Learn from Failure: Fine-Tuning LLMs with Trial-and-Error Data for Intuitionistic Propositional Logic Proving

Chenyang An[1*], **Zhibo Chen**[2*], **Qihao Ye**[1], **Emily First**[1], **Letian Peng**[1]
Jiayun Zhang[1], **Zihan Wang**[1†], **Sorin Lerner**[1†], **Jingbo Shang**[1†]
University of California, San Diego[1] Carnegie Mellon University[2]
{c5an, q8ye, emfirst, lepeng, jiz069, ziw224, lerner, jshang}@ucsd.edu
zhiboc@andrew.cmu.edu

Abstract

Recent advances in Automated Theorem Proving have shown the effectiveness of leveraging a (large) language model that generates tactics (i.e. proof steps) to search through proof states. The current model, while trained solely on successful proof paths, faces a discrepancy at the inference stage, as it must sample and try various tactics at each proof state until finding success, unlike its training which does not incorporate learning from failed attempts. Intuitively, a tactic that leads to a failed search path would indicate that similar tactics should receive less attention during the following trials. In this paper, we demonstrate the benefit of training models that additionally learn from failed search paths. Facing the lack of such trial-and-error data in existing open-source theorem-proving datasets, we curate a dataset on intuitionistic propositional logic theorems and formalize it in Lean, such that we can reliably check the correctness of proofs. We compare our model trained on relatively short trial-and-error information (TRIALMASTER) with models trained only on the correct paths and discover that the former solves more unseen theorems with lower trial searches.

1 Introduction

Automated Theorem Proving is a challenging task that has recently gained popularity in the machine-learning community. Researchers build *neural theorem provers* to synthesize formal proofs of mathematical theorems (Yang et al., 2023; Welleck et al., 2021; Lample et al., 2022; Mikuła et al., 2023; Wang et al., 2023; Bansal et al., 2019; Davies et al., 2021; Wu et al., 2021; Rabe et al., 2020; Kusumoto et al., 2018; Bansal et al., 2019; Irving et al., 2016). Typically, a neural theorem prover, given a partial proof and the current *proof state*, uses a neural model to predict the next likely *proof step*, or *tactics*. The neural models utilize different architectures like LSTMs (Sekiyama et al., 2017), CNNs (Irving et al., 2016), DNNs (Sekiyama and Suenaga, 2018), GNNs (Bansal et al., 2019; Wang et al., 2017) and RNNs (Wang and Deng, 2020), though most recent work has begun to explore the use of transformer-based large language models (LLMs) due to their emerging reasoning abilities.

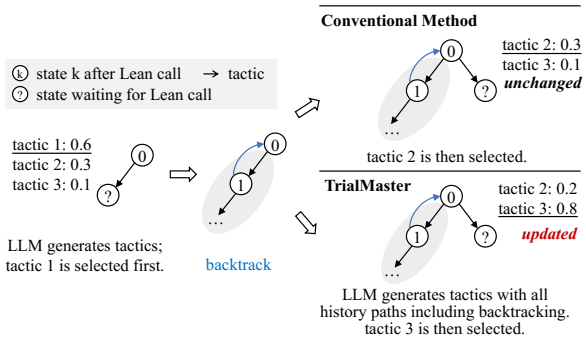


Figure 1: A simple example for how learning trial-and-error data impacts inference distribution. See Figure 4 for a concrete case.

An interactive *proof assistant*, such as Lean (de Moura et al., 2015), Coq (Barras et al., 1997) or Isabelle (Nipkow et al., 2002), evaluates the model's predicted candidate proof steps, returning either new proof states or errors. Neural theorem provers iterate on this procedure, performing *proof search*, e.g., a depth-first search (DFS), to traverse the space of possible proofs. An example of a DFS proof search is illustrated in Figure 2a, where the prover progressively generates new tactics if the attempted tactics result in incorrect proofs.

Such provers are usually trained on a dataset containing only the correct proof paths. This, however, presents a limitation: during inference, the prover does not have the ability to leverage the already failed paths it explored. Such failure information, intuitively, is beneficial, as it could sug-

[*] The first two authors contributed equally to this work.
[†] Corresponding authors.

gest the model to generate tactics similar to the failed ones sparingly. At the very least, the failure information should help the model easily avoid generating already failed tactics. See Figure 1.

In this paper, we wish to empirically verify this intuition. To conduct the experiment, we would compare the conventional model trained on correct proof paths, and TRIALMASTER, the model trained on the whole proof tree, containing both correct paths and incorrect paths. See Figure 2b. As such, TRIALMASTER can make predictions based on the failure information during inference time.

Since current open-source Automated Theorem Proving datasets do not contain complete proof trees, we create such a dataset, PropL, written in Lean. We focus on theorems of intuitionistic propositional logic. A propositional logic formula in intuitionistic propositional logic A is true if and only if $\vdash A$ has a derivation according to the rules of the intuitionistic propositional logic. These rules are listed in Figure 6 in Appendix A with explanations. This is to be distinguished from the classical logic, where a theorem of classical propositional logic admits a proof if and only if under all truth value assignments to all propositions that appear in the theorem, the theorem statement is evaluated to be true.

We give two elementary examples of theorems of intuitionistic propositional logic with proofs. We then give an example of a theorem whose proof contains a backtracking step. The first theorem admits a proof, where as the second doesn't.

```
theorem_1 thm : p1 → p1 ∨ p2 := by
    intro h1
    apply or.inl
    exact h1
```

The first theorem states that p1 implies p1 ∨ p2, where ∨ stands for "or". To show this, we assume p1 and prove either p1 or p2. We prove p1 in this case. The last three lines are tactics (proof steps) in Lean used to prove this fact.

Our second example looks like the following.

```
theorem_2 thm : p1 → p1 ∧ p2 := by
```

The second theorem states that p1 implies p1 and p2. There is no proof to this theorem. When we assume p1, we cannot show both p1 and p2.

Proofs might include backtracking instructions. Here is another possible proof for theorem 1. After step 2, there is no possible proof step that can lead to final solution. Therefore we backtrack to step 1 and try a different proof step.

```
theorem_1 thm : p1 → p1 ∨ p2 := by
    intro h1 #this is step 1
    apply or.inr #this is step 2
    no solution, backtrack to step 1
    apply or.inl
    exact h1
```

Note that "no solution, backtrack to step 1" is not a tactic in Lean. It simply tells the system to backtrack to step 2 and start reasoning from there.

