QDTSynth: Quality-Driven Formal Theorem Synthesis for Enhancing Proving Performance of LLMs

Lei Wang[1], Ruobing Zuo[1], Gaolei He[2], Jianlin Wang[2], Zhengfeng Yang[1]*

[1]East China Normal University, [2]Henan Univeristy

{51265902021, rbzuo}@stu.ecnu.edu.cn
{hegaolei, jlwang}@henu.edu.cn
zfyang@sei.ecnu.edu.cn

Abstract

Automated Theorem Proving is an important and challenging task. Although large language models (LLMs) have demonstrated remarkable potential in mathematical reasoning, their performance in formal theorem proving remains constrained by the scarcity of high-quality supervised fine-tuning (SFT) data. To address this limitation, we propose a **Q**uality-**D**riven **T**heorem **S**ynthesis method (QDTSynth) in Lean4. During the statement synthesis, we enhance Monte Carlo Tree Search (MCTS) with an adaptive adjustment mechanism that dynamically optimizes the search strategy based on the synthesis of statements. In addition, we propose diversity screening and the self-assessment method to select theorems that exhibit both diversity and high quality from the initially synthetic statements, enabling the synthesis of a high-quality Lean4 theorem dataset. After fine-tuning three open-source large language models on our synthetic dataset, experiments on the miniF2F benchmark demonstrate that QDTSynth significantly improves the performance of various open-source LLMs in theorem proving tasks. Our work offers a promising new direction for the future synthesis of high-quality formal mathematical theorems.

1 Introduction

In modern mathematical research and applications, the importance of mathematical proofs is self-evident. Due to the complexity of mathematical reasoning and the potential limitations of manual review, even experienced mathematicians may struggle to identify all potential proof errors. The emergence of formal languages such as Lean (Moura and Ullrich, 2021), Coq (Coq, 1996) and Metamath (Megill and Wheeler, 2019) marks a significant turning point for mathematical proofs. Formal languages ensure that every step in the mathematical proof process can be rigorously checked by computer systems, thereby guaranteeing the correctness and reliability of the final results. Formal mathematics requires a high level of expertise, leading to a scarcity of specialized talent in this field. Additionally, interactive theorem proving demands extensive manual input and meticulous human review, which increases its cost of use. Against this backdrop, automated theorem proving (Bibel, 2013; Loveland, 2016; Kusumoto et al., 2018), as a method capable of significantly reducing manual intervention and improving proof efficiency, has become increasingly important.

Large language models (LLMs) have demonstrated significant potential in the field of mathematical reasoning (Wei et al., 2023; Ahn et al., 2024; Srivastava et al., 2024), with numerous studies integrating them with formal proof assistants to achieve automated theorem proving (Vishwakarma and Mishra, 2023; First et al., 2023; Yang et al., 2024b; Dong et al., 2024). However, the complexity of formal theorem proving and its reliance on deep expertise have resulted in a critical shortage of high-quality formal theorem-proof data suitable for supervised fine-tuning (SFT) of LLMs. Although there are many methods for data synthesis (Lu et al., 2024; Wang et al., 2024b; Zhu et al., 2024; Cao et al., 2025), they are difficult to migrate to the complex formal theorems. To address this challenge, researchers have proposed several methods for the automatic generation of formal theorems. Lin et al., 2024 combines Monte Carlo Tree Search (MCTS) (Chaslot et al., 2008) with language models (LMs), and introduces policy/value networks to optimize the generation process. However, if ineffective tactics are selected during the exploration of MCTS nodes, it may lead to the exploration of low-quality branches and make it challenging to synthesize high-quality theorems. In the process of supervised fine-tuning, the quality of the dataset is often more critical than its quantity (Shen, 2024; Pang et al., 2025), emphasizing the need for

*Corresponding author: Zhengfeng Yang

methods capable of generating high-quality formal theorems. Xin et al., 2024; Ying et al., 2024 have proposed methods for synthesizing formal statements and generating proof steps from informal mathematical problems. However, the potential of formal statements remains underutilized. There are few methods available for the automatic synthesis of formal theorems, and synthesizing high-quality data remains an important task.

In this paper, we propose QDTSynth, focusing on the automatic synthesis of high-quality Lean4 theorems from formal statements. We use theorem statements extracted from Mathlib4 (mathlib Community, 2020) and mathematical problems from high school and undergraduate exercises, exams, and competitions formalized by LLMs as seed data. Throughout the iterative process of MCTS, new statements are continuously synthesized. QDTSynth enhances the traditional MCTS by incorporating an adaptive mechanism, which dynamically adjusts the search rules based on the synthetic statements, thereby optimizing the synthesis process and improving the quality of the synthetic statements. After synthesizing new statements, we employ online dynamic clustering for diversity screening by calculating the cosine similarity between each new statement and the cluster centers. Subsequently, we generate proof steps for the screened statements and introduce a self-assessment mechanism where the LLM evaluates the quality of the theorem proofs. The final selected theorems constitute a new high-quality Lean4 theorem dataset. We perform supervised fine-tuning on three open-source LLMs using the synthesized dataset and evaluate the effectiveness of QDTSynth in Lean4 theorem proving on 488 problems from miniF2F (Zheng et al., 2021). Experimental results demonstrate that the models trained with our method achieve significant performance improvements compared to traditional BFS, MCTS, and MCTS+pvn (Lin et al., 2024).

Our contributions are summarized as follows:

• We integrate an adaptive mechanism into MCTS, dynamically optimizing our tactic selection for synthesizing high-quality statements.

• We propose the QDTSynth framework, designed to synthesize high-quality formal theorems. Based on the adaptive MCTS, we further introduce diversity screening and the self-assessment method to select high-quality theorems.

• QDTSynth has shown notable advantages on the miniF2F benchmark, providing a novel direction for automated formal theorem synthesis.

2 Related Works

LLMs for Data Synthesis. With the advent of LLMs, there are numerous data synthesis methods based on LLMs (Park et al., 2024; Kang et al., 2024), aimed at enhancing the performance of models. Xu et al., 2024 synthesized high-quality instruction data at scale by extracting it directly from an aligned LLM. Wang et al., 2023b addressed distributional discrepancy by iteratively refining the synthesized dataset using error extrapolation via a LLM. Lupidi et al., 2024 takes as input a custom data source and produces synthetic data points with intermediate reasoning steps grounded in real-world sources. Although there are many methods of data synthesis, they are difficult to migrate to the complex formal theorem synthesis.

