An LLM-based Recommendation System for PVS

Nikson Bernardes Fernandes Ferreira
Department of Computer Science, University of Brasilia, Brazil

Mariano M. Moscato
Analytical Mechanics Associates Inc., USA

Mauricio Ayala-Rincón
Exact Sciences Institute, University of Brasília and IME, Federal University of Goiás, Brazil

Abstract

Interactive Theorem Provers are crucial for certifying system correctness, but they impose a steep learning curve and require a high level of user expertise. While Machine Learning techniques have been employed to suggest proof commands, current approaches often rely on resource-intensive pipelines to process the proof data used as input to the model. This work addresses this inefficiency by proposing a methodology that trains autoregressive language models directly on raw proof states, along with *LLM-PVS*, a prototype implementation integrated into the `VSCode` IDE. This approach allows PVS users to receive proof-command recommendations and lemma suggestions from local models without depending on extensive preprocessing of the proof data. While initial empirical evaluation indicates that this direct approach achieves competitive performance in restricted contexts, challenges regarding model hallucinations, lack of explainability, and scalability in premise selection remain to be addressed.

2012 ACM Subject Classification Theory of computation → Logic and verification; Computing methodologies → Machine learning; Computing methodologies → Natural language processing

Keywords and phrases Interactive Theorem Proving, Large Language Models (LLMs), PVS, Proof-step Suggestion, Autoregressive Models, Proof Search, Premise Selection, Code Assistants

Digital Object Identifier 10.4230/LIPIcs.MLSP.2066.Page

Funding *Nikson Bernardes Fernandes Ferreira*: Funded by a Brazilian CAPES PhD scholarship.
Mariano M. Moscato: NASA via RSES 80LARC23DA003.
Mauricio Ayala-Rincón: Brazilian CNPq Grant 407461/2025-6, and Productivity Grant 313290/21.

1 Introduction

Poor software quality has recently been recognized as a significant impediment to the modern economy, costing more than two trillion dollars in the US alone in 2022 [10]. In this context, the formal verification of systems has gained more and more traction in the industry. In particular, the level of assurance provided by approaches backed by interactive theorem provers is particularly attractive now and not only for highly critical applications, as in the past. One distinguished example of this kind of tool is the Prototype Verification System (`PVS`), which has been used in the formal verification community for decades [21]. With strong ties with the NASA Langley Formal Methods Program[1], PVS has been applied to the verification of critical pieces of software of great importance for the aeronautic industry, such as CPR [6] and DAIDALUS [18], which were included as reference implementations in their respective defining standards.

One impediment to the application of this kind of proof assistant is, as it is well accepted, the steep learning curve these tools impose on new users. In particular, PVS implements a deduction engine based on Gentzen's sequent calculus, in which a battery of inference steps

[1] `https://shemesh.larc.nasa.gov/fm/`

can be applied to the current state of the proof, represented by a particular arrangement of formulas called a *sequent*. A sequent can be seen as the description of the known information and targets, usually referred to as *proof state*, at a given moment. The user is responsible for the selection of the kind of inference to be applied to each sequent, and each of these applications results in new sequents until specific trivially valid proof states are generated, and the proof is considered finished by the system. The inference steps available to the user, which in PVS jargon are called *proof commands*, range from connective simplification to the introduction of formulas representing known additional information, either separately proven lemmas or assumedly valid axioms. Even beyond the high level of automation PVS provides for reasoning about specific domains, such as numeric properties or propositional reasoning, these deduction systems heavily rely on the user to guide the proving process.

Coinciding with the growth in theorem proving, the field of Machine Learning (ML) has exploded in the last decade. The continued improvements in parallel computing have enabled the development and scaling of complex deep learning techniques, such as Large Language Models (LLMs). Such models have been successfully applied to a diverse range of tasks, including customer service and support, information retrieval and knowledge management, and even code generation. Ubiquitous in modern life, as the application of these models seems to be these days, it has also been noted that they present attractive opportunities for automated reasoning and assisted theorem proving [2].

This paper presents *LLM-PVS*, a new ML-based agent providing assistance in the proving of properties in PVS. LLM-PVS generates proof commands and lemma recommendations based on the current sequent. Contrary to the traditional approach, in which LLMs are trained on processed data [9], this work proposes training LLMs for interactive proving on raw proof states. This provokes a drastic reduction in the effort needed to preprocess the data being fed into the agent. Another distinguishing feature of LLM-PVS is that it does not require specialized hardware, which allows it to be run locally, avoiding the security concerns raised by the use of externally-hosted AI agents. The main contributions of the work presented in this paper are given below.

- **Lightweight Local Implementation.** A working prototype implementation is presented, providing a low-latency agent that allows users to query the model during an active proof session. The prototype uses Llama 3.2 as the base infrastructure, and it is integrated into the VSCode-PVS plugin, a modern IDE for PVS implemented as an extension of the Microsoft Visual Studio Code environment [16].
- **Evaluation of Representation Efficiency.** As mentioned above, a distinguishing feature of the presented approach is that the model is trained on raw proof sequents. Previous work in this direction by E. Yeh et al. introduced CoProver, an LLM-based tool providing proof-recommendation for PVS that uses preprocessed sequents [27]. This work also presents an empirical comparison of the performance of both approaches, finding that, with current LLM technology, using raw proof data outperforms traditional translation-based approaches.

Related Work. The application of ML techniques in automated and interactive theorem proving is natural because mechanized deduction is usually an intractable task. Indeed, even in the simplest cases, theorem-proving processes have a high complexity: in propositional logic, the problem is NP-hard, in intuitionistic propositional logic, it is PSPACE-complete, and in first-order logic, it is undecidable [8, 24]. This complexity makes it naturally attractive to apply ML techniques not only to increase the degree of automation of proofs but also to help users of proof assistants choose proof-steps and explore a huge variety of proof-decisions during a proof attempt, maintaining the legibility and explainability of proofs. Modern ML

techniques, such as LLMs, have become powerful tools for exploring the complex space of mathematical proofs. Two surveys nicely and exhaustively discuss advances until 2020 in the integration of ML and SAT solvers [11], and until 2024 in the integration of ML and automated reasoning [2].

