TREETHINK: A Modular Tree Search Library for Mathematical Reasoning with LLMs

Burak S. Akbudak[1], **Zeynel A. Ulusan**[2] , **Can S. Erer**[1], **Gözde Gül Şahin**[3, 4, 5]

[1] Computer Engineering Department, Bogazici University, Istanbul, Turkey
[2] Codeway Studios
[3] Friedrich-Alexander-Universität Erlangen-Nürnberg, Intelligent Language Systems
[4] Computer Engineering Department, Koç University, Istanbul, Turkey
[5] KUIS AI Lab, Istanbul, Turkey
https://gglab-ku.github.io/

Abstract

Tree search algorithms enable systematic exploration of the proof space in neural theorem proving. Existing LLM tree search libraries primarily target natural language reasoning and do not provide native integration with formal verifiers, while theorem proving systems often rely on task-specific search implementations. We introduce TREETHINK, an open-source Python library for modular, fully asynchronous tree search in neural theorem proving. It integrates established tree search methods with vLLM-based inference pipelines and diverse node evaluation techniques, ranging from lightweight heuristics to neural evaluators. We support Lean 4, Rocq, and Isabelle/HOL alongside natural language. It connects directly to each language's Read-Eval-Print Loop (REPL) server for real-time verification and proof state extraction. We evaluate TREETHINK on miniF2F and MATH500, demonstrating cross-language formal proof search, natural language reasoning support, and up to 6.3× wall-clock speedup from asynchronous execution. Source code is released under the MIT license at https://github.com/GGLAB-KU/treethink, and the library is accessible as a downloadable package at https://pypi.org/project/treethink/.

1 Introduction

Mathematical reasoning remains a fundamental challenge for artificial intelligence, spanning both informal and formal paradigms (Wang et al., 2026). While large language models (LLMs) have achieved impressive performance in informal mathematical reasoning, they remain susceptible to logical hallucinations and non-monotonic errors (Anh et al., 2025). This has motivated an increasing interest in formal theorem proving, where interactive theorem provers (ITPs) such as Lean (de Moura and Ullrich, 2021), Rocq (The Coq Dev Team, 2024), and Isabelle (Nipkow et al., 2002) verify mathematical statements. Neural theorem proving (NTP) combines these approaches by using LLMs to propose proof steps while relying on the ITP as a strict verification environment. However, NTP remains a highly difficult problem as the action space of valid mathematical tactics is vast. To guide models into successful trajectories, recent works integrate structured search algorithms into the inference process (Li et al., 2024). Fig. 1 illustrates a typical tree search loop: the search algorithm selects which partial proof to expand, an LLM proposes the next tactic, an evaluator scores nodes, and the proof assistant checks validity. Developing novel proof search systems often requires researchers to reimplement standard execution and verification components, resulting in substantial engineering overhead. For instance, to show the effectiveness of their proposed proof level reward evaluator, Wang et al., 2023a implement a tree search mechanism from the ground up, and Li et al., 2025 orchestrate a search system to evaluate the proposed process reward models. General LLM tree search libraries such as

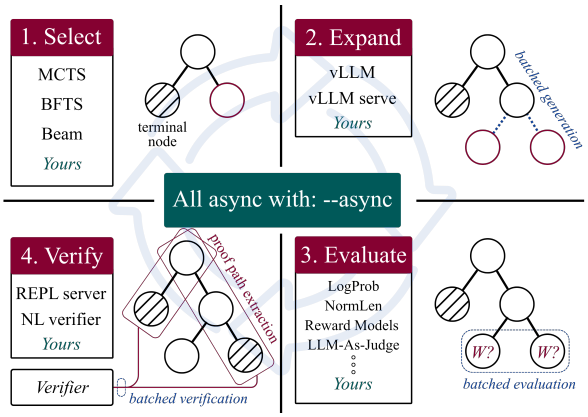

Figure 1: NTP tree search process. *1. Select:* search method selects a node using a search algorithm. *2. Expand*: the policy LLM generates child nodes. *3. Evaluate*: evaluator strategy scores the generated nodes. *W* stands for the value assigned to a node. *4. Verify*: external systems verify the correctness of the proof. Main operations in individual sections are in red while batched processes are in blue.

	Search Methods	#Policy	#Evaluators	Language	Formal Verifier	Async.&Batched	Proof Cache	Graph Vis.
FETCH	MCTS, BFS, Beam	1	1	NL	✗	✗	✗	✗
LLM Reasoners	MCTS, BFS, Beam, DFS, CoT, ToT	6	~	NL	✗	✗	✗	✓
LiTS	MCTS, BFS, Beam	5	~	NL	✗	✗	✗	✓
Ours	MCTS, BFS, Beam	2	8	Lean, Rocq, Isabelle, NL	✓	✓	✓	✓

Table 1: Comparison of the proposed framework with existing search libraries. NL stands for natural language. #Policy: number of supported LLM providers. #Evaluators: number of node evaluation strategies. Formal Verifier: integration capabilities with formal theorem provers. Proof Cache: presence of proof caching mechanisms. ~ symbol means there exists no implementation that would work in NTP scenario out of the box, and users are required to implement it.

