

DT-Solver: Automated Theorem Proving with Dynamic-Tree Sampling Guided by Proof-level Value Function

Haiming Wang^{1*}, Ye Yuan², Zhengying Liu³ Jianhao Shen², Yichun Yin³, Jing Xiong¹, Enze Xie³, Han Shi³, Yujun Li³, Lin Li³, Jian Yin^{1†}, Zhenguo Li³, Xiaodan Liang^{1,4†}

¹Sun Yat-sen University, ²Peking University, ³Huawei Noah's Ark Lab, ⁴MBZUAI
{wanghm39,xiongj69}@mail2.sysu.edu.cn, {yuanye_pku,jhshen}@pku.edu.cn,
{liuzhengying2,yinyichun,xie.enze,shi.han}@huawei.com
{liyujun9,lilin29,Li.Zhenguo}@huawei.com
issjyin@mail.sysu.edu.cn, xdliang328@gmail.com

Abstract

Recent advances in neural theorem-proving resort to large language models and tree searches. When proving a theorem, a language model advises single-step actions based on the current proving state and the tree search finds a sequence of correct steps using actions given by the language model. However, prior works often conduct constant computation efforts for each proving state while ignoring that the hard states often need more exploration than easy states. Moreover, they evaluate and guide the proof search solely depending on the current proof state instead of considering the whole proof trajectory as human reasoning does. Here, to accommodate general theorems, we propose a novel Dynamic-Tree Driven Theorem Solver (**DT-Solver**) by guiding the search procedure with state confidence and proof-level values. Specifically, DT-Solver introduces a dynamic-tree Monte-Carlo search algorithm, which dynamically allocates computing budgets for different state confidences, guided by a new proof-level value function to discover proof states that require substantial exploration. Experiments on two popular theorem-proving datasets, PISA and Mathlib, show significant performance gains by our DT-Solver over the state-of-the-art approaches, with a 6.65% improvement on average in terms of success rate. And especially under low computing resource settings (11.03% improvement on average).

1 Introduction

Automated theorem proving (ATP) (Harrison et al., 2014) has been considered an essential task of Artificial Intelligence (AI) ever since the birth of modern AI (McCarthy et al., 2006). Besides its remarkable theoretical value and huge potential to accelerate research in mathematics, ATP already demonstrates excellent application value in, for ex-

[†] Corresponding authors.

^{*} Work done during internship in Huawei Noah's Ark Lab

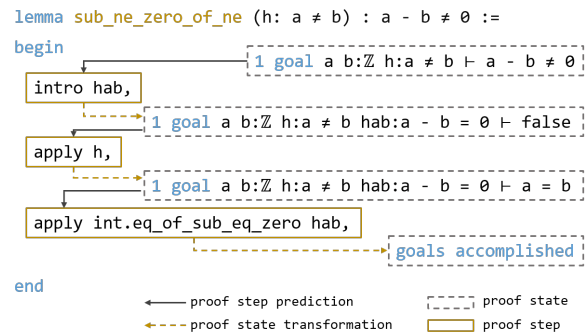


Figure 1: Illustration of a theorem and its proof in Lean (de Moura et al., 2015). The theorem `sub_ne_zero_of_ne` states that if the hypothesis `h` holds (i.e. we have $a \neq b$), then we have $a - b \neq 0$. Three proof steps are used to prove this theorem. After the keyword `begin`, a proof state (containing one or several goals) is initialized. Then each proof step is applied to the current proof state to obtain a new one until a proof state ‘goals accomplished’ is reached, which marks the success of the proof.

ample, formal verification (Barras et al., 1997) and code generation (Howard, 1980).

In general, the goal of a theorem proving task is to construct a *proof* (a sequence of proof steps) that *proves* a given theorem (often within a given time budget, e.g. 300 seconds). The validity of this proof is verified by a *formal environment* in which the theorem and the proof are formalized. We show in Fig. 1 an example of a theorem proven in Lean (de Moura et al., 2015), a formal environment we use in this work. Sec.2.1 gives more details on the formal environment.

To deal with this theorem proving problem automatically, many task-specific algorithms have been proposed and are later implemented as state-of-the-art ATP solvers such as Z3 (de Moura and Bjørner, 2008), Vampire (Kovács and Voronkov, 2013), E (Schulz, 2002) and Zipperposition (Bentkamp et al., 2021). These approaches are task-specific in that they are specific to one given formal environment and rely on solid domain knowledge, symbolic operations, and usually human heuris-

tics. With the exciting development of deep learning (LeCun et al., 2015) over the past decade, approaches (not necessarily for ATP) in the *task-agnostic* paradigm have been proposed, such as AlphaZero (Silver et al., 2017) and more recently Gato (Reed et al., 2022). As for ATP, task-agnostic approaches are also proposed. For example, GPT-f (Polu and Sutskever, 2020) uses a language model to suggest a set of candidates for the next proof step and then search for a complete valid proof within the formal system Metamath (MEGILL and DAVID A, 2019). In contrast to classic ATP algorithms, GPT-f has remarkable generality and can be applied to any ATP tasks in any formal system, such as propositional logic, first-order logic, higher-order logic, typed lambda calculi, etc. With GPT-f, domain knowledge and human heuristics are no more absolutely required. Some works following the workflow of GPT-f are introduced in A.2.

Although approaches such as GPT-f demonstrate impressive generality and performance, they still have two significant drawbacks. **On the one hand**, GPT-f consumes considerable computational resources. According to (Lample et al., 2022), the original GPT-f requires 2000 GPU days (A100) for training, and their approach HTPS (HyperTree Proof Search (Lample et al., 2022)) also consumes more than 1000 GPU days for one training process, which sets a prohibitive threshold for most researchers. Even worse, the cost for inference with GPT models and search can be 5 to 10 times heavier than that of training. **On the other hand**, the search process in GPT-f could reach a state of ‘empty queue’, where all proof step candidates (i.e. all leaf nodes in the search tree) generated by the language model turn out to be inapplicable. This can typically take place within the time budget and an premature ‘Failed’ result is returned, potentially resulting in a low *pass rate* (sometimes we also use the term ‘one-pass success rate’ to be more specific). Also, prior works (Polu and Sutskever, 2020; Han et al., 2021; Polu et al., 2022) often conduct constant computation efforts for each proving state while ignoring that the hard states often need more exploration than easy states. Moreover, they evaluate and guide the proof search solely depending on the current proof state instead of considering the whole proof trajectory as human reasoning does.

In this work, we address the above two issues and propose a novel **Dynamic-Tree** driven Theorem **Solver** (DT-Solver). As an automated theorem-

proving algorithm, DT-Solver uses *dynamic-tree sampling* guided by a *proof-level value function*. We illustrate these two main components as follows.

