization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 17958–17980.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. 2025. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *Preprint*, arXiv:2504.21801.

Sho Sonoda, Shunta Akiyama, and Yuya Uezato. 2026. Why agentic theorem prover works: A statistical provability theory of mathematical reasoning models. *arXiv preprint arXiv:2602.10538*.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2023. An in-context learning agent for formal theorem-proving. *arXiv preprint arXiv:2310.04353*.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Lin Li, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, and 1 others. 2023. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Zheng Yuan, Wenwei Zhang, Dahua Lin, and Kai Chen. 2025. Internlm2. 5-stepprover: Advancing automated theorem proving via critic-guided search. In *2nd AI for Math Workshop@ ICML 2025*.

Shaoxiong Zhan, Yanlin Lai, Ziyu Lu, Dahua Lin, Ziqing Yang, and Fei Tang. 2025. Mathsmith: Towards extremely hard mathematical reasoning by forging synthetic problems with a reinforced policy. *arXiv preprint arXiv:2508.05592*.

Jian Zhang and Si-Cheng Tan. 2026. Automated conjecturing and theorem finding: A survey. *Journal of Computer Science and Technology*, pages 1–21.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and 1 others. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

A Generated Proof of Focused Theorem

```
theorem P2_inter {X : Type*} [TopologicalSpace
    X] {A B : Set X} : P2 A → P2 B → P2 (A ∩
    B) := by
  intro hP2A hP2B x hx
  rcases hx with ⟨hxA, hxB⟩
  -- Auxiliary open neighbourhoods furnished by
     `P2 A` and `P2 B`.
  set UA : Set X := interior (closure (interior
    A)) with hUA
  set UB : Set X := interior (closure (interior
    B)) with hUB
  have hUA_open : IsOpen UA := by
    simpa [hUA] using (isOpen_interior :
      IsOpen (interior (closure (interior A))))
  have hUB_open : IsOpen UB := by
    simpa [hUB] using (isOpen_interior :
      IsOpen (interior (closure (interior B))))
  have hxUA : x ∈ UA := by
    have : x ∈ interior (closure (interior A))
    := hP2A hxA
    simpa [hUA] using this
  have hxUB : x ∈ UB := by
    have : x ∈ interior (closure (interior B))
    := hP2B hxB
    simpa [hUB] using this
  -- Combine the two neighbourhoods.
  have hO_open : IsOpen (UA ∩ UB : Set X) :=
    hUA_open.inter hUB_open
  have hxO : x ∈ UA ∩ UB := ⟨hxUA, hxUB⟩
  -- Main claim: the intersection lies in the
     relevant closure.
  have hO_sub : (UA ∩ UB : Set X) ⊆ closure
    (interior (A ∩ B)) := by
    intro y hy
    rcases hy with ⟨hyUA, hyUB⟩
    -- `y` is simultaneously in the closures of
     `interior A` and `interior B`.
    have hy_clA : y ∈ closure (interior A) := by
      -- `UA ⊆ closure (interior A)`
      have hsub : (UA : Set X) ⊆ closure
    (interior A) := by
        intro z hz
        have hz' : z ∈ interior (closure
    (interior A)) := by
          simpa [hUA] using hz
        exact interior_subset hz'
      exact hsub hyUA
    have hy_clB : y ∈ closure (interior B) := by
      have hsub : (UB : Set X) ⊆ closure
    (interior B) := by
        intro z hz
        have hz' : z ∈ interior (closure
    (interior B)) := by
          simpa [hUB] using hz
        exact interior_subset hz'
      exact hsub hyUB
    -- Show that every open neighbourhood of
     `y` meets `interior (A ∩ B)`.
    have : y ∈ closure (interior (A ∩ B)) := by
      apply (mem_closure_iff).2
      intro V hVopen hyV
      -- First refinement, intersect with `UB`.
      have hV1_open : IsOpen (V ∩ UB) :=
    hVopen.inter hUB_open
      have hyV1 : y ∈ V ∩ UB := ⟨hyV, hyUB⟩
      -- Obtain a point in `interior A`.
      rcases (mem_closure_iff).1 hy_clA (V ∩
    UB) hV1_open hyV1 with
        ⟨a, ⟨haV, haUB⟩, haIntA⟩
      -- `a` is now in `V`, `UB`, and `interior
    A`.
      have ha_clB : a ∈ closure (interior B) :=
    by
        have hsub : (UB : Set X) ⊆ closure
    (interior B) := by
          intro z hz
          have hz' : z ∈ interior (closure
    (interior B)) := by
            simpa [hUB] using hz
          exact interior_subset hz'
        exact hsub haUB
      -- Second refinement, intersect with
     `interior A`.
      have hW_open : IsOpen (V ∩ interior A) :=
    hVopen.inter isOpen_interior
      have haW : a ∈ V ∩ interior A := ⟨haV,
    haIntA⟩
      -- Obtain a point in `interior B`.
      rcases (mem_closure_iff).1 ha_clB (V ∩
    interior A) hW_open haW with
        ⟨z, ⟨hzV, hzIntA⟩, hzIntB⟩
      -- `z` lies in `V`, `interior A`, and
     `interior B`.
      have hzIntAB : z ∈ interior (A ∩ B) := by
        -- The set `interior A ∩ interior B` is
    open and contained in `A ∩ B`.
        have hS_open : IsOpen (interior A ∩
    interior B) :=
          isOpen_interior.inter isOpen_interior
        have hS_sub : (interior A ∩ interior B
    : Set X) ⊆ A ∩ B := by
          intro w hw
          rcases hw with ⟨h1, h2⟩
          exact ⟨interior_subset h1,
    interior_subset h2⟩
        have hS_to : (interior A ∩ interior B :
    Set X) ⊆
            interior (A ∩ B) :=
          interior_maximal hS_sub hS_open
        have hzS : z ∈ interior A ∩ interior B
    := ⟨hzIntA, hzIntB⟩
        exact hS_to hzS
      exact ⟨z, hzV, hzIntAB⟩
    exact this
  -- Use the neighbourhood just constructed.
  have hNhd : (UA ∩ UB : Set X) ∈ 𝓝 x :=
    hO_open.mem_nhds hxO
  have h_mem :
      x ∈ interior (closure (interior (A ∩
    B))) :=
```

```
    (mem_interior_iff_mem_nhds).2
      (Filter.mem_of_superset hNhd hO_sub)
  simpa using h_mem
```

