

ENHANCING AUTOMATED THEOREM PROVING
IN COQ WITH SPECIALIZED LARGE LANGUAGE
MODELS AND TREE OF THOUGHTS

PIERCE MALONEY


ADVISOR: ZACHARY KINCAID

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF ARTS
DEPARTMENT OF COMPUTER SCIENCE
PRINCETON UNIVERSITY

APRIL 2024

I hereby declare that I am the sole author of this thesis.

I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

A handwritten signature in black ink, consisting of a stylized 'P' followed by a series of loops and a long horizontal stroke at the end.

Pierce Maloney

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

A handwritten signature in black ink, consisting of a stylized 'P' followed by a series of loops and a long horizontal stroke at the end.

Pierce Maloney

Abstract

This thesis explores the enhancement of formal theorem proving through the integration of Large Language Models (LLMs) within the Coq proof assistant environment. This work developed and evaluated a fine-tuned LLM capable of generating Coq code, trained to learn the relationship between tactics and proof states. It applied a Tree of Thoughts (ToT) reasoning framework to facilitate the exploration of possible proof paths, combined with the feedback from interaction with the Coq proof assistant at each step. The evaluation was run on three models – LLEMMA-7b, LLEMMA-7b fine-tuned, and GPT-4 – across 572 proofs from the CoqGym validation dataset, using GPT-4 as the state evaluator. A* and greedy search strategies over the ToT were compared to see the relative strengths of each. The results demonstrate that the A* search strategy outperforms the greedy approach as the proof tree grows, suggesting that the strategy’s exploratory nature leads to more effective proof completion over time. Additionally, the fine-tuned model performed the best on a large number of proofs, yet the results are not substantial enough to be conclusive. Moreover, though the fine-tuned model had the highest overall proof success rate, the base LLEMMA model generated legal tactics more frequently than both the fine-tuned model and GPT-4. This project highlights the efficacy of combining specialized LLMs and search strategies in enhancing the performance of automated theorem-proving systems. Code available on [GitHub](#) and dataset and models at [Hugging Face](#).

Acknowledgements

First, I would like to thank my advisor, Zachary Kincaid, whom I was lucky enough to have been advised by for my final three semesters at Princeton. I have deeply enjoyed my independent work, and much of it is due to Zak.

Thank you to Cleo and Keller, who grew up to eventually become wonderful siblings. I love you guys.

I would like to thank some of my mentors, Brian, Peter, Dusty. Additionally, I would like to acknowledge the late Mr. Zwemer, who taught me to love school, and Mr. Walch, who taught me to cherish life.

Thank you to Alex, who is always there for me.

Thank you to my friends, old and new, who always seemed to end up in Pyne.

I would also like to thank my teammates on the Men's Water Polo team, past and present, who have not seen me in weeks.

I want acknowledge Ivy for the wonderful people it has brought into my life, who put up with me every day, and whom I love dearly. It has made my experience at Princeton so special.

Thank you to those of you who supported me in writing this thesis, you know well who you are.

It's worth acknowledging the architects of Firestone Library, Robert B. O'Connor and Walter H. Kilham Jr., who in 1948 built a table on the third floor at a window facing East, at which the majority of this thesis was written.

To my parents.

Contents

Abstract	iii
Acknowledgements	iv
List of Tables	ix
List of Figures	x
1 Introduction	1
2 Background and Related Work	3
2.1 Related Work	4
2.1.1 Machine Learning for Proof Assistants	4
2.1.2 LLMs in Automated Theorem Proving	5
2.1.3 Large Language Model Agents and Reasoning	6
2.1.4 Goals	7
3 Approach	9
3.1 Models	9
3.1.1 Tactic Generator	9
3.1.2 State Evaluator	10
3.2 Data	10
3.3 Fine-Tuning	11
3.4 Query	11
3.5 Interface	11