- **Dynamic-tree sampling.** To remedy the issue of premature failure with ‘empty queue’, we allow the nodes in the search tree to be expanded (i.e. generate child nodes using proof steps predicted by the language model) *several times*, instead of being expanded just once (as proposed by GPT-f). To achieve this, we add an imaginary node or a *virtual node* to each node in the search tree. When this virtual node is selected, no specific *proof step* is applied. Instead, we use the language model again to generate possible *proof steps* and potentially produce new and promising child nodes (proof states). Thus the width of the search tree can be adjusted *dynamically*. Furthermore, we modify the usual PUCT score in the Monte-Carlo tree search to ensure that the computational resources focus on promising but possibly ‘hard’ nodes. We find this dynamic sampling technique helpful and obtain an increase of 39.9% -> 48.4% in terms of the pass rate.
- **Proof-level value function.** Different from GPT-f (Polu and Sutskever, 2020) and HTPS (Lample et al., 2022), which use step-level value function to guide the search, we train another encoder-only transformer, RoBERTa (Liu et al., 2019) to predict whether a new *proof state* is on the right track to finishing the proof. As we consider the whole proof instead of the current proof step, this RoBERTa model is actually a *proof-level* value function. We find that this technique can help increase the value function’s accuracy, for instance, from 63.6% to 70.7%.

2 Background

2.1 Formal mathematics environments

Following (Han et al., 2021; Polu et al., 2022) and (Jiang et al., 2021, 2022), we choose Lean (de Moura et al., 2015) and Isabelle (Paulson, 1994) as our formal environments. As illustrated in Fig. 1, the theorem-proving process in Lean (and similarly in Isabelle) consists of sequentially applying a proof step (also known as a *tactic*) to a given theorem statement. Specifically,

the formal environment first constructs an initial proof state from the theorem statement, which contains one or multiple goals to be proved. Seeking to solve all goals in the proof state, the user produces a proof step that applies an existing theorem (apply int.eq_of_sub_eq_zero), introduces some new assumptions (intro hap), or tells the formal environment to use techniques like proof by contradiction or mathematical induction. If the produced proof step is applicable, the formal environment applies the proof step and transforms the original proof state into a new proof state. This process is repeated until the proof state reaches ‘no goals’ (or ‘goals accomplished’ as illustrated), which means all goals in the theorem are proven.

2.2 Language model guided theorem proving

Recent approaches that use language models to solve the ATP problem mostly follow the work from GPT-f. Given the current proof state, a causal language model (usually a decoder-only transformer like GPT (Radford et al.)) is used to predict possible proof steps that can be applied. Concretely, the language model is trained on sequences in the following form using a language modeling objective:

GOAL \$(proof state) PROOFSTEP \$(proof step) where \$(\cdot)\$ is the placeholder for the actual proof state and proof step. At test time, multiple new proof steps are sampled from the language model giving a prompt as follows:

GOAL \$(proof state) PROOFSTEP

We denote this atomic operation as expansion. To construct a complete proof for a given theorem, GPT-f adopts the best-first search algorithm that iterates among selecting a best-scoring proof state from the priority queue to expand, doing expansion using the language model with the selected proof state, and adding new states to the priority queue. Each state’s score is either calculated by a value function or uses the prior probability of the preceding proof step. Iterations continue until GPT-f finds the theorem’s proof or reaches the limit of computational budgets.

3 Methodology

Starting from the initial state of the theorem, DT-Solver adapts the language model to suggest forward-moving actions and empowers dynamic-tree sampling guided by a proof-level value function to find the complete proof path. This is ac-

complished by a three-step pipeline: (1) Following procedures described in Sec. 2.2, DT-Solver trains a language model using supervised data extracted from formal mathematics libraries to predict proof steps based on proof states. The trained language model is called the *policy model*. (2) Subsequently, dynamic-tree sampling uses the trained policy model to generate steps and searches for theorems proofs in the training set. A data collection procedure collects proof trajectories from successful proof searches. (3) Lastly, DT-Solver trains the proof-level value function (called the *critic model*) to identify promising proof paths from astray ones on collected data with a classification loss. At test time, to evaluate the performance of our method, we use the trained policy model and critic model to perform dynamic-tree sampling on the test set theorems.

In the following section, we first introduce the dynamic-tree sampling algorithm and the data collection procedure (Sec.3.1). After that, we introduce our proof-level value function (Sec. 3.2).

3.1 Dynamic-tree sampling

The dynamic-tree sampling algorithm strives to maximize the efficiency of the search procedure under limited computational budgets. In practice, it controls the exploration of different proof states according to two criteria: state value and model confidence. The state value estimates whether the state is on the correct proof path. Only high-valued states deserve more exploration. The model confidence is calculated with the prior probabilities of the current state’s proof step. A state is considered to require more exploration if the policy model cannot produce confident steps toward the current state.

To achieve the above objectives, we build the dynamic-tree sampling based on the Monte-Carlo tree search algorithm. As shown in Fig. 2, dynamic tree sampling progressively adds new proof states to construct a proof tree. Specifically, each node in the proof tree represents a proof state denoted as $s_i \in \{s_0, s_1, \dots, s_M\}$, where M is the total number of tree nodes. Each edge represents a proof step denoted as $a_j \in \{a_0, a_1, \dots, a_{M-1}\}$, where $M - 1$ is the total number of edges. For notational clarity, the edge from s_i to s_j is always denoted as a_j , using the same subscript as s_j . Similar to the classic Monte-Carlo tree search algorithm, every node-edge pair in the proof tree has a visit count

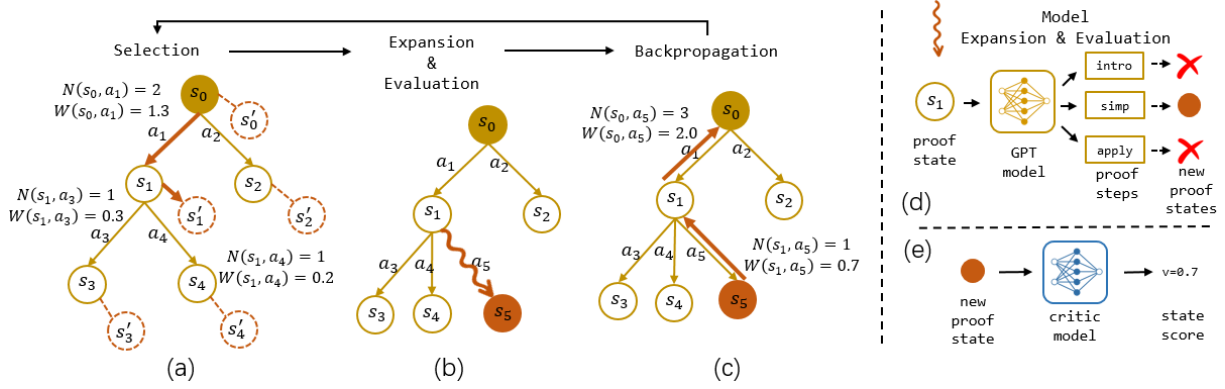


Figure 2: Overview of DT-Solver. The algorithm consists of 3 phases: Selection, Expansion & Evaluation, and Backpropagation. (a) In the Selection phase, the child node (or a virtual node) with the highest PUCT score is selected for expansion. (b) The phase Expansion & Evaluation first expands the selected virtual node’s parent (detailed in (d)) and then evaluates the generated new state with our proof-level value function (detailed in (e)). (c) The Backpropagation phase backpropagates the new state score to the root state by successively adding the score to the parent’s value count W and accumulating visit count N . (d) Detailed Model expansion procedure. The trained policy model (GPT) takes the selected proof state as input and sample e proof steps. We then apply the generated proof steps and use a formal environment to produce new proof states if successful. (e) State evaluation with proof-level value function. Specifically, the critic model takes the current state and auxiliary information (complete proof path, initial proof state) as input and outputs a score for the current state.