B Example of Generated Incorrect Proof

The following is an example of a 'proof' of the theorem that the intersection of two alpha-open sets is alpha-open, generated by GPT-4o. We have only changed the line breaks to adjust the width.

> To prove the given theorem, let's first define what it means for a set to be alpha-open in a topological space.
>
> Let (X, τ) be a topological space and let $A \subseteq X$. The set A is called alpha-open if
>
> $$A \subseteq \text{int}(\overline{\text{int}(A)}),$$
>
> where $\text{int}(A)$ denotes the interior of A, and $\overline{\text{int}(A)}$ denotes the closure of $\text{int}(A)$.
>
> We aim to prove that the intersection of any two alpha-open sets is alpha-open. Let $A, B \subseteq X$ be two alpha-open sets; hence, by definition, we have:
>
> 1. $A \subseteq \text{int}(\overline{\text{int}(A)})$ 2. $B \subseteq \text{int}(\overline{\text{int}(B)})$
>
> We need to show that $A \cap B$ is also alpha-open, i.e.,
>
> $$A \cap B \subseteq \text{int}(\overline{\text{int}(A \cap B)}).$$
>
> Firstly, observe that $\text{int}(A \cap B) \subseteq \text{int}(A) \cap \text{int}(B)$.
>
> Both closure and interior operations are monotonic, so:
>
> $$\begin{aligned}\overline{\text{int}(A \cap B)} &\subseteq \overline{\text{int}(A) \cap \text{int}(B)} \\ &\subseteq \overline{\text{int}(A)} \cap \overline{\text{int}(B)}.\end{aligned}$$
>
> Taking interiors again yields:
>
> $$\begin{aligned}\text{int}(\overline{\text{int}(A \cap B)}) &\subseteq \text{int}(\overline{\text{int}(A)} \cap \overline{\text{int}(B)}) \\ &\subseteq \text{int}(\overline{\text{int}(A)}) \cap \text{int}(\overline{\text{int}(B)}).\end{aligned}$$
>
> By the assumptions that $A \subseteq \text{int}(\overline{\text{int}(A)})$ and $B \subseteq \text{int}(\overline{\text{int}(B)})$, it follows that
>
> $$\begin{aligned}A \cap B &\subseteq \text{int}(\overline{\text{int}(A)}) \cap \text{int}(\overline{\text{int}(B)}) \\ &\supset \text{int}(\overline{\text{int}(A \cap B)}).\end{aligned}$$
>
> Therefore,
>
> $$A \cap B \subseteq \text{int}(\overline{\text{int}(A \cap B)}).$$
>
> This shows that $A \cap B$ is indeed alpha-open. Therefore, the intersection of any two alpha-open sets is alpha-open.
>
> Hence, the theorem is true.

This proof clearly has a gap because, when deriving the last relation from the second-to-last relations, the direction of a subset relation is reversed.