LLM Reasoners (Hao et al., 2024), FETCH (Wang et al., 2025), and LiTS (Li and Tao, 2026) provide reusable search abstractions, however users must still implement formal proof verification, proof-state extraction, and domain-specific evaluators for NTP.

To reduce duplication of effort and accelerate experimentation, we present TREETHINK, a modular Python library for tree search in neural theorem proving. TREETHINK is designed around formal verifier interaction with natural language support. Our contributions are threefold: TREETHINK decouples search algorithms, LLM policies, evaluators, and environment interaction into reusable components; provides unified REPL clients for Lean 4, Rocq, and Isabelle/HOL; and supports fully asynchronous and batched execution with vLLM for high throughput inference (Kwon et al., 2023). We evaluate the system on miniF2F for formal mathematics (Zheng et al., 2021) and MATH500 for natural language mathematical reasoning (Lightman et al., 2024), showing support for both settings and up to 6.3× speedup over synchronous execution. Alongside the open source code[1], we release the framework as a downloadable package in PyPI[2], and provide a demo video[3].

2 Previous Systems

Recent efforts in LLM-based reasoning increasingly treat tree search as a reusable infrastructure by separating search logic from model inference (Wang et al., 2025; Hao et al., 2024; Li and Tao, 2026). These systems provide useful abstractions for natural-language reasoning, but they do not natively integrate formal proof verification, proof-state extraction, or multi-ITP execution. In formal reasoning, existing tools typically address individual parts of the pipeline: extract proof state information (Yang et al., 2023), execute proofs (Aniva et al., 2025), or support premise retrieval (Gao et al., 2026). Furthermore, recent formal math theorem-proving systems (Xin et al., 2025; Li et al., 2025; Xin et al., 2024; Wu et al., 2025) are designed as task-specific prover pipelines rather than reusable, multi-language tree-search libraries. Thus, existing systems either provide a general-purpose LLM tree search without formal verifier integration, or the provided infrastructure lacks reusable, multi-language, asynchronous tree-search abstractions. TREETHINK fills this gap by combining reusable tree-search components, batched LLM inference, heterogeneous evaluators, and unified REPL interaction across Lean, Rocq, and Isabelle/HOL. We compare our framework with other general tree search systems in Table 1.

3 TREETHINK

TREETHINK is designed around three principles: component interchangeability, verifier-agnostic environment interaction, and asynchronous execution. Component interchangeability is achieved by modularizing search methods §3.1, candidate-generating policies §3.2, and step evaluators §3.3. Verifier-agnostic interaction is managed by environment interactors §3.4, which standardize communication with formal proof checkers and informal verifiers. All components support asynchronous, batched execution. In addition, TREETHINK incorporates several features to accelerate experimentation §3.5. An overview is shown in Figure 2.

3.1 Search Algorithms

Search algorithms construct the proof tree by iteratively selecting and expanding nodes via a policy model, guided by leaf value estimates from eval-

[1] https://github.com/GGLAB-KU/treethink
[2] https://pypi.org/project/treethink/
[3] https://youtu.be/vKFXxnjlk8M

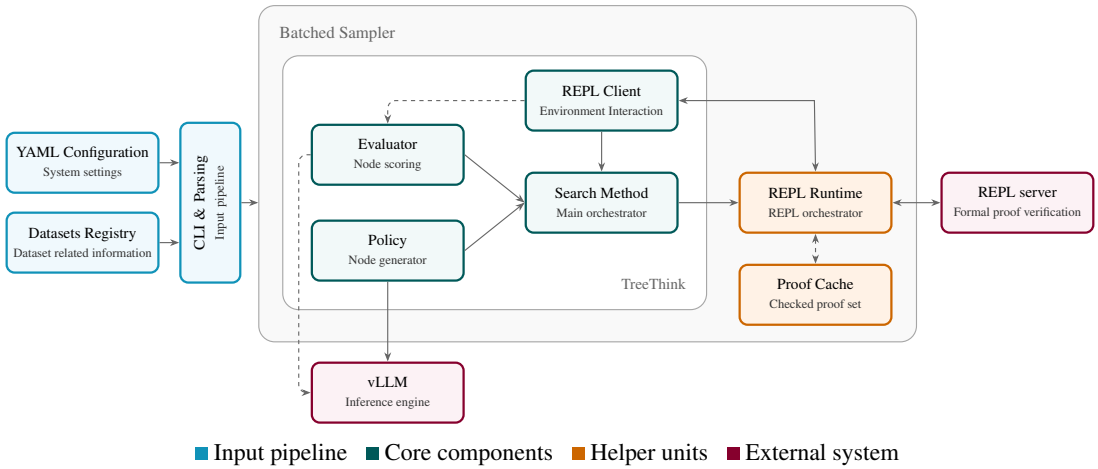

Figure 2: Overview of the system. Our framework accepts search configuration as a YAML file and a dataset registry. Then, we parse the given parameters and initiate search components. We embed the environment interaction logic into helper classes for modularity. Dashed lines denote optional relationships as some features may require the indicated connection under specific configurations (e.g., when the evaluator is selected as neural model).

uators. Each algorithm features an asynchronous counterpart to enable concurrent selection, expansion, and evaluation. We adapt four algorithms with varying exploration behavior and computational cost that have demonstrated effectiveness in recent NTP studies (Polu and Sutskever, 2020; Xin et al., 2025; Wu et al., 2025; Shao et al., 2024).