Specifically, our PropL dataset is created through a two-stage process that first involves generating a comprehensive set of propositional logic theorems by uniformly sampling from all possible theorems, utilizing a bijection between natural numbers and propositions to ensure representativeness. Following theorem generation, proofs are constructed using a focusing method with polarization, incorporating a detailed trial-and-error search process that includes both successful and backtracked steps, thereby capturing the complexity and nuances of theorem proving in intuitionistic propositional logic. Thus, our dataset is complete, scalable, and representative. The proofs in our dataset are combined with trial-and-error information, which is generated by the Focused Proof Search (FPS) algorithm (McLaughlin and Pfenning, 2009; Liang and Miller, 2009; Pfenning, 2017).

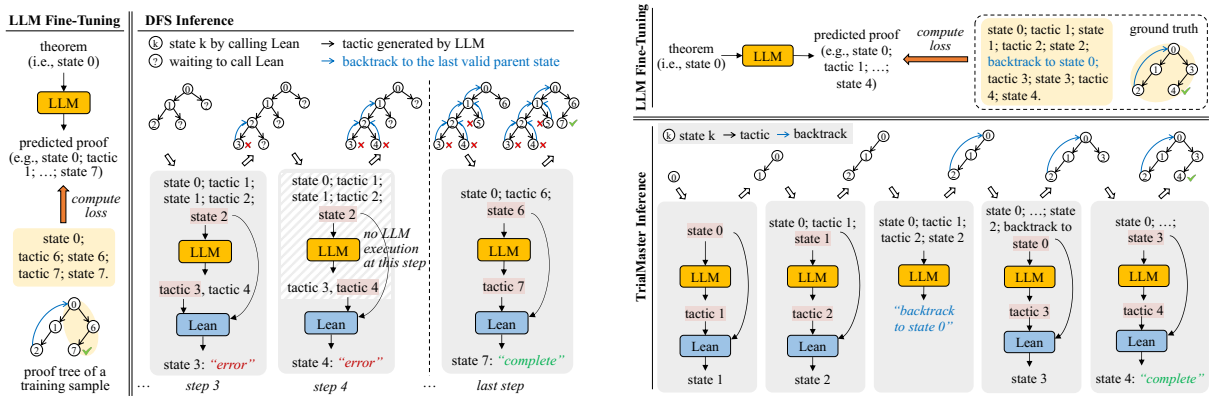
We verify the effectiveness of incorporating the failed trials during training and inference by experiments on PropL, observing that TRIALMASTER achieves a higher proof search success rate and lower search cost over conventional model trained on correct proof paths. Our experiments further indicate that our model can perform backtracking without help from an external system.

Our main contributions are as follows:

- We establish, PropL, a complete, scalable, and representative benchmark for intuitionistic propositional logic theorems formalized in Lean. PropL includes proofs with trial-and-error information, generated by the FPS algorithm.
- We demonstrate that for intuitionistic propositional logic theorem proving, incorporating trial-and-error information into training and proving outperforms a conventional model that is trained on correct proofs only.

2 Related Work

Automated Theorem Proving. Automated Theorem Proving has evolved significantly since its inception, focusing mainly on developing com-



(a) Conventional system with depth-first search

(b) Our training and inference methodology

Figure 2: Method comparison. (a) A conventional system: The tactic generator (i.e., LLM) is fine-tuned on correct proof paths only. During inference, the trained tactic generator produces N_{sampled} (e.g., 2 in the example) tactics at a time. If Lean decides that the current tactic is wrong, the system backtracks to the last valid state and tries other candidate tactics. (b) Our methodology: The tactic generator is fine-tuned on proofs with trial-and-error. During inference, we take the first tactic it generates and feed that into Lean for state checking at each step.

puter programs that can autonomously prove mathematical theorems. Early ATP systems (mechanical theorem proving) were based on first-order logic (Clocksin and Mellish, 2003; Chang and Lee, 2014), where the resolution method (Robinson, 1965) played a crucial role. Recent progress in Automated Theorem Proving has been marked by the integration of machine learning (Bansal et al., 2019; Davies et al., 2021; Wagner, 2021), especially LLMs (Yang et al., 2023; Polu and Sutskever, 2020; Han et al., 2021; Welleck et al., 2021; Jiang et al., 2022), and heuristic methods (Holden and Korovin, 2021), aimed at amplifying the efficiency and capacity of Automated Theorem Proving systems. Within the domain of LLMs, formal mathematical languages like Metamath (Megill and Wheeler, 2019), Lean (de Moura et al., 2015), Isabelle (Nipkow et al., 2002), and Coq (Barras et al., 1997), serve as a bridge, enabling the precise expression and verification of mathematical theorems and concepts through a computer-verifiable format, thereby mitigating hallucination risks (Nawaz et al., 2019). COPRA incorporates backtracking information into a prompt and sends the prompt to GPT-4 without fine-tuning it to perform proof search (Thakur et al., 2023). Baldur fine-tunes LLMs with proofs and error information given by the proof assistant (First et al., 2023). In contrast, our work focuses on fine-tuning LLMs with the complete past proof history without using the error message from the proof assistant.

Propositional Logic Problem. Early implementations of ATP systems demonstrated the potential for computers to automate logical deductions, with notable examples including the Logic Theorist (Crevier, 1993; McCorduck, 2004; Russell and Norvig, 2010) and Gilmore's program (Davis, 2001; Gilmore, 1960). These systems laid the groundwork for the resolution of propositional logic problems, showcasing the ability of automated systems to handle logical reasoning tasks. Recent advancements in Automated Theorem Proving have revisited propositional logic problems, integrating modern computational techniques. Sekiyama et al. (Sekiyama and Suenaga, 2018) have employed Deep Neural Networks (DNNs) as a statistical approach to generate proofs for these theorems, while Kusumoto et al. (Kusumoto et al., 2018) have explored graph representations coupled with reinforcement learning to find proofs. Furthermore, sequence-to-sequence neural networks have been applied for deriving proof terms in intuitionistic propositional logic (Sekiyama et al., 2017). This area of research is particularly intriguing due to the simplicity and importance of propositional logic in mathematics, and there is a growing interest in evaluating the capability of LLMs in tackling this specific mathematical domain.