denoted as $N(s_i, a_j)$ and a value count denoted as $W(s_i, a_j)$. Starting from the root node, dynamic-tree sampling repeats the following three steps until the theorem’s proof is found or the computational budget is exhausted. (1) The selection phase (Fig. 2(a)). Dynamic-tree sampling computes the PUCT score for child nodes and the virtual child node. We select the highest-scoring child node and proceed according to the following conditions: if a non-virtual child node is selected, dynamic-tree sampling continues the selection phrase from the selected child node. If a virtual node is selected, the selection phase ends and proceeds to expansion & evaluation. (2) The expansion & evaluation phase (Fig. 2(b)(d)(e)). Dynamic-tree sampling performs expansion on the parent node of the selected virtual node. A fixed number (denoted as e) of proof steps are sampled from the trained policy model. The formal environment verifies proof steps and produces new proof states, which are then deduplicated and evaluated by the critic model, and finally added to the tree. (3) The backpropagation phase (Fig. 2(c)). Dynamic-tree sampling backpropagates new state scores to the root state by successively adding the score to the parent’s value count W and accumulating visit count N .

We detailed the selection and backpropagation phrases in the following. The expansion phase is well discussed and we leave the details about evaluating states’ value in Sec. 3.2

Selection. We denote the current node to perform selection as s_t , and its children as $s_c \in C(s_t)$ where $C(\cdot)$ is the function that returns all the children for a given node. The virtual child of s_t is denoted as s'_t . The PUCT score for each child node $s_c \in C(s_t)$ is formulated as follow:

$$\text{PUCT}_{s_c} = \frac{W(s_t, a_c)}{N(s_t, a_c)} + c \cdot p(a_c|s_t) \cdot \frac{\sqrt{N(s_t, \cdot)}}{N(s_t, a_c)} \quad (1)$$

where c is a constant balancing the exploration and exploitation trade-off, and $p(a_c|s_t)$ is the probability (estimated by the language model) of generating the proof step a_c given current state s_t . The PUCT score for the virtual child node s'_t is formulated as follow:

$$\text{PUCT}_{s'_t} = \text{val}_{s'_t} + c \cdot \text{conf}_{s'_t} \cdot \frac{\sqrt{N(s_t, \cdot)}}{|C(s_t)|} \quad (2)$$

$$\text{val}_{s'_t} = 1 - \max_{s_c \in C(s_t)} \frac{W(s_t, a_c)}{N(s_t, a_c)} \quad (3)$$

$$\text{conf}_{s'_t} = 1 - \sum_{s_c \in C(s_t)} p(a_c|s_t) \quad (4)$$

where $|C(s_t)|$ is the number of children s_t have. Two parts control the score of selecting the virtual node. $\text{val}_{s'_t}$ estimates the value of the virtual node. When s_t finds high-valued promising children, $\text{val}_{s'_t}$ will have a low score, indicating that

no more exploration is required on s_t . $conf_{s_t}$ estimates the model confidence on s_t . The value of $conf_{s_t}$ remains high until confident proof steps are generated. The model confidence is discounted by the number of children s_t have and keeps annealing with more exploration performed on s_t . In the selection phase, we repeatedly select the child node with the highest PUCT score and continue until we reach a virtual node. The selection process empowers dynamic-tree sampling by being able to backtrack to previously expanded states, eliminating the "empty queue" failure and drastically increasing DT-Solver's stability of finding valid proof.

Backpropagation. The backpropagation follows the classical Monte-Carlo tree search. We denote the function that returns the node's parent as $P(\cdot)$. The value counts $W(P(s_c), a_c)$ and visit counts $N(P(s_c), a_c)$ are both initialized as 0 for newly add child node. Given the newly added leaf state s_c and its estimated score v_c , dynamic-tree sampling repeats the following steps until the root node is reached: (1) Accumulate the visit count $N(P(s_c), a_c) += 1$, and accumulate the value count $W(P(s_c), a_c) += v_c$ (2) Traverse back the tree by changing the current node to the parent $s_c \leftarrow P(s_c)$.

Data collection. To construct a proof-level value function capable of identifying promising states from astray ones, DT-Solver collects supervised training data by performing dynamic-tree sampling in training set theorems¹. Specifically, DT-Solver collects trajectories data in the form of $[(s_0, a_0), (s_1, a_1), \dots, (s_l, a_l)], y$, where s_0 is the root state and s_l is the leaf state. The label $y = 1$ if the trajectory correctly proves the theorem and $y = 0$ otherwise. The same data collection procedure is performed on the validation and test theorems for testing the value function.

3.2 Proof-level value function

The performance and efficiency of the dynamic-tree sampling algorithm depend heavily on correctly evaluated state values. Existing methods like GPT-f only use a representation of the current state to assess the state's quality. However, according to our close-up observation, states along correct proof paths tend to be more similar and consistent, while states along false proof paths exhibit more diversity.

¹At this stage, the proof-level value function is not applicable; thus the state's preceding step's prior probability is used as value estimate v .

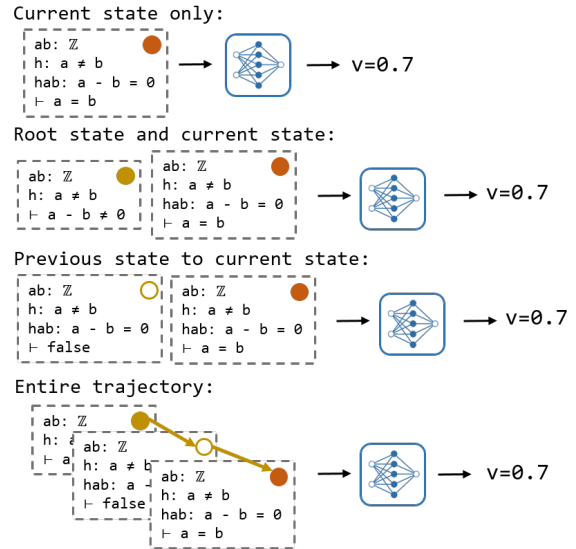


Figure 3: Four types of proof-level value functions. In practice, we use Roberta as our classifier to output the probability of being in a good state.

Thus, it is beneficial to use proof-level information to estimate the value of a state.