Best-First Search (Pearl, 1984) aims to prioritize the most promising frontier nodes according to evaluator scores. It does so by maintaining a priority queue of leaf nodes that are eligible for expansion and expanding the most promising node at each iteration:

$$s^* = \arg\max_{s \in \mathcal{F}} V(s)$$

where $\mathcal{F}$ is the frontier of leaf nodes and V(s) is the previously assigned evaluator score. The algorithm expands the selected node by generating candidate tactics and evaluates the generated states using the specified evaluation strategy.

Beam Search (Bisiani, 1987) focuses on pruning low-quality branches early. The algorithm expands the proof tree level-by-level, keeping the top-k most promising states at each depth:

$$\mathcal{B}_{d+1} = \text{top-}k \left(\bigcup_{s \in \mathcal{B}_d} \text{children}(s) \right)$$

where states are ranked by evaluator scores. At each level, we expand all beam states in parallel, generating multiple candidates per state. The top-k children across all expansions form the next beam, i.e., parent nodes do not survive.

Traditional Monte Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006) systematically explores search spaces by balancing exploration and exploitation. Each iteration involves selection, expansion, simulation, and backpropagation. During selection, the tree is traversed using the Upper Confidence Bounds for Trees (UCT) formula:

$$s^* = \arg\max_{s_i \in \text{child}(s)} \left[Q(s_i) + c\sqrt{\frac{\ln N(s)}{N(s_i)}} \right]$$

where $Q(s_i)$ is the mean value estimate of child s_i, N denotes visit counts, and c controls exploration. The policy expands the state into new child nodes, from which n terminal rollouts are generated. An evaluator scores these rollouts, assigning the mean value V to the child node. Finally, values are backpropagated up to the root via $Q(s) \leftarrow \frac{N(s) \cdot Q(s) + V}{N(s)+1}$.

Rollout-Free Value-Guided MCTS Traditional simulation is computationally expensive in NTP as each node requires terminal rollouts. Therefore, we implement a rollout-free variant inspired by AlphaZero (Silver et al., 2017), replacing simulation with direct value evaluation. The algorithm selects an expandable node using the aforementioned UCT rule and generates candidate proof steps. TREETHINK then directly evaluates these candidates to produce a value estimate, which is backpropagated to update ancestor nodes. Unlike AlphaZero, TREETHINK utilizes generic UCT

rather than PUCT with learned priors. This accommodates arbitrary pluggable evaluators better suiting the computational constraints of NTP.

3.2 Policies

Policies enlarge the search tree by generating child nodes from LLM outputs. TREETHINK supports local and remote inference via vLLM's `LLM` and `vllm serve` interfaces respectively. We use `AsyncLLM` for handling concurrent generation across search branches. When the policy is called, the model is prompted with the current proof trajectory starting from the root node to selected child node. Full model outputs are stored in each node, allowing heuristic evaluators such as `LogprobEvaluator` to leverage it. Output text is parsed in two modes: newline-delimited (default) and XML-tag-delimited. In newline-delimited mode, we give the newline token as the stop token to vLLM's sampling parameters. For XML mode, users specify a tag such as `<PROOF_STEP>`; we set the corresponding closing tag as the stop token and extract the inner content to construct nodes. Using XML tags allows LLMs to separate reasoning from proof steps independent of the selected search components. With this approach, models that produce more structured outputs can be natively integrated into our framework. Optionally, TREETHINK deduplicates child nodes by comparing their text before adding them to the search tree. This eliminates redundant candidates, conserving the expansion budget and ensuring each expansion yields a distinct exploration path.

3.3 Evaluators

Evaluators score generated nodes to determine the search algorithm's expansion priorities (§3.1). We implement adaptable approaches established in the literature. Asynchronous complements for heuristic evaluators wrap synchronous implementations, while neural reward systems leverage vLLM's asynchronous capabilities. Table 2 summarizes the supported evaluators detailed below:

`LogprobEvaluator` scores nodes by cumulative log-probability of the generated text within that individual node, favoring more likely reasoning steps. This technique is a fundamental approach used by GPT-f (Polu and Sutskever, 2020).

