LLM-BASED AUTOMATED THEOREM PROVING HINGES ON SCALABLE SYNTHETIC DATA GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advancements in large language models (LLMs) have sparked considerable interest in automated theorem proving, and a prominent line of research integrates stepwise LLM-based provers into tree search. In this paper, we introduce a lightweight, one-pass data synthesis method that explores a wide range of intermediate proof states to generate diverse proof steps, which facilitates effective one-shot fine-tuning of the LLM policy model. We also propose an adaptive beam size strategy, which effectively takes advantage of our data synthesis method and achieves a trade-off between exploration and exploitation during tree search. Evaluations on the MiniF2F and ProofNet benchmarks demonstrate that our method achieves competitive results compared to strong baselines, attaining an average pass rate of 60.74% on MiniF2F and 21.18% on ProofNet under the stringent *Pass@1* metric, as well as a 68.85% pass rate on MiniF2F under the *Pass@16* metric. These results underscore the impact of large-scale synthetic data in advancing automated theorem proving.

1 INTRODUCTION

Reasoning has emerged as a frontier in large language model (LLM) research (Yu et al., 2024; Giadikiaroglou et al., 2024), with mathematical reasoning receiving special attention as a benchmark for cognitive proficiency and its potential in software engineering (Ahn et al., 2024; Li et al., 2024; Liu et al., 2024). While extensive research focuses on solving mathematical problems in natural language (Muennighoff et al., 2025; Xiang et al., 2025), proving theorems in formal mathematical languages represents a more fundamental challenge. By synergizing interactive theorem provers (ITPs) like Lean (Moura & Ullrich, 2021) with LLMs, we can develop a neuro-symbolic approach towards automated theorem proving (ATP).

In the literature, LLM-based ATP generally falls into two categories: *tree search* methods and *whole-proof generation* methods (Ren et al., 2025). While the latter is simpler and end-to-end, it often struggles with long proofs where intermediate states are hidden. In practice, these methods usually rely heavily on additional techniques, such as natural language comments to assist the proof (Lin et al., 2025; Xin et al., 2024b). In contrast, tree search methods iteratively prescribe tactics for specific proof states, which demand lower capability from the policy model and offer greater flexibility for incorporating customized strategies (Wang et al., 2024; An et al., 2024). Crucially, starting from a small number of seed problems, tree search enables extensive exploration of proof states, facilitating the synthesis of large volumes of new data for iterative training: a task considerably challenging for whole-proof generation (Xin et al., 2024a). We adopt Lean 4 as our base ITP, though our methodology is easily applicable to other proof assistants.

However, LLMs remain limited in formal mathematical reasoning due to the scarcity of high-quality training data (Yang et al., 2024b). Proofs in ITPs are expressed in domain-specific languages like Lean, which are extremely limited in volume compared to general programming languages. For instance, standard Lean 4 libraries (mathlib Community, 2020) provide less than 1GB of data, which is insufficient for effective fine-tuning. Consequently, synthetic data generation is crucial. Recent efforts have explored autoformalization, which translates natural language mathematics into Lean syntax (Wu et al., 2022; Ying et al., 2024), and generating new problems by sampling from learned

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

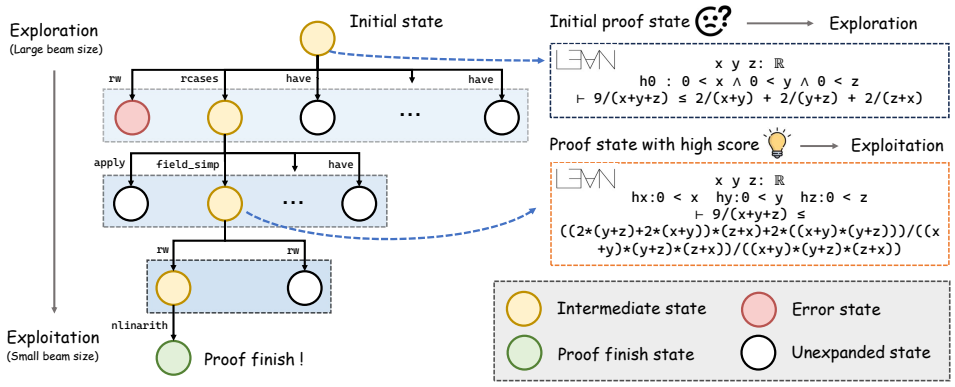


Figure 1: An illustration of the proposed tree search strategy. The example is derived from the MiniF2F benchmark, specifically the problem *algebra_9onxpypzleqsum2onxpy*. For simplicity, only the tactics are retained in the depicted proof states, while the associated premises are omitted. The basic functionality of the tactics used in our examples (e.g. `rw`, `rcases`, `have`, `apply`, etc.) is further explained in Appendix A.4.

distributions (Dong & Ma, 2025; Ying et al., 2024; Xin et al., 2024a). Another prevalent approach is Expert Iteration (EI) (Polu & Sutskever, 2020), which enhances model capabilities by collecting successful proofs from solvable cases for fine-tuning. Despite its widespread use (Polu & Sutskever, 2020; Polu et al., 2022; Wu et al., 2024; Xin et al., 2025), EI suffers from significant inefficiency when applied with tree search; the number of newly solvable problems in each iteration is often limited, while performing a full search over the entire dataset incurs substantial computational cost. This makes EI impractical in resource-limited settings, highlighting *the need for a more direct and resource-efficient data generation strategy*.

In this paper, we propose a data synthesis method called *proof state exploration* for fine-tuning LLMs in ATP, along with a complementary *adaptive beam size strategy* for tree search during the proving process. Our data synthesis method begins with a set of formalized mathematical problems as seeds and leverages the existing policy model to explore related intermediate proof states via tree search. These intermediate states may correspond to goals that are logically equivalent to the original ones after a sequence of transformations, or to subgoals resulting from original problem decomposition. During the process, we explicitly decouple the generation of tactics from that of premises. Specifically, we adopt a constrained decoding algorithm (Hokamp & Liu, 2017) that restricts the model to sample from and de-duplicate within a curated set of commonly used tactics, followed by premise completion handled by the policy model. In addition, we introduce a heuristic pruning strategy to balance the trade-off between diversity in the generated outputs and computational efficiency during the exploration. This enables large-scale synthesis of intermediate proof states and transformation steps within tree search, thereby reducing the distribution gap between the training data and real-world proving scenarios. Moreover, our method does not rely on EI. Instead, our policy model is trained on data generated from a single, exhaustive exploration over existing formalized problems. This one-pass generation approach significantly reduces computational overhead while achieving competitive performance.

Although the LLM fine-tuned on this synthetic dataset can serve as a policy model capable of generating diverse tactics, it may not always prioritize proof completion during actual theorem proving. To mitigate this limitation, we propose a simple yet effective *adaptive beam size strategy* tailored to our data synthesis framework and integrated into the tree search process (cf. Figure 1). The strategy starts with a relatively large beam size to encourage broad exploration of the space of proof states. As the search proceeds, the beam size is gradually reduced, enabling the policy model to concentrate on high-scoring proof states. This dynamic adjustment improves the focus of the search and enhances the overall success rate of proof completion. Besides, we develop Dojo-BeamSearch-Visualization (DoBeVi) (cf. Appendix A.3) as the interaction tool between the model and the Lean 4 prover with a visualization module for analysis. We evaluate our method on two widely adopted benchmarks, MiniF2F and ProofNet, achieving average pass rates of 60.74% and 21.18%, respectively, under the constraints of *Pass@1* and a limited computational budget. Furthermore, our method achieves a

Pass@16 rate of 68.85% on the MiniF2F benchmark. These results demonstrate competitive performance against strong tree-search baselines under the same evaluation protocol.

The main contributions of this paper are as follows.

- We propose a novel data synthesis method, *proof state exploration*, enabling large-scale generation of exploration data for fine-tuning LLMs used in ATP.
- We propose a simple but efficient *adaptive beam size strategy* that synergistically operates with the data synthesis method, effectively guiding the search process towards proof completion.
- We carry out extensive experiments demonstrating that our method can effectively balance exploration and exploitation during tree search, achieving competitive performance against strong tree-search baselines.

2 BACKGROUND

ITP and ATP. In a nutshell, interactive theorem proving is a process to develop and verify formal proofs with the assistance of a computer, usually within a proof assistant (e.g., Lean, Isabelle, Rocq), and involves human guidance to structure and complete the proof. The general process is to formalize the theorem to be proven (i.e., the goal) in the language of the proof assistant first, and then repeatedly apply tactics to break the goal into subgoals. The tactic may involve case analysis, induction, rewriting, application of lemmas or axioms, etc. As such, the human user iteratively refines the proof to guide the prover, choosing which rules to apply and in what order. The prover checks that each inference step is valid according to the rules of the formal system. A major drawback is that this process requires heavy human expertise and thus largely lacks automation. LLM-based Automated Theorem Proving aims to find a proof automatically by simulating this process, effectively replacing the human user by an LLM.

Tree Search LLM-based ATP. As mentioned in Section 1, we adopt a tree search approach towards LLM-based ATP. Generally speaking, tree search is a class of decision-making algorithms which consists of searching combinatorial spaces represented by trees, where nodes denote states and edges denote transitions (actions) from one state to another. In our setting, they enable an efficient navigation of the large and complex proof space.

At a high level, tree search LLM-based ATP can be cast as a reinforcement learning (RL) problem, where the state space comprises the proof states (aka proof terms). In general, proof states represent the current status of the proof at any given point in the interactive session between LLM and the proof assistant. Typically, they comprise the context (i.e., a list of assumptions, hypotheses and local variables), the current (sub)goals (i.e., the proposition needing to prove to complete the proof) and the number of subgoals to be proven. LLM acts as an actor, which, given the current proof state s_i, generates a tactic. In other words, LLM serves as the *policy model*. In theorem proving, tactics refer to commands, or instructions, that guide the process of constructing a proof. In RL terms, tactics can be regarded as an action a_i, based on which the proof assistant produces a new proof state s_{i+1}. This process continues until either the theorem is successfully proved or the maximum number of iterations is reached.

There are a plethora of search strategies, for instance, Monte Carlo tree search (MCTS) or best-first search (BFS). Although MCTS is well suited for balancing exploration and exploitation, its classical simulation step is impractical in the context of ATP. Once a simulation reaches a successful proof state, the proof is already complete, and no further search is required. In BFS, each expansion step involves using the policy model to perform beam search decoding, generating a fixed number of top candidates (called the beam size). Typically, the process begins at the root node and iteratively expands the current node to produce potential child nodes. These candidates are then evaluated using a heuristic scoring function, and the node with the highest score is selected for the next expansion. The expansion-score-selection steps are repeated until a goal is reached or a termination condition is met. In our work, LLM, as the policy model, generates a set of tactics $\{t_i^j\}_{j=1}^B$ for a proof state s_i, which are executed by the proof assistant one by one to obtain the corresponding set of next proof states $\{s_{i+1}^j\}_{j=1}^B$.

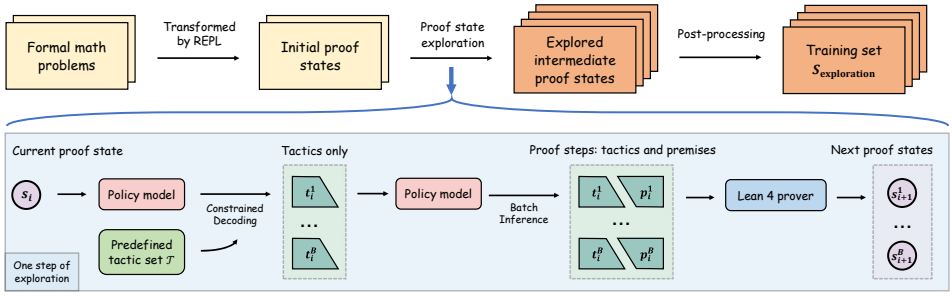


Figure 2: An illustration of the data synthesis pipeline.

3 METHODS

3.1 DATA SYNTHESIS BY PROOF STATE EXPLORATION

In ITP, tactics play a central role. Normally, a tactic (e.g., `rw [add_assoc add_comm]`) consists of the tactic operator (e.g., `rw`) and the premises (e.g., `add_assoc` and `add_comm`). In this paper, to facilitate a finer-grained decomposition, what is commonly referred to as tactics are called *proof steps*, while *tactic* refers to the operator only. We propose to carry out *proof state exploration* to synthesize data. It is similar to tree search, but the crucial difference is that we prioritize the diversity of tactics. Our data synthesis method is described as Algorithm 1. (cf. Figure 2 as well).