CoProver [27] is the most similar work to the one presented in this paper. As mentioned above, it presents an LLM-based agent to propose proof commands and lemmas to PVS users. The distinguishing difference is the preprocessing stage of the proof states. An empirical evaluation between both approaches is presented in Section 4. A selection of recent works that propose integrating LLMs with other ITPs is described below.

The authors in [15] analyze, using GPT-3.5, common mistakes LLMs make when generating formal proofs, concluding that it often identifies the correct high-level structure of a proof but fails to get the correct lower-level details. Based on this observation, they propose a framework that first generates an LLM proof and then iteratively leverages targeted symbolic methods to repair low-level problems in Coq.

In [20], LLMs and Lean4 [3] are integrated into a framework named Apollo. An LLM-generated proof is fixed by combining syntax cleaning, automatic solvers, and LLM-driven sub-proofs, which are verified by the prover. The process is repeated until a fully verified Lean4 proof is obtained. Experiments demonstrate the practical relevance of this integration by generating Lean4 proofs across a variety of examples.

In [26], Best-First Tree Search is applied to explore proof paths and automate Lean4 proofs with LLMs. The LLM interactively processes new proof nodes to generate tactics verified by Lean4. According to the outcome, the proof paths are expanded, adding valid or error nodes. When the tactic completes the proof, it generates successful proof paths.

In [22], state-of-the-art LLMs are applied to obtain formal representations verifiable in Isabelle/HOL [19] from natural language explanations, i.e., Natural Language Interpretations (NLIs). The crucial task in this kind of *autoformalization* is to translate NLIs into semantically correct and precise representations of the explanations, which requires an LLM's action to extract information from the errors provided by the prover and further human action to verify the translation's consistency.

In [5], the authors present experiments integrating Lean and Isabelle/HOL with LLMs via reinforcement learning, iteratively generating new (LLM-generated) conjectures and verifying them with the provers. The proposed framework assumes the roles of generating conjectures and proving theorems, each role providing training signals to the other. This iterative workflow generates increasingly challenging conjectures over time.

Organization. Section 2 discusses the proposed workflow and the implementation of the integration. Section 3 shows how the user interacts with LLM-PVS. Section 4 presents the model evaluation, including quantitative metrics compared with CoProver, as well as a qualitative evaluation. Section 5 presents the conclusions and future work. An extended version of this work is in progress.

2 Recommendation Approach

Providing a seamless local interaction between LLMs and an interactive proof assistant such as PVS requires some customization in the way the model is trained and handled. One of the guiding principles for this work is that LLM-PVS should be lightweight enough to run on a user-grade computer while maintaining the ability to learn non-trivial patterns in the structure of proofs and provide useful proof suggestions. This section presents the proposed approach, which enables the low-latency local integration of LLMs into the PVS workflow,

providing a natural interaction between the user, the model, and the proof assistant. LLM-PVS provides two kinds of suggestions: proof-command recommendation and lemma selection. The peculiarities of each feature are explained in subsections 2.1 and 2.2 respectively.

2.1 Proof-Command Recommendation

In the proof-command recommendation interaction, also called *proof-step* recommendation in the literature [14], the goal is to provide the user with a command name and applicable arguments in order to advance the proof to conclusion. As is standard practice in ML, the approach can be divided into two phases: Training, which occurs offline, and Inference, the interactive querying process taking place during a proof session.

2.1.1 Training Phase

Training an ML model means optimizing parameters to minimize or maximize a particular mathematical expression defined for a specific task, namely, the *loss function*. The proof-command recommendation training phase is performed on the task of next-token prediction, i.e., predicting the next token based on previous text. A token is a fundamental concept in natural language processing, representing an indivisible unit of text. Tokens can be frequent words, subwords, bytes, or other units, allowing the model to process text even when a word is unknown. The intuition comes from how words are formed [13]; for instance, consider the word "snowboarding". You can guess its meaning even without knowing it beforehand, based on its three pieces (namely, tokens): "snow", "board", and "ing". For instance, given the stylized sequent shown in Listing 1, the model learns to complete the text until "`Rule? (`" with the token "`EXP`" (first token `EXPAND` command). Indeed, it actually provides (predicts) a certainty for each next token, e.g., 49% for "EXP", 27% for "CASE", 5% for "INST".

Listing 1 Example of training mask for a given sequent and the proof command applied to it. The function `leq_elements` returns the list of elements of a given list l that are smaller than or equal to a given element x. The infix function $|.|$ returns the length of the list.

```
[-1]  ∀_{y:list[T]}: ∀_{x:T}: |y| < |l| ⟹ |leq_elements(y, x)| ≤ |l|
  |-------
{1}   |leq_elements(l, x)| ≤ |l|

Rule? (EXPAND "leq_elements" 1)
```

Because the approach aims to predict proof commands rather than memorize proof states, this work proposes the loss function to be computed only from proof steps. The highlighting in Listing 1 shows the proof-step information fed to the loss computation, i.e., the piece of information that the model learns using the previous text (i.e., the sequent) as context.

The training phase, depicted on the left-hand side of Figure 1, takes information collected from existent PVS proofs, represented by the `PVS Libs` box, to produce a model capable of recommending proof-commands for the current sequent, the `Trained LLM`. The initial proof data is collected in form of what is called `Proof Traces` in PVS jargon and consists of pairs formed by a sequent and the proof command applied to it. These `Proof Traces` are collected by executing PVS in batch mode through the script application `proveit`.