`NormLenEvaluator` pioneered by Xin et al., 2025, applies a length-normalized scoring function,

$$V(s) = \frac{\Sigma_{t=0}^{L-1} \log p(a_t|s_t)}{L^\alpha}$$

Evaluator	Signal Type	REPL?	RM?
`LogprobEvaluator`	Continuous	✗	✗
`NormLenEvaluator`	Continuous	✗	✗
`ProofLevelRewardEvaluator`	Discrete	✗	✓
`StateLevelRewardEvaluator`	Discrete	✗	✓
`JudgeEvaluator`	Continuous	✓	✓
`TournamentEvaluator`	Continuous	✓	✓
`RMaxTSEvaluator`	Continuous	✗	✗
`REPLEvaluator`	Binary	✓	✗

Table 2: Supported evaluators with their signal type, REPL requirement, and neural reward model (RM) usage. Due to the modular design, all evaluators are suitable for both informal and formal math problem solving.

where L is node depth, α controls the strength of the length penalty, s_t and a_t are proof step and applied tactic at time t respectively, and $p(a_t|s_t)$ the predicted probability of model generating the next tactic a_t at state s_t. Normalizing by L^α enables more fine-grained control over the shape of the tree, e.g., increasing α would lead the tree into deeper proof paths.

`ProofLevelRewardEvaluator` scores a node by evaluating the complete generated proof with a discriminative reward model via vLLM pooling mode. The pooling operation can be either "classify" for sequence reward models outputting a scalar value, or "token_classify" for process reward models (PRM) producing a per-token score, which can be reduced to a single scalar by taking the mean. The final score reflects how promising the partial proof is, judged in the context of the entire proof rather than a single step. Wang et al., 2023a use this approach in a formal proof search setting with their DT-Solver system.

`StateLevelRewardEvaluator` is similar to the proof-level variant but uses only the current node's text rather than the full proof, acting as a PRM. It judges the sensibility of an individual step independent of history. Zhang et al., 2024 uses a similar evaluator system in their self-training pipeline.

`JudgeEvaluator` utilizes a REPL to extract state information and a generative reward model (GRM) that assigns quality scores to proof steps. The model is prompted with current goals, applied tactic, and solved goals, and is asked to give a score on a fixed scale (e.g. 0–20). Given score is then parsed, and normalized to $[0, 1]$. We can see this evaluator in action in STILL (Jiang et al., 2024) framework, where authors use a GRM.

`TournamentEvaluator` ranks candidate proof steps using a single-elimination tournament. After extracting proof states via batched REPL calls, a separate LLM judge performs batched pairwise

comparisons. Candidates are scored based on their elimination round, with the overall winner receiving the total number of rounds. Normalizing these scores to $[0, 1]$ yields a continuous quality signal reflecting tournament advancement. Mahdavi et al., 2026 employ a similar pairwise strategy for proof-level selection.

`RMaxTSEvaluator` designed by Xin et al., 2024, assigns a novelty bonus to previously unseen proof states, which are tracked via SHA-256 hashing. When extrinsic rewards from proof verification are sparse, it gives the search a continuous incentive to explore new parts of the proof space.

`REPLEvaluator` returns binary type-check feedback by executing proof steps in a formal REPL. It is mainly intended for traditional MCTS rollouts where binary signals suffice.

3.4 Environment Interaction

We implement a unified client interface for communicating with formal proof checkers' REPL servers: i) Kimina Lean Server (Santos et al., 2025) for Lean 4 by adapting its existing sync/async clients, ii) isabelle-server (Nipkow et al., 2002) via isabelle-client (Koepke et al., 2022) for Isabelle, and iii) `rocq-ml-server` from rocq-ml-toolbox (Stoskopf, 2025) for Rocq. Both rocq-ml-server and isabelle-server accept proof paths rather than raw strings. Therefore, we provide proofs inside temporary files that are cleaned up after use, incurring a negligible I/O overhead. To batch the inputs and allow concurrency, we connect to servers via multiple clients.

TREETHINK supports three modes of formal-language REPL interaction: (1) evaluator-driven interactions, where evaluators query the REPL during the search process (e.g., JudgeEvaluator); (2) post-search verification, which validates completed proof candidates after search termination; and (3) online verification, which evaluates candidates at terminal nodes (e.g. those with triple ticks as their text) during search to enable early stopping. All verification methods support batched execution to balance accuracy against runtime overhead.

3.5 Additional Features

Beyond core components, TREETHINK includes features that improve search speed, enable visual inspection, and simplify usage.

Proof Caching An LRU cache keyed by SHA-256 hash of proof text prevents duplicate REPL submissions. Proof caching is especially important in tree search with NTP as branches often produce duplicate terminal proof candidates.

Graph Interaction Graphviz-based (Ellson et al., 2004) visualizations with auto-computed statistics facilitate visual debugging of search components. Additionally, search states can be saved and reloaded to resume interrupted experiments.

Configuration and Execution Dataset metadata is stored in TOML (Preston-Werner et al., 2021) files to ensure reproducibility, while experiments run via a unified command-line interface that accepts all search configurations, making the framework well-suited for HPC systems (see Appendices B and C).