3.6	System	12
3.7	Tree of Thoughts	12
3.8	Evaluation	12
4	Implementation	13
4.1	Data Preparation	13
4.1.1	Creating Import Context	13
4.1.2	Training Data	14
4.1.3	Query Context	16
4.2	Training	17
4.2.1	Getting Model	17
4.2.2	Preparation and Loss Masking	17
4.2.3	Data Chunking	18
4.2.4	Training Runs	18
4.3	Model Inference	19
4.3.1	Custom Handler	20
4.4	Server	22
4.5	Tactic Tree	23
4.6	Solver	24
4.6.1	Executing Coq Code to Build Proofs	25
4.6.2	Generation and Evaluation Inference	25
4.6.3	Search Strategies	25
4.6.4	Building Queries	26
4.6.5	Adding Nodes and Handling Errors	27
4.6.6	Saving Outputs	27
4.7	Frontend	27
4.8	Evaluation	29

5 Results	31
5.1 Quantitative Results	31
5.2 Qualitative Results	35
6 Conclusions and Future Work	38
6.1 Future Work	38
6.2 Discussion	38
A Prompts	40

List of Tables

5.1	Summary of Proof Results on Common Subset	32
5.2	Summary of All Proof Results	33
5.3	Tactic Generation Statistics	34

List of Figures

4.1	Training Steps vs. Val Loss on four training runs	19
4.2	A diagram of the system and its interactions	23
4.3	Frontend v0: an example proof, mult-is-zero, is shown.	28
4.4	Frontend v1: Introduction of tree navigation. An example proof, mult-is-zero, is shown.	28
4.5	Frontend v2: a straightforward system of analyzing proof trees. LEMMA-base using A* search generated this complete proof of reduces-transitive from coq-library-undecidability/Problems/Reduction.v	29
5.1	Tactics Generated vs. Proof Completion, Common Subset of Proofs	32
5.2	Tactics Generated vs. Proof Completion, All Tested Proofs	34

Chapter 1

Introduction

The advent of formal theorem proving represents a significant step in the intersection of Mathematical Logic and Computer Science, enabling the definitive validation of theorems solely through logical deduction. This field enhances the reliability of critical technological systems by verifying they work as expected. In recent years, the emergence and integration of Interactive Theorem Provers (ITPs) and Automated Theorem Provers (ATPs) have augmented human capabilities, allowing for complex proofs to be developed and verified with greater efficiency.

Despite this, the process of formal verification remains intricate and labor-intensive. The latest developments in Machine Learning (ML), particularly the evolution of Large Language Models (LLMs), present a promising avenue for addressing the challenges inherent in formal theorem proving. LLMs have demonstrated potential in understanding and generating complex textual content, which may be leveraged to automate and refine theorem-proving processes.

This project proposes a novel approach to enhance theorem proving using a fine-tuned LLM, capable of generating and evaluating Coq code—a prominent language in formal theorem proving. By integrating enhanced ML techniques with traditional theorem-proving methods, this project aims to demonstrate the promise of what can

be achieved in automated theorem-proving in this domain.

Chapter 2

Background and Related Work

Formal theorem proving aims at establishing the truth of mathematical theorems through logical reasoning. It ensures conclusions follow logically from premises without the need for experimental verification. As a result, formal theorem proving has become an invaluable tool in fields requiring high levels of mathematical certainty, such as formal verification of software and hardware systems, where it guarantees that systems behave as intended. This technology paves the way for advancements in our approach and theoretical understanding of the validation of systems by bridging the gap between mathematical concepts and their practical applications.

Interactive Theorem Provers, such as Coq, Isabelle, and Lean, offer a collaborative approach to the development and verification of formal proofs, using human input to guide the proof process. ITPs allow mathematicians and computer scientists to build proofs interactively, with the software verifying the correctness of each step. This method leverages human intuition for more complex or nuanced proofs, providing a platform for verifying mathematical concepts.

Automated Theorem Provers are designed to automatically find proofs for theorems within specific logical frameworks without human intervention. By employing algorithms to navigate the space of possible proofs, ATPs attempt to handle tasks

that are either too tedious or complex for manual processing. As the capabilities of ATPs improve, so do the abilities of mathematicians and computer scientists who employ them.