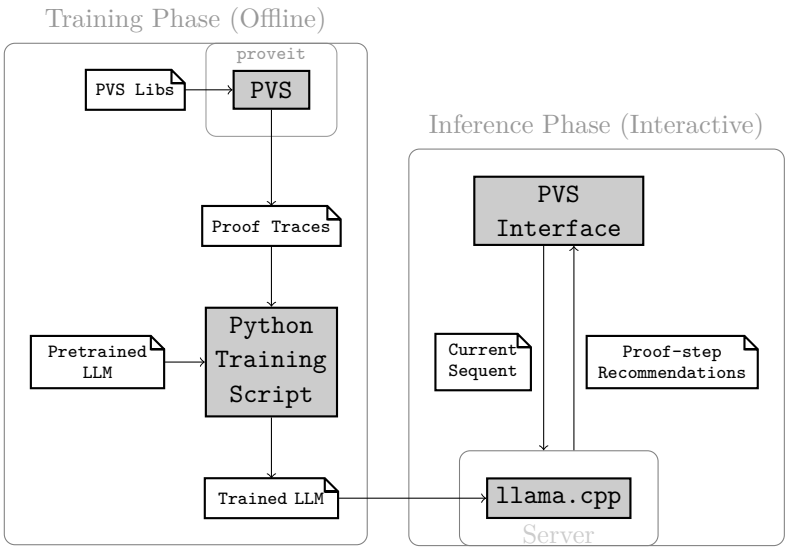


Figure 1 Workflow for training and integrating the LLM for proof-command recommendation in PVS.

For the purposes of evaluating this work, the traces are collected from NASALib[2], a collaborative collection of formalizations written in PVS and maintained by NASA. The collection spans diverse areas of Mathematics and Computer Science, like algebra, analysis, number theory, and algorithmics, among others. The selected data was taken only from one library due to the wide variety of the topics covered in NASALib. Greater diversity requires more data to represent it, leading to more time- and resource-consuming experiments. The case study was focused specifically on the Sorting library. This library is actively used for didactic purposes since it serves as an introduction to algorithmic analysis with step-by-step proofs, easing proof knowledge acquisition. The size of the Sorting library amounts to 7.6 MB, while the collected proof traces comprise 13,903 proof sequents and steps, totaling 14.8 MB. This data set was randomly split into 70% (9,732) for training, 20% (2,794) for validation, and 10% (1,377) for testing.

After collecting the `Proof Traces`, the ad-hoc `Python Training Script` uses them to fine-tune a `Pretrained LLM`, generating the resulting `Trained LLM` that will be queried by the user to obtain proof-command recommendations. The current implementation of LLM-PVS uses Meta's Llama 3.2 1B as `Pretrained LLM` [17]. The model was selected based on the performance it showed on the accompanying benchmarks and its small size, compared with other alternatives. The data used for the initial training of Llama 3.2 1B is not publicly available. Therefore, it is not possible to precisely determine the model's initial knowledge.

In the fine-tuning, the proof knowledge in `Proof Traces` is added to the `Pretrained LLM` by adjusting its parameters (weights matrices). Adjusting a pretrained model leverages transfer learning to apply prior knowledge to a new domain. The computational resources required for training are directly related to the number of parameters modified. Thus, the script fine-tuned the model through LoRA (LOw Rank Adaptation) [12]. The technique modifies a weight matrix (w) by approximating the weight change matrix (Δw) through a low-rank matrix decomposition. Therefore, for each fine-tuned matrix, two small parameter matrices are adjusted instead of the full matrix, significantly reducing the number of fit

[2] NASALib is publicly available at https://github.com/nasa/pvslib.

parameters and, consequently, the computational resources required. For instance, there is a 2048×8192 matrix on Llama 3.2 1B. Instead of changing 16,777,216 parameters directly, two matrices A and B of shapes 2048×8 and 8×8192 are trained so that their product approximates the original matrix change induced by the training, totaling only 81,920 parameters adjusted. This work uses a variant of LoRA that reduces the precision of the original floating-point weights via quantization (q-LoRA) [4]. In the experiments, the matrix decompositions use rank 8, and alpha 16, fitting 5,636,096 parameters, which corresponds to 0.4540% of the original 1,241,450,496 Llama 3.2 1B parameters, quantized in 4 bits.

Additionally, a bound on the input length was set to optimize the training phase. Samples (pairs sequent, proof commands) larger than this are cropped. The maximum input length was set based on statistics to keep most samples unchanged. The average is 203.24 tokens per sample. 95.58% of samples have fewer tokens than the average plus two times the standard deviation, i.e., 500.46 tokens. For optimization, the maximum was set to the nearest power of 2 (512), which covers 99.82% of the samples.

The model was trained over 20 epochs. In each epoch, one weight updating step was performed using a batch of 256 samples. For each update, the model predicts the certainty of the tokens and updates the weights, thereby reducing the loss function. To avoid overfitting, if the validation loss value does not decrease for 2 epochs, the training stops early. The training took 2 h 59 min on a single 40GB NVIDIA H100 GPU. The output of the training phase is a `Trained LLM` that learned to extract complex statistical patterns between the structure of the proof goal and the applied proof commands.

2.1.2 Inference Phase

The inference phase is responsible for obtaining proof-command recommendations during the interactive proof section. The presented prototype of LLM-PVS extends VSCode-PVS, which serves as the `PVS Interface`. However, any other user interface could be used, such as the one based on Emacs, which is still provided along with PVS. The `Current Sequent` that is presented to the user by the `PVS Interface` is also submitted to the model via an intermediary server that executes the `Trained LLM` and gets from it a list of `Proof-step Recommendations`. This list, ranked according to the certainty computed by the model, is finally presented back to the user in a dedicated panel in VSCode-PVS.

The LLM suggestion needs to be light and fast enough to be helpful. Here, the inference framework `llama.cpp` is used to perform low-level interaction with the `Trained LLM` to obtain suggestions. The framework is an open-source C/C++ engine designed to efficiently execute LLMs on non-dedicated hardware, specifically CPUs rather than GPUs. The model was quantized in `llama.cpp` at 4 and 8 bits to reduce the inference resource requirements. However, the latter performed best during the empirical evaluation.