Accordingly, shown in Fig. 3, we propose four types of proof-level value functions. Current-state-only value function construct supervised data formatted as (s_l, y) from the collected trajectory data. A Roberta language model is fine-tuned on the current-state-only data with a classification loss. Root-state-and-current-state value function constructs sentence pair data formatted as $([s_0, s_l], y)$ from the collected trajectory. The two states are concatenated and fed into the Roberta to predict the state's value. Conversely, previous-state-to-current-state uses more recent state information and formats the training data as $([s_{l-1}, s_l], y)$. Entire-trajectory value function concatenates the entire proof path as follows:

GOAL $\$(s_0)$ PROOFSTEP $\$(a_0)$ </s></s> GOAL $\$(s_1)$ PROOFSTEP $\$(a_1)$...

where the </s></s> separate different state-action pair. Empirically, the entire-trajectory value function performs the best in the value function test set, but slightly worse than the Root-state-and-current-state value function in end-to-end proof successful rate.

4 Experiments

4.1 Experimental setup

Implementation details. In this paper, we adopt the same model setup from PACT (Han et al.,

Table 1: Performance comparison between our DT-Solver and baselines. Columns three and four show the one-pass success rate of the theorems proven in the test set. The low-resources setting uses $e = 8$ for one expansion, and the high-resources setting uses $e = 32$ for one expansion. On-Track Rate (OTR) averages the rate of selecting the correct nodes to expand in each proof search. Success On Track Rate (SOTR) only calculates the OTR of successful proof searches. Time shows the average time used per proof search. The values expressed in ‘X/Y’ format illustrate the outcome for low-resource and high-resource settings, and the up arrow symbol signifies these values are higher the superior. The best results without using the expert iteration are underlined.

Dataset	Methods	low. \uparrow	high. \uparrow	OTR \uparrow	SOTR	Time (s)	STime (s)
PISA	Lisa [†] (Jiang et al., 2021)	16.8	27.3	15.2 / 23.1	57.0 / 47.4	58.6 / 229.4	26.4 / 108.7
	Lisa+MCTS	13.3	21.1	11.7 / 17.3	53.3 / 47.2	37.8 / 209.3	24.2 / 51.3
	Lisa+DT-Solver	26.7	27.6	19.7 / 20.6	53.0 / 44.5	176.5 / 229.1	45.8 / 88.7
mathlib	GPT-f [†] (Polu and Sutskever, 2020)	12.6	23.4	11.9 / 19.2	61.3 / 48.2	7.6 / 77.6	15.1 / 83.2
	PACT [†] (Han et al., 2021)	22.6	35.1	20.2 / 30.1	54.4 / 53.5	23.1 / 165.4	29.1 / 98.6
	Expert iter. [†] (Polu et al., 2022)	39.9	45.9	<u>35.5</u> / 40.5	53.6 / 50.8	61.4 / 149.2	34.1 / 87.3
	PACT+MCTS	23.2	36.1	20.5 / 29.6	48.6 / 45.0	32.8 / 170.1	61.3 / 115.4
	Expert iter.+MCTS	40.9	47.1	35.4 / 40.2	48.1 / 46.1	75.3 / 150.8	64.3 / 101.2
	PACT+DT-Solver	<u>37.3</u>	<u>39.3</u>	<u>28.3</u> / <u>31.7</u>	43.0 / 42.4	203.9 / 204.3	95.5 / 136.6
	Expert iter.+DT-Solver	48.4	48.2	39.5 / 40.2	45.5 / 45.6	167.7 / 175.2	75.6 / 103.9

2021)² in Lean formal environment and Thor(Jiang et al., 2022) in Isabelle formal environment for better comparison. Detailed model configuration and training procedure are described at Sec. A.3 in Appendix.

Dataset. To validate our proposed DT-Solver, we choose the Mathlib dataset in the Lean formal environment and the PISA dataset constructed from Archive of Formal Proofs (AFP) in the Isabelle formal environment. Although both datasets are extracted from the world’s largest formal mathematical libraries, these two datasets have very different characteristics. Mathlib’s proof steps are instructional, such as applying a theorem or simplifying a goal. Contrary, Isabelle’s proof steps are more human-friendly and declarative. The proof step is usually led by a conjecture to proof and followed by short instructions to prove it. Empirically, it is much harder for the model to suggest a correct conjecture. Detailed statics regarding the training dataset are shown in Table.4 and Table.5 in Appendix.

Baseline methods. For the Isabelle formal environment, we compare our DT-Solver with Lisa (Jiang et al., 2021)³, the first work that applies the GPT-f model to the Isabelle environment. In addition, Lisa proposes to add previous proof context to the original proof state to help the lan-

guage model predict the following proof states. For the Lean formal environment, besides the original GPT-f model (Polu and Sutskever, 2020), we compare our DT-Solver with PACT (Han et al., 2021) and Expert iteration (Polu et al., 2022) PACT (Proof Artifact Co-training) built upon the GPT-f model and proposed to co-train the language model with nine auxiliary tasks. Expert iteration bootstraps the language model by training the model with self-generated data from proof searches. All the baseline methods described above use the best-first search algorithm described in Sec. 2.2.

DT-Solver follows previous work to construct a policy model. For fair comparisons, we use the same policy models from baseline methods. Specifically, three policy models are trained: Lisa, PACT, and Expert iter. We evaluate our DT-Solver by substituting the best-first search with dynamic-tree sampling and entire-trajectory proof-level value function. For ablation, we substitute dynamic-tree sampling with classic Monte-Carlo tree search but leave the policy and critic models unchanged. All the baseline methods are re-implemented by ourselves since none of them releases the code.

4.2 Main results

4.2.1 Comparison with state-of-the-art

As shown in Table 1, our proposed DT-Solver dramatically outperforms all the baseline methods under all scenarios. Specifically, Expert iter.+DT-Solver improves from 39.9% to 48.4% in the low-resource setting, surpassing the model performance of 45.9% for the Expert iter. method in high-

²Unfortunately, we cannot use models with the same pre-training configuration since the data for pre-training have not been released.

³We leave the comparison of Thor (Jiang et al., 2022) in the future work since the use of the Sledgehammer is compatible with our DT-solver and is a guaranteed improvement.

Table 2: Performance comparison among different value functions on mathlib dataset. The accuracy (acc.) is the classification accuracy calculated on the value function test set. All the results are conducted with Expert iter. policy model and dynamic tree sampling in the low-resources setting.