Despite their importance, formal verification tools involve a highly intricate and time-consuming process that requires a deep understanding of the use of tactics to simplify and prove assertions. With the rapid advancement and increasing ubiquity of Machine Learning technologies, leveraging these technologies to assist with the more complex and tedious aspects of formal verification becomes a compelling prospect.

Some of the most notable advancements of recent in ML come in the form of LLMs, which have demonstrated remarkable abilities in understanding and generating human-like text based on vast amounts of data. These models could revolutionize the field of formal verification by automating aspects of the theorem-proving process, synthesizing lemmas and auxiliary propositions and thereby reducing the cognitive load on human experts.

2.1 Related Work

2.1.1 Machine Learning for Proof Assistants

Learning to Prove Theorems via Interacting with Proof Assistants addresses the challenge of automating theorem proving within the Coq proof assistant environment by leveraging ML techniques and constructing a novel dataset. It presents a novel encoder-decoder model that generates tactics as programs, represented in the form of abstract syntax trees. The resulting ATP, Proverbot9001 [11], is a neural machine learning-based system designed to automate the generation of proofs for software correctness within interactive theorem provers. By applying Proverbot9001 to the CompCert [7] verified C compiler—a significant practical proof project—the system demonstrated its capability to automate proof tasks. Specifically, Proverbot9001

achieved automatic proof generation for 28% of theorems in the test dataset with additional solver support, marking a fourfold improvement over previous machine learning models for Coq proof generation.

“TacticToe” [3], designed for the HOL4 interactive theorem prover, utilizes machine-learned insights from human-generated proofs. It employs a Monte Carlo tree search algorithm to navigate viable tactic level proof paths, adopting a goal-oriented proof approach that progressively evolves the proof state towards a solution. Similarly, Tactician [4] related search-based methods but for Coq—contributing to tactical proof generation—using a modified k-NN algorithm.

2.1.2 LLMs in Automated Theorem Proving

As explained in the introduction, LLMs present an interesting new area of exploration with regards to Automated Theorem Provers. With their capability to generate large amounts of coherent output, an LLM may be used to generate entire proofs at once [5].

LLMs trained on large amounts of Coq code, such as LLEMMA [2], which takes the base Code Llama [10] model and further pretrains it on a collection of mathematical proofs and code, seem promising in handling Coq code. However, these models may lack a deeper understanding of the underlying tasks. A limitation of this approach is that it overlooks the dynamic relationship between the model and the nature of Interactive Theorem Provers. Specifically, when a tactic is applied within an ITP, it alters the state of the proof, affecting how subsequent tactics will be interpreted. Yet the LLM learns an entire proof without seeing how the proof state is affected sequentially. Thus, the omission of this strategic interaction deprives autoregressive models of crucial feedback to update their understanding as the proof evolves. Ostensibly, the model would learn tactic generation without an updated understanding of the proof state, rendering it more prone to hallucination when put into practice.

A technique bridging this gap is COPRA [13], an in-context learning agent tailored for formal theorem-proving within environments such as Lean and Coq. COPRA leverages the dynamic feedback loop inherent in theorem proving. It employs the high-capacity, general-purpose language model GPT-4 [9] to recommend tactics using a stateful backtracking search. The process is iterative; each proposed tactic is executed in the proof environment, and the outcomes are used to refine subsequent queries. This feedback loop enriches the model’s understanding by incorporating execution results, selected search history, and lemmas from an external database into the prompt construction for future queries. COPRA’s efficacy is demonstrated through its superior performance on benchmarks for Lean and Coq tasks from the CompCert project. This construction of queries provides the foundation for effectively leveraging the capabilities of LLMs; however, the selection of a general-purpose language model leaves a more specialized model selection to be desired.

2.1.3 Large Language Model Agents and Reasoning

The impressive performance of Large Language Models on human tasks has spurred extensive research into expanding their capabilities. This includes enabling agents to act as independent agents and developing structured methods to enhance their reasoning processes. Work in this domain aims to extend the capabilities of LLMs.