2.2 Lemma Selection

In the lemma-selection interaction, the main objective is to provide one or more lemmas that can simplify the proof of the current sequent. The recommendations are selected from the set of available lemmas in a proving session, namely those in the theory where the original formula is declared and its dependencies. As expected, the complexity of the problem increases with the number of available lemmas [14]. The problem of training LLMs to suggest useful lemmas can then be fundamentally reduced to the question of how to represent lemmas and sequents to facilitate their recovery. As with the proof-command recommendations,

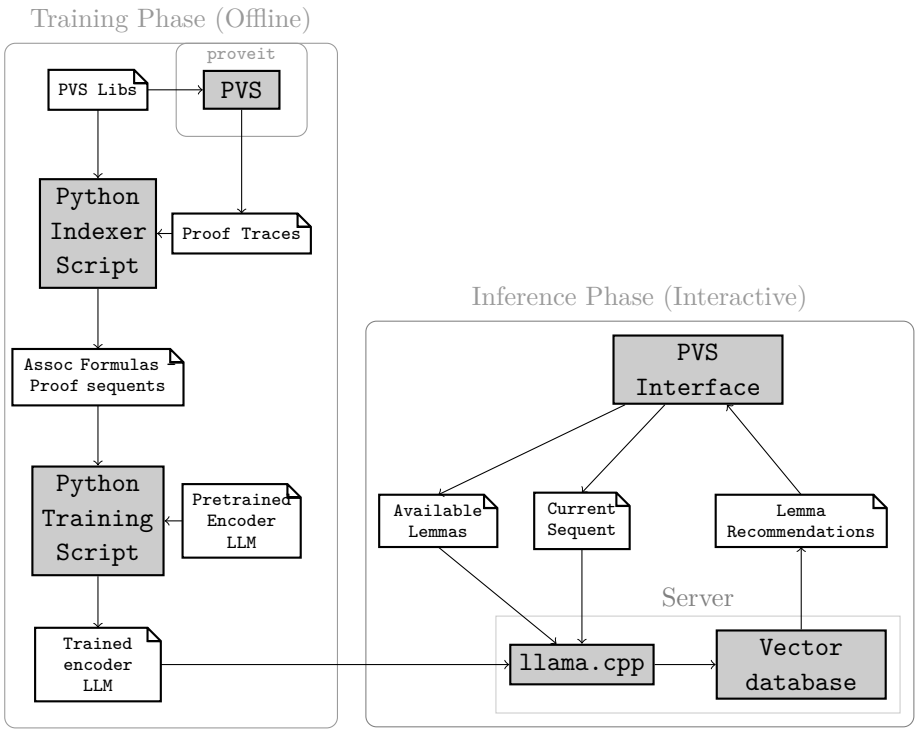


Figure 2 Workflow for training and integrating the LLM for lemma suggestion in PVS.

the approach for Lemma selection can be separated into two phases: Training (offline) and Inference (during proof).

2.2.1 Training Phase

The Training phase for lemma selection, depicted on the left-hand side of Figure 2, uses data from selected `PVS Libs` to obtain the `Trained encoder LLM`, a model capable of representing text in a high-dimensional space, where the proximity of a proof state to a lemma statement may indicate the applicability of such a lemma at that point of the proof.

The first step of the Training phase for lemma selection is to collect `Proof Traces` by running `proveit` on `PVS Libs` as was done for the proof-command recommendations. In the second step, an ad-hoc Python script, represented in the figure by the box `Python Indexer Script`, takes each sequent from the `Proof Traces` in which a lemma has been introduced to the proof and pairs the sequent with the actual declaration of the introduced lemma. For example, the lemma shown in Listing 3 is introduced in a proof when the sequent in Listing 2 is the current sequent. Then, the `Python Indexer Script` combines both the lemma and the sequent as a sample for the training. The collection of training data is represented as `Assoc Formulas - Proof sequents` in the figure.

Listing 2 Example of Sequent that is associated with lemma

```
|-------
{1} ∀_{l:list,x:T,r:list}
l = cons(x, r)  ⟹  |leq_elements(r, x)| < |l|
```

Listing 3 Example of lemma that is associated with sequent.

```
leq_elements_size : LEMMA
```

$\forall_{l:\texttt{list},x:T}$: |leq_elements(l, x)| $\leq$ |l|

In the case study, 873 pairs of sequent and lemma specifications were collected from the Sorting Library, comprising 438.1 KB of data. The data was split into 90% (785) for training and 10% (88) for testing.

After the data collection, the `Python Training Script` fine-tunes a `Pretrained Encoder LLM` using the `Assoc Formulas - Proof sequents` data, obtaining the `Trained encoder LLM`. The `Pretrained Encoder LLM` has an architecture that encodes the input on a vector, in contrast to the `Pretrained LLM` from the previous section, which predicts next-token statistics. This work used DistilRoberta [23] as `Pretrained Encoder LLM` due to its small size (82.1M parameters) and robust performance reported on benchmarks. In this step, all model weights are adjusted since the process does not require high computational resources. The model has less than 1% of Llama 3.2 1B parameters. A common loss function for training the model to encode data uses triples: an anchor (reference), a positive (a sample that should be closer), and a negative (a sample that should be farther away). However, the `Assoc Formulas - Proof sequents` data contains only sequents (anchors) and applicable lemmas (positive), i.e., those should be closer in the vectorial space. This work used the Multiple Negative Symmetric Rank Loss [7], a loss function that requires only the anchor and positive, removing the necessity of choosing negative samples.

2.3 Inference Phase

The inference phase is responsible for selecting and recommending applicable lemmas during the interactive proof session. As in the proof-command inference phase, there is an interaction between the `PVS Interface` and the llama server.