Trial-and-Error. The Chain-of-Thought (CoT) (Wei et al., 2022; Wang et al., 2022; Zhou et al., 2022; Fu et al., 2022; Chu et al., 2023; Yu et al., 2023) approach demonstrates that LLMs can be guided to perform step-by-step reasoning by incorporating intermediate reasoning steps in

their prompts. This concept is expanded in later research, such as the Tree of Thoughts (ToT) (Yao et al., 2023), which organizes reasoning into a tree structure, and the Graph of Thoughts (GoT) (Besta et al., 2023), which adopts a graph format for thought structuring. Trial-and-error complements structured reasoning by allowing the model to empirically test hypotheses generated, thereby refining its reasoning process based on feedback from interactions or emulations. The Boosting of Thoughts (BoT) (Chen et al., 2024) prompting framework iteratively explores and evaluates multiple trees of thoughts to gain trial-and-error reasoning experiences, using error analysis from the LLMs to revise the prompt. Lyra introduces correction mechanisms to improve the performance of the tactic generator (Zheng et al., 2023).

3 PropL: A New Dataset for Intuitionistic Propositional Logic Theorems in Lean

Our aim is to experimentally validate that trial-and-error information can enhance models' ability to do backtracking and tactic generation for theorem-proving tasks. Given that existing open-source theorem proving datasets lack information on trial-and-error processes, we have developed PropL, which is based on theorems of intuitionistic propositional logic. This dataset uniquely includes proofs that encapsulate the complete search process, incorporating the trial-and-error data generated by the FPS algorithm. Our dataset has two other benefits. It is formalized in Lean, so that the validity of the theorems and proofs are guaranteed. The tactics generated by the model trained on PropL can also be directly sent to Lean to be checked. PropL is also representative of all the intuitionistic propositional logic theorems, since by uniformly sampling integers, we can use a bijection between natural numbers and propositions to uniformly sample propositions. This bijection is explained in the Theorem Generation section.

3.1 Data Generation of PropL

PropL comprises theorems uniformly sampled from the entire set of propositional logic theorems. It includes various proof types for each theorem. We only report proof types that are used in this paper. For additional information about the dataset, please refer to the GitHub repository and Huggingface.

The construction of PropL involves two primary stages: the generation of propositional logic theorems and the generation of proofs for these theorems from an existing algorithm.

Theorem Generation. Consider the set of propositions A with at most p atomic propositions. A can be inductively generated by the following grammar:

$$A, B ::= P_i \mid T \mid F \mid A \wedge B \mid A \vee B \mid A \to B,$$

where P_i is the i-th atomic proposition with $1 \leq i \leq p$, T stands for True, and F stands for False. We note that an atomic proposition is just a statement like "Today is sunny". The concrete meaning of the statement is irrelevant with respect to the theorem proving task in this paper. A connective $\wedge$, $\vee$, $\to$ is called an internal node of the proposition. We assign the following lexicographic order to propositions:

1. Number of internal nodes (increasing order)
2. Number of internal nodes of the left child (decreasing order)
3. Top level connective, $T < F < P_1 < \cdots < P_p < \wedge < \vee < \to$
4. The (recursive order (2 - 5)) of the left child
5. The (recursive order (2 - 5)) of the right child

For a fixed upper bound p for the number of atomic propositions, we establish a bijection between the natural numbers and the set of propositional logic formulas. The counting can be made efficient using the Catalan Numbers (Atkinson and Sack, 1992). Figure 3 gives an example of mapping between propositions and natural numbers. Details about the encoding and decoding algorithms are provided in Appendix B.

Proof Generation. Given a randomly sampled theorem, the proof of the theorem is constructed using the focusing method with polarization (McLaughlin and Pfenning, 2009; Liang and Miller, 2009; Pfenning, 2017). Proof search is divided into two stages: inversion and chaining. The inversion phase mechanically breaks down negative connectives (e.g. implications) in the goal and positive connectives (e.g. disjunctions) in the premises. After inversion, chaining will pick an implication in the premise or show one of the disjuncts in the conclusion, with backtracking. The proof search procedure terminates when the same atomic proposition appears in both the premise and the conclusion. An example of

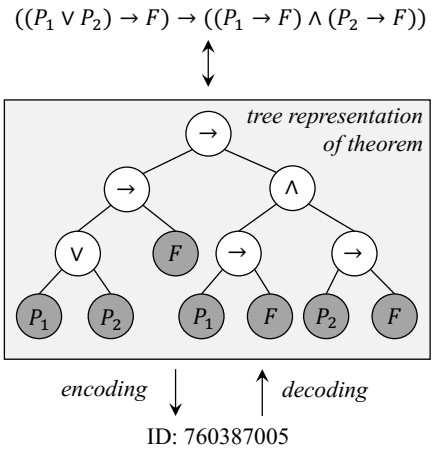


Figure 3: An illustration of the bijection between a proposition and a natural number, where gray nodes are leaf nodes. ID is computed using Algorithm 1 with $n = 6$ and $p = 2$ in this case.

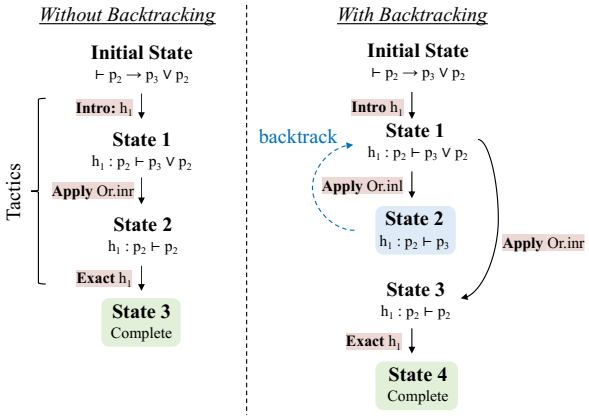


Figure 4: Two proofs for one propositional logic theorem with tactics and states in Lean.

proofs with trial-and-error information (backtracking) and with trial-and-error information removed are shown in Figure 4.

Once proofs are generated, we use them to fine-tune models and start the proof search on the test set.

The polarization of the connectives affects the behavior of the inversion and the search procedure. We choose to uniformly polarize conjunctions that occur negatively (e.g. on the right-hand side of an arrow) as negative and polarize conjunctions that occur positively (e.g. on the left-hand side of an arrow) as positive. Atomic propositions are assigned polarities based on the connective that they first appear under.