Motivated explanation. The intuition is that proof assistants (e.g., Lean 4) normally provide a versatile range of tactics, which is considerably larger than, e.g., the set of reserved words in programming languages, and can even be defined by users. It is practically infeasible to adopt all of them in tree search. On the other hand, in practice, their actual use in theorem proving may not be as wild as, e.g., user-defined functions (which are virtually infinite, or at least cannot be enumerated exhaustively). Hence, as a trade-off, a reasonably curated set of commonly used tactics is sufficient to solve the vast majority of problems in practice. Besides that, for tree search, critically we need an "effective" set of (proof state, tactic) pairs. The observation is that in theorem proving, the same tactic may have multiple meanings and can be used across various scenarios; for instance, the `apply` tactic may be used in five different scenarios.[1] Moreover, sometimes a tactic can be used in much broader cases; for instance, the tactic `rfl` can handle all *definitional equalities*. However, as the policy model, the LLM, like an inexperienced problem solver, may fail to realize that a tactic can actually be used when a proof state is encountered. Ideally, such domain-specific knowledge should be injected into LLMs via training, as they largely fall short of a certain level of abstraction capabilities and so would be hard to learn via the standard tree search.

The proposed proof state exploration method. We intend to identify those tactics that occur rarely (with low probability) at a proof state. Just like a human problem solver normally has a "comfort zone" for certain methods when facing a problem, in tree search, the policy model likely picks up the tactic that has a higher probability. Quite often, this is sub-optimal or may represent a missed opportunity. Our proof state exploration may force the policy model to explore those tactics that it does not typically explore (i.e., force it to jump out of its comfort zone), by which a more diverse training set can be constructed.

To this end, we first identify the frequently used tactics. We scan the entire Mathlib to collect all defined tactics.[2] There are hundreds of them, most of which are rarely used. To filter these out, we need a representative dataset from which the frequency of each tactic can be collected. In our work, we use the STP dataset (Dong & Ma, 2025) from which a total of 173 distinct tactics are obtained. To further refine the set, we apply nucleus sampling with a threshold of $\mathcal{P} = 0.999$ to filter out rarely used tactics, ultimately yielding a set $\mathcal{T}$ of 60 commonly used tactics.

The design of the proof state exploration is guided by the above intuition and resembles a tree search. For each expansion step, we apply Constrained Decoding to force the LLM to output B different tactics (without premises) in $\mathcal{T}$. We set a large beam size (e.g., $B = 32$). After obtaining $\{(s_i, t_i^j)\}_{j=1}^{B}$ with $t_i^j \in \mathcal{T}$, we feed these pairs back into the policy model to complete the premises,

[1] `https://www.ma.imperial.ac.uk/~buzzard/xena/formalising-mathematics-2024/Part_C/tactics/apply.html#`

[2] `https://github.com/haruhisa-enomoto/mathlib4-all-tactics`

Algorithm 1 Proof State Exploration for Data Synthesis

Input: Initial proof states set S, initial exploration budget $\mathcal{B}$, tactic set $\mathcal{T}$, branch factor B, pruning ratio α, top size β, decay factor γ
Output: Exploration dataset $S_{\text{exploration}}$
$S_{\text{exploration}} \leftarrow \emptyset$
for each $s_0 \in S$ **do**
  $q \leftarrow \{(s_0, \emptyset)\}$, $b \leftarrow 0$, $\mathcal{B}_s \leftarrow \mathcal{B}$, $\mathcal{P}_s \leftarrow \emptyset$
  while $q \neq \emptyset$ **and** $b < \mathcal{B}_s$ **do**
    Pop (s_i, π_i) from q
    Generate candidate tactics $\{t_i^j\}_{j=1}^B$ using constrained decoding
    Complete premises $\{p_i^j\}_{j=1}^B$ and compute proof-step scores $\{r_i^j\}_{j=1}^B$ via batch inference
    Select αB branches by retaining the top-β according to r_i^j and randomly sampling the remaining $\alpha B - \beta$ branches
    for each selected branch j **do**
      Execute (t_i^j, p_i^j) at s_i to obtain s_{i+1}^j
      $\pi_{i+1}^j \leftarrow \pi_i \circ (t_i^j, p_i^j)$
      Add $(s_{i+1}^j, t_i^j, p_i^j, s_i)$ to $S_{\text{exploration}}$
      if s_{i+1}^j is valid **then**
        Push (s_{i+1}^j, π_{i+1}^j) into q
      end if
      if s_{i+1}^j is proof-finished **and** $\pi_{i+1}^j \notin \mathcal{P}_s$ **then**
        $\mathcal{P}_s \leftarrow \mathcal{P}_s \cup \{\pi_{i+1}^j\}$
        $\mathcal{B}_s \leftarrow \gamma \mathcal{B}_s$
      end if
    end for
    $b \leftarrow b + 1$
  end while
end for
$S_{\text{exploration}} \leftarrow \text{post_processing}(S_{\text{exploration}})$

Note: We retain all valid proof transitions to maintain data density; see Appendix A.11 for detailed rationale and empirical results.

yielding a set $\{(s_i, t_i^j, p_i^j)\}_{j=1}^B$ of proof steps from s_i. By applying each (t_i^j, p_i^j) to s_i using the Lean 4 prover, we generate the corresponding set of next proof states s_{i+1}^j. Note that for exploration, even if *proof finish* state is encountered, the exploration continues until the entire budget is exhausted.

Potential issues for this proof state exploration include (1) as B is often large, obtaining deeper proof states (i.e., those far away from s_0) can take a long time under limited computational resources; (2) in certain proof states, a large beam size can lead to the generation of meaningless proof steps. For example, the model may construct hypotheses that do not contribute to the progress of the proof.

To address these, we apply two heuristic strategies: (1) During each expansion, we prune the set by retaining only αB branches, typically setting $\alpha = 0.25$. When pruning, we keep the top β branches with the highest joint probabilities and then randomly sample the remaining $(\alpha B - \beta)$ branches uniformly from the rest. This balances exploration diversity and efficiency while preserving some of the policy model's inherent tendency toward proof finish. (2) Some seeds are overly simple and do not require extensive exploration; too much exploration may even degrade the quality of the synthetic data. To control this, we use proof finish as a signal: if a distinct path achieving proof finish is found during an expansion, we reduce the expansion budget by a factor of $\gamma = 0.9$. This allows a simple seed to quickly terminate exploration after generating multiple distinct solutions, improving overall data synthesis efficiency.

After the proof state exploration, we obtain a dataset $S_{\text{exploration}}$. To produce the final dataset, we apply three post-processing steps: (1) De-duplication, to remove redundant samples; (2) Decontamination, where we use BLEU as a similarity metric to eliminate examples with high overlap with the evaluation benchmark; and (3) Rejection Sampling, where we discard all transitions that are invalid

from the Lean 4 prover. The generated data is ultimately used to train policy LLM by performing Supervised Fine-Tuning on a base model using the full accumulated training data corpus.

3.2 ADAPTIVE BEAM SIZE STRATEGY

The tree search approaches used for ATP are conceptually similar (Polu & Sutskever, 2020; Polu et al., 2022; Wu et al., 2024; Xin et al., 2025), differing mainly in (1) the policy model, (2) the scoring function, (3) the beam size, and (4) other budget-related parameters. In our tree search procedure, the policy model is trained using data collected by the proof state exploration, as in Section 3.1. For the score function, we take the sum of the LLM's logarithmic joint probability over the beam (during beam search) and the parent node's score as the current node's score, that is:

$$\text{score}(s_{i+1}^j) = \log P_\theta(t_j|s_i) + \text{score}(s_i) \tag{1}$$

where $P(t_j|s_i)$ is the model's predicted probability of applying the tactic t_j at the state s_i and θ denotes the policy model's parameters.

In this work, we focus the pruning strategy on dynamically decaying the *beam size* during the tree search process. This design is motivated by the empirical finding that beam size has a particularly significant impact on overall performance compared to other factors in tree search. The details of the finding are presented in Section 4.3. The optimal beam size is difficult to determine in advance. The general observation is that, if the beam size is too small, the search may fail to discover proof states, missing potential opportunities; if the beam size is too large, the search may risk being trapped within one sub-tree rooted at a high-scoring intermediate node, resulting in futile expansions. Hence, it is necessary to adapt the beam size to better control the tree search.

We adopt a larger beam size at the early stages of tree search; as the search proceeds (i.e., the depth increases), we gradually reduce the beam size. This is largely aligned with the aforementioned observation. In particular, during the later stages of tree search, being trapped is particularly harmful as it may dump the search budget. By reducing the beam size, we effectively prune away less relevant branches and prevent the search from over-committing to one sub-tree too much. One may intuitively understand the design by taking an analogy of an IMO contestant. At the early stage, (s)he may want to widen the horizon by exploring different problem-solving strategies (e.g., reading books, surfing AoPS). However, when sitting in the competition (i.e., at a later stage), (s)he would better focus on relatively small strategies to attempt, prioritizing solving the problems successfully. To this end, we design an adaptive beam size function, namely,

$$\text{beam_size} = B_{\text{min}} + (B_{\text{max}} - B_{\text{min}}) \times \max(1 - \lambda M, 0) \tag{2}$$

where $M = e/E$, with e denoting the current expansion step and E representing the maximum number of expansions allowed by the budget. The parameter λ is a hyperparameter controlling the decay rate of the beam size. The beam size gradually decreases from $B_{\max}$ to $B_{\min}$ as the search proceeds, thereby realizing the adaptive behavior described earlier. The relationship between beam size and search depth is further analyzed empirically in Appendix A.14 (Figure 7).

4 EXPERIMENTS

4.1 EVALUATION SETTINGS

Below are the basic experimental settings. All experiments were conducted on a server with 8 H800 GPUs; the code and model checkpoint are available[3].

Models. We use Qwen2.5-Math-7B (Yang et al., 2024a) as the base model to train the policy model. During full fine-tuning, we use a batch size of 2,048 and a learning rate of 2×10^{-5}, training for one epoch. A cosine annealing schedule is employed for the learning rate, reaching 10% of its initial value by the end of the epoch. Notably, the model training process does *not* incorporate EI.

Datasets. We use the STP dataset (Dong & Ma, 2025) as the seed and adopt BFS-Prover (Xin et al., 2025) as the policy model for *proof state exploration*. We first generate the synthetic dataset

[3]The implementation code is accessible at `https://anonymous.4open.science/r/llm_based_atp-8B67`. The policy model is accessible at `https://huggingface.co/RYbWnlL0v8/llm_atp_model`

Table 1: Main evaluation results on MiniF2F-test and ProofNet-test (all values are in %). We report *Pass@1* under two strategies, where *Max* and *Avg.* denote the maximum single-run and mean success rates, respectively. Subscripts in *Fixed* columns indicate the optimal beam size $B \in \{4, 8, 16, 32\}$ for average *Pass@1*. *Pass@16* represents the cumulative success rate across 16 independent search processes on MiniF2F-test. Due to limited resources, *Pass@16* comparisons are conducted only against the stronger baseline. Full evaluation results are available in Appendix A.5.2.

Model	MiniF2F (*Pass@1*)				ProofNet (*Pass@1*)				MiniF2F (*Pass@16*)	
	Fixed		Adaptive		Fixed		Adaptive		Fixed (B^*)	Adaptive
	Max	Avg. (B^*)	Max	Avg. (Δ)	Max	Avg. (B^*)	Max	Avg. (Δ)		
InternLM2.5-StepProver	59.83	58.93_{16}	59.83	59.10 +0.17	20.96	19.89_{32}	19.89	19.46 –0.43	-	-
BFS-Prover	61.47	61.07_{8}	**62.30**	**61.56** +0.49	20.43	19.78_{32}	20.96	20.54 +0.76	68.03_{4}	66.80
Ours	60.24	59.51_{8}	**62.30**	60.74 +1.23	**21.51**	$\mathbf{20.75}_{32}$	**22.04**	**21.18** +0.43	$\mathbf{68.85}_{4}$	67.21

following the method in Section 3.1. Then, before the actual training begins, the dataset is further combined with human-authored data from Mathlib, followed by deduplication and decontamination. This results in a final training set comprising approximately 20 million proof transitions.

Baselines. We select two strong representative models that use tree search methods, InternLM2.5-StepProver (Wu et al., 2024) and BFS-Prover (Xin et al., 2025), as baseline models. Both baselines are trained via multiple rounds of EI, requiring a larger volume of training samples and higher total computational resources than ours. We apply the same search budget to these baselines in our experiments. All comparisons in this paper are established under the tree search paradigm; for a discussion regarding whole-proof methods, please refer to Appendix A.12.