4 Evaluation of the System

Our evaluation approach consists of three stages: cross-language formal proof search §4.1 where we illustrate our framework's capability of interacting with different ITPs, comparison between asynchronous and synchronous runs §4.2 to observe the gained speedups, and informal proof search with fixed-budget §4.3 to showcase the effectiveness of tree search. We demonstrate the system's modularity in Appendix D.1 by detailing the lines of code required to switch languages. Additionally, Appendix D.2 outlines the smoke tests used to validate approaches outside our main evaluation.

Our aim with these experiments is to demonstrate our framework's ability to act as a reusable tree-search system rather than a SOTA theorem proving system. Unless otherwise stated, the default experimental configuration uses a search budget of 512 expansions with 3 children per expansion, a vLLM policy backend, a NormLenEvaluator formal evaluator, the miniF2F test and MATH500 as formal and informal datasets respectively, and a single NVIDIA A40 GPU. We provide a comprehensive list of experiment parameters in Appendix E for reproducibility purposes.

4.1 Cross-Language Formal Proof Search

We evaluate our system on formal reasoning using the miniF2F dataset, which serves as an appropriate benchmark for cross-language evaluation, as it is available in all three formal languages that TreeThink supports. We use DeepSeekProverV2-7B (Ren et al., 2025) for Lean, Qwen2.5-Coder-14B-Instruct (Hui et al., 2024) for Isabelle, and

Method	Lean (%)	Isab. (%)	Rocq (%)
pass@1	49.4	42.2	0.8
BFTS	54.9	65.1	1.2
RF-MCTS	57.4	61.8	2.0

Table 3: Formal proof-search pass rates on miniF2F. RF-MCTS denotes our rollout-free value-guided MCTS variant. Tree-search methods use the same expansion budget, child budget, policy backend, and evaluator.

Qwen3.5-9B (Qwen Team, 2026) for Rocq for our runs. Since model availability differs across formal languages, we select a strong publicly available model for each language, using prior work when available (Hu et al., 2025).

Table 3 shows tree search consistently improves pass@1 rates across all three languages. RF-MCTS outperforms BFTS in Lean and Rocq, while the reverse holds for Isabelle, demonstrating the value of modular search methods. The overall performance in Rocq remains comparatively low, which we attribute to the limited availability of Rocq-specialized public models and prior benchmarks.

4.2 Asynchronous Execution

We evaluate the asynchronous features by varying the concurrency level, with pass rate and average wall-clock time per question results presented in Table 4. Across all runs, we maintain Isabelle as the formal language, and Qwen2.5-Coder-14B-Instruct as the policy model. We aim to demonstrate how concurrency affects the average solution time, rather than an improvement in accuracy. For demonstration purposes, we use a subset of miniF2F with 32 random samples.

Mode / Concur.	Pass (n=32)	Avg. Time / Q.	Speedup
Sync / 1	18	2.72m	x1.0
Async / 2	18	2.34m	x1.1
Async / 4	18	0.78m	x3.5
Async / 8	20	0.75m	x3.6
Async / 16	16	0.43m	x6.3

Table 4: Effect of asynchronous execution and concurrency level. All settings use the same model, evaluator, expansion and child budget. Avg. Time/Q. denotes average wall-clock time per question.

As shown in Table 4, asynchronous runs almost preserve pass rates while significantly reducing the wall-clock time. Specifically, we achieve x6.3 speedup when concurrency level is set to 16.

4.3 Informal Mathematical Reasoning

To demonstrate our framework's support for natural language mathematical reasoning, we use MATH500, a subset of the MATH (Hendrycks et al., 2021) dataset. It contains challenging competition-level problems stated in natural language. We select LLama3-8B (Grattafiori et al., 2024) as our policy model. For the evaluator, we use `JudgeEvaluator` with the judge model as RISE-Judge-Qwen2.5-7B (Yu et al., 2025).

To compare the models' answers with the ground truth (GT), we prompt the model to provide the final answer inside `\boxed{}`. After extracting the contents, we parse the text into expressions compatible with the symbolic mathematics library SymPy (Meurer et al., 2017) to verify equality between GT and the proposed solution.

Method	Pass (%)
pass@1	23.0
maj@12	34.8
RF-MCTS	40.0

Table 5: Natural-language mathematical reasoning results on MATH500. RF-MCTS indicates rollout-free value-guided MCTS with a judge-based evaluator.

We report pass rates for pass@1, majority voting@12 (Wang et al., 2023b) and our tree search run on MATH500 at Table 5. We choose maj@12, since RF-MCTS produces approximately 12 terminated paths per problem on average. Among tested methods, RF-MCTS achieves the highest score, suggesting that TreeThink's search abstraction can also benefit natural language mathematical reasoning. Moreover, this experiment demonstrates our framework's capability to model natural language reasoning without structural changes.

5 Conclusion

We present TREETHINK, a modular, open-source library for tree search in neural theorem proving. By decoupling infrastructure into reusable search algorithms, evaluators, and a vLLM-backed policy layer, it eliminates significant engineering overhead. Its fully asynchronous architecture yields substantial wall-clock speedups without accuracy loss. Furthermore, a unified REPL client enables identical search configurations across three ITPs. We also test the support for natural language reasoning, and achieve higher results compared to single-pass inference. We believe that TreeThink's extensible design simplifies the integration of new components, providing a practical foundation for future research in formal and informal reasoning.