In the first step, the set of known lemmas is constructed by storing in the `Vector database` all the available lemma declarations represented in the high-dimensional space by the `Trained encoder LLM`. This step reduces the running time of the inference phase by processing each lemma only once and caching it in the knowledge base. In the next step, the `PVS Interface` provides the current sequent to the local server, which uses `llama.cpp` to execute the `Trained encoder LLM` to obtain its vectorial representation. It is compared to all lemmas in `Vector database` in the high-dimensional space through cosine similarity. The lemmas associated with the most similar vectors are then presented to the user as lemma recommendations in the `PVS Interface`.

3 LLM-PVS Integration Framework

This section describes how users can interact with LLM-PVS thanks to the integration into the VSCode-PVS extension, one of the main contributions of this work. Figure 3 shows a guiding example where the first five consecutive steps of the proof of the theorem `merge_sort_size` are depicted This theorem states that the function implementing merge-sort preserves the length of the input list. Because of space limitations, this Section focuses only on recommendations of proof commands. It is worth noting that the proof-command recommendations may include commands related to the inclusion of lemmas, but these are independent from the lemma-suggestion feature described in the previous section.

Figure 3a depicts how the initial sequent of the proof is presented to the user in VSCode-PVS. The current sequent is displayed in the proof console (on the right-hand side), and the model recommendations are shown in the LLM-PVS panel (on the left-hand side), ranked by model certainty. It should not be surprising that the top-ranked recommendation

Figure 3 Interaction with LLM-PVS during the proof of the lemma `merge_sort_size` in VSCode-PVS. The depicted Visual Studio Code screenshots are used with permission from Microsoft.

for the initial sequent points to the Skolemization of the target formula, via the `SKEEP` command with 51% of certainty, since most of the proofs in the library start in this way. In second place, LLM-PVS proposes to apply an induction scheme over the list `l` using the `MEASURE-INDUCT+` command, which is a direct way to prove this property over lists. Notably, the command arguments in the recommendation are correct in content and position. The following recommendation in the rank is to introduce a lemma via the `LEMMA` command. The proposed lemma could actually be useful in this proof, but it is declared after the one being currently proved, and thus it is not applicable. The `SKOSIMP` command in the fourth place is another flavor of Skolemization. Finally, the fifth recommendation is to rewrite the formula in the sequent using the lemma `length_merge_sort`, but this is a hallucination, since there is no lemma with that name declared in the library.

In the recommendations generated for this sequent, the manifestation of the *data-bias* issue can be seen. For instance, there are two main ways to perform induction in PVS: `MEASURE-INDUCT` and `INDUCT`. While the latter command usually produces simpler proofs for data structures such as lists, the model suggested the former, reproducing the preference of the authors of the proofs from the training library. This is because, like any other deep learning model, LLM-PVS tries to extract hidden patterns and reproduce them, thereby repeating the bias from the training data and, in this case, actually favoring the attempt of more complicated proofs. Similarly, both `SKEEP` and `SKOSIMP` are commands for Skolemizing; there is a clear preference for `SKEEP` in the training data, reflected in the model certainty: 51% versus 1%, respectively.

The user can tap one recommendation, and the command is sent to the proof console. After performing the induction, the screen shown in Figure 3b is presented. In order to use the inductive hypothesis added to the sequent as formula -1, it is necessary to expose the recursive nature of merge-sort by expanding its definition. The model is highly confident (82%) about this expansion, specifically on Formula 1. The second recommendation proposes analyzing the base case, where the list is empty. The `ASSERT` command is a powerful mechanism that simplifies the goal using decision procedures and automatic rewriting. Next in order of certainty, LLM-PVS recommends instantiating the inductive hypothesis using the `INST` command, although the suggested instantiation is a dead end. As for the initial sequent, the fifth recommendation is the introduction of a non-existent lemma. Note that in both hallucinations, the model certainty for such suggestions was close to zero.

Figure 3c shows the result of expanding the `merge_sort` function. The model is highly confident (72%) in lifting the conditionals (`LIFT-IF`), a step needed to work on each of the cases in the definition of the function. The other suggestions are also applicable. For instance, the expansion of `merge` enables the instantiation of the inductive hypothesis.

The sequent in the Figure 3d is the result of applying the `LIFT-IF` command. The first recommendation, `PROP`, would produce a split in the proof following the branches of the conditional. The model also suggests `LIFT-IF` with low certainty (13%), the same command used before. As shown in the inference phase (Section 2.1), the model input is solely the current sequent, and the model extracts complex statistics of the input, which does not provide a global proof context, leading to repetitions of the same recommendation in sequential steps. In this case, it is possible that the model learned the relation between conditional keywords in sequent and the `LIFT-IF` application.

Figure 3e shows the sequent after applying the first recommendation. At this point, the `Trained LLM` suggests two top options with near the same certainty; either expand the `merge` definition, or instantiate the inductive hypothesis with the correct list. Both alternatives are possible steps to advance the proof. Indeed, this approach *assists* the user during the

interactive proof section. The responsibility to *prove* remains on the user, checking the applicability of the recommendations and choosing the more convenient proof plan.

4 Empirical Evaluation

This section presents an empirical evaluation of the proposed approach, in which LLM-PVS is compared against CoProver [27]. CoProver also provides proof-command recommendations and lemma suggestions for PVS, but employs an extensive preprocessing of the sequent. CoProver collects a rich JSON representation of each sequent that is then translated by replacing variables, functions, predicates, and formula numbers with tags. In contrast, LLM-PVS is fed with the raw sequent, i.e., as presented to the user by the `PVS Interface`. Additionally, LLM-PVS provides command parameters, while CoProver does not. The empirical evaluation that follows is split according to the kind of suggestion provided: proof-command recommendation, and lemma selection.