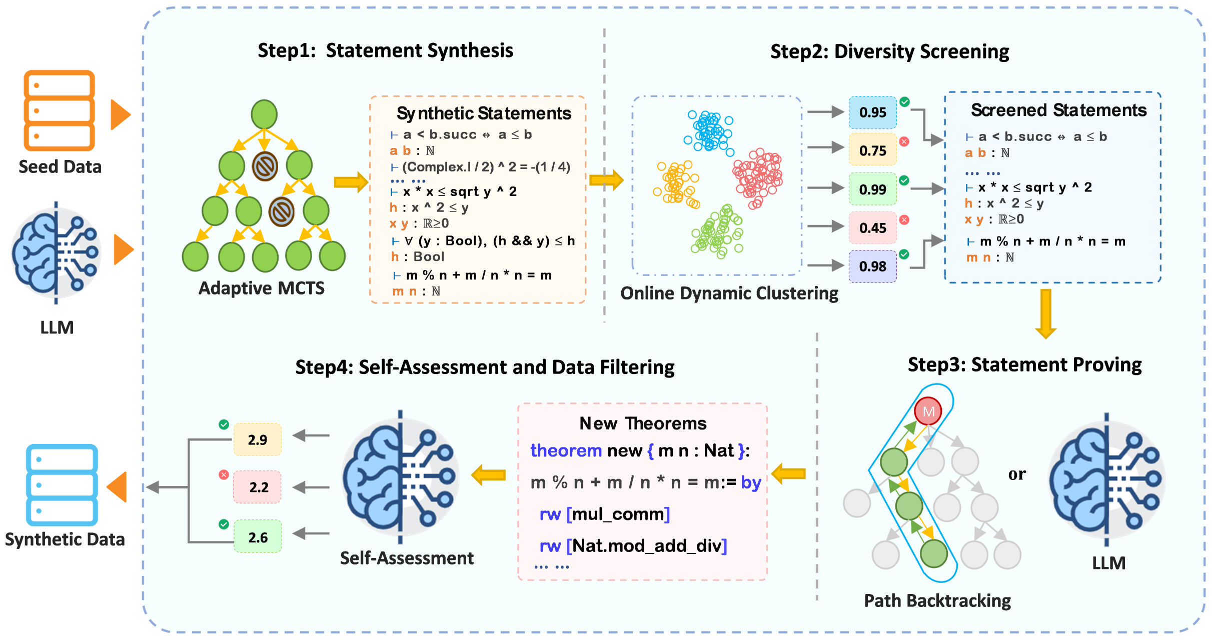
Formal Theorem Synthesis. In previous studies, formal theorem synthesis methods have enhanced the performance of provers. MetaGen (Wang and Deng, 2020) is the first neural generator for synthetic training data, using reinforcement learning to synthetic theorems that resemble those written by humans. PACT (Han et al., 2021) is an approach for extracting abundant self-supervised data from kernel-level proof terms for joint training alongside the usual tactic prediction objective. DeepSeek-Prover (Xin et al., 2024) and Lean Workbook (Ying et al., 2024) generated Lean4 statements from broad natural language mathematical problems. Lin et al. (2024) combined MCTS with LMs and learned policy and value models to generate a new dataset. Unlike prior work, QDTSynth presents a unique approach for theorem synthesis from formal statements, combined with the optimization of the synthesis process and quality filtering.

3 QDTSynth

QDTSynth is an approach to quality-driven synthesis of formal theorems, focusing on the synthesis of high-quality Lean4 theorems from formal statements. QDTSynth consists of following steps: Statement Synthesis, Diversity Screening, Statement Proving, Self-Assessment and Data Filtering. The process is illustrated in Figure 1.

3.1 Statement Synthesis

We continuously synthesize new statements through iterative processes of selection, expansion,



Synthetic Statements
```
⊢ a < b.succ ↔ a ≤ b
a b : ℕ
⊢ (Complex.I / 2) ^ 2 = -(1 / 4)
… …
⊢ x * x ≤ sqrt y ^ 2
h : x ^ 2 ≤ y
x y : ℝ≥0
⊢ ∀ (y : Bool), (h && y) ≤ h
h : Bool
⊢ m % n + m / n * n = m
m n : ℕ
```

Screened Statements
```
⊢ a < b.succ ↔ a ≤ b
a b : ℕ
… …
⊢ x * x ≤ sqrt y ^ 2
h : x ^ 2 ≤ y
x y : ℝ≥0
⊢ m % n + m / n * n = m
m n : ℕ
```

New Theorems
```
theorem new { m n : Nat }:
m % n + m / n * n = m:= by
  rw [mul_comm]
  rw [Nat.mod_add_div]
… …
```

Figure 1: Overview of QDTSynth framework. QDTSynth consists of four steps, starting with seed data and the LLM trained from Mathlib4. **(1) Statement Synthesis:** We introduce an adaptive mechanism into MCTS to optimize the statement synthesis process. **(2) Diversity Screening:** We employ online dynamic clustering for diversity screening by calculating the cosine similarity between each new statement and the cluster centers. **(3) Statement Proving:** We generate proof steps for the screened statements, synthesize complete theorems. **(4) Self-Assessment and Data Filtering:** We introduce a self-assessment mechanism to evaluate the quality of theorems, and employ data filtering to obtain high-quality theorems.

and backpropagation in MCTS. Starting from the root node, we select tactics for nodes based on policy/value models, and interact with the Lean proof assistant. After receiving feedback from Lean, we expand the current node to generate new nodes and determine whether a new statement has been synthesized based on the node's state. To optimize the selection process in MCTS, we introduce an adaptive mechanism that allows search rules to dynamically adjust based on the generation conditions, thereby enhancing both the quality and efficiency of statement generation.

Monte Carlo Tree Structure. The root node of the search tree is derived from formal statements extracted from Mathlib4 and mathematical problems from high school and undergraduate exercises, exams, and competitions formalized by LLMs. Each node in the tree records its state, including intermediate results of theorems or outcomes of reasoning steps. The state of a node is closely tied to the results of interactions with Lean. If the state is neither success nor failure, the state of the node is considered as a statement. Additionally, we use the input LLM trained by Mathlib4 to generate candidate tactics for each state, which form the edges of the search tree.

In the process of statement generation, we classify the generated nodes into three types: error nodes, duplicate statement nodes, and new statement nodes. We design different rewards for each of these three distinct node types. For error nodes, if a tactic results in an erroneous state, the reward of -1 is given. For duplicate statement nodes, if the newly generated statement is a duplicate of an existing statement, the reward is defined as 0. If the statement is new, which does not exist in the current statement database, the reward is determined to be 1.