Methods	acc. \uparrow	low. \uparrow	OTR \uparrow	SOTR \uparrow	Time(s) \downarrow
\triangleright Ablation:					
Log prob (Polu and Sutskever, 2020)	-	47.9	39.4	51.7	150.3
Outcome (Han et al., 2021)	63.62	47.6	39.5	48.1	175.7
\triangleright Expert iter.+DT-Solver:					
Current state only	66.43	48.2	38.6	47.6	169.2
Root state and current state	68.01	48.9	38.7	48.7	167.1
Previous state to current state	69.39	49.6	39.3	44.5	167.9
Entire trajectory	70.70	48.4	39.5	45.5	167.7

resource settings. Moreover, in the low-resources setting, PACT+DT-Solver achieves a success rate of 37.3%, only 2.6% worse than the Expert iter. baseline of 39.9%. This result implies that our proposed DT-Solver can substantially close the gap brought by different policy models and empower weaker models to solve more difficult problems. The same improvement applies to the PISA dataset, where the Lisa+DT-Solver model improves from 16.8% to 26.7% in the Lisa baseline. In the high resources setting, the effectiveness of DT-Solver is marginalized by high expansion samples in the best-first search algorithm. Nevertheless, Our DT-Solver can still improve upon baseline methods 2.25% on average.

Focusing on how well different search algorithms can find the correct proof state to perform expansion, we calculated the global on-track rate (OTR) and successful on-track rate (SOTR). On-track rate averages the rate of selecting the correct nodes to expand in each proof search. As shown in the Table, our proposed DT-Solver has the highest global on-track rate in most scenarios. The result shows that DT-Solver with proof-level value function can locate states requiring more exploration and finds the proof efficiently.

The best success on-track rate and shortest average search time are biased toward weaker policy models. The high successful on-track rate with weak policy models is accomplished by only being able to solve simple theorems within one or two steps long. Meanwhile, the short average search time is because a weak policy model with the best-first search quickly exhausted the plausible action, resulting in a state named "empty queue" that quickly declares the search's failure. Although DT-Solver takes more time to solve a theorem, the algorithm makes maximum utility within the given time limitation. On average, Expert iter.+DT-Solver

only uses 10.2 seconds more to find 5.4% more proofs that Expert iter. in the high-resource setting. Calculating only the successful proof search, Expert iter.+DT-Solver in the low-resources setting uses 75.6 seconds on average to solve a problem, which is 11.6 seconds less than the high-resources setting counterpart Expert iter.

4.2.2 The effect of dynamic-tree sampling

We compare our proposed dynamic tree sampling with the classic Monte-Carlo tree search algorithm (MCTS). MCTS disables dynamic tree sampling by restricting the selection process until a non-expanded leaf node, denoting no backtracking capabilities. As shown in Table. 1, PACT+MCTS and Expert iter.+MCTS both improved against PACT and Expert iter. 0.95% on average. This shows MCTS a better algorithm to locate correct proof states in proof searches. Compared to DT-Solver, the model's performance dropped drastically without the ability to backtrack to previous states. In the low-resources setting, MCTS, on average, drops 11.25% in success rate compared to DT-Solver. We observe similar performance drops in the PISA dataset. From these results, we believe that the back-tracking capability in DT-Solver plays a vital role in improving the algorithm effect.

4.2.3 The effect of proof-level value function

To validate the effectiveness of our proof-level value function, we substitute the proof-level value function with outcome value function (Han et al., 2021) and log prob value function (Polu and Sutskever, 2020). Additionally, We calculate the value function's accuracy on the created value function test set to evaluate the performance of different value functions. As shown in Table 2, the critic model using Roberta as the classifier backbone performs better than the GPT counterpart (Log prob and Outcome). The entire-trajectory value func-

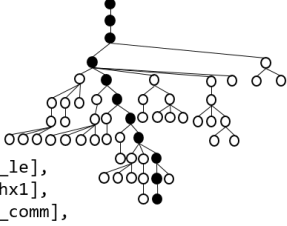

[Expert iter.+DT-Solver] Time: 216.88s OTR: 36.5%	[Expert-iter.] Time: 85.19s OTR: 21.8%
<pre>lemma rpow_le_rpow_left_iff_of_base_lt_one (hx0 : 0 < x) (hx1 : x < 1) : x ^ y ≤ x ^ z + z ≤ y := begin rw [rpow_def_of_pos hx0], rw [rpow_def_of_pos hx0], rw exp_le_exp, rw [← not_lt], rw [← not_lt], rw mul_comm, apply not_iff_not_of_iff; simp [lt_iff_le_and_ne, not_le], norm_num [le_antisymm_iff, hx1], simp [*, mul_comm, mul_left_comm], simp [*, log_pos_iff hx0, log_neg_iff hx0], end</pre> 	<pre>lemma rpow_le_rpow_left_iff_of_base_lt_one (hx0 : 0 < x) (hx1 : x < 1) : x ^ y ≤ x ^ z + z ≤ y := begin rw [rpow_def_of_pos hx0], rw [rpow_def_of_pos hx0], rw [← not_lt], rw exp_lt_exp, rw [← not_lt], rw [mul_comm], sorry, end</pre> 

Figure 4: Case Study. We use a concrete theorem as an example to demonstrate how DT-Solver (left) differs from an approach using best-first search (right). Both approaches use the same language model (Expert-iter) for predicting proof steps. The trees beside the proof are the proof tree created in the proof search.

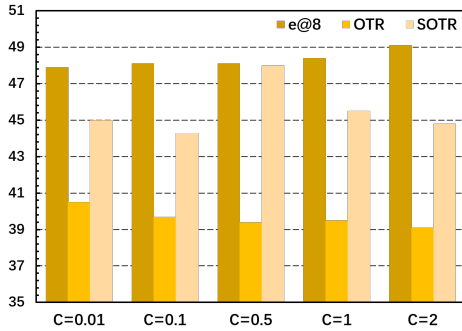


Figure 5: The effect of balancing exploration and exploitation. We explore different chooses of c in Eq. 1 and Eq. 2

tion substantially outperforms the outcome objective baseline in test set accuracy. Furthermore, the previous-to-current-state value functions provide the best end-to-end result. The result indicates that the state value is better estimated by more recent representations, instead of a remote anchor.

The minor disadvantage of the entire-trajectory value function might be the over-length trajectories, which need to be truncated to feed into the language model for prediction. Specifically, trajectories with lengths exceeding the maximum LM input were truncated from the left, resulting in the absence of the root state being preserved as a prefix. Among all value function queries, truncation occurred at a frequency of 53.00%, covering 53.54% of theorems in the Lean test set. However, when employing the "root-state-and-current-state" strategy, truncation only took place 3.10% of the time. We have extracted a sub-dataset that solely contains theorems with shorter proofs. This subset enables one to calculate the "entire trajectory" value function without resorting to any truncation. Table 3 shows the performance of root-state-and-current-state and

Table 3: Performance comparison among different value functions in the short-proof subset.

Methods	low. \uparrow	OTR \uparrow	SOTR \uparrow	Time(s) \downarrow
Root & current	90.67	82.9	56.7	41.0
Entire trajectory	93.22	86.7	57.6	32.4

entire-trajectory strategies in the short-proof subset. This outcome validates our initial hypothesis that truncation harms the overall performance of the value function.

4.2.4 The effect of balancing exploration and exploitation

In this section, we seek to understand how c in Eq. 1 and Eq. 2 affect model performance. As shown in Fig. 5, with larger c , the dynamic tree sampling weights more on the model prior to determining which node to select. DT-Solver achieves a better end-to-end success rate with larger c . This result indicates that more exploration and back-tracking are beneficial to find more plausible actions. However, the SOTR achieves the best performance in $c = 0.5$; this shows better selection accuracy when we focus more on exploiting the value function's state estimates.