4.1 Proof-Command Recommendation

As shown in the proof-command inference phase in Section 2.1, LLM-PVS provides a list of possibilities sorted by aggregate certainty. The `Trained LLM` was evaluated by comparing the recommendation list with the `Proof Traces`, i.e., checking if the proof step used in the existing proof matches the suggestions. The metric is expressed in terms of accuracy on the first k recommendations (acc_{top_k}). Denoting by $\langle s, c\rangle$ each sample from `Proof Traces`, where s is a sequent and c is the proof command applied to it, and using N to denote the number of samples evaluates, i.e., the size of `Proof Traces`, the Formula 1 expresses that the accuracy top_k is calculated as an average of hits and miss suggestions.

$$\text{acc}_{top_k} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}1, \text{ if } c_i \text{ is among the top } k \text{ recommendations for } s_i\\ 0, \text{ otherwise}\end{cases} \tag{1}$$

It is important to note that the metric defined above does not evaluate the capacity of the recommendations in helping prove the original formula, since there are often multiple ways to prove a property, and proof commands that apply similar reasoning steps. Instead, acc_{top_k} is intended to measure how capable the model is of mimicking a particular existing proof. For instance, Figure 3a illustrates a case where the model suggested `MEASURE-INDUCT` to perform induction; if the original proof contains `INDUCT` instead, it would count as a miss since that command is not recommended, even though it applies the same kind of deductive step. In Figure 3e, both expanding the definition and instantiating the inductive hypothesis would have advanced the proof. Still, if the existing proof prefers instantiation at this point, it counts as a miss for acc_{top_1}, where only the first option is considered a hit. Still, this metric is considerably more practical to implement than a more equitable alternative, and it can serve as a baseline for comparing the models.

The Table 1 shows the evaluation of the accuracy top_k on three different models. For this experiment, the `Proof Traces` from the Sorting library of NASALib were randomly split into three sets: 70% (9,732) for training, 20% (2,794) for validation, and 10% (1,377) for testing. The model *CoProver A* is the artifact as presented in [27], i.e., with no retraining applied. *CoProver B* is the CoProver model but trained using the `Proof Traces` extracted from the Sorting Library. LLM-PVS is the artifact presented in this paper, trained using the same collection of traces. The three models were evaluated using the same fragment of traces from the Sorting library. As can be seen in the table, LLM-PVS outperforms CoProver for all numbers of recommendations in the case study, providing the user's proof step as one of

Table 1 Comparing CoProver and LLM-PVS command prediction top-k accuracy (acc_{top_k}).

k	CoProver A	CoProver B	LLM-PVS
1	23.37%	48.33%	55.56%
3	45.95%	76.43%	82.72%
5	57.82%	85.91%	90.92%
7	66.27%	91.39%	95.21%
10	72.50%	95.32%	97.17%

10 suggestions on 97.17% of sequents. The original CoProver model (A) is considerably worse than its version adjusted on the domain data (B). It raises questions about the generalization capability of the approach. Indeed, ML approaches only extract hidden statistical patterns from the training data. If the test data is not in the same distribution, performance is expected to be lower. In the same sense, LLM-PVS was trained in a specific domain and may not perform well in other libraries without additional training. It is worth noting that, while the percentages shown in the table are useful for comparing the models, they should not be taken as an indicator of performance in the general case. The way in which the traces were separated does not prevent different sequents from the same proof from appearing in both the training and the evaluation sets. In such cases, both LLM-PVS and CoProver are expected to perform better than on sequents from a proof not present in the training set. The authors are currently working on designing an experiment that can be used as an indicator of the expected usefulness of the approach in the general case.

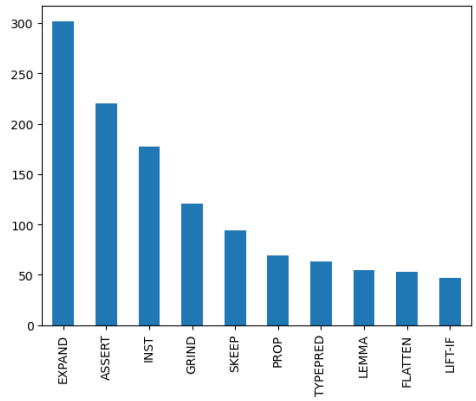
Another way to evaluate the approach is to investigate bias in suggestions. For instance, the model may become prone to predicting only the most frequently used command. Figure 4 shows the 10 most frequently used commands in the proofs from the Sorting library, and the most frequent top_1 recommendation by LLM-PVS. While both distributions follow a similar shape, the two most frequent commands swapped positions with the two most recommended commands. LLM-PVS is more prone to recommend `ASSERT` than `EXPAND`, but the difference is slight, which means that the model is biased to use this command, but it is not a significant bias. It is important to highlight the case of the `GRIND` recommendation. `GRIND` is a versatile command frequently used for two purposes: automatically completing proof branches and exhaustively applying obvious simplifications until a fixed point is reached. It is the fourth most frequent command used on `Proof Traces`, while it is the sixth most suggested by LLM-PVS. While `GRIND` is very powerful and usually able to close proof branches containing all the information needed to complete the proof, its application can lead to illegible sequents or even start a non-terminating rewriting process when applied to non-ground recursive expressions. Therefore, recommending less `GRIND` could be considered a positive bias.

4.2 Lemma Selection

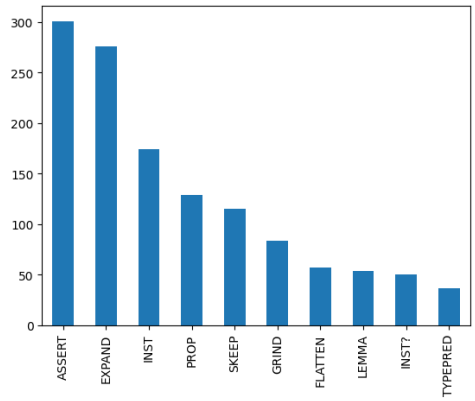
As described in Section 2.2, in this modality LLM-PVS suggests the most similar lemmas applicable to the current sequent. The approach was evaluated on sequents of `Proof Traces` where the existing proof invoked a lemma by comparing the first k lemmas proposed by the model with the one actually introduced. In this section, two metrics are used: Mean Reciprocal Rank (MRR) as defined in Formula 2, and Recall, as defined in Formula 3.