Adaptive Optimization in Selection. The selection phase is central to the statement synthesis process, determining both the direction and efficiency of the search. During the selection phase, we utilize the Predictor + Upper Confidence for Trees (PUCT) search strategy. This strategy leverages prior probabilities $\pi(s_t, a)$ for selecting specific edges, which are produced by a policy model. The selection of an action is based on the average value and exploration value of state s_t. Each time state s_t is traversed, the cumulative total reward $W(s_t)$ is updated by adding the value $v(s_t)$ of the expanded nodes,which is computed by our value model. The cumulative reward is then divided by the number of visits $N(s_t)$ to state s_t, resulting in the average reward of state s_t. The details of the selection process are illustrated in Figure 2a. Building on the original PUCT formula (Silver et al., 2017), we introduce a policy penalty term $Pen(s_t, a)$, where the value of the penalty increases with the proportion of repeated theorems. At each state s_t, for every time step t, a new action a will be selected according to the formula:

$$a = argmax_a(\frac{W(s_t)}{N(s_t)} + U(s_t, a)) \tag{1}$$

where

$$\begin{aligned} U(s_t, a) = c\pi(s_t, a)\tfrac{\sqrt{N(s_t)}}{1+N(s_t,a)} \\ -\lambda(s_t)Pen(s_t, a) \end{aligned} \tag{2}$$

During statement generation, if the same tactic is repeatedly executed, it will seriously affect the efficiency and quality of the statements. For example, in Lean, continuously applying the "have" tactic to declare the same lemma results in no new statements being obtained and leads to excessively long and useless generation steps, which negatively impacts the effectiveness of SFT for LLMs. Therefore, we introduce a tactic penalty term $Pen(s_t, a)$ as a constraint mechanism for selection. The computation of this penalty term is based on the ratio between the repetition count $Repeat(s_t, a)$ of tactic a and the length of the generation path $Len(s_t)$, aimed at reducing the use of repeated tactics during the search process. The specific formula is as follows:

$$Pen(s_t, a) = \frac{Repeat(s_t, a)}{Len(s_t)} \tag{3}$$

To further enhance the effect of tactic penalty term, we introduce an adaptively adjusted penalty weight $\lambda(s_t)$. As the search depth increases, the impact of the penalty term also grows. The calculation formula is as follows:

$$\lambda(s_t) = \lambda_0 \cdot (1 + \alpha \cdot Len(s_t)) \tag{4}$$

For the exploration coefficient c in PUCT, its primary role is to balance the exploration and exploitation. In the initial stages of the search, we aim to encourage the algorithm to explore a wider range of possible statements by setting a larger c, avoiding convergence to local optima. As the search progresses and the number of visits increases, we tend to reduce c to focus on exploiting higher-quality statements, improving the efficiency of the search and the quality of the statements. To dynamically adjust the value of c to adapt to different stages of the search process, the formula for our adaptive mechanism is as follows:

$$c = c_0 \cdot \exp\left(-\gamma \cdot \frac{N}{1+N}\right) \tag{5}$$

Here, c_0 is the initial exploration coefficient, γ controls the decay rate, and N represents the total number of visits.

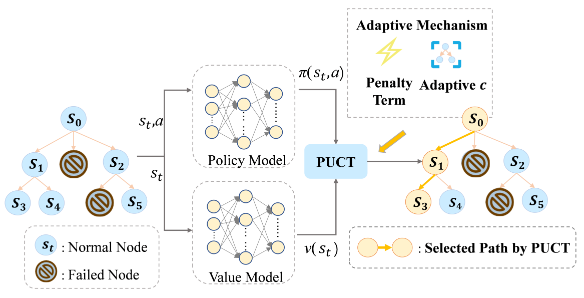
Expansion and Backpropagation. During the expansion phase, the selected node is expanded by randomly selecting an action from the candidate tactics provided by LLMs and executing it on the current state, thereby generating a node with a new state. The left part of Figure 2b illustrates this process, where the selected leaf node s_3 is expanded by executing the action a_8, resulting in the creation of a new node s_6. If the state is neither success nor failure, the state of the node can be considered as a statement.

The backpropagation process is illustrated in the right portion of Figure 2b. The current action sequence is updated based on the output of the newly generated leaf node s_6 . The impact of the leaf node's expansion on its parent node is considered by backtracking from the leaf node along its decision path. This process involves updating the associated values $W(s_t)$ += $v(s_t)$ and visit counts $N(s_t)$ += 1.

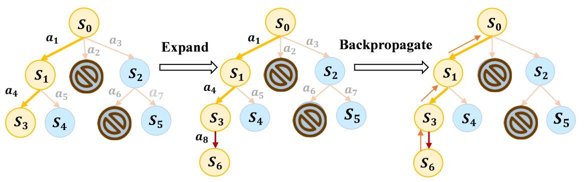
Policy Model and Value Model. QDTSynth incorporates a policy model and a value model in selection phase, which are obtained through online training. These models enhance the efficiency and quality of MCTS-based decision-making during the statement synthesis process.

The objective of the policy model is to generate the probability values for different candidate tactics with a given state. In our approach, when a state s_t and an action a are given, the policy model returns the probability of this action in the given state s_t, which will be used to guide the search processes.

The objective of the value model is to assess the potential for generating more new statements from a given state s_t. Specifically, when the system is in a certain state, the value model estimates whether further exploration in this state is likely to successfully produce valuable new statements. This estimation helps guide the search algorithm by focusing resources on paths that are most likely to synthesize new statements without time-consuming



(a) Selection with adaptive mechanism.



(b) Expansion and Backpropagation.

Figure 2: Adaptive MCTS for Statement Synthesis.

simulation process, thereby optimizing the synthesis process.

3.2 Diversity Screening

When generating statements starting from the same root node, it is easy to produce repetitive or highly similar statements. Therefore, introducing a diversity screening mechanism for the generated statements becomes particularly important.

QDTSynth introduces an online dynamic clustering method to assess the novelty of generated statements. We employ the BERT model (Devlin et al., 2019) to generate context-aware vector representations for text-based statements, achieving incremental clustering through real-time computation of cosine similarity between synthetic statements and dynamic cluster centers. Specifically, for each new synthetic statement, the statement is encoded to a high-dimensional semantic embedding vector e_i using BERT. Subsequently, we compute the cosine similarity between e_i and historical cluster centers in set $C = \{c_1, c_2, ..., c_k\}$, formally defined as:

$$sim(e_i, c_j) = \frac{e_i \cdot c_j}{|e_i||c_j|} \tag{6}$$

If the maximum similarity between the statement embedding and existing cluster centers falls below the predefined threshold, we initialize the statement as a new cluster center($C \leftarrow C \bigcup \{e_i\}$). Otherwise, the position of the most similar cluster center is refined by an exponential smoothing strategy:

$$c_i^{new} = \beta c_i^{old} + (1-\beta)e_i \tag{7}$$

The smoothing factor $\beta \in [0, 1]$ controls the blending weight between historical cluster center and new statement.

3.3 Statement Proving

Through adaptive MCTS and diversity screening, we have successfully obtained many high-quality statements. However, generating proof steps for these statements poses a significant challenge. Employing LLMs for automated proof generation often entails substantial resource consumption and time investment, with no guarantee of successful proof completion. To relieve this issue, for statements generated from seed data extracted from Mathlib4, we prioritize a path backtracking approach to derive proof steps. Specifically, if the node is derived by expanding from an existing statement in Mathlib4, during the proof generation phase, we employ a reverse backtracking strategy to trace the generation path from the current node back to the root node. Throughout this backtracking process, we interact with the Lean proof assistant, providing feedback on the node states and tactic execution. If the Lean proof assistant confirms a successful proof upon reaching the root node, it indicates that the corresponding proof steps for the statement have been successfully generated. Conversely, if the proof fails, these statements, along with those not derived from Mathlib4, are subsequently processed by LLMs for proof generation. Appendix B provides a detailed display of reverse path backtracking

3.4 Self-Assessment and Data Filtering

To ensure the high-quality output of automatically generated theorems, we establish a self-assessment and data filtering mechanism, focusing on quality control for newly generated theorems and their proof steps. The quality of these theorems and their proof steps directly impacts model performance. However, the quality of the generated theorem proofs largely depends on the formal mathematical reasoning capabilities of the LLMs themselves. Due to the inherent complexity and technical challenges of automated theorem proving, the quality of the generated theorem proofs exhibits considerable uncertainty, necessitating systematic quality evaluation and filtering mechanisms for further optimization and refinement.