To improve the runtime of the search procedure, we make an additional assumption that once an implication is picked, the implication cannot be used to show its premise. In theory, this introduces incompleteness into the search procedure, but it only affects 1 theorem out of around 1000 provable theorems randomly sampled.

3.2 Construction of Training and Test Sets

In this section, we explain how we construct the datasets for training and evaluation. We want to avoid training and testing on similar data. In order to test the model performance on harder out-of-distribution (OOD) tasks, we need to ensure that the lengths of the proofs in the training data are shorter than the lengths of the proofs in the test data.

Given PropL, we fix the number of internal nodes in the theorem statement to be 16 (explained in the dataset generation section). We then uniformly randomly sample 200,000 theorems from PropL, which can be achieved by using the integer-theorem bijection as explained before. Our method ensures that the theorems we study in this paper are representative of the propositional logic theorems in general.

We first apply our deterministic algorithms to generate the proofs of the 200,000 theorems, and then remove the trial-and-error information of those proofs. We get a word-length distribution of those proofs without trial-and-error information. Next, to ensure the diversity of the trial-and-error information, we randomly select propositions to focus on during the chaining phase of the proof search, and then generate 10 different proofs with backtracking. By using the average length of the 10 different proofs, we have another word length distribution of proofs with trial-and-error information.

We then split the 200,000 theorems into training and test sets based on both of the word length distributions mentioned above. The word lengths of the proofs of the training data theorems fall within the lower 0.66 quantile of the two distributions of the word length of all the proofs of the 200,000 theorems (109,887 in total). The word lengths of the in-distribution test set also fall in that category (1000 in total). The word lengths of the proofs among the out-of-distribution test theorems are above 0.8 quantile (1000 total) of the two distributions of the word lengths of all the proofs of the 200,000 theorems. We note that the baseline model and TRIALMASTER are trained on proofs of the same set of theorems.

Table 1: The scale of training and test split in our PropL dataset.

Subset	Number of theorems
Training set	109,887
In-dist. test set	1,000
Out-of-dist. test set	1,000

4 Methodology

4.1 LLM Fine-Tuning

We utilize the training set formed in PropL for training both TRIALMASTER and the tactic generator in the DFS system. The numbers of theorems used in the training and test datasets are presented in Table 1.

LLM fine-tuning with trial-and-error. In our approach, we randomly select two out of the shortest five among the ten proofs with trial-and-error information for each theorem in the training set and utilize them to train TRIALMASTER. Refer to Figure 2b for a description of this training process.

LLM fine-tuning in a DFS system. For the tactic generator of the DFS system, we employ the deterministic FPS algorithm to generate the proofs of the theorems in the training set. The trial-and-error information is removed from the proofs. The LLM is then fine-tuned on the proofs without trial-and-error information as the conventional methods do. Figure 2a illustrates the training process of the DFS system.

4.2 Inference

Inference method of model trained with trial-and-error. TRIALMASTER conducts inference on itself without any help from a backtracking system like DFS or BFS. It outputs two kinds of tactics: tactics in Lean and backtrack instructions. An example of a backtrack instruction would be like *"no solution, return to state 2 [that leads to state 4]"*, where state 4 is the current state. When TRIALMASTER is doing a proof search on the test set, it is prompted with all history paths, including previous tactics, states, the backtracking it made before, and the failed search path. It then outputs the entire proof path after. Nonetheless, we only utilize the first tactic in the output and employ Lean as a calculator to determine the next state, thereby ensuring the correctness of the state following the tactic. If the tactic outputted by TRIALMASTER is a backtrack instruction, it is then prompted with all the proof search history including the backtrack instruction and the state that the backtrack instruction says to return to. If that tactic is not a backtrack instruction, the tactic and the current state will be fed into Lean for producing the state after. TRIALMASTER is then prompted with the entire proof tree including the state that Lean calculated, and it should output a tactic again. This process is repeated until Lean identifies that the proof is complete or any Lean error occurs. We also note that TRIALMASTER only outputs one tactic at each state using greedy search.

Inference method of the DFS system. There are two hyperparameters in the DFS system: temperature t and the number of sampled tactics N_{sampled}. The temperature t decides the diversity of the model outputs. As t increases, the outputs of the model become more varied. The second parameter determines how many tactics the tactic generator of the DFS system produces for a new proof state.

During inference, the LLM in the DFS system produces N_{sampled} of candidate tactics at each new state. For each proof state, the DFS only makes one inference. If any two of the generated tactics for the same state are the same, we remove one of them to ensure efficiency. We also remove the tactic suggestions that fail to follow the grammar of Lean. The system follows the depth-first order to keep trying untried tactics. If the system exhausts all the tactics for a given state but has not found a valid one, the system returns to the parent state and then keeps trying untried tactics for the parent state. The overview is presented in the Figure 2a.

To fully exploit the ability of the DFS system, we varied the parameters of it, such as temperature and the number of sampled tactics. We count how many times Lean has been called to check tactics for both the DFS system and TRIALMASTER during the inference stage.

Why do we choose DFS over BFS? While the breadth-first-search (BFS) system is also popular for building neural provers in Automated Theorem Proving, we have opted for DFS as our baseline over BFS in the context of propositional logic theorem proving. This is due to the finite number (around 20) of tactics available at any step for the search process of intuitionistic propositional logic theorems, making DFS more efficient than BFS without compromising the success rate.

5 Evaluation

5.1 Experimental Setup

Base LLM. We used `Llama-2-7b-hf` (Touvron et al., 2023) as the backbone LLM for tactic generation. Models are trained on two A100 GPUs for a single epoch with batch size set to 4. Huggingface (Wolf et al., 2019) is used for fine-tuning the models, and VLLM (Kwon et al., 2023) is used for inference of the models for efficiency. The learning rate of all training processes is set to be 2×10^{-5}.

Hyperparameters. In the experiment, we vary temperature t from $\{0.3, 0.7, 1.2, 1.5, 2.0\}$ and number of sampled tactics N_{sampled} from $\{2, 5, 10, 15, 20\}$. We notice that in all experiments for temperature t less than 1.5, there were only very few (less than 15) theorems that were still under search when the total steps for that search attempt reached 65. For t higher than 1.5, N_{Lean} starts to increase dramatically. At temperature 2 with N_{sampled}=20, N_{Lean} climbs up to 32,171 when the number of total steps reaches 65. Therefore, in our experiment, we set 65 as the search step limit to control time complexity.