MRR represents the inverse of the average position of the invoked lemma in the recommendations. The closer to the first suggestion, the better.

$$\text{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{p_i} \tag{2}$$



(a) Histogram of the 10 most user-used commands.



(b) Histogram of 10 most suggested commands in top_1.

Figure 4 Comparing the expected histogram of command usage based on user data and the one suggested in the top_1 suggestion of LLM-PVS. The y-axis shows the number of times the command is used, and the x-axis shows the commands.

In the formula above, i is the index of the sample, p_i is the position of the introduced lemma in the recommendations for sample i, and N is the number of samples in `Proof Traces`.

The Recall (Recall_{top_k}) indicates the rate of samples where one of the recommendations is the lemma introduced in the existing proof. It is expressed in terms of the number of suggestions k, as shown in Formula 3.

$$\text{Recall}_{top_k} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}1, \text{ if the introduced lemma is among the first } k \text{ suggestions}\\ 0, \text{ otherwise}\end{cases} \tag{3}$$

As mentioned in Section 2.2.1, 10% of the data in `Assoc Formulas - Proof sequents` was separated for testing, comprising 88 pairs of sequents and introduced lemmas. Aware that the bigger the search space, the more complex the task in information retrieval, the lemma-selection modality was evaluated in two scenarios: selecting lemmas among the 187 lemmas specified in the Sorting library, and among all 19,959 lemmas available in NASALib, aiming to understand the impact of the lemma knowledge base expansion.

Table 2 shows MRR and Recall_{top_k} (with $k = 1$, 3, 5, and 10) measures on LLM-PVS, either limiting the lemma selection to the Sorting library or using the totality of the lemmas from NASALib. An MRR of 0.51 was reported for CoProver in [27], which indicates that, on average, the user-chosen lemma is near the second position. In its turn, the 0.75 of LLM-PVS indicates that it is between the first and the second position. While this result seems promising for the LLM-PVS approach, these measurements are not directly comparable since different data sets were used in the experiments. The Recall_{top_3} of LLM-PVS expresses that in 88% of the tested samples, the user-chosen lemma is among the 3 lemmas recommended.

These results suggest that the lemma selection of LLM-PVS could produce better results than the analogous modality of CoProver. However, it is necessary to conduct more experiments to confirm or disprove this hypothesis. Indeed, the lemma selection in LLM-PVS can recommend an applicable lemma in 99% of unseen sequents when suggesting 10 options by limiting the search. The lemma knowledge expansion to all lemmas specified in NASALib provokes a metric decay, recovering the lemma invoked by the user in only 84% of the samples in the same conditions. This could indicate a scalability issue in the lemma selection process, where the `Trained encoder LLM` did not seem to learn to move away from different domains.

Table 2 LLM-PVS performance as measured by MRR and Recall$_{top_k}$, with $k = 1, 3, 5, 10$.

Metric	Sorting Library	Full NASALib
MRR (0-1) ↑	0.75	0.63
Recall$_{top_1}$	61%	52%
Recall$_{top_3}$	88%	68%
Recall$_{top_5}$	95%	80%
Recall$_{top_{10}}$	99%	84%

Table 3 LLM-PVS Inference time on CPU and a user-grade GPU.

Device	Number of recommendations	mean (secs.)	std (secs.)	min (secs.)	25% (secs.)	50% (secs.)	75% (secs.)	max (secs.)
CPU	1	0.992	0.000	0.992	0.992	0.992	0.992	0.992
	10	2.766	0.429	1.933	2.476	2.688	2.966	4.568
GPU	1	0.127	0.074	0.034	0.082	0.111	0.155	0.502
	10	0.613	0.150	0.380	0.496	0.590	0.700	1.182
GPU	10 accum. certainty	6.040	1.219	3.732	5.191	5.779	6.861	10.536

4.3 Latency

The latency for the proof-command recommendation modality was evaluated by obtaining suggestions for 10% of the testing samples (137). Table 3 describes the time statistics to get 1 or 10 recommendations from LLM-PVS executing the framework on a CPU or a user-grade GPU. The framework uses two mechanisms to calculate the model's certainty: *first token* and *accumulated certainty*. Since `llama.cpp` returns the estimated certainty of each token, the former is calculated simply by taking the estimation for the first token in an LLM-PVS recommendation. In its turn, the accumulated certainty for the whole proof command is obtained by applying the General Multiplication Rule on the certainties of all the tokens in the recommendation. The reported experiments were performed on a 16 GB Intel i7-12700H computer with an NVIDIA RTX 3060 GPU.

Table 3 shows that obtaining 10 recommendations on CPU takes less than 3 seconds for 75% of cases, while the worst case took approximately 4.5s. Usually, it is faster than reading a long sequent. On the other side, if the user has a user-grade GPU, this time decreases to 0.7s and 1.2s, respectively. Calculating the certainty by accumulating each token's certainty introduces a significant overhead; it took approximately 10 *times* more time than the first token approach. Thus, LLM-PVS uses the latter as default. The results indicate that LLM-PVS provides low-latency recommendations, even on user-grade computers.

5 Conclusion and Future Work

This work proposes an approach that integrates LLMs into the PVS workflow to improve user productivity by providing useful proof-command and lemma recommendations during interactive proving. It is implemented as the framework LLM-PVS using Llama3.2 1B. LLM-PVS acts as a non-intrusive, low-latency intermediary between PVS and the ML model. The framework runs locally on standard computers without requiring specialized hardware, enabling it to be used in real-world industrial scenarios where confidentiality concerns prevent the use of remote LLM APIs.

LLM-PVS was evaluated on a case study containing a subset of NASALib and compared

to CoProver [27]. LLM-PVS results surpass CoProver across the evaluated metrics, even without considering the extensive preprocessing required by the latter. The recommendations were also qualitatively evaluated by analyzing practical scenarios. The suggestions seem to be reasonable and able to help the user advance the proof. Indeed, in the case study, by restricting the set of recommendations to the range from one to ten, the percentage of applicable proof-commands always exceeded that of CoProver.