We introduce a self-assessment mechanism, wherein the LLMs evaluate the quality of the theorems and proof steps they generate. A well-defined evaluation framework is designed, requiring the models to score the theorems based on three dimensions: redundancy, clarity, and relevance. We employ few-shot prompting for self-assessment, providing several expert-crafted and representative examples in prompts. Upon completion of the self-assessment, only those theorems and proofs whose composite scores exceed a predefined threshold are incorporated into the high-quality theorem database. This filtering mechanism effectively ensures the high quality of all theorems and their corresponding proof steps included in the database, providing a reliable data foundation for subsequent model training.

4 Experiments

4.1 Experimental Setup

Models. We selected three popular open-source LLMs as our base models, including Mathstral-7B (Jiang et al., 2023), Llama-3-8B (Dubey et al., 2024) , and Qwen2.5-7B (Yang et al., 2024a). These base models will be fine-tuned using our synthetic dataset. In our approach, the entire process employs a single large language model for all stages, including statement generation, statement proving, self-assessment, and supervised fine-tuning.

Evaluation. We employ the best-first search approach to explore and validate intermediate proof steps within the tactic space generated by large models until the proof is successfully completed or resources are exhausted. For each test theorem, we perform an independent search. At each generation step, the LLM generates 32 candidate proof tactics for the current state. The maximum proof duration for each theorem is limited to 10 minutes.

In this work, we use the miniF2F benchmark to evaluate the performance of our models in formal theorem proving. The miniF2F is a standard test dataset consisting of 244 validation and 244 test formal statements of mathematical problems, sourced from mathematical competitions such as AMC, AIME, and IMO. Our evaluation metric is the proving pass rate of each theorem within ten minutes.

Baselines. We evaluate the effectiveness of our approach by comparing its Lean theorem proving performance against multiple baseline approaches. Specifically, we fine-tune Llama-3-8B, Mathstral-7B, and Qwen2-7B by our synthetic dataset combined with Mathlib4, and assess their performance in Lean theorem proving tasks. To establish a comprehensive benchmark, we compare our models against the original untuned models, models fine-tuned solely on Mathlib4, models trained on datasets synthesized by traditional BFS combined with Mathlib4, models trained on datasets synthesized using conventional MCTS combined with Mathlib4, and models trained on datasets synthesized using MCTS+pvn proposed by Lin et al. (2024) combined with Mathlib4.

Training Details and Dataset. In this study, we utilized LlamaFactory to perform SFT on three base models with LoRA method. Our training configuration was as follows: a learning rate of 2.0×10^{-5}, a cosine learning rate scheduler, and a warm-up ratio of 0.03. We also set the floating-point precision to bfloat16 and used a batch size of 4. During online training of policy and value model, we employ the Adam optimizer to train the networks, with a learning rate set at 5.0×10^{-4}. All training is conducted on a machine running Ubuntu 22.04, equipped with A800-80G $\times$ 4 GPUs.

We decompose the synthetic theorems step-by-step based on their proof traces, extracting each goal and the corresponding tactic applied at each step. The proof traces of all synthetic theorems collectively form our training dataset.

Interaction Tool. The interaction tool with proof environment is essential, which enables us to execute tactics in the current state and receive feedback from the proof environment. We develop an interactive interface called Lean4Repl, implemented directly in Lean over the standard input/output. Through Lean4Repl, we can interact with Lean, allowing provers to observe Lean's proof state, execute tactics to alter the state, and receive feedback from Lean. Additionally, we develop a tool called Lean4Client, which converts Lean files into JSON files for fine-tuning and use with LLMs. This tool systematically breaks down a complete Lean theorem into a step-by-step "goal-tactic-goalAfter" format. Each JSON object contains the current proof state, the tactic executed, and the resulting new state. The version of Lean used in this paper is leanprover/lean4:v4.10.0.

4.2 Main Results

Table 1 presents the performance of our QDTSynth compared with five baseline approaches across Mathstral-7B, Llama3-8B and Qwen2.5-7B. Figure 3 provides a clear comparison of the performance between QDTSynth and the baseline meth-

Training Data	Mathstral-7B		Llama3-8B		Qwen2.5-7B	
	miniF2F-valid	miniF2F-test	miniF2F-valid	miniF2F-test	miniF2F-valid	miniF2F-test
Origin	22.13%	20.90%	23.77%	24.59%	18.44%	20.49%
Mathlib4	31.97%	31.97%	25.82%	25.82%	25.00%	23.36%
Mathlib4+BFS	31.97%	31.15%	26.64%	25.82%	28.69%	29.51%
Mathlib4+MCTS	32.38%	32.79%	27.87%	27.05%	29.92%	30.33%
Mathlib4+MCTS+pvn	32.79%	33.20%	28.69%	28.69%	31.15%	30.33%
Mathlib4+QDTSynth(ours)	**37.70%**	**36.89%**	**33.61%**	**32.79%**	**36.07%**	**35.25%**

Table 1: Results of QDTSynth, compared the pass rates on miniF2F among Mathstral-7B, Llama3-8B and Qwen2.5-7B trained on different datasets.

ods on miniF2F. We analyze the experimental results as follows:

(1) The proposed QDTSynth method demonstrates significant advantages over baseline approaches across three models. The experimental results on Mathstral-7B show that QDTSynth achieves pass rates of 37.70% and 36.89% on the miniF2F-valid and miniF2F-test, surpassing the suboptimal method (MCTS+pvn at 32.79% and 33.20%) by margins of 4.91% and 3.69%. This marked improvement confirms that QDTSynth enhances the quality of synthetic theorems through quality-driven mechanisms, generating high-quality Lean4 theorem datasets that substantially enhance model performance in proving tasks.

(2) Although the integration of policy/value networks (pvn) with traditional MCTS has resulted in a slight increase in pass rates, QDTSynth introduces three critical refinements: adaptive mechanisms, diversity screening, and self-assessment, which enable QDTSynth to outperform MCTS+pvn by 4.91% and 3.69% on Mathstral-7B. The results demonstrate that optimizing search strategies alone has limited effectiveness in improving data quality. Due to the complexity of formal theorem synthesis and proving, it is necessary to assess and filter the synthetic theorems.