4.3 Case Study

In this section, we conduct a detailed case study for a close-up look at the result DT-solver produced. More examples are shown in Sec. A.4. As shown in Fig. 4, we compare proof from DT-solver with the expert iteration policy model and best-first search-based expert iteration policy model. The theorem `rpow_le_rpow_left_iff_of_base_lt_one` aims to prove that for $x \in (0, 1)$ and $y, z \in \mathbb{R}$, we have $x^y \leq x^z$ if and only if $z \leq y$. Both approaches can produce (by prediction and search)

the same first 6 proof steps. These steps rewrite (with the tactic `rw`) the goal with previously proved theorems such as `rpow_def_of_pos` (if $x > 0$ then $x^y = \exp(x \cdot \log y)$) and `exp_le_exp` ($e^x \leq e^y$ if and only if $x \leq y$).

Here, we focus on Fig. 4 left. After the tactic application of the first `rw` [`<- not_lt`], DT-solver fails to generate the second `rw` [`<- not_lt`] subsequently. Ordinarily, such a malfunction in the best-first search algorithm would lead to the persistence of incorrect states without the opportunity to re-explore the node after the first application of `rw` [`<- not_lt`]. Nevertheless, the utilization of our Dynamic-tree sampling approach and proof-level value function facilitates the algorithm’s return to the node after 9 attempts to explore other search branches. Upon the decision to re-expand the state, there were 20 unexpanded states, and the total number of different states stood at 41. Thus the proof-level value function plays an essential role in identifying the appropriate states for re-expansion.

With the best-first search algorithm, we see that the search tree has a width of 1 or 2 and one easily falls into a state of ‘empty queue’, where no candidate generated by the language model is applicable. The approach stops the search process and returns failed before the time budget (300 seconds) is used up. While for DT-Solver, the number of child nodes varies from 1 to 5 and has greater variance than the best-first search. Although the number of child nodes can be relatively larger, the search process of DT-Solver manages to achieve a good balance between exploration and exploitation. Eventually, DT-Solver finds proof of 11 steps within the time budget with higher OTR (36.3% > 21.8%) than the best-first search.

5 Conclusion

In this work, we introduced a new ATP method, DT-Solver, which uses a proof-level value function to guide the dynamic tree sampling algorithm. DT-Solver smartly allocates computational budget to states that require more exploration and reduces the cost on easy states. The proof-level value function effectively locates difficult states. Extensive tests show that our method can indeed improve success rates on both PISA and mathlib datasets while being efficient enough to find proofs within a limited time budget. Although our method brings substantial advantages, there remain multiple aspects for improvement in the future, such as utilizing the for-

mal environment’s feedback or reducing enormous search spaces with high-level proof planning.

6 Limitations

Due to the limited ability of the policy model, most theorem still failed to find the proof because of poorly suggested proof steps. Predicting the proof step from the proof state requires substantial reasoning ability. It’s observed in the experiment that the language model tends to produce the same proof step in training data, and is unsatisfactory in generalizing for new states. Another limitation resides in the proof-level value functions. Although the performance of the proof-level value functions shows promising improvement in the value function test set. The end-to-end pass rate diminishes the performance gap. This accounts for two major reasons: 1) weak policy model fails to produce correct action even if our value function correctly located the state to expand. 2) Our value function’s performance is still behind the performance threshold where the value really helps the search drastically. One future direction is to enhance the language model for better reasoning ability by using a larger language model or adding symbolic reasoning into the language model to produce more reasonable proof steps and better evaluate states’ value.

7 Acknowledgement

This work was supported in part by the National Key R&D Program of China under Grant No. 2020AAA0109700, Shenzhen Science and Technology Program (Grant No. RCYX20200714114642083) and Shenzhen Fundamental Research Program (Grant No. JCYJ20190807154211365)

References

- Alexander A. Alemi, François Chollet, Niklas Eén, Geoffrey Irving, Christian Szegedy, and Josef Urban. 2016. DeepMath - deep sequence models for premise selection. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS’16, pages 2243–2251, Red Hook, NY, USA. Curran Associates Inc.
- Kshitij Bansal, Christian Szegedy, Markus Norman Rabe, Sarah M. Loos, and Viktor Toman. 2020. [Learning to Reason in Large Theories without Imitation](#).
- Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo

- Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. 1997. *The Coq Proof Assistant Reference Manual : Version 6.1*. report, INRIA. Pages: 214.
- Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirović. 2021. [Superposition for Full Higher-order Logic](#). In *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 396–412, Cham. Springer International Publishing.
- Leonardo de Moura and Nikolaj Bjørner. 2008. [Z3: An Efficient SMT Solver](#). In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 337–340, Berlin, Heidelberg. Springer.
- Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. [The Lean Theorem Prover \(System Description\)](#). In *Automated Deduction - CADE-25*, Lecture Notes in Computer Science, pages 378–388, Cham. Springer International Publishing.
- Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.
- Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. 2021. [Proof Artifact Co-training for Theorem Proving with Language Models](#). *ICLR 2022*. ArXiv: 2102.06203.
- John Harrison, Josef Urban, and Freek Wiedijk. 2014. History of interactive theorem proving. In *Computational Logic*, volume 9, pages 135–214.
- William Alvin Howard. 1980. The Formulae-as-Types Notion of Construction. In Haskell Curry, Hindley B, Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press.
- Albert Q Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding hammers to integrate language models and automated theorem provers. *arXiv preprint arXiv:2205.10893*.
- Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. 2021. Lisa: Language models of isabelle proofs.
- Laura Kovács and Andrei Voronkov. 2013. [First-Order Theorem Proving and Vampire](#). In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 1–35, Berlin, Heidelberg. Springer.
- Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. 2022. [HyperTree Proof Search for Neural Theorem Proving](#). Technical Report arXiv:2205.11491, arXiv. ArXiv:2205.11491 [cs] type: article.
- Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. [Deep learning](#). *Nature*, 521(7553):436–444. Number: 7553 Publisher: Nature Publishing Group.
- Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. 2020. [IsarStep: a Benchmark for High-level Mathematical Reasoning](#). In *ICLR 2021*.
- Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. [Roberta: A robustly optimized BERT pretraining approach](#). *CoRR*, abs/1907.11692.
- John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. 2006. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4):12–12.
- NORMAN. WHEELER MEGILL and DAVID A. 2019. *METAMATH: a computer language for mathematical proofs*. Lulu Press, Place of publication not identified. OCLC: 1105224041.
- Lawrence C. Paulson. 1994. *Isabelle a Generic Theorem Prover*. Springer Verlag.
- Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. 2022. [Formal Mathematics Statement Curriculum Learning](#). Technical Report arXiv:2202.01344, arXiv. ArXiv:2202.01344 [cs] type: article.
- Stanislas Polu and Ilya Sutskever. 2020. [Generative Language Modeling for Automated Theorem Proving](#). *arXiv:2009.03393 [cs, stat]*. ArXiv: 2009.03393.
- Markus Norman Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. 2020. [Mathematical Reasoning via Self-supervised Skip-tree Training](#).
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. page 24.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2018. [Language models are unsupervised multitask learners](#).
- Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. 2022. [A Generalist Agent](#). Technical Report arXiv:2205.06175, arXiv. ArXiv:2205.06175 [cs] type: article.
- Stephan Schulz. 2002. E - a brainiac theorem prover. *AI Communications*, 15(2,3):111–126.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneshelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. [Mastering the game of Go with deep neural networks and tree search](#). *Nature*, 529(7587):484–489. Number: 7587 Publisher: Nature Publishing Group.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharsan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. [Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm](#). *arXiv:1712.01815 [cs]*. ArXiv: 1712.01815.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. [Attention is all you need](#). In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Mingzhe Wang and Jia Deng. 2020. [Learning to Prove Theorems by Learning to Generate Theorems](#). In *Advances in Neural Information Processing Systems*, volume 33, pages 18146–18157. Curran Associates, Inc.