Environments. We develop DoBeVi (cf. Appendix A.3) as the interaction tool between the model and the Lean 4 prover (v4.10.0).

Benchmarks, metrics and computation budgets. We evaluate on two popular benchmarks, MiniF2F (Zheng et al., 2021) and ProofNet (Azerbayev et al., 2023), using *Pass@1* as the metric. We follow the commonly adopted search budget in tree search methods: $K \times B \times E$, where K is the number of searches per theorem (i.e., k in *Pass@k*); B is the beam size per expansion; and E is the maximum number of expansions allowed in a single search. Particularly, we fix $E = 600$ for all evaluation in this paper.

4.2 Main results

Table 1 presents the results of our model alongside baseline methods on MiniF2F and ProofNet. Since most existing baseline results were not evaluated under our specific search budget, we re-evaluate all baselines using our own implementation to ensure a fair comparison. Notably, some baselines achieved even higher results in our environment than those reported in original papers. Detailed environment settings and search techniques can be found in Appendix A.5.1.

The results demonstrate that our method achieves competitive performance without relying on EI, while requiring only a single data-synthesis pass instead of the multi-round EI pipeline used by BFS-Prover and achieving comparable policy quality with substantially lower cumulative data-synthesis time in our controlled comparison (Appendix A.9.1). Moreover, the adaptive beam size strategy generalizes across different policy models and improves Pass@1 performance in most settings. Under Pass@16, our model further outperforms BFS-Prover. This result is consistent with the objective of our data synthesis method to enhance diversity: by encouraging broader exploration within individual search trees, the resulting policy promotes greater diversity across independent searches, thereby improving cumulative success over multiple runs.

Interestingly, when employing a fixed beam size strategy, the optimal beam size (B^*) varies across different models, benchmarks, and $Pass@k$ metrics. We provide further discussion of this phenomenon in Section 4.3 and Appendix A.5.2.

Additional experiments and insights. We conduct a set of supplementary experiments to better understand the limitations of existing search strategies and the behavior of our proposed method. First, we report a series of early unsuccessful attempts in Appendix A.6, which indicate that many seemingly reasonable modifications to the search procedure yield limited improvements when the policy model and scoring function remain unchanged. These findings motivate a deeper investigation of scoring mechanisms. Accordingly, Appendix A.7 provides a focused experimental analysis of different scoring functions, showing that existing designs exhibit trade-offs across problems and that

intuitive normalizations are not consistently effective. Finally, we include additional experiments on the adaptive beam size strategy in Appendix A.8, demonstrating that its performance gains are broadly consistent across different problem types rather than being concentrated in specific domains.

4.3 DISCUSSIONS

Beyond policy model capability, tree search performance is heavily influenced by search configurations, with beam size being a particularly critical factor. As shown in Table 1, the optimal beam size (B^*) varies across MiniF2F and ProofNet for both our model and baselines, likely due to training data distributions. Most Lean 4 datasets derive from Lean-Workbook (Ying et al., 2024), which aligns closely with the MATH subset of MiniF2F. In such in-distribution settings, larger beam sizes often introduce redundant branches that exhaust the search budget without yielding improvements. Conversely, for out-of-distribution benchmarks like ProofNet, a larger beam size facilitates broader exploration, helping the model identify tactics that lead to familiar intermediate proof states.

The choice of beam size fundamentally dictates the search tree topology. A smaller beam size produces "tall and thin" trees with greater average depth (cf. Appendix A.14), which is effective when the policy model can confidently identify proof-terminating steps. However, under the *Pass@1* metric, this leads to higher variance if early expansions are suboptimal. In contrast, a larger beam size creates "shallow and wide" structures; while this reduces variance, it may compromise search efficiency by introducing irrelevant proof steps.

As the number of independent searches K increases, system diversity is increasingly driven by inter-tree exploration rather than internal tree breadth. This shift suggests that individual searches should prioritize exploitation (smaller B) to maximize cumulative success. This is evidenced in Table 1, where B^* for our model and BFS-Prover shifts from 8 at *Pass@1* to 4 at *Pass@16*. From our perspective, the adaptive beam size strategy is optimized for the exploration-exploitation trade-off within a single tree, explaining its significant gains in *Pass@1* settings. In high-K scenarios, where external exploration dominates, the relative advantage of the adaptive strategy over a specialized, exploitation-heavy fixed beam (e.g., $B = 4$) becomes less pronounced as the necessity for internal exploration diminishes.

4.4 ABLATION STUDY

We conduct comprehensive ablation studies on both the data synthesis method and the adaptive beam size strategy.

Ablation on Data Synthesis. (cf. Appendix A.9) We perform a two-level ablation study to systematically evaluate our proposed synthesis method. At the *macro-level*, we compare our single-pass approach against the multi-round Expert Iteration (EI) pipeline and investigate the impact of hyperparameters α and β. As illustrated in Figure 3, our method consistently outperforms the model trained through four iterations of EI, while demonstrating high robustness across various hyperparameter configurations. At the *micro-level*, we conduct a causal analysis to isolate the effects of Constrained Decoding and Budget Decay, which confirms the specific contributions of both components in alignment with our expectations. Beyond these performance metrics, we provide detailed insights into the roles of α, β, and γ in regulating the exploration-exploitation trade-off. We also analyze the advantages of our method through comprehensive statistical profiling of the generated search trees.

Ablation on Adaptive Beam Size Strategy. (cf. Appendix A.10) We conduct a focused ablation study on the hyperparameters of the adaptive beam size strategy to investigate their impact on evaluation performance and to establish empirical guidelines for their selection. Our results demonstrate that choosing B_{max} and B_{min} within reasonable ranges improves the effectiveness of tree search.

5 RELATED WORK

Automated theorem proving. ATP has been a classic research area which is historically dominated by symbolic approaches (Pastre, 1993; Schürmann & Pfenning, 1998; Leino, 2013). In recent years, the rise of LLMs has spurred renewed interest. One representative work is GPT-f (Polu & Sutskever, 2020), which uses the Expert Iteration approach to train the LLM for the next proof step generation

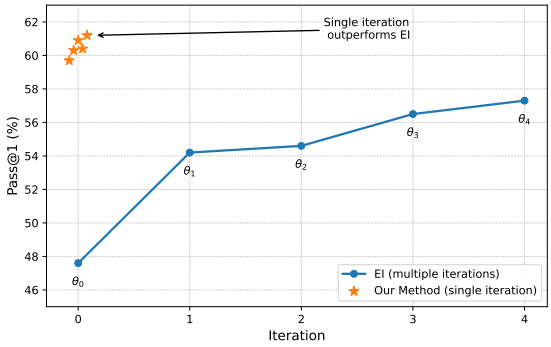


Figure 3: Main evaluation results of the ablation study on data synthesis methods. For EI, we perform one round of bootstrapping followed by four subsequent iterations, while our method requires only a single round of tree search. The five results shown in the upper-left corner correspond to our method with different hyperparameter settings.

based on the current state, combined with tree search methods to prove theorems. Subsequent tree search methods have followed similar ideas, e.g., Polu et al. (2022); Lample et al. (2022); Lin et al. (2024). Additionally, some work improves tree search tailored in ATP. For example, Wu et al. (2024) uses the DPO algorithm to train an auxiliary value network to evaluate intermediate proof states, avoiding the issue of directly using beam search probabilities as scoring signals.

In contrast, whole-proof generation methods (Xin et al., 2024b; Lin et al., 2025; Zhang et al., 2025; Wang et al., 2025a) aim to produce an entire proof in a single forward pass from the initial goal. While these methods do not leverage intermediate proof states, they are typically more efficient and allow for multiple rollouts to increase the likelihood of success. A notable example is DeepSeek-Prover-V1.5 (Xin et al., 2024b), which enhances performance by inserting natural language chain-of-thought (CoT) commentary into Lean 4 proofs during inference. Additionally, MA-LoT (Wang et al., 2025b) introduces a multi-agent architecture where separate models are responsible for analyzing and revising incorrect proofs, enabling iterative refinement toward a correct solution.

Data synthesis for theorem proving. One of the major bottlenecks in LLM-based automated theorem proving remains the lack of high-quality training data, particularly for Lean 4, a relatively new language with limited existing formalized data. Existing data generation methods can be broadly categorized into two groups.

The first category leverages LLMs to generate Lean 4 problems by building on the abundance of natural language mathematics. These approaches are generally referred to as *autoformalization*. However, current efforts primarily focus on autoformalizing problem *statements*, with less attention given to the autoformalization of proof processes (Wang et al., 2025a; Lu et al., 2024).

The second category bootstraps from a small set of existing Lean 4 problems to generate new ones through iterative synthesis. These methods typically involve a two-model framework: one model generates new conjectures, and the other attempts to prove them, enabling co-evolution of data and model capabilities. For instance, STP (Dong & Ma, 2025) uses problems from the Lean-Workbook dataset (Ying et al., 2024) as seeds. A conjecturer proposes conjectures related to the original problems, while a prover attempts to validate themallowing both the training set and the model to be refined through this iterative process.

6 Conclusions

In this paper, we presented a data synthesis method and an adaptive tree search pruning strategy for automated theorem proving in Lean 4. Our approach enables large-scale generation of diverse intermediate proof states and transformation steps without ground-truth solutions. Combining with the adaptive beam size strategy, our method effectively balances exploration and exploitation throughout the tree search process. Together with the developed DoBeVi Lean 4 interaction tool, we expect our contributions will support and inspire further research within the ATP community, which is burgeoning but under development.

AI USE STATEMENT

Generative AI tools were used to assist with language polishing, LaTeX formatting and consistency checks, and the retrieval and discovery of relevant literature. Any references identified with AI assistance were independently verified by the authors against the original sources. All AI-assisted manuscript changes were reviewed by the authors. The authors take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

The experimental setup, training configuration, search budgets, and implementation details are described in Section 4.1. More detailed experimental settings and evaluation results are provided in Appendix A.5. Our implementation code is available at `https://anonymous.4open.science/r/llm_based_atp-8B67`, and the model checkpoint is available at `https://huggingface.co/RYbWnlL0v8/llm_atp_model`.

REFERENCES

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

Chenyang An, Zhibo Chen, Qihao Ye, Emily First, Letian Peng, Jiayun Zhang, Zihan Wang, Sorin Lerner, and Jingbo Shang. Learn from failure: Fine-tuning llms with trial-and-error data for intuitionistic propositional logic proving. *arXiv preprint arXiv:2404.07382*, 2024.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023.

Kefan Dong and Tengyu Ma. Stp: Self-play llm theorem provers with iterative conjecturing and proving. *arXiv e-prints*, pp. arXiv–2502, 2025.

Panagiotis Giadikiaroglou, Maria Lymperaiou, Giorgos Filandrianos, and Giorgos Stamou. Puzzle solving using reasoning of large language models: A survey. *arXiv preprint arXiv:2402.11291*, 2024.

Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. *arXiv preprint arXiv:1704.07138*, 2017.

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

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

K Rustan M Leino. Automating theorem proving with smt. In *International Conference on Interactive Theorem Proving*, pp. 2–16. Springer, 2013.

Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Huang, Kashif Rasul, Longhui Yu, Albert Q Jiang, Ziju Shen, et al. Numinamath: The largest public dataset in ai4maths with 860k pairs of competition math problems and solutions. *Hugging Face repository*, 13:9, 2024.

Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. Lean-star: Learning to interleave thinking and proving. *arXiv preprint arXiv:2407.10040*, 2024.

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

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

Jianqiao Lu, Yingjia Wan, Zhengying Liu, Yinya Huang, Jing Xiong, Chengwu Liu, Jianhao Shen, Hui Jin, Jipeng Zhang, Haiming Wang, et al. Process-driven autoformalization in lean 4. *arXiv preprint arXiv:2406.01940*, 2024.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pp. 367–381, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

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

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. minif2f-lean revisited: Reviewing limitations and charting a path forward. *arXiv preprint arXiv:2511.03108*, 2025.

Dominique Pastre. Automated theorem proving in mathematics. *Annals of Mathematics and Artificial Intelligence*, 8:425–447, 1993.

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

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

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, 2025. URL https://arxiv.org/abs/2504.21801.

Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for lf. In *International Conference on Automated Deduction*, pp. 286–300. Springer, 1998.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

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

Haiming Wang, Huajian Xin, Zhengying Liu, Wenda Li, Yinya Huang, Jianqiao Lu, Zhicheng Yang, Jing Tang, Jian Yin, Zhenguo Li, et al. Proving theorems recursively. *arXiv preprint arXiv:2405.14414*, 2024.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025a.