(3) It is noteworthy that, although BFS shows performance improvements in Qwen2.5-7B, it only yields a slight improvement on Llama3-8B and even experienced performance degradation on Mathstral-7B (a decrease of 0.82% compared to Mathlib4 on the miniF2F-test). This phenomenon underscores the importance of data quality in the supervised SFT of LLMs. Although BFS can generate a larger volume of training data through exhaustive search, the low-quality proof paths it produces lead to the model learning incorrect reasoning patterns. These experimental results suggest that in LLM-based theorem proving systems, blindly increasing the scale of data may be counterproductive, and the quality of the training dataset is crucial for performance enhancement.

4.3 Ablation Study

We use data synthesized from Mathstral-7B and conduct a series of ablation experiments to further investigate the effects of the adaptive mechanism, diversity screening, and self-assessment on model training. The experimental results are obtained from the pass rates of the trained Mathstral-7B on miniF2F.

Training Data	miniF2F-valid	miniF2F-test
QDTSynth	37.70%	36.89%
- w/o Pen	36.07% (-1.63)	35.66% (-1.23)
- w/o Dynamic c	36.89% (-0.81)	36.48% (-0.41)
- w/o Both	35.66% (-2.04)	35.25% (-1.64)

Table 2: Ablation results of the adaptive mechanism in statement synthesis on Mathstral-7B. 'Pen' represents the penalty term, and 'Dynamic c' refers to the adaptive dynamic exploration coefficient c.

Adaptive Mechanism. Table 2 demonstrates the critical impact of the penalty term and dynamic exploration coefficient c in the adaptive mechanism. Removing the penalty term leads to pass rates reductions of 1.63% and 1.23% on the miniF2F-valid and miniF2F-test, confirming its essential role in suppressing invalid proof paths. This observation indicates that low-quality proof steps significantly disrupt the training effect. Removal of the dynamic exploration coefficient c results in performance declines of 0.81% and 0.41%, illustrating its role

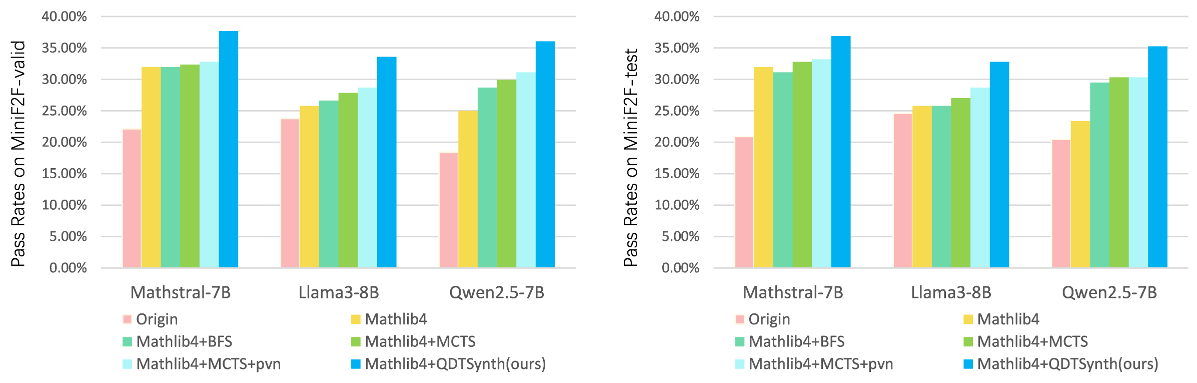


Figure 3: Comparison of QDTSynth with Baseline Methods on miniF2F pass rates across Mathstral-7B, Llama3-8B, and Qwen2.5-7B.

in optimizing the efficiency of the search strategy through dynamic adjustment of the exploration weights. Notably, the penalty term has a more pronounced influence on model performance than dynamic c, further underscoring the importance of data quality in theorem synthesis tasks. The detrimental effects of low-quality proof paths far outweigh the limitations of localized search strategy optimizations. When both the penalty term and dynamic c are removed, the performance degradation (-2.04% and -1.64%) exceeds the sum of their individual losses (-2.44% and -1.64%). This finding reveals a mutually dependent enhancement mechanism between the two components. The results highlight the effectiveness of our adaptive mechanism in model training.

Training Data	miniF2F-valid	miniF2F-test
QDTSynth	37.70%	36.89%
- w/o Diversity	36.48% (-1.22)	35.66% (-1.23)
- w/o SA	34.43% (-3.27)	34.84% (-2.05)
BFS	31.97%	31.15%
- w/ Both	32.79% (+0.82)	32.79% (+1.64)
MCTS+pvn	32.79%	33.20%
- w/ Both	35.66% (+2.87)	35.25% (+2.05)

Table 3: Ablation results of the diversity screening and self-assessment on Mathstral-7B. 'Diversity' and 'SA' denote diversity screening and self-assessment respectively

Diversity Screening and Self-Assessment. From Table 3, it is evident that diversity screening (Diversity) and self-assessment (SA) play an important role in theorem synthesis. Removing diversity screening caused performance drops of 1.22% on miniF2F-valid and 1.23% on miniF2F-test, indicating that similar training data restrict the model's ability to learn diverse reasoning patterns. The removal of the self-assessment module results in a more significant performance degradation (-3.27% and -2.05%), indicating that our self-assessment method effectively filters out high-quality formalized theorems, thereby enhancing the model's proof performance. To further validate the effectiveness of these two components, we integrated them into BFS and MCTS+pvn. Experimental results show that our components are highly effective in filtering out similar or low-quality theorems, contributing to the synthesis of high-quality training theorems.

5 Conclusion

In this work, we propose QDTSynth, an approach to quality-driven synthesis of formal theorems, focusing on the synthesis of high-quality Lean4 theorems from formal statements. QDTSynth enhances Monte Carlo Tree Search (MCTS) with an adaptive adjustment mechanism that dynamically optimizes the statement synthesis process, and further enhances theorem quality by incorporating diversity screening and self-assessment mechanisms, thereby significantly improving the diversity and high quality of the synthetic theorems. We perform supervised fine-tuning on three open-source LLMs using the synthetic dataset and evaluate the effectiveness of QDTSynth in Lean4 theorem proving on miniF2F. Experimental results demonstrate that QDTSynth significantly improves the performance of various open-source LLMs in theorem proving tasks. QDTSynth provides a novel direction for automated formal theorem synthesis.

Limitations

Despite QDTSynth's outstanding performance in theorem proving, several limitations must be acknowledged. QDTSynth uses seed data as the root node for statement expansion, which limits the diversity and quality of the generated data based on the coverage of the initial seed data. In the future, we can extract seed data from open source theorem sets such as DeepSeek-Prover(Xin et al., 2024) and Lean Workbook(Ying et al., 2024). Furthermore, for formal statements not derived from the Mathlib4 library, the method relies on LLMs to autonomously generate proof steps. This can lead to resource wastage, and the ability to successfully generate theorems depends on the proof capabilities of the LLMs. These issues need to be further addressed and improved, which may further enhance the quality of the synthetic data and synthesis efficiency.