However, hallucination, lack of global proof context, and data bias appear during usage. Hallucinations are inherent in LLMs [1] and, consequently, inherited by the approach. Sometimes, the solution predicts a correct proof command but applies it to the wrong formula or recommends the introduction of a nonexistent lemma. However, in the context of the proving task, the same proof assistant acts as a supervisor for the LLM, verifying the outputs' applicability and correctness. Therefore, the major problem caused by hallucinations is ineffective recommendations. Future work will explore recommendation refinement and filtering approaches to reduce the volume of unusable feedback presented to the user. A possible solution is to use the multi-threading capabilities of PVS to check each of the top-ranked suggestions, showing the user only those that provoke changes.

The proposed approach uses only the current sequent as context for the LLM and provides no global information about the proof. This leads to several issues. Notably, LLM-PVS sometimes repeats recommendations that are valid for previous moments of the proof but do not apply to the current sequent. Further work will try to address this issue by enriching the context information provided to the model, for instance, with the original formula to be proven and a processable part of the proof context.

Data bias is an issue for ML as it is for the proposed approach. ML models are meant to extract and identify patterns in the training data. In this way, the model replicates bias. Indeed, LLM-PVS reproduces the preferences of the authors of the proofs used for training. For instance, in the case study, it often suggests the PVS command `MEASURE-INDUCT` instead of `INDUCT`, even when the latter yields simpler proofs when applied to data structures such as lists and sequences. Different ways to prevent bias in LLMs are planned to be explored and applied to LLM-PVS.

Because of the causal nature of generative LLMs, recent LLMs have introduced reasoning as a prior step, analyzing the input before generating an explainable result. This addition increased their performance on more complex tasks, such as solving mathematical problems [25]. Thus, it is an interesting future work to check whether training a model to *reason* about the proof before suggesting a proof step improves recommendations and provides a natural-language explanation for its choice.

As expected, the performance of lemma recommendation degrades substantially as the search space grows. The suggested approach is efficient during the inference phase, as each lemma statement is processed only once and stored in a vector database. However, as the entire sequent is processed, it may include formulas that are not relevant for suggesting lemmas. A pairwise approach could access both the vector database and the current sequent, but it would still need to process all lemmas for every new prediction, affecting the overall performance of LLM-PVS. Thus, it remains as future work to propose a better representation for the sequent provided to the model, such as splitting the goal by formula and ranking them according to a given metric, such as the time of last modification or introduction.

References

1 Sourav Banerjee, Ayushi Agarwal, and Saloni Singla. LLMs will always hallucinate, and we need to live with this. *CoRR*, abs/2409.05746, 2024. `arXiv:2409.05746`.

2 Lasse Blaauwbroek, David M. Cerna, Thibault Gauthier, Jan Jakubuv, Cezary Kaliszyk, Martin Suda, and Josef Urban. Learning guided automated reasoning: A brief survey. In *Logics and Type Systems in Theory and Practice - Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday*, volume 14560 of *Lecture Notes in Computer Science*, pages 54–83. Springer, 2024.

3 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

4 Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: efficient finetuning of quantized llms. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

5 Kefan Dong and Tengyu Ma. STP: self-play LLM theorem provers with iterative conjecturing and proving. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net, 2025.

6 Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, Gregory Anderson, and François Bobot. Formal analysis of the Compact Position Reporting algorithm. *Formal Aspects Computing*, 33(1):65–86, 2021.

7 Luyu Gao, Yunyi Zhang, Jiawei Han, and Jamie Callan. Scaling deep contrastive learning batch size under memory limited setup. In Anna Rogers, Iacer Calixto, Ivan Vulić, Naomi Saphra, Nora Kassner, Oana-Maria Camburu, Trapit Bansal, and Vered Shwartz, editors, *Proceedings of the 6th Workshop on Representation Learning for NLP (RepL4NLP-2021)*, pages 316–321, Online, August 2021. Association for Computational Linguistics.

8 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

9 Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Sebastopol, CA, 2017.

10 Herb Krasner. The cost of poor software quality in the US. `https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-ReportNov-22-2.pdf`., 2022. Accessed: 2026-01-30.

11 Sean B. Holden. Machine learning for automated theorem proving: Learning to solve SAT and QSAT. *Found. Trends Mach. Learn.*, 14(6):807–989, 2021.

12 Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

13 Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, pages 66–71. Association for Computational Linguistics, 2018.

14 Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving. *CoRR*, abs/2404.09939, 2024. `arXiv:2404.09939`.

15 Minghai Lu, Benjamin Delaware, and Tianyi Zhang. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pages 1509–1520. ACM, 2024.

16 Paolo Masci and César A. Muñoz. An integrated development environment for the Prototype Verification System. In *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019*, volume 310 of *EPTCS*, pages 35–49, 2019.

17 Meta AI. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. `https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/`, 2024. Accessed: 2026-01-30.

18 César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, and María Consiglio. DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In *Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015)*, Prague, Czech Republic, September 2015.

19 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

20 Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: automated LLM and Lean collaboration for advanced formal reasoning. *CoRR*, abs/2505.05758, 2025. `arXiv:2505.05758`.

21 Sam Owre, John M. Rushby, , and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

22 Xin Quan, Marco Valentino, Louise A. Dennis, and André Freitas. Faithful and robust llm-driven theorem proving for NLI explanations. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 17734–17755. Association for Computational Linguistics, 2025.

23 Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019. `arXiv:1910.01108`.

24 Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theor. Comput. Sci.*, 9:67–72, 1979.

25 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

26 Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. 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), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 32588–32599. Association for Computational Linguistics, 2025.

27 Eric Yeh, Briland Hitaj, Sam Owre, Maena Quemener, and Natarajan Shankar. CoProver: A Recommender System for Proof Construction. In *Intelligent Computer Mathematics - 16th International Conference, CICM 2023, Cambridge, UK, September 5-8, 2023, Proceedings*, volume 14101 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2023.