Limitations

Evaluation strategy Our evaluation showcases the framework's capabilities; large-scale benchmarking across diverse datasets and model families is left for future work.

Inference provider support TREETHINK currently supports only vLLM-based inference (local or `vllm serve`). Other backends are not yet integrated, though the policy abstraction can accommodate them via subclassing.

Tool-assisted reasoning TREETHINK does not yet support interleaving proof steps with external tool calls (e.g., symbolic computation). Incorporating tool-use into the search loop is left as future work.

Domain scope The library is currently tailored to mathematical theorem proving. Extending the framework to other reasoning domains such as code generation, agentic task planning, or scientific discovery would require additional environment interactors and domain-specific evaluation strategies.

Search visualization While the framework provides Graphviz-based static visualizations of completed search trees, it does not support animated or interactive exploration of the search process (e.g., step-by-step playback or live tree expansion in a browser interface).

Component benchmarking TREETHINK supports multiple search methods and evaluators, but our main experiments evaluate only representative configurations. A full factorial benchmark over all search methods, evaluators, models, and proof assistants is computationally expensive and left for future work.

Broader Impact Statement

By allowing researchers to easily extend its capabilities with new search methods, evaluators, policies, or formal languages, TreeThink lowers the engineering barriers to automated formal reasoning research. It provides mathematicians and researchers in neural theorem proving with accessible, LLM-driven tools to explore complex proof spaces, aiding the advancement of mathematical understanding. Furthermore, the library's asynchronous and batched execution reduce the computational overhead and environmental footprint typically associated with large-scale reasoning tasks.

Acknowledgements

We gratefully acknowledge KUIS AI Lab for providing computational support.

References

Dang Hoang Anh, Vu Tran, and Le Minh Nguyen. 2025. Analyzing logical fallacies in large language models: A study on hallucination in mathematical reasoning. In *New Frontiers in Artificial Intelligence*, pages 179–195, Singapore. Springer Nature Singapore.

Leni Aniva, Chuyue Sun, Brando Miranda, Clark Barrett, and Sanmi Koyejo. 2025. Pantograph: A machine-to-machine interaction interface for advanced theorem proving, high level reasoning, and data extraction in lean 4. In *Tools and Algorithms for the Construction and Analysis of Systems: 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part I*, page 104–123, Berlin, Heidelberg. Springer-Verlag.

R. Bisiani. 1987. Beam search. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 56–58. John Wiley and Sons.

Leonardo de Moura and Sebastian Ullrich. 2021. 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*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer.

John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2004. *Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools*, pages 127–148. Springer Berlin Heidelberg, Berlin, Heidelberg.

Guoxiong Gao, Zeming Sun, Jiedong Jiang, Yutong Wang, Jingda Xu, Peihao Wu, Bryan Dai, and Bin Dong. 2026. Leansearch v2: Global premise retrieval for lean 4 theorem proving. *Preprint*, arXiv:2605.13137.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Shibo Hao, Yi Gu, Haotian Luo, Tianyang Liu, Xiyan Shao, Xinyuan Wang, Shuhua Xie, Haodi Ma, Adithya Samavedhi, Qiyue Gao, and 1 others. 2024. Llm reasoners: New evaluation, library, and analysis of step-by-step reasoning with large language models. *arXiv preprint arXiv:2404.05221*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Jilin Hu, Jianyu Zhang, Yongwang Zhao, and Talia Ringer. 2025. Hybridprover: Augmenting theorem proving with llm-driven proof synthesis and refinement. *Preprint*, arXiv:2505.15740.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Jinhao Jiang, Zhipeng Chen, Yingqian Min, Jie Chen, Xiaoxue Cheng, Jiapeng Wang, Yiru Tang, Haoxiang Sun, Jia Deng, Wayne Xin Zhao, Zheng Liu, Dong Yan, Jian Xie, Zhongyuan Wang, and Ji-Rong Wen. 2024. Enhancing llm reasoning with reward-guided tree search. *arXiv preprint arXiv:2411.11694*.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, page 282–293, Berlin, Heidelberg. Springer-Verlag.

Peter Koepke, Anton Lorenzen, and Boris Shminke. 2022. Cicm'22 system entries. In *Intelligent Computer Mathematics*, pages 344–348, Cham. Springer International Publishing.

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

Xinzhe Li and Yaguang Tao. 2026. LiTS: A modular framework for LLM tree search. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 47–56, San Diego, California, United States. Association for Computational Linguistics.

Yang Li, Dong Du, Linfeng Song, Chen Li, Weikang Wang, Tao Yang, and Haitao Mi. 2025. Hunyuanprover: A scalable data synthesis framework and guided tree search for automated theorem proving. *Preprint*, arXiv:2412.20735.

Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. 2024. A survey on deep learning for theorem proving. In *First Conference on Language Modeling*.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Sadegh Mahdavi, Branislav Kisacanin, Shubham Toshniwal, Wei Du, Ivan Moshkov, George Armstrong, Renjie Liao, Christos Thrampoulidis, and Igor Gitman. 2026. Scaling generative verifiers for natural language mathematical proof verification and selection. In *Forty-third International Conference on Machine Learning*.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, and 8 others. 2017. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103.

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

Judea Pearl. 1984. Heuristics - intelligent search strategies for computer problem solving. In *Addison-Wesley series in artificial intelligence*.

Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393.

Tom Preston-Werner and 1 others. 2021. TOML: Tom's obvious, minimal language. `https://toml.io/en/`.

Qwen Team. 2026. Qwen3.5: Towards native multimodal agents.

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

Marco Dos Santos, Haiming Wang, Hugues de Saxcé, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. 2025. Kimina lean server: Technical report. *Preprint*, arXiv:2504.21230.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, L. Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv*, abs/1712.01815.

Théo Stoskopf. 2025. rocq-ml-toolbox. https://github.com/LLM4Rocq/rocq-ml-toolbox.

The Coq Dev Team. 2024. The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman.

Ante Wang, Linfeng Song, Ye Tian, Dian Yu, Haitao Mi, Xiangyu Duan, Zhaopeng Tu, Jinsong Su, and Dong Yu. 2025. Don't getlost in the trees: Streamlining llm reasoning by overcoming tree search exploration pitfalls. *Preprint*, arXiv:2502.11183.

Haiming Wang, Ye Yuan, Zhengying Liu, Jianhao Shen, Yichun Yin, Jing Xiong, Enze Xie, Han Shi, Yujun Li, Lin Li, Jian Yin, Zhenguo Li, and Xiaodan Liang. 2023a. DT-solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12632–12646, Toronto, Canada. Association for Computational Linguistics.

Peng-Yuan Wang, Tian-Shuo Liu, Chenyang Wang, Ziniu Li, Yidi Wang, Shu Yan, Chengxing Jia, Xu-Hui Liu, Xinwei Chen, Jiacheng Xu, and Yang Yu. 2026. A survey on large language models for mathematical reasoning. *ACM Comput. Surv.*, 58(8).

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023b. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Zheng Yuan, Wenwei Zhang, Dahua Lin, and Kai Chen. 2025. InternLM2.5-stepprover: Advancing automated theorem proving via critic-guided search. In *2nd AI for Math Workshop @ ICML 2025*.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. 2024. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *CoRR*, abs/2408.08152.

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. 2025. BFS-prover: Scalable best-first tree search for LLM-based automatic theorem proving. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32588–32599, Vienna, Austria. Association for Computational Linguistics.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. Leandojo: theorem proving with retrieval-augmented language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Jiachen Yu, Shaoning Sun, Xiaohui Hu, Jiaxu Yan, Kaidong Yu, and Xuelong Li. 2025. Improve llm-as-a-judge ability as a general ability. *Preprint*, arXiv:2502.11689.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024. ReST-MCTS*: LLM self-training via process reward guided tree search. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

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

Appendix

A Full List of Controllable Parameters

We provide the full list of controllable parameters in our framework below. Note that, REPL client, model and sampling parameters are not given multiple times as they share the same dataclass.

```yaml
treethink:
  method_name: "BFTS"
  max_children: 3
  expansion_count: 512
  timeout: 480
  graph_path: /path/to/graph/output
  termination_str: "```"
  store_method_class: false
  store_graph_stats: true
  remove_duplicate_children: true
  parse_tag: "\n"

  # REPL / termination
  language: "lean"
  client_args:
    host: "localhost"
    port: "5000"
    batch_size: 8
    num_proc: 4
    timeout: 200
    enable_cache: true
    cache_maxsize: 4096
  termination_on_encounter:
      batch_size: 2
  termination_on_paths:
      max_repl: 16
  max_concurrent_expansions: 8

  # Special to Search Methods
  beam_width: 4
  exploration_weight: 1.414
  final_decision_mode: "native"
  tie_breaker: "random"

  # Rollout (TraditionalMCTS)
  rollout_evaluator_args: null
  rollout_max_tokens: 4096
  rollout_n: 1
  rollout_temperature: 1.0
  rollout_top_p: 0.9

policy:
  func_name: "vllm_policy"
  model:
    model: "deepseek-ai/DeepSeek-Prover-V2-7B"
    max_model_len:
    tensor_parallel_size: 1
    gpu_memory_utilization: 0.95
  sampling:
    max_tokens: 2048
    temperature: 1.0
    top_k: -1
    top_p: 0.95
    seed:
    stop: ["\n"]
    n: 3
    logprobs: 1
  server: # vLLM server mode
    base_url: "http://localhost:8000/v1"
    api_key: ""
    timeout: 600
  visible_devices: "0"

evaluator:
  func_name: "cumulative_logprob_evaluator"
  length_norm: 0.5 #
  client_args: # client args as before

  # for JudgeEvaluator
  llm_as_judge_model: # model args as before
  llm_as_judge_sampling: # sampling args as before
  llm_as_judge_system_prompt: "You are
  an LLM judge..."
  llm_as_judge_visible_devices: "1"

  # for Discriminative reward models
  reward_model: # model args as before
  reward_pooling_task: "classify"
  reward_score_reduction: "mean"
  reward_system_prompt: "Give reward based on..."
  reward_visible_devices: "0"
```