Ethics Statement

In our work, we use LLMs for generating candidate tactics, statement proving, and self-assessment. We have utilized Mathstral-7B, Llama3-8B and Qwen2.5-7B, as well as open-source software such as Hugging Face and PyTorch. Our models can output untrue hallucinations, just like any language model.We adhere to the policies and licenses of these resources and acknowledge the role they have played in our work.

Acknowledgement

This work was supported in part by the National Key Research and Development Program of China under Grant (2023YFA1009402), the National Natural Science Foundation of China under Grant (No.12171159), the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant (No.22510750100), and the Shanghai Trusted Industry Internet Software Collaborative Innovation Center. This work was also supported by the GPU computing platform of the Academy of Mathematics and Systems Science, Chinese Academy of Science.

References

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. *Preprint*, arXiv:2402.00157.

Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An environment for machine learning of higher order logic theorem proving. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR.

Richard Bellman. 1966. Dynamic programming. *science*, 153(3731):34–37.

Wolfgang Bibel. 2013. *Automated theorem proving*. Springer Science & Business Media.

David Brandfonbrener, Sibi Raja, Tarun Prasad, Chloe Loughridge, Jianang Yang, Simon Henniger, William E Byrd, Robert Zinkov, and Nada Amin. 2024. Verified multi-step synthesis using large language models and monte carlo tree search. *arXiv preprint arXiv:2402.08147*.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

Maosong Cao, Taolin Zhang, Mo Li, Chuyu Zhang, Yunxin Liu, Haodong Duan, Songyang Zhang, and Kai Chen. 2025. Condor: Enhance llm alignment with knowledge-driven data synthesis and refinement. *Preprint*, arXiv:2501.12273.

Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. 2008. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 216–217.

Projet Coq. 1996. The coq proof assistant-reference manual. *INRIA Rocquencourt and ENS Lyon, version*, 5.

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. *Preprint*, arXiv:1810.04805.

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

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

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.

Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. 2021. Tactictoe: learning to prove with tactics. *Journal of Automated Reasoning*, 65(2):257–286.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. 2021. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203*.

Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. Deepmath-deep sequence models for premise selection. *Advances in neural information processing systems*, 29.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Andrea Kang, Jun Yu Chen, Zoe Lee-Youngzie, and Shuhao Fu. 2024. Synthetic data generation with llm for improved depression prediction. *Preprint*, arXiv:2411.17672.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

Mitsuru Kusumoto, Keisuke Yahata, and Masahiro Sakai. 2018. Automated theorem proving in intuitionistic propositional logic by deep reinforcement learning. *Preprint*, arXiv:1811.00796.

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

Xiaohan Lin, Qingxing Cao, Yinya Huang, Zhicheng Yang, Zhengying Liu, Zhenguo Li, and Xiaodan Liang. 2024. Atg: Benchmarking automated theorem generation for generative language models. *arXiv preprint arXiv:2405.06677*.

Donald W Loveland. 2016. *Automated theorem proving: A logical basis*. Elsevier.

Zimu Lu, Aojun Zhou, Houxing Ren, Ke Wang, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024. Mathgenie: Generating synthetic data with question back-translation for enhancing mathematical reasoning of llms. *Preprint*, arXiv:2402.16352.

Alisia Lupidi, Carlos Gemmell, Nicola Cancedda, Jane Dwivedi-Yu, Jason Weston, Jakob Foerster, Roberta Raileanu, and Maria Lomeli. 2024. Source2synth: Synthetic data generation and curation grounded in real data sources. *Preprint*, arXiv:2409.08239.

The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20. ACM.

Norman Megill and David A Wheeler. 2019. *Metamath: a computer language for mathematical proofs*. Lulu. com.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pages 625–635, Cham. Springer International Publishing.

Jens Otten and Wolfgang Bibel. 2003. leancop: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1-2):139–161.

Jinlong Pang, Na Di, Zhaowei Zhu, Jiaheng Wei, Hao Cheng, Chen Qian, and Yang Liu. 2025. Token cleaning: Fine-grained data selection for llm supervised fine-tuning. *Preprint*, arXiv:2502.01968.

Jeiyoon Park, Chanjun Park, and Heuiseok Lim. 2024. Chatlang-8: An llm-based synthetic data generation framework for grammatical error correction. *Preprint*, arXiv:2406.03202.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. 2022. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*.

Arthur L. Robinson. 1980. New Ways to Make Microcircuits Smaller—Duplicate Entry. *Science*, 208:1019–1026.

Ming Shen. 2024. Rethinking data selection for supervised fine-tuning. *Preprint*, arXiv:2402.06094.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Pragya Srivastava, Manuj Malik, Vivek Gupta, Tanuja Ganu, and Dan Roth. 2024. Evaluating llms' mathematical reasoning in financial document question answering. *Preprint*, arXiv:2402.11194.

Zhen Tan, Dawei Li, Song Wang, Alimohammad Beigi, Bohan Jiang, Amrita Bhattacharjee, Mansooreh Karami, Jundong Li, Lu Cheng, and Huan Liu. 2024. Large language models for data annotation and synthesis: A survey. *Preprint*, arXiv:2402.13446.

Rahul Vishwakarma and Subhankar Mishra. 2023. Enhancing neural theorem proving through data augmentation and dynamic sampling method. *arXiv preprint arXiv:2312.14188*.

Haiming Wang, Ye Yuan, Zhengying Liu, Jianhao Shen, Yichun Yin, Jing Xiong, Enze Xie, Han Shi, Yujun Li, Lin Li, et al. 2023a. 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)*, pages 12632–12646.

Ke Wang, Jiahui Zhu, Minjie Ren, Zeming Liu, Shiwei Li, Zongye Zhang, Chenkai Zhang, Xiaoyu Wu, Qiqi Zhan, Qingjie Liu, and Yunhong Wang. 2024a. A survey on data synthesis and augmentation for large language models. *Preprint*, arXiv:2410.12896.

Mingzhe Wang and Jia Deng. 2020. Learning to prove theorems by learning to generate theorems. *Advances in Neural Information Processing Systems*, 33:18146–18157.

Ruida Wang, Wangchunshu Zhou, and Mrinmaya Sachan. 2023b. Let's synthesize step by step: Iterative dataset synthesis with large language models by extrapolating errors from small models. *Preprint*, arXiv:2310.13671.

Zifeng Wang, Chun-Liang Li, Vincent Perot, Long T. Le, Jin Miao, Zizhao Zhang, Chen-Yu Lee, and Tomas Pfister. 2024b. Codeclm: Aligning language models with tailored synthetic data. *Preprint*, arXiv:2404.05875.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

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

Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. 2021. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34:9330–9342.

Yuhuai Wu, Albert Qiaochu Jiang, Jimmy Ba, and Roger Grosse. 2020. Int: An inequality benchmark for evaluating generalization in theorem proving. *arXiv preprint arXiv:2007.02924*.