Daniel Whalen. 2016. [Holophrasm: a neural Automated Theorem Prover for higher-order logic](#). Technical Report arXiv:1608.02644, arXiv. ArXiv:1608.02644 [cs] type: article.

Yuhuai Wu, Albert Qiaochu Jiang, Jimmy Ba, and Roger Grosse. 2021. [INT: An Inequality Benchmark for Evaluating Generalization in Theorem Proving](#). *ICLR 2021*. ArXiv: 2007.02924.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2021. [miniF2F: a cross-system benchmark for formal Olympiad-level mathematics](#).

A Appendix

A.1 Overview

In the appendix, we first discuss related works in Section A.2. We give more implementation details and dataset statistics of DT-Solver in Section A.3. More examples of theorem proven by DT-Solver are shown in Section A.4.

A.2 Related works

Deep learning has already been applied to automated theorem proving prior to

GPT-f (Polu and Sutskever, 2020). Methods such as DeeMath (Alemi et al., 2016), Holophrasm (Whalen, 2016), HOList (Bansal et al., 2020; Rabe et al., 2020) and MetaGen (Wang and Deng, 2020) all apply one or several neural networks to recommend premises and proof steps. Then the search can either be guided by a neural network or not.

GPT-f (Polu and Sutskever, 2020) uses a fine-tuned GPT-2 (Radford et al.) as the language model to recommend the next proof step for the current *proof state* (called *tactic state* in Lean) and apply a best-first search based on the log-prob of the sequence predicted by the language model. Based on this framework, PACT (Han et al., 2021) provides a multi-tasking training scheme for a slightly larger GPT-2 model. Polu et al. (Polu et al., 2022) further introduce expert iteration (Silver et al., 2017) to achieve a sort of data augmentation to improve the training of the language model further. Most recently, HTPS (Lample et al., 2022) plugs in Monte-Carlo Tree Search (Silver et al., 2016) in this framework and applies an online version of expert iteration, which further advances the state-of-the-art performance on Metamath and the Lean benchmark miniF2F (Zheng et al., 2021).

Approaches in a similar paradigm have also been successfully applied to other formal systems such as Isabelle (Paulson, 1994; Li et al., 2020; Jiang et al., 2021, 2022), Lean (de Moura et al., 2015; Han et al., 2021; Polu et al., 2022; Lample et al., 2022) and other customized systems such as INT (Wu et al., 2021; Polu and Sutskever, 2020) and Equations (Lample et al., 2022).

A.3 Experimental details

A.3.1 Experimental setup

Model specification. The policy model is a decoder-only transformer (Vaswani et al., 2017) language model with 774M parameters, 36 layers, 20 attention heads, a hidden dimension of 1280, and a GPT-2 (Radford et al., 2018) tokenizer with 50400 vocabulary size. The model is pre-trained on Github python codes and the arXiv library. For lean, we further pre-train our policy model on the PACT dataset (mix1 and mix2 as denoted in PACT). During per-training, we use a global batch size of 512 with 2500 steps warmup using the AdamW optimizer. The cosine learning rate scheduling strategy is used with a maximum learning rate of $5 * 10^{-5}$ and a minimum learning rate $5 * 10^{-6}$.

The policy model is pre-trained for 250000 step. For fine-tuning in Lean, we use the tactic dataset in PACT to train the model, with a batch size of 16, and a warmup step of 1000 with a maximum learning rate of $5 * 10^{-5}$ and a minimum learning rate of $5 * 10^{-6}$. We are early-stopping the fine-tuning in 15000 steps with a total training budget of 100000 step. For Isabelle, we follow instructions from Lisa (Jiang et al., 2021) and reproduce the AFP dataset. The policy model is further fine-tuned on the AFP dataset with the same fine-tuning configuration used in Thor.

For the critic model, we use a pre-trained RoBERTa-base model (Liu et al., 2019) as our language model classifier. The critic model is fine-tuned for one epoch on the dataset generated following procedures described in Sec.3.1. Fine-tuning critic model uses a global batch size of 128, a maximum sequence length of 512, a linear learning secluding strategy with 2000 warmup steps, and a maximum learning rate of $1 * 10^{-5}$. We early stop the training with the lowest evaluation loss.

Dynamic tree sampling configuration. We set the balancing factor $c = 1$ in Eq. 1 and Eq. 2. To sample from the policy model, we use temperature $T = 1.2$ for Isabelle and $T = 1$ for Lean. Each proof step has a timeout limit of 10 seconds, and the search is terminated in the following condition: (1) A proof of the theorem has been found (2) Global timeout for 300 seconds (3) A total expansion step of 128 is reached.

Machine Configuration. We use Nvidia V100 GPU with 32GB of GPU memory for pre-training and fine-tuning. The training server has 104 CPU cores and 768GB of CPU memory. For training the policy model and the critic model, we use 8 GPUs with data parallel to speed up the training, with estimated 1100 GPU hours to run the training. For running the dynamic tree sampling algorithm, we use 32 GPUs to speed up the proof-finding procedure. We estimate it takes 64 GPU hours to run a single evaluation on the mathlib test set.

Interactive theorem provers To expediently verify proofs in Lean and Isabelle, we use REPL wrappers that formulate the interactions with ITPs in a REPL style. For lean, we use lean-gym following (Polu et al., 2022). In every step, lean-gym takes a proof state and a proof step as input and outputs a new proof state. For Isabelle, we created a REPL verifier named isabelle-gym based on PISA, with the same IO specification as lean-gym.

Table 4: Data statistics on the training data

Data split	size
GitHub python code	159GB
GitHub (the pile)	95GB
Arxiv documents	56GB
PACT-mix1	2GB
PACT-mix2	22GB
PISA fine-tuning	1102MB
Mathlib fine-tuning	119MB

Table 5: Theorem counts on Mathlib and PISA

dataset	train	valid	test
Mathlib	36960	1621	1580
PISA	156809	1627	6491

With these formal environments, we can check if a proof step is applicable or whether we have proven the theorem.