B Example Dataset Registry Entry

We implement a simple dataset registry for enhanced reproducibility in our experiments. An entry includes a name, dataset path, system and user prompts, what data key to use and its renamed counterparts for compatibility with the framework. An example dataset registry is given below:

```
[configs.minif2f_lean]
name="minif2f_lean"
path_or_name="/path/to/minif2f/lean/dataset"
dataset_split="test"
prompt_format="""Complete the following lean
code: \n```\n{}"""
system_prompt = "You are an expert in Lean 4
formal proof language."
data_keys=["formal_with_headers", "problem_id"]
renamed_data_keys=["problem", "problem_id"]
format_type="huggingface_disk"
```

C `treethink run` Command Example

Below is an example command for running a tree search using our framework:

```
treethink run \
  --gen-config-path "search_params.yaml" \
  --data-config-path "dataset_config.yaml" \
  --data-config-name "minif2f_lean" \
  --batch-size 8 \
  --num-iterations 1 \
  --output-dir "/output_folder/" \
  --run-name "minif2f_lean" \
  -v debug \
  --continue-from-prev \
  --async # auto-converts all components to
          # async implementations
```

Classification	Components
Language-Agnostic	Search method, Evaluator, Policy, Search&Child budget
Language-Specific	REPL client, Policy model, Prompt template

Table 6: Classification of TreeThink's components in terms of language dependency. Notably, REPL client is not choosable, as it is determined upon selecting the language.

D Further Discussion on System Evaluation

D.1 Modularity

As detailed in Section 3, the modular architecture of TREETHINK allows individual components to be swapped independently, facilitating highly reusable search processes. For example, migrating a tree search from Lean to Isabelle requires modifying only four lines in configuration files: language name, selected policy model, prompt and the dataset used. To clarify this separation of concerns, Table 6 categorizes all components as either language-dependent or language-agnostic.

D.2 Component Coverage

While we do not exhaustively evaluate every combination of search method, evaluator, and policy, our experiments focus on a representative subset that demonstrates the core functionality of our framework. To validate the remaining components, we employ smoke testing, i.e. successful end-to-end execution on a small held-out set shown in Table 7.

Scope	Covered Components
Main eval.	Best-first search, RF-MCTS, NormLenEvaluator, JudgeEvaluator
Smoke test	Best-first search, Beam search, Traditional MCTS, RF-MCTS, LogprobEvaluator, NormLenEvaluator, Reward evaluators, JudgeEvaluator, REPLEvaluator, TournamentEvaluator, RMaxTSEvaluator

Table 7: Coverage of TREETHINK components in our evaluation. Main experiments use representative configurations, while the remaining components are verified through integration smoke tests.

In our formal proof search experiments, we select `NormLenEvaluator` as our evaluator due to its lightweight and deterministic features. For informal language test, we use `JudgeEvaluator` to demonstrate our framework's ability to integrate reward models into the search process.

E Experiment Variables for Reproducibility Purposes

Parameter	Value
TreeThink commit hash	4887eb86a15f2b2bd5c4 cf2fa0bdd34502eac357
TreeThink PyPI package version	v0.1.0
vLLM version	v0.21.0
Lean/Rocq/Isabelle versions	v4.21.0 / 9.0 / 2025-2
REPL server versions	v1.0.1 / v0.1.0 / v0.3.5
Exact HuggingFace model IDs	deepseek-ai/DeepSeek-Prover-V2-7B, Qwen/Qwen2.5-Coder-14B-Instruct, Qwen/Qwen3.5-9B
Exact HuggingFace dataset IDs	HaimingW/miniF2F-lean4 / LLM4Rocq/miniF2F-rocq / erer-can/minif2f-isabelle
Sampling parameters	temperature: 0.3, top_k: -1, top_p: 0.95
Timeout behavior	Max. 480s for a single proof search. Max. 120s for REPL .
Hardware (CPU/RAM/GPU)	AMD EPYC 9224 24-Core Processor / 32 GB RAM / Nvidia A40
Is averaged over multiple runs?	No, single runs were performed.

Table 8: Experiment details: version numbers, HuggingFace model/dataset IDs, sampling parameters, timeout behavior, hardware and whether the experiments are averaged over multiple runs or not.

The list of experiment variables can be found in Table 8. For the datasets we use, we construct columns containing both the header and the problem statement. To incentivize proof generation by the model, we modify the target language syntax: replacing `sorry` with `by` in Lean 4 and with `proof` in Isabelle, while appending the `Proof.` keyword to Rocq statements.