Ruida Wang, Rui Pan, Yuxin Li, Jipeng Zhang, Yizhen Jia, Shizhe Diao, Renjie Pi, Junjie Hu, and Tong Zhang. Ma-lot: Multi-agent lean-based long chain-of-thought reasoning enhances formal theorem proving. *arXiv preprint arXiv:2503.03205*, 2025b.

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

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Jiayu Wang, Dahua Lin, and Kai Chen. Internlm2. 5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems. *arXiv preprint arXiv:2410.15700*, 2024.

Violet Xiang, Charlie Snell, Kanishk Gandhi, Alon Albalak, Anikait Singh, Chase Blagden, Duy Phung, Rafael Rafailov, Nathan Lile, Dakota Mahan, et al. Towards system 2 reasoning in llms: Learning how to think with meta chain-of-thought. *arXiv preprint arXiv:2501.04682*, 2025.

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

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

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

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*, 2024a.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

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

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

Fei Yu, Hongbo Zhang, Prayag Tiwari, and Benyou Wang. Natural language reasoning, a survey. *ACM Computing Surveys*, 56(12):1–39, 2024.

Jingyuan Zhang, Qi Wang, Xingguang Ji, Yahui Liu, Yang Yue, Fuzheng Zhang, Di Zhang, Guorui Zhou, and Kun Gai. Leanabell-prover: Posttraining scaling in formal reasoning. *arXiv preprint arXiv:2504.06122*, 2025.

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

A APPENDIX

A.1 PROMPTS USED FOR POLICY MODEL

For the policy model, we use the similar prompt template as in GPT-f (Polu & Sutskever, 2020), where the prompt only describes the current proof state without introducing any additional information. The template and an example are shown below:

Template:

[GOAL]
<state >
[PROOFSTEP]

Example: (aime_1983_p1 in MiniF2F)

[GOAL]
x y z w : $\mathbb{N}$
ht : 1 <x $\wedge$ 1 <y $\wedge$ 1 <z
hw : 0 $\leq$ w
h0 : Real.log w / Real.log x = 24
h1 : Real.log w / Real.log y = 40
h2 : Real.log w / Real.log (x * y * z) = 12
$\vdash$ Real.log w / Real.log z = 60
[PROOFSTEP]

A.2 CASE STUDY

Below we present two example proofs generated by our model on MiniF2F and ProofNet, respectively.

Example: amc12a_2009_p6 in MiniF2F

```
theorem amc12a_2009_p6
  (m n p q : ℝ) ℝ
  (h0 : p = 2 ^ m)
  (h1 : q = 3 ^ n) :
  p^(2 * n) * (q^m) = 12^(m * n) := by
    subst_vars
    simp [mul_comm, Real.rpow_mul]
    have h0 : (0 : ℝ) < 2 := by norm_num
    have h1 : (0 : ℝ) < 3 := by norm_num
    field_simp [Real.rpow_def_of_pos, h0, h1]
    rw [ Real.exp_add]
    rw [mul_comm]
    rw [
        show (n * 2 * (Real.log 2 * m) + Real.log 3 * n * m) =
    (Real.log 12 * m * n) by
        have h3 : Real.log 12 = Real.log (2 ^ 2 * 3) := by norm_num
        rw [h3]
        field_simp [Real.log_mul, Real.log_rpow, mul_assoc]
        ring_nf
    ]
```

Example: Rudin_exercise_4_11a in ProofNet

```
theorem exercise_4_11a
    {X : Type*} [MetricSpace X]
    {Y : Type*} [MetricSpace Y]
    (f : X → Y) (hf : UniformContinuous f)
    (x : ℕ → X) (hx : CauchySeq x) :
    CauchySeq (λ n => f (x n)) :=
        rw [Metric.cauchySeq_iff] at hx ⊢,
        intro ε hε,
        obtain δ, hδ, hδ' := Metric.uniformContinuous_iff.mp hf ε
    hε,
        obtain ⟨ N, hN ⟩ := hx δ hδ,
        exact ⟨ N, fun m hm n hn => hδ' <| hN _ hm _ hn ⟩
```

A.3 Our DoBeVi REPL

To enable better interaction with Lean 4 prover and visualization of the search process, we develop Dojo-BeamSearch-Visualization (DoBeVi) REPL, a tool built upon a simplified and customized version of LeanDojo (Yang et al., 2023). The tool consists of two main components: *interactive theorem proving environment* and *visualization module*.

A.3.1 Interactive theorem proving environment

In the component of interactive theorem proving environment, we present a streamlined adaptation of the LeanDojo framework, establishing a lightweight but robust foundation for distributed interactive theorem proving. Our implementation retains two core functionalities from LeanDojo:

- AST extraction and semantic analysis of target Lean files with precise theorem declaration identification.
- Core interactive logic execution via the `lean_dojo_repl` custom tactic.

Our DoBeVi REPL refines the architecture by removing three categories of non-essential components from LeanDojo:

- Partial initialization workflows.
- Premise and dependency tracking modules.
- Secondary data processing components.

Then we add the following new useful features for better interaction with the prover:

- Proof state tracking using globally unique identifiers (excluding `ProofFinish` nodes).
- Decoupled session management architecture enabling parallel theorem processing.
- Resilient recovery mechanisms for:
  - Tactic execution timeouts.
  - Lean process failures.

After that, the DoBeVi provides an optimized architecture that reduces runtime overhead while maintaining stable interfaces for integrating model-based theorem proving components.

A.3.2 Visualization of the search tree

The visualization module serves to clearly illustrate the entire proof search process by graphically representing the evolving search tree structure. In this module, each proof state encountered during the search is abstracted as a node in a graph, and each proof step applied to transition between states is represented as a directed edge. To better reflect the semantics of the search process, nodes are categorized into three types:

- ProofFinished node: Indicates that the proof has been successfully completed at this node.
- Error node: Represents a failed search node due to errors such as Lean syntax errors, time-outs, or unexpected crashes of the Lean process.
- Open node: Denotes an intermediate state that can still be expanded during the search.

Similarly, edges are divided into two categories:

- Tree edge: Connects to a previously unseen state, resulting in the creation of a new node. Tree edges are essential for expanding the search space and discovering new proof paths.
- Back edge: Leads to a state that has already been visited earlier in the search process. Back edges help identify convergence and redundancy in the search, enabling cycle detection and pruning.

Although the structure is referred to as a "search tree", it is, strictly speaking, a directed acyclic graph (DAG). This distinction stems from two key characteristics:

- Non-tree structure: Different but semantically similar proof steps may both lead from the same current state to the same next state, resulting in multiple paths converging to a single node. The property that there are no loops inside the tree is violated.
- Acyclic design: Cycles in the search path (e.g., $s_1 \to s_2 \to \cdots \to s_1$) are explicitly detected and pruned in advance, as we consider such loops to be semantically meaningless and computationally redundant.

This visualization module is part of the DoBeVi system, which is organized in a modular fashion. The system consists of the following four components:

- Search tree module: Defines the core data structures, including Node, Edge, and the overall Tree layout.
- Tactic generation module: Interfaces with LLMs. Given the current state as input, it returns a set of candidate proof steps along with their associated scores. This module is extensible and allows users to integrate custom models as needed.
- Search strategy module: Specifies the logic for node expansion. We provide several default strategies such as Best-First Search, which always expands the node with the highest current score that has not yet been explored. Users are free to implement additional strategies tailored to their tasks.
- Visualization module: Responsible for rendering the entire search tree graphically, enabling an intuitive understanding of the search process and state transitions.

Figure 4 presents the visualization result of applying the Best-First Search strategy to the problem amc12_2000_p6 from the MiniF2F dataset. Node IDs increment from 0, indicating the order in which nodes were expanded during the search. In this example, the search sequentially expanded nodes with IDs 0, 3, 11, 14, 16, 19, and 21, ultimately completing the proof successfully after the seventh expansion.

In summary, this module visualizes the search tree to record the entire search process, making it easier to review after execution and to inspire the design of better search and pruning strategies.

A.4 Basic functionality of the example tactics

Below we summarize the Lean 4 tactics used in our examples, together with minimal illustrations. Each illustration uses comments to indicate the context and the effect of the tactic.

- **`rw`** (rewrite): Rewrites the goal using a known equality.

```
-- Context: h : a = b, goal: f a = c
rw [h]
-- After: goal becomes f b = c
```

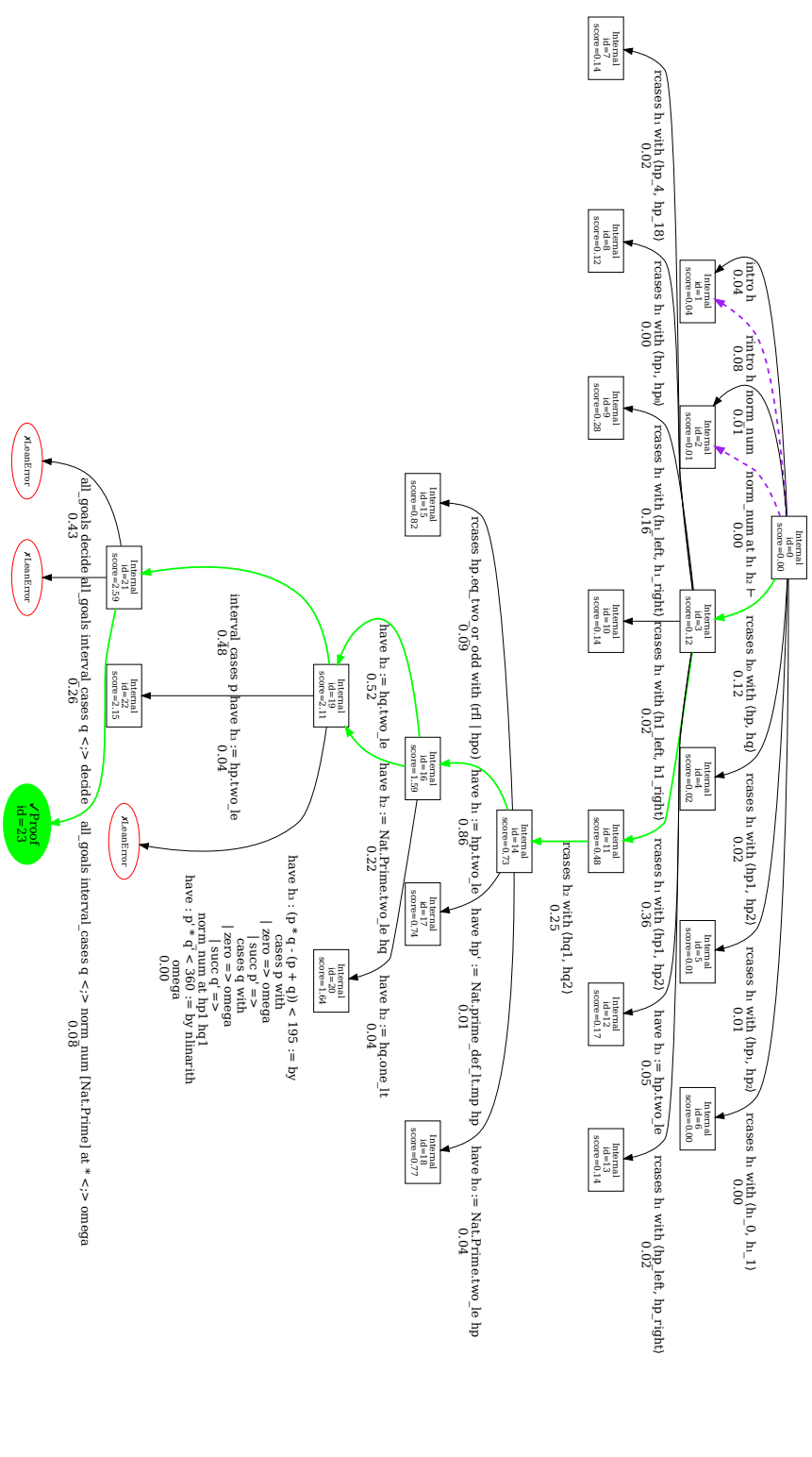


Figure 4: An example of search tree (amc12_2000_p6 in MiniF2F). Rectangular nodes represent `Open Nodes`, red elliptical nodes indicate `Error Nodes`, and green nodes denote `ProofFinished Nodes`. Edges are labeled with the applied tactic and its associated beam probability (`beam_prob`). Each node is annotated with its unique id and current score.