A.3.2 Dataset statistics

We provide statistics for our training dataset. Detailed statics for pre-training are shown in Table.4. We use 159G Python code collected from GitHub and PACT-mix{1,2} dataset for Mathlib pre-train. Moreover, we follow (Jiang et al., 2022) for Isabelle pre-train with GitHub and Arxiv dataset from the-pile (Gao et al., 2020). We also provide the number of theorems for Mathlib and PISA datasets in Table.5. The Mathlib dataset is created with versions of b72300f3455ae73c3ab9ed40fc1f80bbb9c85ba4 and lean core versions of 3.39.1. The PISA dataset is created with Isabelle2021. For running proof searches in the test set in Isabelle, we use the ‘quick test problem’ crated following Thor, which contains 600 theorems randomly sampled from the test split of the PISA dataset.

A.4 Proof examples

In this section, we provide more examples of proof found by our DT-Solver. For each proof, we also compare our method with the best-first search counterpart using the same policy model. Specifically, we provide two cases for proofs found in the Lean formal environment shown in Fig.6 and Fig.7. As well as two cases for proofs found in the Isabelle formal environment shown in Fig.8 and Fig.9.

[Expert iter.+DT-Solver] : Time: 58.1s OTR:47.6%	[Expert iter.] : Time: 15.07s OTR: 75.0%
<pre>lemma lipschitz_with.comp {Kf Kg : ℝ≥0} {f : β → γ} {g : α → β} (hf : lipschitz_with Kf f) (hg : lipschitz_with Kg g) : lipschitz_with (Kf * Kg) (f ∘ g) := begin unfold lipschitz_with, simp, intros x y, simp only [ennreal.coe_mul, mul_assoc, edist_comm], apply le_trans (hf (g x) (g y)), apply ennreal.mul_left_mono, simp only [ennreal.coe_mul] using hg.edist_le_mul x y, end</pre>	<pre>lemma lipschitz_with.comp {Kf Kg : ℝ≥0} {f : β → γ} {g : α → β} (hf : lipschitz_with Kf f) (hg : lipschitz_with Kg g) : lipschitz_with (Kf * Kg) (f ∘ g) := begin unfold lipschitz_with, intros, simp only [ennreal.coe_mul, mul_assoc, edist_comm], apply le_trans (hf _ _), sorry, end</pre>

Figure 6: An example proof in Mathlib. Our approach DT-Solver manages to find a proof for the theorem `lipschitz_with.comp` in 58.1 seconds. While with best-first search, a premature ‘Failed’ result is return after only 15 seconds, knowing that the time budget for proving one theorem is 300s.

[Expert iter.+DT-Solver] Time: 197.4s OTR: 24.0%	[Expert-iter.] Time: 143.34s OTR: 15.4%
<pre>lemma indicator_eventually_eq (hf : f =^[1] [1] ∩ P s] g) (hs : s =^[1] [1] t) : indicator s f =^[1] [1] indicator t g := begin simp only [eventually_eq, pi.zero_apply, iff_true], simp only [eventually_eq, eventually_inf_principal] at *, filter_upwards [hf, hs], assume x hx hxs, simp * at *, simp only [set.indicator], split_ifs; simp * at *, tauto, tauto, end</pre>	<pre>lemma indicator_eventually_eq (hf : f =^[1] [1] ∩ P s] g) (hs : s =^[1] [1] t) : indicator s f =^[1] [1] indicator t g := begin simp only [eventually_eq, eventually_le] at *, simp only [eventually_iff, mem_inter_eq, and_iff_left_iff_imp, filter.eventually] at *, revert hs, classical, intro h simp only [mem_inf_principal] at *, simp only [set.indicator] at hf ⊢, filter_upwards [h, hf], intros x hx hxs, split_ifs, tidy, end</pre>

Figure 7: An example proof in Mathlib. For this example, we see that both DT-Solver and best-first search successfully find a proof for the theorem `indicator_eventually_eq`, under a duration of similar scale. DT-Solver achieves a higher on-track-rate (24.0%) than that of best-first search (15.4%).

[Lisa.+DT-Solver] Time: 14.81s OTR: 57.1%	[Lisa.] Time: 3.12s OTR: 50%
<pre>lemma sumZero: fixes P :: pi shows "P \<oplus> \<zzero> \<simeq> P" proof - have "P \<oplus> \<zzero> \<sim> P" by(rule sumZero) thus?thesis by(rule strongBisimWeakEq) qed</pre>	<pre>lemma sumZero: fixes P :: pi shows "P \<oplus> \<zzero> \<simeq> P" proof - have "P \<oplus> \<zzero> \<sim> P" oops</pre>

Figure 8: An example proof in Isabelle. While the best-first search counterpart returns a premature ‘Failed’ result under only 3.12 seconds, our DT-Solver finds a two-step proof under 14 seconds with a high on-track-rate.

[Lisa.+DT-Solver] Time: 80.77s OTR: 62.0%	[Lisa.] Time: 4.90s OTR: 50%
<pre>lemma kD_nsqn: assumes "dip \<in> kD(rt)" shows "nsqn rt dip = nsqn \<^sub>r(the (\<sigma>\<^bsub>route\<^esub>(rt, dip)))" proof - from assms show?thesis unfolding nsqn \<^sub>r_def using assms unfolding nsqn_def by auto qed</pre>	<pre>lemma kD_nsqn: assumes "dip \<in> kD(rt)" shows "nsqn rt dip = nsqn \<^sub>r(the (\<sigma>\<^bsub>route\<^esub>(rt, dip)))" using assms unfolding nsqn_def by simp'</pre>

Figure 9: An example proof in Isabelle. Similar to the results in Fig.7, both approaches successfully find a proof while DT-Solver manages to achieve a higher OTR than best-first (62.0% > 50%).

ACL 2023 Responsible NLP Checklist

A For every submission:

- A1. Did you describe the limitations of your work?
Section 6
- A2. Did you discuss any potential risks of your work?
Section 6
- A3. Do the abstract and introduction summarize the paper’s main claims?
Section 1
- A4. Have you used AI writing assistants when working on this paper?
I use chatGPT and use it to polish the language of my paper and correct the grammar error. This happens in section 4.2.3 and section 4.3

B Did you use or create scientific artifacts?

Section 4

- B1. Did you cite the creators of artifacts you used?
References
- B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
Section A.3
- B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
Section A.3
- B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
Not applicable. Left blank.
- B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
Section A.3
- B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
Section A.3.2

C Did you run computational experiments?

Section A.3

- C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
Section A.3

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

- C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?

Section A.3

- C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?

Section 4

- C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?

Section A.3

D Did you use human annotators (e.g., crowdworkers) or research with human participants?

Left blank.

- D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?

No response.

- D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?

No response.

- D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?

No response.

- D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?

No response.

- D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?

No response.