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

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

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

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2024. Magpie: Alignment data synthesis from scratch by prompting aligned llms with nothing. *Preprint*, arXiv:2406.08464.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024a. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2024b. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. 2024. Lean workbook: A large-scale lean problem set formalized from natural language math problems. *Preprint*, arXiv:2406.03847.

Di Zhang, Jiatong Li, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. 2024. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*.

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

He Zhu, Junyou Su, Tianle Lun, Yicheng Tao, Wenjia Zhang, Zipei Fan, and Guanhua Chen. 2024. Fanno: Augmenting high-quality instruction data with open-sourced llms only. *Preprint*, arXiv:2408.01323.

A Finetuning Details

We present the hyperparameters used for LoRA training in the LLamaFactory as Figure 4.

We convert the synthetic theorems into the Alpaca format. In the Lean proof environment, one formalized theorem is as Figure 5.

```
Hyperparameters for LoRA Training

--stage: sft
--do_train: true
--finetuning_type: lora
--lora_target: all
--deepspeed-examples/deepspeed/ds_z0_config.json
--template: mistral
--cutoff_len: 4096
--max_samples: 100000000
-- overwrite_cache: true
-- preprocessing_num_workers: 16
-- output_dir: saves/
-- logging_steps: 10
-- save_steps: 100
-- plot_loss: true
-- overwrite_output_dir: true
-- per_device_train_batch_size: 4
-- gradient_accumulation_steps: 4
-- learning_rate: 2.0e-5
-- num_train_epochs: 3
-- lr_scheduler_type: cosine
-- warmup_ratio: 0.01
-- bf16: true
-- ddp_timeout: 180000000
-- val_size: 0.01
-- per_device_eval_batch_size: 2
-- eval_strategy: steps
-- eval_steps: 2000
```

Figure 4: Hyperparameters for LoRA Training.

```
theorem div_mul_add_mod (m n : Nat) : m / n *
n + m % n = m := by
  rw [←mul_comm]
  rw [div_add_mod]
```

Figure 5: An example of Lean4 Theorem.

After reading the state of a theorem, the model should provide effective tactics. The `GOAL` and `tactic` from the converted theorems are extracted as the "input" and "output" portions of the fine-tuning dataset respectively. The supervised dataset format is as Figure 6.

B Details of Statement Proving

In statement proving phase, for statements synthesized from seed data extracted from Mathlib4, we prioritize a path backtracking approach to derive proof steps. Specifically, if the node is derived by expanding from an existing statement in Mathlib4, we employ a reverse backtracking strategy to trace the generation path from the current node back to the root node. Taking the Mathlib4 theorem in Figure 8 as an example, this theorem serves as the root

```
[
  {
    "input": "[GOAL]m n : ℕ ⊢ m / n * n + m % n = m[tactic]",
    "output": "rw [← mul_comm]",
    "instruction": "",
    "history": []
  },
  {
    "input": "[GOAL]m n : ℕ ⊢ n * (m / n) + m % n = m[tactic]",
    "output": "rw [div_add_mod]",
    "instruction": "",
    "history": []
  }
]
```

Figure 6: Supervised Dataset Format.

node to synthesize new statements, and backtrack their synthesis path. Figure 7 shows in detail the statement proving process synthesized by this theorem. Starting from the root node, a_1, a_4, and a_8 are selected. During the statement proving, we trace the synthesis path back from the reverse a_8 until reaching the root node, and finally executed "rw" tactic and assumption (may not be necessary). The complete proof steps of the new statement obtained is shown in Figure 9.

C Prompts

For better reproduction, we have provided all prompt templates in the appendix. We list the following for reference:

Figure 10: Generating candidate tactics for the input state.

Figure 11: Generate proof steps for the current statement, using them in both the statement proving and evaluation stages.

Figure 12: Self-assessment and scoring for synthetic theorems based on three dimensions: redundancy, clarity, and relevance.

Figure 13: An example for self-assessment.

D Interactive Tool

We develop an interactive interface called `Lean4Repl`, implemented directly in Lean over the standard input/output. Through Lean4Repl, we can interact with Lean, allowing provers to observe Lean's proof state, execute tactics to alter the state, and receive feedback from Lean. `Lean4Repl` presents the following API:

• `Lean4Gym(lean_workdir, lean_file)`: Initializes an instance of the Lean4Gym class, based on the root path of the Lean project and the file path of the initial theorem.

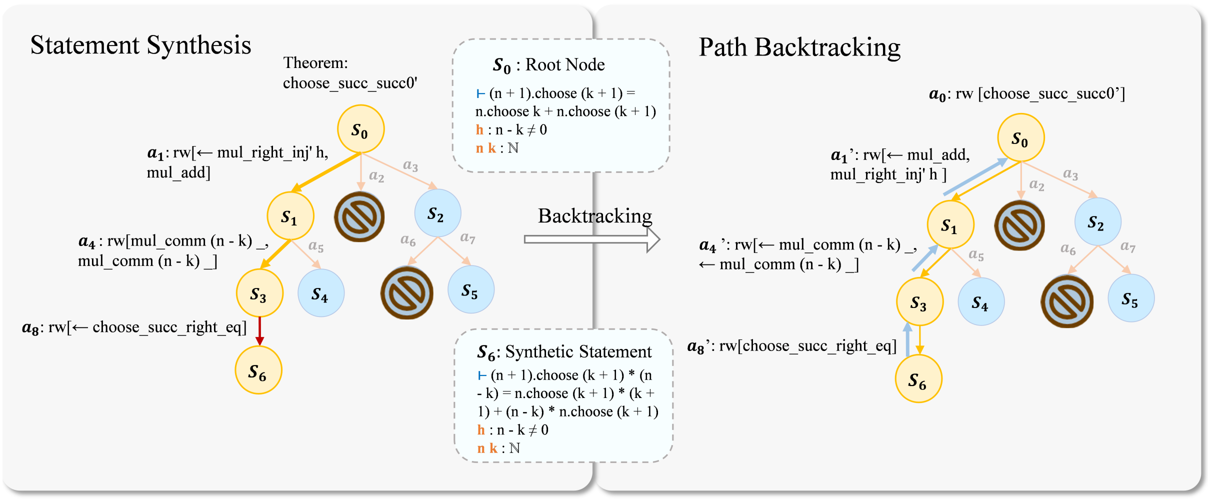

Statement Synthesis
Theorem: choose_succ_succ0'
S_0
a_1: rw[← mul_right_inj' h, mul_add]
a_2
a_3
S_1
S_2
a_4 : rw[mul_comm (n - k) _, mul_comm (n - k) _]
a_5
a_6
a_7
S_3
S_4
S_5
a_8: rw[← choose_succ_right_eq]
S_6
S_0 : Root Node
⊢ (n + 1).choose (k + 1) = n.choose k + n.choose (k + 1)
h : n - k ≠ 0
n k : ℕ
Backtracking
S_6: Synthetic Statement
⊢ (n + 1).choose (k + 1) * (n - k) = n.choose (k + 1) * (k + 1) + (n - k) * n.choose (k + 1)
h : n - k ≠ 0
n k : ℕ
Path Backtracking
a_0: rw [choose_succ_succ0']
a_1': rw[← mul_add, mul_right_inj' h]
a_4 ': rw[← mul_comm (n - k) _, ← mul_comm (n - k) _]
a_8': rw[choose_succ_right_eq]