- **have**: Introduces an intermediate fact for later use.

```
-- Before: no lemma about x + y
have h1 : x + y = y + x := by apply add_comm
-- After: h1 is available as an additional assumption
```

- **rcases**: Decomposes a structured hypothesis (e.g., an existential or a conjunction).

```
-- Context: h :  n, n > 0  n < 5
rcases h with n, hn_pos, hn_lt5
-- After: n : Nat, hn_pos : n > 0, hn_lt5 : n < 5
```

- **apply**: Matches the goal with the conclusion of a lemma, turning it into its premises.

```
-- Context: goal is x  z, lemma le_trans : x  y  y  z  x  z
apply le_trans
-- After: two subgoals: x  y  and  y  z
```

- **field_simp**: Simplifies expressions in fields by clearing denominators.

```
-- Context: goal contains a rational expression, assume h : x  0
field_simp [h]
-- After: denominators are cleared, e.g., (a / x) * x becomes a
```

- **nlinarith**: Automatically solves nonlinear arithmetic goals.

```
-- Context: hx : x  0, hy : y  0, goal: x * y  0
nlinarith
-- After: goal is discharged automatically
```

- **rfl**: Closes goals that are definitionally equal.

```
-- Context: goal: (x + 1) - 1 = x
rfl
-- After: goal closed because both sides reduce to the same
    expression
```

A.5 Detailed experimental settings and results

A.5.1 Setup

In our experiments, similar to most tree search methods, we separate the tree search and model generation components to improve overall evaluation efficiency. We refer to the implementation in ReProver (Yang et al., 2023) and use Python's Ray [4] and vLLM (Kwon et al., 2023) libraries to implement the system. We launch multiple processes for tree search, where each process pulls unprocessed theorems from a queue and uses LeanDojo as the interaction tool between Python and the Lean 4 prover. When fetching proof steps, multiple tree search workers share several asynchronous LLM engines launched by vLLM, with one LLM instance deployed per GPU. In practice, we use dozens of tree search processes to maximize GPU utilization. For each search, we set a global timeout of 1,800 seconds; if the search time exceeds this value, the search process is forcibly terminated. For each proof step, we impose a timeout of 20 seconds. While there are occasional cases - such as when using `aesop` - where longer execution times may be required, we consider any proof step exceeding this threshold as a timeout. This constraint is intended to prevent individual proof steps from disproportionately consuming the overall time budget.

During tree search, we use the REPL functionality provided by DoBeVi (Appendix A.3). When executing proof steps, errors can broadly be categorized into two types: (1) syntax errors or similar issues that directly cause the Lean 4 prover to return an error, which are usually caught quickly with an error message; and (2) timeout errors. To prevent infinite loops caused by certain tactics, we set a timeout for tactic execution. If such a timeout occurs, the Lean 4 prover becomes unable to execute new proof steps, forcing the entire proof process to stop. To handle this, we save the proof context before executing each tactic; if a timeout occurs, we restart the Lean REPL environment and restore

[4] https://github.com/ray-project/ray

the context. This avoids cases where a proof that should have succeeded with tree search ultimately fails just because a faulty tactic caused the Lean 4 prover to hang.

In addition, based on observations of certain failure cases during tree search, we identified an important phenomenon: erroneous proof steps tend to appear in batches. Specifically, when performing beam search from a proof state, if several consecutive proof steps are incorrect, it is highly likely that the remaining proof steps in the beam will also be incorrect. This pattern may stem from the fact that the current proof state represents an out-of-distribution input for the policy model, making it difficult for the model to generate valid Lean 4 code. Under such circumstances, these erroneous proof steps can substantially consume the global timeout allocated for the tree search, leading to inefficient exploration. To mitigate this issue, we adopt an early termination strategy: during expansion at a given state, if the number of incorrect child nodes exceeds a predefined threshold (usually $0.5\times$ beam size), we immediately discard the current node from further search. We found that this pruning technique did not degrade overall proving performance while significantly reducing computational time.

Regarding the choice of hyperparameters in the adaptive beam size strategy, we set $B_{\text{max}} = 16$, $B_{\text{min}} = 4$, and $\lambda = 15$ for MiniF2F, and $B_{\text{max}} = 48$, $B_{\text{min}} = 24$, and $\lambda = 2$ for ProofNet. Further discussion of these hyperparameters can be found in Appendix A.10.

A.5.2 Full experimental results

This section provides a comprehensive breakdown of the experimental results across all evaluated benchmarks and configurations. To ensure the highest level of rigor and reproducibility, we report the performance of our model and the key baselines across a wide range of beam sizes $B \in \{4, 8, 16, 32\}$, as well as our proposed adaptive beam size strategy.Table 2 details the *Pass@1* performance on MiniF2F-test and ProofNet-test.

Table 2: Complete *Pass@1* evaluation results across MiniF2F-test and ProofNet-test. All results are obtained through re-evaluation in our unified environment to ensure a fair comparison under a fixed expansion budget $E = 600$.

Model	Model Size	Beam Size (B)	MiniF2F-test	ProofNet-test
InternLM2.5-StepProver	7B	4	$57.05\% \pm 0.66\%$	$13.44\% \pm 1.23\%$
		8	$58.69\% \pm 0.54\%$	$16.77\% \pm 0.63\%$
		16	$\mathbf{58.93\% \pm 0.60\%}$	$18.06\% \pm 0.87\%$
		32	$58.20\% \pm 0.00\%$	$\mathbf{19.89\% \pm 0.68\%}$
		Adaptive	$\mathbf{59.10\% \pm 0.60\%}$	$\mathbf{19.46\% \pm 0.40\%}$
BFS-Prover	7B	4	$60.90\% \pm 0.61\%$	$17.85\% \pm 0.92\%$
		8	$\mathbf{61.07\% \pm 0.45\%}$	$19.25\% \pm 1.04\%$
		16	$60.49\% \pm 0.61\%$	$19.68\% \pm 0.73\%$
		32	$59.75\% \pm 0.31\%$	$\mathbf{19.78\% \pm 0.53\%}$
		Adaptive	$\mathbf{61.56\% \pm 0.71\%}$	$\mathbf{20.54\% \pm 0.53\%}$
Ours	7B	4	$56.56\% \pm 1.13\%$	$13.01\% \pm 0.79\%$
		8	$\mathbf{59.51\% \pm 0.79\%}$	$17.53\% \pm 0.26\%$
		16	$59.02\% \pm 1.13\%$	$19.35\% \pm 0.96\%$
		32	$57.05\% \pm 0.40\%$	$\mathbf{20.75\% \pm 0.94\%}$
		Adaptive	$\mathbf{60.74\% \pm 0.88\%}$	$\mathbf{21.18\% \pm 0.55\%}$

Table 3 presents the complete evaluation results on MiniF2F-test with $K = 16$. These results provide additional evidence regarding the scaling behavior of different search strategies and further validate the observations discussed in the main text.

First, as the number of independent searches K increases, the system's diversity is increasingly driven by inter-tree exploration. Consequently, optimal cumulative performance is achieved when each individual search prioritizes exploitation (e.g., $B = 4$), as evidenced by the performance degradation observed with larger fixed beam sizes for both models. This aligns with our core finding that the search budget should be carefully balanced between exploration and exploitation. Notably, by further reducing B_{max} in our adaptive strategy under this high-K setting, we can achieve even superior results than those reported in Table 1.

Second, our adaptive beam size strategy demonstrates superior robustness compared to fixed beam settings. While its relative advantage is slightly moderated at high $K$where multi-run diversity becomes the dominant factorit maintains stable, high-tier performance across various hyperparameter configurations. In contrast, fixed beam strategies exhibit significant performance drops when B is sub-optimally tuned (e.g., $B = 32$), highlighting the reliability of the adaptive approach in diverse search scenarios.

Table 3: *Pass@16* performance (%) on MiniF2F-test under a fixed expansion budget $E = 600$ per search. For the adaptive beam size strategy, we explore various hyperparameter sets to evaluate the robustness and peak performance.

Search Setting	Hyperparameters	BFS-Prover	Ours
Fixed Beam Size	$B = 4$	68.03	**68.85**
	$B = 8$	66.39	66.80
	$B = 16$	65.98	64.75
	$B = 32$	63.11	63.52
Adaptive Beam Size	$B_{max} = 6, B_{min} = 2, \lambda = 10$	67.21	**68.03**
	$B_{max} = 8, B_{min} = 2, \lambda = 10$	67.62	68.03
	$B_{max} = 16, B_{min} = 4, \lambda = 10$	65.98	67.62
	$B_{max} = 16, B_{min} = 4, \lambda = 15$	66.80	67.21

Finally, to verify the generalizability of the adaptive beam size strategy across different evaluation sets, we conducted additional experiments on four distinct benchmarks (Table 4). These tests maintained a consistent configuration ($B_{max} = 16, B_{min} = 4, \lambda = 15$) and were compared against a Fixed Beam Size of 8. For Lean-Workbook and NuminaMath-LEAN, we randomly sampled 2% of the data for evaluation, as the original datasets are too large to serve directly as standard benchmarks.

The adaptive strategy consistently achieves performance gains across all tested benchmarks, ranging from specialized competition sets like miniF2F_v2c to large-scale synthetic datasets like NuminaMath-LEAN. These results strongly suggest that the mechanism of progressively pruning search breadth in favor of depth is a generalizable principle for formal theorem proving, rather than an optimization over-fitted to specific datasets.

Table 4: Performance (%) of adaptive beam size strategy across multiple formal math benchmarks. We report the average *Pass@1* over 3 independent runs. The results demonstrate consistent gains across all evaluated sets.

Benchmark	Fixed ($B = 8$)	Adaptive	Gain (Δ)
miniF2F_v2s (Ospanov et al., 2025)	53.14	54.10	+0.96
miniF2F_v2c (Ospanov et al., 2025)	50.41	51.09	+0.68
Lean-Workbook (Ying et al., 2024)	97.24	97.77	+0.53
NuminaMath-LEAN (Wang et al., 2025a)	55.21	57.61	+2.40

A.6 Unsuccessful attempts in tree-search methods

In this section, we summarize several design choices and algorithmic variants that we explored during early experiments but ultimately found to be ineffective. These unsuccessful attempts provide useful insights into the limitations of current tree-search methods and help motivate the design choices adopted in the main body of this work.

Separating tactics and premises. In our early experiments, we explored a more straightforward separation of tactics and premises. Specifically, we adopted a two-model design: one model directly outputs a valid tactic to serve as the starting token of the code, while a second model, analogous to the policy model, completes the required premises. However, this approach yielded unsatisfactory results. We frequently observed that the second model either failed to generate complete premises or produced outputs that violated the Lean 4 grammar. A key challenge lies in the highly imbalanced data distribution for a model that generates only tactics without premises. To quantify this imbalance, we analyzed the STP dataset and computed the tactic usage statistics, presented in Appendix A.13. For example, tactics like `rw` and `have` appear in nearly every proof context, leading

the model to overwhelmingly favor these high-frequency tactics. In contrast, the policy model must guide the proof state toward completion, which does not always align with simply selecting the most common tactics. This mismatch introduces conflicting objectives that ultimately degrade performance. Therefore, we conclude that forcibly separating the generation of tactics and premises is infeasible. Instead, it may be more effective to adopt external augmentation strategies, such as the retrieval-augmented generation (RAG) method in Yang et al. (2023) or the lemma library construction approach in Wang et al. (2023).

An ineffective scoring function. One major drawback of tree search methods compared to whole-proof methods lies in the complexity of the neural-symbolic system: it involves many interacting components, making it difficult to diagnose failure modes when performance is suboptimal. In some cases, the coupling between components may even be inherently unavoidable. As discussed in the main text, we identify several key factors in tree search methods that have a non-negligible impact on performance:

- **Policy model.** As the actor in the system, the policy model is clearly the most critical determinant of overall performance.
- **Scoring function (sometimes value network).** The scoring function determines the order in which nodes are expanded within the tree structure. In some cases, a value network may be used for scoring instead of a rule-based method.
- **Beam size.** The beam size represents a special type of budget. As discussed in Section 4.3, increasing the beam size does not necessarily lead to better results. On the contrary, for certain settings, a small beam size (such as 4) can sometimes unlock greater potential from the policy model. Moreover, the empirically optimal beam size varies across benchmarks and is not a fixed value.
- **Other budget-related factors.** These include parameters like K, E, and the timeout threshold. Compared to the previous factors, these are relatively straightforward: given fixed settings for other components, larger budgets generally increase the likelihood of achieving better results.

In our early experiments, we hypothesized that the problem setting resembled reinforcement learning, and thus sought inspiration from algorithms such as GRPO Shao et al. (2024). We aimed to avoid introducing a separate value network, instead using the average performance of a set of actions generated by the actor as a scoring criterion. Concretely, for any intermediate proof state s, we computed the following score:

$$\text{score}(s) = \frac{1}{B} \sum_{i=1}^{B} \log P_\theta(t_i \mid s) \tag{3}$$

Here, θ denotes the policy models parameters, and $P_\theta(t_i \mid s)$ represents the joint probability of the i-th beam. The intuition behind this scoring function is that, on an average sense, higher beam probabilities suggest that the policy model is more likely to have encountered similar examples during training. Therefore, the current proof state s is more likely to lie on a path leading to proof completion.