5.2 Evaluation Metrics

Proof search success rate. We use proof search success rate as our primary metric, which represents the ratio of successful searches by the model. A search attempt for a theorem is marked as successful if Lean outputs state *"no goal, the proof is complete"*, implying the theorem is effectively proved after applying a tactic produced by the model. For TRIALMASTER, a search attempt ends and fails immediately after one of the following conditions: 1) the word length of the proof tree with trial-and-error exceeds 1500 (for the sake of context length of the model), or 2) the tactic produced by the model at any step induces a Lean error. For the conventional DFS system, a search attempt fails when one of the following conditions happens: 1) the word length of the proof tree without trial-and-error exceeds 1500, or 2) all tactics generated for the initial states have been explored and failed, or 3) the total search steps exceed 65 (see Section 5.1 for the choice of this value). We note that the 1500-word limit is stricter for our method since it produces entire proof paths including trial-and-error and thus would be easier to hit the limit.

Search cost. We define a metric to assess the search cost—the total number of Lean calls for tactic checking during proof search for the entire test set, denoted as N_{Lean}. Given the same proof search success rate, a lower N_{Lean} indicates a more efficient system for proof search. Note that backtracking instructions from our method do not require applying Lean to check the state and, consequently do not add search cost.

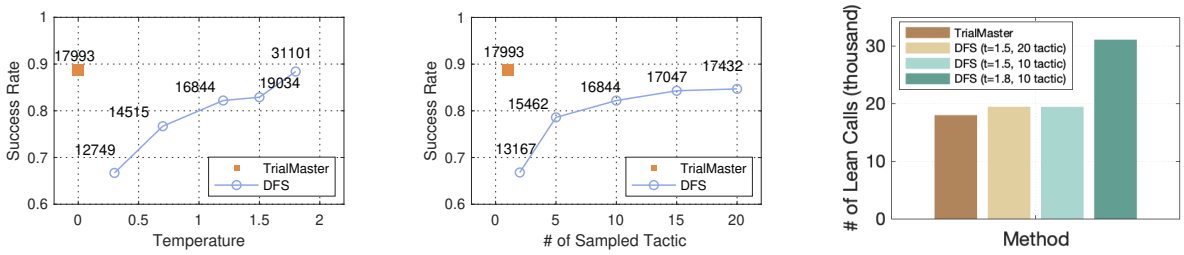
Table 2: Performance on **in-distribution** task. Both methods perform well for propositional logic.

Model			Success Rate
TRIALMASTER			**100%**
DFS	$t = 1.2$	$N_{\text{sampled}} = 2$	99.5%
		$N_{\text{sampled}} = 5$	99.9%
		$N_{\text{sampled}} = 10$	99.6%
	$t = 2.0$	$N_{\text{sampled}} = 2$	75.9%
		$N_{\text{sampled}} = 5$	97.3%
		$N_{\text{sampled}} = 10$	99.0%

5.3 Results and Analysis

TRIALMASTER outperforms conventional DFS system. We begin by evaluating the methods of the in-distribution test set. Table 2 illustrates that both our method and the DFS system perform exceptionally well, achieving a success rate of nearly 100% in most configurations. This suggests that Llama-7b effectively masters in-distribution intuitionistic propositional logic theorems. Then, we compare the performance of the methods on the out-of-distribution task. The results are presented in Figure 5. Our method with trial-and-error significantly outperforms the DFS system across various hyperparameter configurations. Additionally, we observe that feeding more proofs without trial-and-error for LLM fine-tuning does not further improve the performance.

Impact of hyperparameters in the DFS system. As shown in Figure 5, on the OOD task, although the success rate of the DFS system gets higher when we increase the temperature t or the number of sampled tactics N_{sampled}, the search cost (reflected by N_{Lean}) also goes up. Specifically, when we increase N_{sampled}, the DFS system explores a larger pool of candidate tactics during the search, leading to a higher number of Lean calls. In contrast, our method does a greedy search to generate only one tactic for each new state. Likewise, as t increases, the tactic generator of the DFS system tends to produce more diverse tactics at each proof state, improving the system's performance but leading to higher search costs.



(a) Temperature t (b) Sampled Tactics N_{sampled} (c) Search Cost

Figure 5: Experiment results on **OOD** task. (a) We fix $N_{\text{sampled}} = 10$ to see the impact of temperature on the DFS system. (b) We fix $t = 1.2$ to see the impact of the number of sampled tactics. The number of Lean calls is noted beside the marker. (c) Comparison of N_{Lean} among our method and top 3 DFS systems with the highest success rate. In summary, training with trial-and-error achieves a higher success rate with a relatively lower search cost compared to the DFS systems.

Table 3: Ablation study: Comparison of TRIALMASTER and model trained without trial-and-error information on OOD task

Model	Success rate
TRIALMASTER	**88.7%**
Model - proof w/o t.a.e.	59.3 %

TRIALMASTER achieves high success rates at lower search cost. For a direct comparison of search costs, we plot the N_{Lean} values of our method alongside those of the top three DFS systems with the highest success rates among all the hyperparameters we experimented with, i.e., t=1.5, N_{sampled}=20 (87.2%), t=1.5, N_{sampled} = 10 (86.1%), and t=1.8, N_{sampled}=10 (88.4%). This comparison is illustrated in Figure 5c. Notably, we observe that the DFS systems that closely approach our model's performance exhibit significantly higher search costs. With t=1.8, N_{sampled}=10, N_{Lean} of the DFS system, which has 0.3% lower success rate than our method, has reached 31,101, which is 72% higher than that of our method with trial-and-error. The high search cost makes the DFS system with high temperatures unfavorable. These results demonstrate that training with trial-and-error produces higher-quality tactics, achieving a higher success rate with relatively lower search cost.

Model learns backtracking capability from trial-and-error data. In the experiments, we find out that our TRIALMASTER successfully acquires the backtracking capability from proofs with trial-and-error information. This is evidenced by the fact that during TRIALMASTER's proof search for theorems in the test set, all backtracking instructions produced by the LLM adhere to the correct format and point to existing state numbers.