Figure 7: An example of path backtracking during statement proving.

```
theorem Nat.choose_succ_succ0' (n : ℕ)
(k : ℕ)(h: n - k ≠ 0) :
Nat.choose (n + 1) (k + 1) = Nat.choose
n k + Nat.choose n (k + 1)
```

Figure 8: An example as the seed data from Mathlib4.

```
theorem new_choose (n : ℕ) (k : ℕ)(h: n - k ≠ 0) :
(n + 1).choose (k + 1) * (n - k) = n.choose (k + 1)
* (k + 1) + (n - k) * n.choose (k + 1) := by
  rw[choose_succ_right_eq]
  rw[← mul_comm (n - k) _, ← mul_comm (n - k) _]
  rw[←mul_add, mul_right_inj' h]
  rw[Nat.choose_succ_succ0']
  assumption
```

Figure 9: The synthetic theorem and its proof steps.

• `getInitState()`: Extracts the initial state of the theorem from the `lean_file`

• `run_tactic(state, tac)`: Facilitates interaction with Lean through Lean4Repl by inputting the state and tactic, and returns feedback from Lean.

Additionally, we develop a tool called Lean4Client, converting Lean files into JSON files. The tool breaks down a complete Lean theorem into a step-by-step "goal-tactic-goalAfter" format. Each JSON object contains the current proof state, the tactic executed, and the resulting new state. Figure 14 displays the converted data format.

Prompt for Statement Synthesis

You are using Lean4 for theorem generation. Now give you the current state of the theorem you need to give me a tactic to generate high-quality theorems as:
[GOAL] <Input State> [tactic]
You should generate a tactic that can help generate high-quality theorems. If you use a "rw" or "simp" tactic, please do not rewrite multiple theorems at the same time. It is necessary to ensure that the executed tactics can be executed successfully without errors
(Note: Do not output any extra content, only the tactic itself in plain text.)

Figure 10: Prompt template to generate tactics for statement synthesis.

Prompt for Theorem Proving

You are using Lean4 for theorem proving. Now give you the current state of the theorem you need to prove in Lean4 language as:
[GOAL] <Input State> [tactic]
You should generate a tactic that can help prove the theorem.
(Note: Do not output any extra content, only the tactic itself in plain text.)

Figure 11: Prompt template for theorem proving.

Prompt for Self-Assessment

The user will provide a section of the Lean4 theorem and its proof steps. You need to score the proof of the theorem from the following three aspects, provide detailed evaluation reasons, and give a total score for the theorem based on the following three aspects:

1. Redundancy (0-1 points):
Evaluate whether there are unnecessary repetitions or redundancies in the proof steps. A high rating indicates efficiency and no redundancy.
Scoring criteria: 0 indicates complete redundancy, 1 indicates no redundancy.

2. Clarity (0-1 points):
Evaluate whether the logic of the proof is clear and whether the steps are easy to understand. A high rating indicates that it is easy to understand and well structured.
Scoring criteria: 0 indicates very unclear, 1 indicates very clear.

3. Relevance (0-1 points):
Evaluate whether the proof always revolves around the goal and avoids irrelevant steps or lemmas. A high rating indicates proof of focusing on the goal.
Scoring criteria: 0 indicates complete deviation from the target, 1 indicates complete focus on the target.

Please strictly follow the following format for output:
1. Redundancy rating: X/1
Explanation: [Briefly explain the reasons for the redundancy rating]
2. Clarity rating: X/1
Explanation: [Briefly explain the reasons for the clarity rating]
3. Relevance score: X/1
Explanation: [Briefly explain the reasons for the relevance rating]
Total score: X/3
Summary: [Briefly summarize the overall quality of the theorem, with a focus on redundancy, clarity, relevance, and other aspects]
[Examples]

Now, the Lean4 theorem entered by the user is as follows:
[Theorem]
<Input Theorem>
[Response]

Figure 12: Prompt template for self-assessment.

An Example for Self-Assessment

Example:
The Lean4 theorem entered by the user is as follows:

```
  [Theorem]
  theorem mathd_numbertheory_3012525(j : ℕ) (h₀ : 0 < j):(3 * 3 + 3 * (7 * j)) % 7 = 2 :=
by
   simp [mul_comm]
   simp [Nat.add_comm, Nat.mul_comm, Nat.add_assoc, Nat.mul_assoc]
   rw [mul_comm, mul_assoc, mul_comm, mul_assoc, add_comm]
   simp [mul_comm]
  [Response]
```

Response:
1. Redundancy rating: 0.6/1
Explanation:
The proof contains multiple simp steps with overlapping lemmas, such as mul_comm and Nat.mul_comm, which are repeated unnecessarily. The use of rw [mul_comm, mul_assoc, mul_comm, mul_assoc, add_comm] is also redundant, as some of these rewrites could be combined or simplified. The proof could be made more efficient by reducing repetitive steps.

2. Clarity rating: 0.5/1
Explanation:
The proof logic is somewhat clear, but the repeated use of simp with similar lemmas and the lengthy rw step make it harder to follow. The proof could benefit from clearer structuring and fewer overlapping steps. A more streamlined approach would improve readability.

3. Relevance score: 0.8/1
Explanation:
The proof stays focused on the goal of showing (3 * 3 + 3 * (7 * j)) % 7 = 2, and all steps are relevant to this goal. However, the inclusion of unnecessary lemmas (e.g., Nat.add_comm, Nat.mul_comm) slightly detracts from the relevance, as they do not contribute significantly to the proof.

Total score: 1.9/3
Summary:
The theorem's proof is relevant and mostly focused on the goal, but it suffers from redundancy and lacks clarity due to repetitive and overlapping steps. The proof could be significantly improved by streamlining the use of simp and rw to eliminate redundancy and make the logic clearer. Overall, the proof is functional but not optimal in terms of efficiency and readability.

Figure 13: An example for self-assessment.

```
{
   "tactics": [
      {
         "tactic": "rw [← mul_comm]",
         "proofState": 0,
         "pos": {
           "line": 11,
           "column": 0
          },
          "goalsAfter": "unknown goal",
          "goals": "m n : N ⊢ m / n * n + m % n = m",
          "endPos": {
           "line": 11,
           "column": 14
          }
      },
      {
          "tactic": "rw [div_add_mod]",
          "proofState": 1,
          "pos":{
             "line": 12,
             "column": 0
           },
           "goalsAfter": "no goals",
           "goals": "m n : N ⊢ n * (m / n) + m % n = m",
           "endPos": {
            "line": 12,
            "column": 15
           }
       }
    ]
}
```

Figure 14: Theorem with "goal-tactic-goalAfter" format.