However, subsequent experiments indicate that the evaluation results obtained using this scoring function are nearly indistinguishable from those derived using Equation 1, suggesting that this approach may not be particularly effective within the context of tree search methods.

Balancing exploration and exploitation. In search process, a common optimization strategy is to balance exploration and exploitation, with classical algorithms such as MCTS following this principle. In the ATP context, a number of methods with similar ideas have also been proposed, for instance the length-normalized BFS introduced in Xin et al. (2025). However, in our experiments, we find that these approaches show no clear difference from naive BFS in terms of evaluation results.

Applying the standard MCTS framework to ATP presents several significant challenges. First, the selection step in MCTS relies on the Upper Confidence Bound (UCB), which requires access to the parent nodes visit count. Although the ATP search process is often described as a tree search, the underlying structure is in fact a directed acyclic graph (DAG), since the same intermediate proof

state can be reached through multiple derivation paths. This makes it ambiguous, or even impossible, to assign a unique parent visit count to each node. Second, the classical simulation step in MCTS is impractical for ATP. Once a simulation reaches a successful proof state, the proof is complete and no further search is necessary. Moreover, simulations consume interactions with the Lean server step by step, which also exhausts the allocated budget. As a result, applying the classical simulation of MCTS to ATP effectively reduces to a form of best-first search. In practice, we experiment with modified versions of MCTS to address these issues. On the MiniF2F-test benchmark under the $1 \times 8 \times 600$ setting, the modified MCTS achieves an average *pass@1* of 56.15%, which is slightly lower than that of naive BFS.

We also explored normalized BFS, which normalizes by the current depth of the tree node or the length of the proof step. However, we find that their generalization ability is limited, and when combined with our policy model, it has virtually no effect to the tree search process.

Dynamic beam size via *top_p* strategy. Another adaptive beam search strategy we explored is inspired by the *top_p* (nucleus sampling) strategy commonly used in decoding. Specifically, after normalizing the joint probabilities of each beam, we sort the beams by their joint probabilities and then filter out beams with lower probabilities according to the *top_p* threshold. Despite its conceptual appeal, this strategy did not lead to measurable improvements under our experimental settings.

Summary. Overall, these unsuccessful attempts suggest that many seemingly reasonable modifications to the search procedure have limited impact when the policy model and scoring mechanism remain unchanged. In particular, ad hoc exploration strategies or heuristic normalizations are often insufficient to overcome the inherent limitations of existing scoring functions. A more detailed analysis of scoring functions is therefore necessary, which we present in the Appendix A.7.

A.7 Further analysis of scoring function

To examine the effectiveness of different scoring functions, we select several representative methods and conduct additional experiments on the MiniF2F-Test split using our model. The experimental budget is set to $K = 1$, $B = 8$, and $E = 600$, with a global timeout of 1800 seconds per problem. The results are shown in Table 5.

Table 5: Comparison of representative scoring functions on MiniF2F-Test under fixed search budget. The evaluation results reported here and those in Table 1 are obtained from different experimental runs, and therefore exhibit slight variations.

Scoring Function	Results
log probability	$59.9\% \pm 0.9\%$
log probability + *depth normalization* (α = 0.5) (Xin et al., 2025)	$59.5\% \pm 1.7\%$
log prob + *length normalization*	$57.7\% \pm 1.7\%$
Value network (InternLM2.5-StepProver-critic) (Wu et al., 2024)	$57.2\% \pm 1.0\%$

From these results, we observe that the performance differences between scoring functions are small. Several normalization-based strategies that are theoretically appealing (such as depth normalization and length normalization) did not lead to the expected improvements. Similarly, using a value-network-based scoring function did not offer clear benefits under the current setup. For this reason, we prefer *log probability* as the default scoring function, as it provides more stable performance across problems.

To more concretely illustrate the limitations of existing scoring functions in current tree-search methods, we take the example of solving the MiniF2F-Test *mathd_algebra_598* as an example, evaluated under the scoring function *log probability* + *depth normalization.* The initial state of the theorem is:

```
a b c d :
 h : 4 ^ a = 5
 h : 5 ^ b = 6
 h : 6 ^ c = 7
```

```
h : 7 ^ d = 8
 a * b * c * d = 3 / 2
```

During the search process, each state's score is computed as: score = parent_score + current_score, where current_score is obtained from the scoring function (*log probability + depth normalization*), and parent_score is the accumulated score along the path. Using this method, the search is able to correctly produce the first eight proof steps:

```
have h : (4 : ) ^ a = 5 := by linarith
have h : (5 : ) ^ b = 6 := h
have h : (6 : ) ^ c = 7 := h
have h : (7 : ) ^ d = 8 := h
have h : (4 : ) > 0 := by norm_num
have h : (5 : ) > 0 := by norm_num
have h : (6 : ) > 0 := by norm_num
have h : (7 : ) > 0 := by norm_num
```

At the next expansion step, a correct proof step would be :

```
have h : Real.log 4 > 0 := Real.log_pos (by norm_num)
```

However, the policy model assigns it a relatively low score -9.29, while three other incorrect proof steps receive scores around -4. As a result, the search prefers the incorrect steps and subsequently moves into an unproductive subtree. At this point, the search depth has already reached 8. Due to depth normalization, states at different depths are mapped to a similar scale, the search struggles to escape once it enters a wrong subtree, often consuming a large portion of the budget without making progress (i.e., solve the problem). Without depth normalization, the scores of such incorrect subtrees would decrease quickly as the depth grows, allowing the search to abandon them much earlier.

This example shows that depth normalization can have a negative impact in certain cases. However, this does not imply that depth normalization is always harmful. When the policy model assigns strong and consistent preferences to correct steps, depth normalization can help keep the search focused on the correct proof trajectory. Our intention with this example is to illustrate that existing scoring functions in tree-search methods can have both positive and negative effects depending on the specific problem, and each approach carries its own strengths and limitations.

A.8 Applicability of adaptive beam size strategy to different problem types

To further examine whether the adaptive beam size strategy behaves differently across problem types, we conduct an additional set of experiments using the official category labels from MiniF2F (e.g., original competition source or mathematical subfield: *aime*, *algebra*, *amc*, *induction*, *numbertheory*, etc.; see Zheng et al. (2021)). For each category, we perform 15 independent evaluations on the MiniF2F-Test split for both the fixed-beam and adaptive-beam strategies. Specifically, we run five evaluations for each fixed beam size $B = 8, 16, 32$, and 15 evaluations for the adaptive beam size strategy. We then compute success rates under each category using two metrics below.

Cumulative solved. This metric measures the proportion of problems that are solved at least once across all runs. For instance, if a category contains 100 problems and 82 of them are solved in at least one of the 15 runs, then the cumulative solved score is $82/100 = 0.82$.

Overall solved. This metric measures the average solve rate across all runs. It is computed as the total number of solved instances across all runs divided by the number of problems times the number of runs. For example, with 100 problems and 15 runs, if the total number of solved instances is 1,080, then the overall solved metric is $1080/(100 \times 15) = 0.72$.

The results are presented in Table 6 and Table 7. Across all subcategories of MiniF2F-Test, the adaptive beam size strategy consistently improves performance, rather than yielding gains concentrated in specific domains. As a general enhancement of the search procedure, its benefits appear broadly applicable across diverse mathematical problem types.

Table 6: Performance of **fixed beam size strategy** across MiniF2F-Test subcategories, evaluated over 15 runs.

Subcategories	Cumulative Solved	Overall Solved
aime	5/15 (33.33%)	64/225 (28.44%)
algebra	10/18 (55.56%)	147/270 (54.44%)
amc	21/45 (46.67%)	245/675 (36.30%)
imo	3/20 (15.00%)	35/300 (11.67%)
induction	5/8 (62.50%)	74/120 (61.67%)
numbertheory	4/8 (50.00%)	53/120 (44.17%)
mathd_numbertheory	51/60 (85.00%)	701/900 (77.89%)
mathd_algebra	60/70 (85.71%)	813/1050 (77.43%)

Table 7: Performance of **adaptive beam size strategy** across MiniF2F-Test subcategories, evaluated over 15 runs.

Subcategories	Cumulative Solved	Overall Solved
aime	5/15 (33.33%)	70/225 (31.11%)
algebra	10/18 (55.56%)	149/270 (55.19%)
amc	22/45 (48.89%)	271/675 (40.15%)
imo	4/20 (20.00%)	46/300 (15.33%)
induction	7/8 (87.50%)	78/120 (65.00%)
numbertheory	4/8 (50.00%)	60/120 (50.00%)
mathd_numbertheory	53/60 (88.33%)	716/900 (79.56%)
mathd_algebra	62/70 (88.57%)	833/1050 (79.33%)

A.9 Ablation study and further analysis of data synthesis methods

To better understand the impact of our proposed data synthesis method on the performance of the policy model, we conduct a systematic suite of ablation studies across multiple dimensions. Our analysis begins with a macro-level comparison against Expert Iteration to verify the overall effectiveness and hyperparameter robustness of our single-pass synthesis approach. We then proceed to a micro-level causal analysis to isolate the specific contributions of core pipeline components. Furthermore, we provide qualitative insights into the design logic of our exploration-exploitation hyperparameters and conclude with a structural analysis of search trees to reveal the underlying advantages of our method in terms of tactic diversity and expansion efficiency.

A.9.1 Comparison with Expert Iteration: Effectiveness and Efficiency

Dataset. Since EI requires performing a complete tree search over a seed problem set at each iteration, which incurs substantial computational cost, we adopt a small-scale dataset for this ablation study. Specifically, we sample two disjoint datasets, denoted as dataset $\mathcal{A}$ and dataset $\mathcal{B}$, from STP, each consisting of 1,000 original problems. Dataset $\mathcal{A}$ is used as the bootstrapping problem set for EI, while dataset $\mathcal{B}$ is used for subsequent EI iterations and for evaluating the capability of the current policy model.

Model. We follow the same model configuration as in the main paper. BFS-Prover is used as the policy model for the initial data synthesis stage in EI, and the synthesized data are used to train the Qwen2.5-Math-7B base model to obtain a policy model with basic theorem-proving capability, denoted as θ_0. This model serves as the starting point for training in each EI iteration.

Search budget. The search budget for each EI iteration is set to $1 \times 8 \times 600$.

Data synthesis procedure. In the EI procedure, we first use BFS-Prover to perform proof tree search on dataset $\mathcal{A}$ and collect all $(\text{state}, \text{proof step})$ pairs encountered during the search that do not trigger Lean 4 errors (following the same setting as in Polu et al. (2022)). These pairs form the initial training dataset D_0. We then train the Qwen2.5-Math-7B base model on D_0 to obtain model θ_0. In each subsequent iteration, we use the model obtained from the previous iteration to perform a full tree search on dataset $\mathcal{B}$, extract $(\text{state}, \text{proof step})$ pairs along successful proof paths, and progressively expand the training dataset. We perform a total of four iterations, producing datasets D_1, D_2, D_3, D_4 and models $\theta_1, \theta_2, \theta_3, \theta_4$, respectively.

For our proposed data synthesis method, we use BFS-Prover to perform tree search on dataset $\mathcal{A}$, following the same procedure as described in Section 3.1, but with only a single round of tree search. In addition, to study the effect of hyperparameters in the data synthesis process, we further use multiple hyperparameter settings while fixing the beam size to 8 after dropout at each expansion.

To control for training-data volume, we approximately match the serialized dataset size between the EI and our synthesis settings.

Efficiency analysis. A key advantage of our method is that it replaces the repeated search–training cycles of Expert Iteration (EI) with a single proof-state exploration pass followed by one SFT stage. BFS-Prover constructs its final policy from data accumulated over ten EI rounds, followed by an additional DPO refinement, but does not report the end-to-end runtime or total GPU-hours of this pipeline. We therefore compare efficiency under the controlled setting described above, where EI and our method use the same initial BFS-Prover checkpoint, Qwen2.5-Math-7B base model, problem sets, and aligned search budgets. In this 1K-problem setting, one EI search stage takes approximately 2 hours, while our single proof-state exploration pass takes approximately 5 hours. Nevertheless, our one-pass pipeline already outperforms the policy obtained after the initial bootstrapping stage and four subsequent EI iterations, which require approximately $5 \times 2 = 10$ hours of cumulative search. Thus, at this level of policy quality, our method reduces the measured data-synthesis time by approximately $2\times$.