Including failed search paths helps TRIALMASTER to learn. The following experiment shows that adding failed search paths to the training data for TRIALMASTER results in an overall gain. In this experiment, the model is only trained to learn the correct search paths and the backtracking instructions. The model is not trained to learn the failed search paths (we don't compute the loss for the failed search paths during the training stage in this case). The proof success rate in this case is 75.6%, which is lower than TRIALMASTER's proof success rate of 88.7%. The N_{Lean} for the model is 13,600, which is lower than that of TRIALMASTER. This is expected since the model does not learn to predict failed search paths. Our explanation for why TRIALMASTER has a higher proof search success rate than the model trained in the previously mentioned experiment is that the failed search paths also contribute to improving the proof search success rate of the model. TRIALMASTER strategically tried some potentially failed search paths to gain a more comprehensive view of the problem, which then led to the correct search paths.

5.4 Ablation Study

To evaluate the effectiveness of training with trial-and-error, we craft an ablated version of our method where the LLM is fined-tuned with data of the correct path only and do inference in the same way as our method (i.e., producing one tactic at a time and applying Lean for state checking). We denote the ablated version as *Model - proof w/o t.a.e.*. For both methods, we mark the search attempt as failed if the tactic induces a Lean error,

Table 4: Comparison of models trained on different lengths of proofs with trial-and-error on OOD task.

Model	Success rate
Model - short proof w/ t.a.e.	**88.7%**
Model - long proof w/ t.a.e.	72.4 %

or the search exceeds the 1500-word limit. The result is shown in the Table 3. The difference between the success rates of the two models is 29.4%, which is significant. This clearly shows that failed search states and trial-and-error information tremendously enhance the model's capability to solve theorem-proving tasks.

5.5 Exploratory Study: Effect of Training Proof Length on Model Performance

Since the FPS algorithm of PropL dataset can generate multiple proofs with variable length, we conduct an exploratory study to assess the impact of proof length on model performance. We fine-tune two models using proofs with different lengths of trial-and-error information. For the first model, which is our TRIALMASTER, the training data is derived by randomly selecting two out of the shortest four proofs from the ten available proofs for each theorem in PropL. We denote it as *Model - short proof w/ t.a.e.* In contrast, the training data of the second model is formed by randomly selecting two proofs from the ten available for each theorem, irrespective of their lengths. We denote it as *Model - long proof w/ t.a.e.* For both models, we use greedy search to let them generate one tactic for each state. We evaluate the models on our 1000 OOD test set. The results are shown in the Table 4. A higher success rate is observed in the model trained with shorter proofs. This can be attributed to the fact that as the proof with trial-and-error information becomes longer, there is too much trial-and-error information that may detrimentally affect the model's performance, as too many failed search paths may lower the quality of the training data.

6 Conclusion and Future Work

In this paper, we study Automated Theorem Proving in formalized environments. We create a complete, scalable, and representative dataset of intuitionistic propositional logic theorems in Lean. We demonstrate that leveraging information from failed search states and backtracking not only teaches models how to backtrack effectively, but also helps in developing better tactics than those generated by models trained without access to backtracking insights. We release our datasets on GitHub and Huggingface.[1]

A natural extension of our research involves investigating whether trial-and-error information is beneficial for more general mathematical theorem-proving settings. Exploring this avenue could provide valuable insights into the effectiveness of our approach across broader mathematical domains.

Limitations

One limitation of our study is that some proof attempts are forced to stop due to the prompt exceeding the context length of 1500 tokens. This constraint may potentially influence our results by truncating the available information during the proof search process.

Furthermore, our method was not evaluated on general mathematical theorems. This limitation arises from both the scarcity of proofs containing trial-and-error information in current math libraries and the intrinsic challenges associated with producing proofs, whether with or without backtracking, for general mathematical theorems in a formalized setting.

Automated theorem proving with LLMs is an emerging area in machine learning. There is still a lack of baselines on LLMs to compare with our method. We establish a fundamental baseline, but we still need accumulative work to provide methods for comparison.

Ethical Consideration

Our work learns large language models to automatically prove propositional logic theorems, which generally does not raise ethical concerns.

Acknowledgments

This work is supported by the National Science Foundation under grants CCF-1955457 and CCF-2220892. This work is also sponsored in part by NSF CAREER Award 2239440, NSF Proto-OKN Award 2333790, as well as generous gifts from Google, Adobe, and Teradata.

[1]PropL dataset is available at https://huggingface.co/datasets/KomeijiForce/PropL. Model weights are available at https://huggingface.co/KomeijiForce/llama-2-7b-propositional-logic-prover. Generation codes for theorems and proofs are available at https://github.com/ucsd-atp/PropL.

References

Michael D Atkinson and J-R Sack. 1992. Generating binary trees at random. *Information Processing Letters*, 41(1):21–23.

Kshitij Bansal, Christian Szegedy, Markus N Rabe, Sarah M Loos, and Viktor Toman. 2019. Learning to reason in large theories without imitation. *arXiv preprint arXiv:1905.10501*.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 1997. *The Coq proof assistant reference manual: Version 6.1*. Ph.D. thesis, Inria.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. 2023. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*.

Chin-Liang Chang and Richard Char-Tung Lee. 2014. *Symbolic logic and mechanical theorem proving*. Academic press.

Sijia Chen, Baochun Li, and Di Niu. 2024. Boosting of thoughts: Trial-and-error problem solving with large language models. In *The Twelfth International Conference on Learning Representations*.

Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2023. A survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402*.

William F Clocksin and Christopher S Mellish. 2003. *Programming in PROLOG*. Springer Science & Business Media.

Daniel Crevier. 1993. *AI: the tumultuous history of the search for artificial intelligence*. Basic Books, Inc.

Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, et al. 2021. Advancing mathematics by guiding human intuition with AI. *Nature*, 600(7887):70–74.

Martin Davis. 2001. The early history of automated deduction. *Handbook of automated reasoning*, 1:3–15.

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.

Emily First, Markus 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.

Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. 2022. Complexity-based prompting for multi-step reasoning. *arXiv preprint arXiv:2210.00720*.

P. C. Gilmore. 1960. A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development*, 4(1):28–35.

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*.

Edvard K Holden and Konstantin Korovin. 2021. Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In *International Conference on Intelligent Computer Mathematics*, pages 107–123. Springer.

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

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

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

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.

Chuck C. Liang and Dale Miller. 2009. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768.

Pamela McCorduck. 2004. *Machines Who Think (2Nd Ed.)*. A. K. Peters.

Sean McLaughlin and Frank Pfenning. 2009. Efficient intuitionistic theorem proving with the polarized inverse method. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 230–244. Springer.