LLM Agents

The emergence of LLM agents represents a new field in the realm of automated reasoning and problem-solving. As outlined by recent studies [16][12], these agents utilize LLMs to interact dynamically with environments, gathering information to refine their prompts and behaviors. In-context learning capabilities enable LLM agents to adapt and respond more effectively to complex tasks. LLM agents demonstrate a remarkable ability to evolve their understanding and approach, showcasing the po-

tential of LLMs as adaptive tools in various applications such as ATPs.

Reasoning Techniques

OpenAI’s work using Reinforcement Learning from Human Feedback—an iterative method to train models using direct feedback from human reviewers—underscores the significance of enhancing LLMs’ multi-step logical reasoning abilities, especially given their propensity for logical errors [8]. By comparing outcome supervision with process supervision, the latter emerges as a more effective strategy for training models on mathematical problem sets. Process supervision, particularly when coupled with active learning, significantly outperforms outcome supervision by providing feedback at each step of the reasoning process, leading to enhanced problem-solving capabilities.

Taking a different approach to improving LLM reasoning capabilities, the “Tree of Thoughts” (ToT) ([15]) framework offers a novel approach to problem-solving by allowing LLMs to explore multiple reasoning paths and make deliberate decisions through a structured inference process. By enabling models to self-evaluate and adjust their course of action through backtracking or lookahead strategies, ToT significantly enhances LLMs’ capacity for complex reasoning and decision-making across diverse tasks. Substantial performance improvements in tasks requiring intricate planning on search capabilities demonstrate the effectiveness of the ToT framework, exemplifying how exploratory reasoning can expand the problem-solving abilities of LLMs.

2.1.4 Goals

To address the gaps in previous related work, this project fine-tunes an LLM capable of producing Coq code onto a new dataset of Coq proof files. This method teaches the model to interact between tactics and proof states. After training the model, I combine this new model with ToT reasoning techniques and an improved tree search. Finally, I allow the model to verify each tactic is legal and meaningful by giving the

ATP the ability to run Coq code. The project’s goal is to evaluate this novel approach on a subset of Coq proofs and compare it to preceding approaches – contributing to the field at the intersection of LLMs and automated theorem proving.

Chapter 3

Approach

3.1 Models

I use LLEMMA as the base model of the project, as its pretraining on Coq code provides a substantial learned foundation for understanding the nature of proofs in Coq. I also use GPT-4, primarily as a state evaluator model, for its state-of-the-art instruction capabilities. The ToT framework outlines two distinct steps when navigating through the tree: generation and evaluation. To accomplish this, I assign one model to tactic generation and one to state evaluation.

3.1.1 Tactic Generator

This model’s job is to generate n tactics given a proof state. There are two main ways to do this with LLMs: completion and instruction. Completion leverages the base ability of language models’ proficiency in predicting the next most likely tokens. This involves creating a query where the next tokens to be predicted would be the next tactic to apply. Instruction, on the other hand, is when the user describes what the model should output explicitly, and the model follows the instructions.

For models like LLEMMA, which are trained primarily on large files of code, com-

pletion is logical, as the model has a better understanding of the next token after code compared to after an instruction. However, this dynamic changes for models like GPT-4, which have been fine-tuned on instruction datasets. I experiment and evaluate both approaches.

3.1.2 State Evaluator

The state evaluator model needs to take a proof state and evaluate it by either giving it an explicit score or choosing the best option from a set. For an effective tree search, proper state evaluation is fundamental for efficiently exploring possible tactic paths through a proof. I experiment with different models and techniques for state evaluation to determine which is best.

3.2 Data

The goal of fine-tuning the LLM is to create a model that learns not just when a human would employ certain tactics, but also how tactics affect proof states. To do this, I create a dataset that includes the proof’s goals before every tactic is used, any imported modules as context, and get rid of extraneous text. To prepare the data, I took the CoqGym dataset, which comes with train, test, and validation splits. I then cleaned the Coq proof files to streamline the import of necessary theorems and definitions while excluding extraneous information, such as unnecessary comments or how the imported definitions or theorems were proved. This approach both ensures a cleaner file and also leaves more tokens available for context, which is critical for this type of task.