For reference, applying the ten EI rounds reported by BFS-Prover to the same controlled setting would correspond to $10 \times 2 = 20$ hours of search, or $4\times$ the cost of our one-pass synthesis, if the per-round workload remained constant. Since BFS-Prover progressively removes solved problems, we further consider a sensitivity analysis in which 80–90% of the active problem set remains after each round and search cost scales with the number of active problems. This yields 4.46–6.51 full-corpus-equivalent rounds, corresponding to a normalized synthesis-cost ratio of approximately 1.8–$2.6\times$. Because BFS-Prover does not report its per-round corpus sizes or end-to-end runtime, these values should be interpreted as normalized estimates under our controlled setting rather than estimates of its actual training cost. Moreover, they account only for proof-search and data-synthesis cost, excluding the additional policy-optimization stages required by iterative EI.

Together with the effectiveness results and Table 1, this analysis shows that our one-pass synthesis pipeline achieves comparable policy quality with substantially fewer iterative search–training cycles.

Results and analysis. The evaluation results of the models obtained by EI and by our method are shown in main paper Figure 3. For EI, the models trained through successive iterations exhibit a steadily improving performance on dataset $\mathcal{B}$. From the initial model θ_0 to θ_4, the number of solved problems consistently increases, and the Pass@1 reaches 57.30% after the final iteration. As illustrated in the figure, although the model capability improves as the number of iterations increases, the gain in each iteration remains relatively limited. In contrast, our method achieves better performance on dataset $\mathcal{B}$ than θ_4 using only a single round of data synthesis and one model training stage. To further control for training-set size, we additionally downsample the constrained-decoding dataset to exactly 384,093 transitions, matching the final EI dataset. Under this stricter transition-count-matched setting, constrained decoding achieves 58.20% Pass@1, compared with 57.30% for EI, showing that the performance advantage is retained when the number of training transitions is exactly matched. In practice, a single round of data synthesis in our method takes approximately 5 hours, while one EI iteration takes about 2 hours. Although constrained decoding is used in our method and the inference speed is affected by the efficiency of the inference engine, the overall time cost is still lower than that of multiple EI iterations.

In addition, Table 8 reports the evaluation results of policy models trained on datasets synthesized with 5 different hyperparameter settings. We observe that under different hyperparameter configu-

rations, our method consistently achieves around 600 solved problems on dataset $\mathcal{B}$. Among these settings, $B = 32, \alpha = 0.25, \beta = 4$ yields the best performance, which is consistent with the setting adopted in the main experiments.

It is worth noting that in Table 8, we maintain a fixed post-pruning effective breadth (i.e., $\alpha B = 8$) across different configurations. This design is essential to decouple the pruning mechanism from the search scale; fixing αB instead of the pre-pruning beam size B prevents the search tree structure and total data volume from becoming confounding variables, thereby ensuring the performance variations stem solely from the selection quality governed by α and β. While the optimal value for this constant is itself a meaningful research question, it remains outside the primary scope of this study. Following the empirical practice in prior work like BFS-Prover, we set $\alpha B = 8$ to ensure sufficient breadth and sample diversity for our single-pass synthesis pipeline.

Table 8: Evaluation results on dataset $\mathcal{B}$ of policy models trained using different data synthesis hyperparameter settings. We fix $\alpha B = 8$ throughout, ensuring that the beam size is 8 at the stage after dropout and before deduplication.

B	α	β	pass@1
8	1	8	59.70%
16	0.5	8	60.30%
32	0.25	8	60.90%
32	0.25	6	60.50%
32	0.25	4	61.20%

We also conduct an additional evaluation on the MiniF2F-test, and the results are reported in Table 9. In this evaluation, we include only the final model θ_4 obtained from EI and the 5 policy models from ablation study in this section. The results show that on MiniF2F-test, the models trained using our data synthesis method remain slightly superior to the EI model.

Table 9: Evaluation results on MiniF2F-test for θ_4 from EI and the 5 policy models from the ablation study in this section. We use a unified search budget of $1 \times 8 \times 600$, and each experiment is run three times.

B	α	β	results
8	1	8	$50.27\% \pm 0.47\%$
16	0.5	8	$49.32\% \pm 1.03\%$
32	0.25	8	$50.41\% \pm 0.41\%$
32	0.25	6	$49.45\% \pm 0.24\%$
32	0.25	4	$49.18\% \pm 0.41\%$
θ_4 from EI			$47.40\% \pm 1.18\%$

A.9.2 Effect of the Data-Generation Policy

We further study the effect of the data-generation policy using a substantially weaker policy than BFS-Prover. We independently sample two disjoint subsets of 1,000 theorems from STP, denoted as D_{gen} and D_{eval}. The former is used exclusively for data synthesis, while the latter is excluded from both synthesis and training and serves as a held-out evaluation set. We first fine-tune Qwen2.5-Math-7B on Mathlib to obtain a policy with basic Lean proving capability. This Mathlib-only policy solves 135 out of 1,000 theorems on D_{eval} and is used as the shared data-generation policy for both methods.

We compare standard EI with constrained decoding under a transition-count-matched setting. The EI branch collects synthetic transitions over three rounds of best-first proof search following the EI procedure described in Appendix A.9.1, while the constrained-decoding branch performs a single round of proof-state exploration. The resulting synthetic data are combined with the same Mathlib data, yielding training sets of approximately 414K proof transitions for both methods. The final policy models are then trained from the same Qwen2.5-Math-7B base model using identical training

configurations. On the held-out D_{eval} set, EI solves 171 out of 1,000 theorems, whereas constrained decoding solves 183. These results show that constrained decoding remains effective even when a substantially weaker data-generation policy is used, achieving better performance than multi-round EI under matched training-data volume.

A.9.3 MICRO-LEVEL ABLATION: COMPONENT-WISE CAUSAL ANALYSIS

Setup. To isolate the causal effects of our synthesis pipeline, we conduct a "remove-one-component" ablation study by systematically disabling Constrained Decoding (CD) and Budget Decay (BD). In this analysis, the *Full Pipeline* setting corresponds to the optimal configuration identified in the previous section ($B = 32, \alpha = 0.25, \beta = 4$). Following the macro-level experimental setup, we use dataset $\mathcal{A}$ for data synthesis and dataset $\mathcal{B}$ for evaluation. All other configurations remain aligned with those described in the macro-level comparison. To ensure a fair comparison, the total expansion budget per problem is held constant across all settings. Unlike standard Expert Iteration, the ablation variants (unless otherwise specified) retain all non-error states to maintain a comparable training data density.

Results and analysis. The evaluation results are reported in Table 10. The results indicate that the full pipeline, which integrates both Constrained Decoding and Budget Decay, achieves the highest performance on dataset $\mathcal{B}$. Specifically, the significant drop in performance upon removing CD confirms its critical role in ensuring tactic diversity. Additionally, the decrease observed without BD suggests that adaptive budget allocation is essential to prevent the model from over-fitting to redundant, easy proof trajectories, allowing the synthesis process to focus on more challenging problems.

Table 10: Component-wise ablation of the proposed synthesis pipeline. "CD" and "BD" denote Constrained Decoding and Budget Decay, respectively. Results are reported as Pass@1 on dataset $\mathcal{B}$. (*) In the w/o CD + w/ BD setting, BD effectively mimics the standard EI behavior of terminating search upon the first success.

Setting	Description	Pass@1
Full Pipeline	Our proposed synthesis method (Main Setting)	63.9%
w/ CD + w/o BD	No decay; exhausts fixed budget for every problem	61.2%
w/o CD + w/ BD*	Equivalent to single-iteration EI; stops at first success	47.6%
w/o CD + w/o BD	No CD; exhausts budget without early stopping	52.2%

A.9.4 INSIGHTS INTO DATA SYNTHESIS HYPERPARAMETERS

We provide further insights into the roles and design motivations of the three hyperparameters used in our data synthesis procedure, namely α, β, and γ, which jointly regulate the exploration–exploitation trade-off during tree search.

Depth-wise exploration–exploitation control (α). The hyperparameter α controls the effective branching factor across tree depths by limiting the number of retained branches at each expansion. Given an initial beam size B, only the top αB branches are preserved. In our data synthesis setup, we typically adopt a relatively large beam size (e.g., $B = 32$) to allow the policy model to propose multiple high-confidence proof steps. However, an overly large beam size tends to produce excessively wide search trees, which restricts the ability to reach states that differ substantially from the initial state after several proof steps. This phenomenon is also reflected during evaluation on familiar benchmarks such as MiniF2F, where large beam sizes are empirically suboptimal. By retaining only a fraction αB of the branches, the search process is biased toward deeper regions of the tree, thereby enhancing exploration over longer proof trajectories.

Branch-wise exploration–exploitation control (β). The hyperparameter β regulates the trade-off between exploitation and exploration among candidate branches within beam search. Purely exploitation-oriented strategies tend to repeatedly select the most likely branches, resulting in limited diversity in the synthesized data. Conversely, unconstrained exploration may cause the search to drift away from meaningful proof construction. To balance these extremes, we deterministically retain the top-β most probable branches to ensure exploitation, while randomly sampling additional

branches from the remaining candidates to encourage exploration. This hybrid strategy maintains diversity in the synthesized trajectories while preserving the model's focus on solving the underlying theorem-proving task.

Adaptive budget decay for easy problems (γ). The hyperparameter γ is introduced to adaptively reduce the search budget for problems that are relatively easy. In the STP dataset, certain problems can reach a proof-finished state within only a few steps. Unlike evaluation, where the search terminates once a single proof is found, data synthesis aims to collect multiple distinct proof-finished trajectories. Allocating a uniform search budget to all seed problems is therefore inefficient. To address this issue, whenever a new proof-finished path is discovered, the remaining search budget is multiplied by a decay factor $\gamma < 1$. As a result, easy problems that admit many completed proofs naturally exhaust their effective budgets more quickly, leading to earlier termination of the search. Smaller values of γ induce faster budget decay and more aggressive early stopping.

A.9.5 Analysis of the advantages of our method

During the data synthesis process, we observe that constrained decoding is necessary in a substantial number of cases, even though it may reduce the efficiency of distillation. A typical scenario is when the model employs the `have` tactic to add a new hypothesis, but the generated outputs differ only in naming. By enforcing the model to attempt different tactics, we largely mitigate this issue and introduce the diversity required to improve the cumulative success rate across independent search trees.

Driven by this observation, in this subsection, we analyze the sources of the advantages of our proposed method over EI from two perspectives: the data synthesis stage and the evaluation stage. For both stages, we collect statistics on the tree structure during tree search as well as the diversity of applied tactics. The results are summarized in Table 11 and Figure 5.

Data synthesis stage. During data synthesis, EI exhibits a smaller average number of children per node, as well as a smaller number of valid children, compared to our method. A primary reason is that among the multiple proof steps generated by the model, a substantial fraction are identical or lead to equivalent next states. In addition, EI consistently selects proof steps with higher confidence from the policy model, which results in a lower proportion of invalid children. Consequently, the tree search tends to concentrate on a limited number of paths that the policy model is already familiar with, leading to a restricted exploration space. From the perspective of tactic diversity, EI also shows limited diversity at each expansion step: for more than half of the nodes, all children expanded from the node use exactly the same tactic. In contrast, our method enforces a broader range of candidate tactics through constrained decoding, explicitly introducing diversity. This enables the data synthesis process to cover a wider variety of tactics and results in more diverse synthesized data.

Evaluation stage. During evaluation on dataset $\mathcal{B}$, our method yields a higher average number of valid children per node than EI, together with improved tactic diversity in the generated proof steps. At the same time, the proportion of invalid children remains comparable to that of EI. This indicates that during evaluation, the policy model is able to effectively increase diversity without introducing an excessive number of invalid branches, which ultimately leads to improved evaluation performance.

Table 11: Statistics of the average number of deduplicated children per tree node during the data synthesis and evaluation stages for EI and our method. The numbers in parentheses denote the counts of valid and invalid children respectively. Our method uses the setting $B = 32, \alpha = 0.25, \beta = 4$, under which the beam size after dropout is $32 \times 0.25 = 8$, aligning that of EI.