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

Maciej Mikuła, Szymon Antoniak, Szymon Tworkowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. 2023. Magnushammer: A transformer-based approach to premise selection. *arXiv preprint arXiv:2303.04488*.

M Saqib Nawaz, Moin Malik, Yi Li, Meng Sun, and M Lali. 2019. A survey on theorem provers in formal methods. *arXiv preprint arXiv:1912.03028*.

Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer.

Frank Pfenning. 2017. Lecture notes on focusing.

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

Markus N Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. 2020. Mathematical reasoning via self-supervised skip-tree training. *arXiv preprint arXiv:2006.04757*.

John Alan Robinson. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41.

Stuart J Russell and Peter Norvig. 2010. *Artificial intelligence a modern approach*. London.

Taro Sekiyama, Akifumi Imanishi, and Kohei Suenaga. 2017. Towards proof synthesis guided by neural machine translation for intuitionistic propositional logic. corr abs/1706.06462 (2017). *arXiv preprint arXiv:1706.06462*.

Taro Sekiyama and Kohei Suenaga. 2018. Automated proof synthesis for propositional logic with deep neural networks. *arXiv preprint arXiv:1805.11799*.

Richard P Stanley. 2015. *Catalan numbers*. Cambridge University Press.

Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. 2023. A language-agent approach to formal theorem-proving. *arXiv preprint arXiv:2310.04353*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Adam Zsolt Wagner. 2021. Constructions in combinatorics via neural networks. *arXiv preprint arXiv:2104.14516*.

Haiming Wang, Ye Yuan, Zhengying Liu, Jianhao Shen, Yichun Yin, Jing Xiong, Enze Xie, Han Shi, Yujun Li, Lin Li, et al. 2023. 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.

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

Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. 2017. Premise selection for theorem proving by deep graph embedding. *Advances in neural information processing systems*, 30.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

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.

Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. 2021. Naturalproofs: Mathematical theorem proving in natural language. *arXiv preprint arXiv:2104.01112*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

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.

Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. *arXiv preprint arXiv:2306.15626*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.

Fei Yu, Hongbo Zhang, and Benyou Wang. 2023. Nature language reasoning, a survey. *arXiv preprint arXiv:2303.14725*.

Chuanyang Zheng, Haiming Wang, Enze Xie, Zhengying Liu, Jiankai Sun, Huajian Xin, Jianhao Shen, Zhenguo Li, and Yu Li. 2023. Lyra: Orchestrating dual correction in automated theorem proving. *arXiv preprint arXiv:2309.15806*.

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

$$\frac{}{\Gamma, A \vdash A}(\text{Assumption}) \qquad \frac{}{\Gamma \vdash T}(\text{T-I})$$

$$\frac{\Gamma \vdash F}{\Gamma \vdash A}(\text{F-E}) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}(\wedge\text{-I})$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}(\wedge\text{-E1}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}(\wedge\text{-E2})$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}(\vee\text{-I1}) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}(\vee\text{-I2})$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}(\vee\text{-E})$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}(\to\text{-I})$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}(\to\text{-E})$$

Figure 6: Natural Deduction Rules for Intuitionistic Propositional Logic

A Inference Rules for Intuitionistic Propositional Logic

Figure 6 shows the inference rules for intuitionistic propositional logic. Rules relate sequents of the form $\Gamma \vdash A$, where Γ is an unordered list of assumptions. A derivation is constructed from the axiom (Assumption) using the derivation rules until Γ is empty.

For example, the ($\to$-I) rule says that from a derivation of $\Gamma \vdash B$ under the assumption of Γ, A, we can get a derivation of $\Gamma \vdash A \to B$. The ($\to$-E) rule says that from a derivation of $\Gamma \vdash A \to B$ and a derivation of $\Gamma \vdash A$ we can derive $\Gamma \vdash B$.

Here are an example derivation.

$\mathsf{p1} \to \mathsf{p1} \vee \mathsf{p2}$.

$$\frac{\dfrac{\dfrac{}{\mathsf{p1} \vdash \mathsf{p1}}\,(\text{Assumption})}{\mathsf{p1} \vdash \mathsf{p1} \vee \mathsf{p2}}\,(\vee\text{-I1})}{\vdash \mathsf{p1} \to \mathsf{p1} \vee \mathsf{p2}}\,(\to\text{-I})$$

B Uniformly Distributed Data Explanation

In this section, we discuss the uniform characteristics of our dataset, particularly emphasizing the one-to-one mapping between propositions and natural numbers. This bijection allows us to simply sample from the natural numbers to ensure the dataset exhibits uniformity.

Algorithm 1 Encoding

Require: A tree $\mathcal{T}$ representing a proposition, with n indicating the number of internal nodes and p representing the number of atomic propositions

Ensure: Output a natural number

return $3^n(p+2)^{n+1} \times$ SHAPENUMBER($\mathcal{T}$) + ASSIGNMENTNUMBER($\mathcal{T}$)
function SHAPENUMBER($\mathcal{T}$)
  if $\mathcal{T}$ is a single node **then return** 0
  end if
  $\mathcal{T}_l, \mathcal{T}_r \leftarrow$ left and right sub-trees of $\mathcal{T}$
  $n_l, n_r \leftarrow$ number of internal nodes in $\mathcal{T}_l, \mathcal{T}_r$ ▷ Total $n_r + n_l + 1$ internal nodes
  return $\sum_{i=1}^{n_l} C_{i-1} C_{n_l+n_r+1-i} + C_{n_r} \times$ SHAPENUMBER($\mathcal{T}_l$) + SHAPENUMBER($\mathcal{T}_r$)
end function
function ASSIGNMENTNUMBER($\mathcal{T}$)
  $N \leftarrow$ NODENUMBER(root node of $\mathcal{T}$)
  if $\mathcal{T}$ is a single node **then return** N
  end if
  $\mathcal{T}_l, \mathcal{T}_r \leftarrow$ left and right sub-trees of $\mathcal{T}$
  $n_r \leftarrow$ number of internal nodes in $\mathcal{T}_r$
  $A_r \leftarrow 3^{n_r}(p+2)^{n_r+1}$ ▷ Compute all possible assignments in the right sub-tree
  return $3A_r \times$ ASSIGNMENTNUMBER($\mathcal{T}_l$) + $A_r \times N +$ ASSIGNMENTNUMBER($\mathcal{T}_r$)
end function
function NODENUMBER($\mathcal{N}$)
  switch $\mathcal{N}$
    case $\wedge$ or T: **return** 0
    case $\vee$ or F: **return** 1
    case $\to$: **return** 2
    case P_i: **return** $i+1$ ▷ $1 \le i \le p$
end function

B.1 Catalan Number

The Catalan number C_n is applicable for counting full binary trees that consist of $n+1$ leaf nodes, corresponding to exactly n internal nodes (Stanley, 2015). Additionally, it can be calculated through recursion as shown:

$$C_n = \sum_{i=1}^{n} C_{i-1} C_{n-i}.$$

The first Catalan numbers for $n = 0, 1, 2, 3, 4$ are $1, 1, 2, 5, 14$.