	EI	Our Method ($B = 32, \alpha = 0.25, \beta = 4$)
Average #children during data synthesis	2.51 (2.16 / 0.35)	4.12 (3.34 / 0.78)
Average #children during evaluation	3.82 (3.46 / 0.36)	4.85 (4.41 / 0.44)

A.10 Ablation study and further analysis of adaptive beam size strategy

We additionally evaluate the adaptive beam size strategy on MiniF2F using our model as the policy model. In this study, we vary the three hyperparameters associated with adaptive beam size and

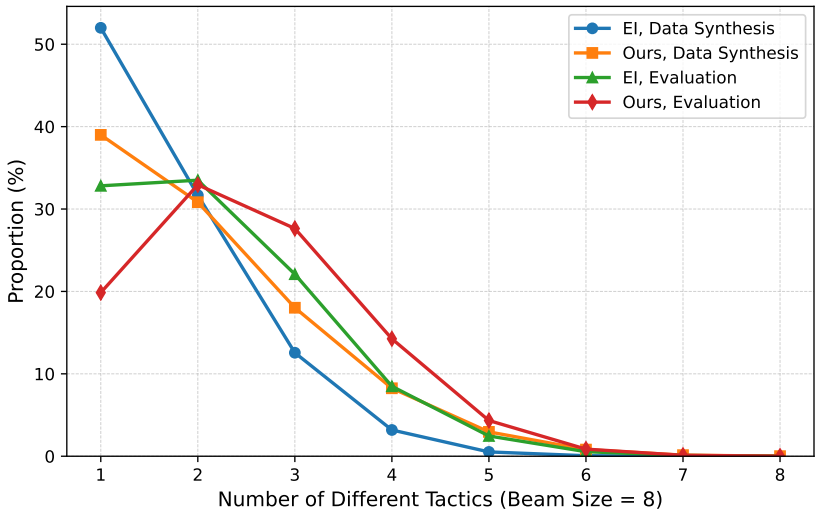


Figure 5: Tactic diversity statistics for EI and our method during the data synthesis and evaluation stages. The x-axis denotes the number of distinct tactics used in the proof steps generated by the policy model at each expansion; for example, a value of 1 indicates that all 8 generated proof steps use the same tactic. The y-axis represents the proportion of expansions of each level of tactic diversity.

evaluate each configuration twice. The results are summarized in Table 12. Overall, the results indicate that as long as B_{max} and B_{min} are chosen within a reasonable range, the performance is relatively robust to the specific choices of the three hyperparameters.

Table 12: Evaluation of adaptive beam-size decay hyperparameters on MiniF2F. Each configuration is run twice; we report the average Pass@1.

B_{max}	B_{min}	λ	Average Pass@1
16	4	15	60.86%
16	4	5	60.45%
16	4	30	60.04%
32	16	15	59.22%
32	4	15	60.04%
16	2	15	60.04%
8	2	15	59.22%

Based on these findings, we summarize the practical guidelines for selecting hyperparameters in the adaptive beam size strategy as follows:

B_{max}: Given a fixed evaluation benchmark, let B^* denote the optimal fixed beam size. We recommend setting B_{max} slightly larger than B^*, but generally not exceeding $2B^*$. This ensures sufficient exploration without unnecessarily enlarging the search space.

B_{min}: We recommend values of 2 or 4. Empirically, we observe no substantial difference between these choices. Because B_{min} governs the beam size in deeper layers of the search tree, using a relatively small value can slightly improve Pass@1 by encouraging deeper exploration.

A.11 Selection of proof states: valid vs. successful trajectories

In our data synthesis pipeline, we retain all "valid" proof transitions rather than restricting the training set to successful proof trajectories. This section clarifies the definition of valid states and provides the rationale and empirical justification for this design choice.

A.11.1 Definition and Rationale

A proof state is defined as **valid** if it does not trigger a Lean 4 kernel error during verification. This includes any intermediate state representing a syntactically correct proof step, regardless of whether it eventually leads to a complete proof. We incorporate all such states based on the following considerations:

Data Density and Volume: Restricting the dataset to successful trajectories via strict rejection sampling reduces the volume of training data by approximately 90%. Given that our pipeline performs full fine-tuning in a single iteration, maintaining a high-density, diverse sample size is critical for model convergence and generalization.

Supervision Signals: While some valid steps may be "goal-irrelevant," including them provides broader supervision signals that help the model learn to navigate the search space more effectively by understanding what constitutes a valid transformation within the formal system.

A.11.2 Empirical Evidence

To evaluate the impact of this choice, we conducted experiments comparing our "All Valid" approach with a "Successful Only" setting. As shown in Table 13, training exclusively on successful trajectories leads to a significant drop in both the total count of training samples and the final model performance (Pass@1).

Table 13: Comparison of training data volume and performance on the evaluation dataset. "Count" denotes the total number of synthesized training samples.

Synthesis Setting (B, α, β)	All Valid (Count / Pass@1)	Successful Only (Count / Pass@1)
$B = 8, \alpha = 1, \beta = 8$	736,635 / 59.7%	91,285 / 58.1%
$B = 16, \alpha = 0.5, \beta = 8$	699,966 / 60.3%	79,457 / 58.6%
$B = 32, \alpha = 0.25, \beta = 8$	498,950 / 60.9%	56,012 / 59.0%
$B = 32, \alpha = 0.25, \beta = 6$	505,131 / 60.5%	55,280 / 58.7%
$B = 32, \alpha = 0.25, \beta = 4$	504,123 / 61.2%	55,235 / 58.0%

The results confirm that the increased data volume and diversity provided by all valid proof states enable the model to achieve superior performance compared to training only on the narrow subset of successful trajectories.

A.12 Discussion on whole-proof generation paradigms

While this work focuses on the efficiency and quality of step-wise tree search, we provide a brief discussion to distinguish this approach from the *whole-proof generation* paradigm. From our perspective, although both share the goal of formal theorem proving, they represent fundamentally different methodologies with distinct evaluation protocols and computational characteristics.

In our view, a key conceptual difference lies in reasoning granularity. Whole-proof generation typically treats proving as a single-shot sequence generation task, which may lack real-time environment feedback. In contrast, our step-wise approach interacts with the Lean 4 kernel at each transition, enabling iterative error correction and state-aware tactic adjustments. Furthermore, while mathematical proofs often decompose into multiple sub-goals, tree search allows for incremental resolution and effective pruning of the search space. Whole-proof methods, by comparison, must serialize the entire proof into a single linear sequence, which can be particularly challenging for maintaining long-range logical consistency in 7B-scale models.

We compared the inference-time costs of these two paradigms to assess their practical efficiency. To obtain candidate quality comparable to tree-search methods, whole-proof generation often necessitates a large sampling budget k (e.g., 128 to 3200). In our tests using DeepSeek-Prover-V1.5 on 4×H800 GPUs, we observed that while a Pass@8 run takes approximately 1 hour, extrapolating this to Pass@128 suggests a significant increase in total compute time. In contrast, our tree-search framework completes a comparable workload in under 2 hours on the same hardware. These results

suggest that whole-proof generation does not necessarily offer a computational advantage when the goal is to achieve high-quality synthesis results at a realistic sample budget.

To provide a more intuitive context for comparison, we summarize the performance of representative 7B-scale whole-proof generation models on the MiniF2F-test benchmark in Table 14. It should be noted that we exclude models that primarily rely on knowledge distillation from much larger teacher models (such as 671B DeepSeek-Prover-V2), as they represent different data cost and compute scale.

Table 14: Performance comparison with representative 7B-scale whole-proof generation models on MiniF2F-test.

Model	Sample Budget (k)	Pass@k
DeepSeek-Prover-V1.5-SFT/RL	3200	53.3% / 54.9%
Goedel-Prover-SFT	3200	62.7%
Ours (7B)	$K = 1, E = 600$, Adaptive Strategy	60.7%

A.13 TACTIC USAGE STATISTICS ON THE STP DATASET

We performed a comprehensive scan of all tactics used across the entire STP dataset, identifying a total of 173 unique tactics. By applying a threshold of $\mathcal{P} = 0.999$ under the *top_p* strategy to exclude infrequently used tactics, we obtained a final set of 60 common tactics. The frequency distribution of these tactics is shown in Figure 6.

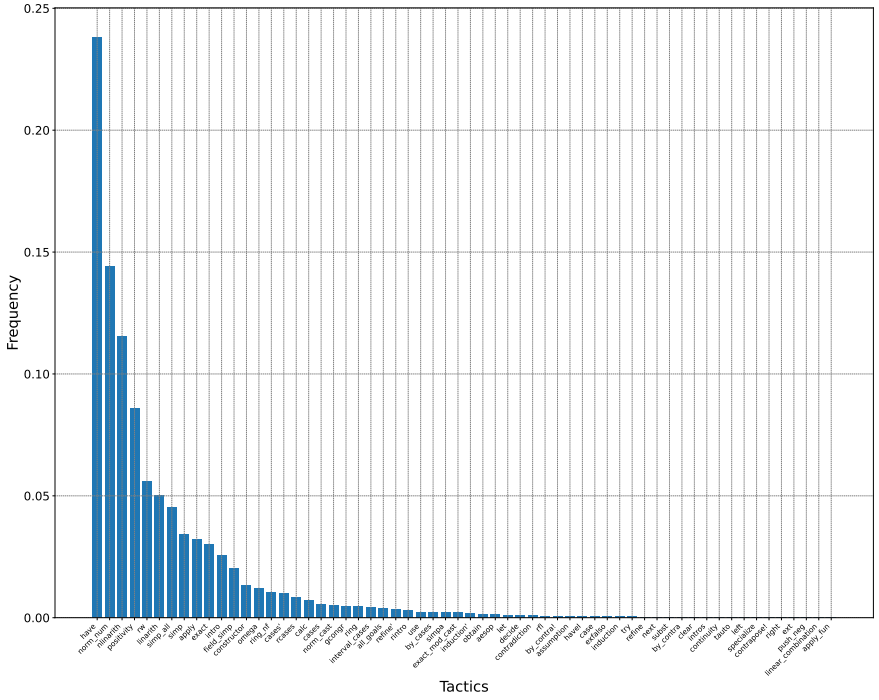


Figure 6: Distribution of the top 60 most frequent tactics in the STP dataset.

A.14 SEARCH DEPTH STATISTICS ON THE MINIF2F BENCHMARK

We conducted experiments using our policy model on the MiniF2F benchmark with varying beam sizes during search. The results, shown in Figure 7, indicate that under a fixed maximum number of expansions, smaller beam sizes tend to produce proving paths with greater search depth.

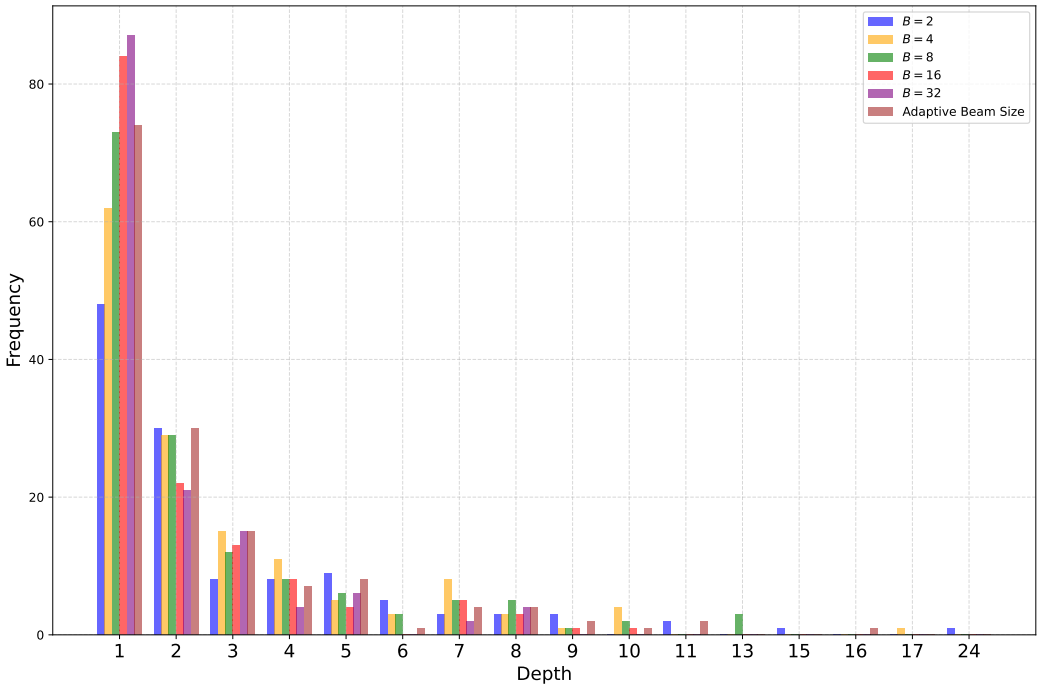


Figure 7: Frequency distribution of search depths of proving paths generated by our policy model on the MiniF2F benchmark, with the maximum number of expansions fixed at 600 ($E = 600$).