A concise interpretation of this recursion is as follows: it involves counting the number of internal nodes in the left sub-tree, amounting to $i - 1$, and then in the right sub-tree, amounting to $n - i$, for each of the n scenarios in computing C_n.

Algorithm 2 Decoding

Require: A natural number ID, with n indicating the number of internal nodes and p representing the number of atomic propositions

Ensure: Output a proposition tree

quotient, remainder $\leftarrow$ DIVMOD(ID, $3^n(p + 2)^{n+1}$)
$S \leftarrow$ TREESHAPE(quotient, n)
$A \leftarrow$ TREEASSIGNMENT(S, remainder)
return a tree with shape S and assignment A
function TREESHAPE(N, n)
    if n is 0 **then return** a single node
    end if
    $n_l \leftarrow \max(\{k | \sum_{i=1}^{k} C_{i-1}C_{n-i} \leq N\} \cup \{0\})$
    $n_r \leftarrow n - n_l - 1$
    remaining $\leftarrow N - \sum_{i=1}^{n_l} C_{i-1}C_{n-i}$
    $N_l, N_r \leftarrow$ DIVMOD(remaining, C_{n_r})
    return a tree with left and right sub-trees shaped by TREESHAPE(N_l, n_l) and TREESHAPE(N_r, n_r), respectively
end function
function TREEASSIGNMENT(S, N)
    Perform an inorder traversal of the tree with shape S. For each leaf node, interpret it as a digit in base $p+2$, and for each internal node, interpret it as a digit in base 3. Compute the assignment that gives N by utilizing the NODENUMBER function introduced in Algorithm 1
end function

B.2 Bijection between Propositions and Natural Numbers

As depicted in Figure 3, every proposition corresponds to a unique tree representation. Consequently, it only requires the identification of a bijection between full binary trees and natural numbers.

For every full binary tree possessing n internal nodes and p atomic propositions, there exist

$$3^{\#\text{internal node}}(p+2)^{\#\text{leaf node}} = 3^n(p+2)^{n+1}$$

distinct cases. Observe that the choices available for internal nodes include conjunction ($\wedge$), disjunction ($\vee$), and implication ($\rightarrow$); whereas for leaf nodes, the choices encompass true (T), false (F), and a set of propositions $P_1, \ldots, P_p$.

This counting facilitates an efficient ranking of all full binary trees with n internal nodes. This ranking process inherently establishes a bijection with the set of natural numbers, allowing for a clear correspondence between each full binary tree and a unique natural number. Consequently, this sets the stage for a detailed examination of two critical processes: encoding (see Algorithm 1), which involves mapping a proposition tree to a natural number, and decoding (see Algorithm 2), which entails mapping a natural number back to a corresponding proposition tree.

Having established the bijection between full binary trees and natural numbers, it becomes apparent that uniformly sampling from the set of natural numbers will, in turn, result in a uniform sampling of full binary trees.

The inclusion of the parameter n in Algorithm 1 is not critical for the functionality of the algorithm as n can be counted from $\mathcal{T}$; however, it is retained to denote that the most basic encoding and decoding approaches are designed for a fixed number of internal nodes n. For example, ID 0 denotes different propositions depending on the specified value of n. As a result, the decoding algorithm, as outlined in Algorithm 2, is specifically intended for uniformly generating proposition trees with a fixed n. To achieve a uniformly distributed proposition tree from a set of trees with various n, one might simply merge the rankings of fully binary trees with different n, a task which can be performed with relative ease. This approach of merging can be similarly applied to trees with varying p as well.

Given the uncertainty surrounding the proof lengths when generating propositions, our approach involves uniform sampling for proposition selection. This selection is later refined by excluding propositions according to their proof lengths as computed by the proof generation algorithm.

C Examples

In Figure 7, we show an example Lean proof for a theorem in Figure 3. Lines preceded by '–' are comments solely for explanatory purposes.

```
variable (p1 p2 p3 p4 p5 : Prop)
theorem : (((p1 ∨ p2) → False) → ((p1 → False) ∧ (p2 → False))) := by
  -- Implications on the right can always be decomposed.
  -- Introduce an assumption h1 that says ((p1 ∨ p2) → False)
  intro h1
  -- Now we want to show ((p1 → False) ∧ (p2 → False))
  -- Conjunctions on the right can always be decomposed.
  -- We then need to show (p1 → False) and (p2 → False) separately.
  apply And.intro
  -- We are showing (p1 → False).
  -- Implications on the right can always be decomposed.
  -- We introduce assumption h2 for p1. And we try to show False.
  intro h2
  -- We want to use the implication h1. So we show its premise.
  have h3 : (p1 ∨ p2) := by
    -- Show the left disjunct. (The right adjunct leads to an TAE)
    apply Or.inl
    -- One of the premise coincides with the conclusion.
    exact h2
  -- We have shown the premise of h1 (p1 ∨ p2),
  -- we can now drive its conclusion (False), denoted by h4.
  let h4 := h1 h3
  -- False on the left can always be used.
  apply False.elim h4
  -- We have shown (p1 → False) and now we show (p2 → False).
  -- Implications on the right can always be decomposed.
  -- We introduce assumption h2 for p2. And we try to show False.
  intro h5
  -- We want to use the implication h1. So we show its premise.
  have h6 : (p1 ∨ p2) := by
    -- Show the right disjunct. (The left adjunct leads to an TAE)
    apply Or.inr
    -- One of the premise coincides with the conclusion.
    exact h5
  -- We have shown the premise of h1 (p1 ∨ p2),
  -- we can now drive its conclusion  (False), denoted by h7.
  let h7 := h1 h6
  -- False on the left can always be used.
  apply False.elim h7
```

Figure 7: An example of an intuitionistic propositional logic theorem with its proof in Lean