Then, I add the proof goals after each tactic is employed and insert the import contexts by detecting module imports from the Coq files. The data preparation is a crucial and substantial part of the training process and is equally valuable during

evaluation, when I leverage the cleaned Coq code to build the queries.

3.3 Fine-Tuning

To fine-tune LLEMMA, I employ a Colab environment to fine-tune the seven-billion parameter model using the Quantized Low-Rank Adaptation (QLoRA) technique to optimize computational efficiency. I add special tokens for comment delimiters to the tokenizer and apply loss masking to exclude those specific tokens from learning. I do this as I do not want the model to learn to predict goal states or imported context, but rather which tactics to use given a proof state. I chunk the data and train the model, saving and deploying the result.

3.4 Query

To build an effective query for the LLMs, the goal is to include as much useful context in as few tokens as possible. Like in training data preparation, building clean and useful queries includes removing unnecessary information and including information that is left out. I use the same technique to remove comments, include the import context, yet I do not remove the tactics within the target proof file, as oftentimes other proofs in the same file use similar tactical techniques as the one we wish to prove. Including the tactics of proofs before the target proof may provide a useful guide for the theorem-prover.

3.5 Interface

To present a clean display, I create a simple frontend that allows a user to explore proofs that the theorem-prover generates. This is useful to visualize the proof tree and how the model elects to choose certain tactics, and it was particularly useful in

the early days of the project before the theorem-prover ran on its own.

3.6 System

The system has many different aspects – where the models are trained, deployed, and employed are all different – and its complexity is increased due to the computational requirements of running such large models. I create a server that manages interactions between the theorem-prover and the endpoints of the models on the web, and CoqGym handles the interactions between the theorem-prover and Coq.

3.7 Tree of Thoughts

I create a structure for maintaining the tree that facilitates simple navigation to all nodes in the tree, where each node represents a tactic and proof state, and the paths to nodes represent the sequence of tactics from the beginning of the proof. In these nodes, I include costs and evaluation scores for search techniques such as A* or Greedy, so I may compare the relative effectiveness of each strategy.

3.8 Evaluation

I leverage the CoqGym evaluation framework to facilitate the interaction with Coq proofs, which includes an environment that abstracts the interactions between the theorem-prover and Coq. I build my own evaluation environment that utilizes this to manage the proof tree, tactic generation, tactic selection, and Coq feedback.

Chapter 4

Implementation

This section details the implementation of the project, from the data preparation, model training, system, ToT framework, and to the evaluation.

4.1 Data Preparation

4.1.1 Creating Import Context

First, I construct the context necessary to include when a Coq file is being imported into the current proof, as these files may contain useful theorems and definitions relevant to the proof. Subsequently, I refined them by removing extraneous information. The advantage of removing comments is that it leaves many more tokens available for context. However, the drawback is that this approach removes human explanations of different proofs. Although, oftentimes these explanations do not directly assist with proving but rather contextualizing the necessity of certain proofs within the greater framework of the project. The comments are more useful in determining which previously proved theorems or definitions are useful for retrieval, as discussed in the Future Work section.

Additionally, after removing all comments, I also removed unnecessary white space

and indentations. In certain cases, whitespace is important to understand the structure of the proof, so I only removed lines where there was more than one empty line between them. In doing so, I condensed the file significantly.

Further, while theorems and definitions are essential to the context of a proof file, the extensive list of tactics required to prove them are often unnecessary. Given each proof ends in a proof-end keyword, such as “**Qed.**” or “**Defined.**”, I used these as signals that tactics were being used. I then removed the tactics by going through the lines of the Coq file in reverse order, detecting one of the proof-end keywords, and then removing all tactics until the proof statement. The proof statement is identifiable by a proof-start keyword, such as “**Theorem**”, “**Lemma**”. However, sometimes proof statements occupied multiple lines, so once I detected a proof statement, I looked forward until the end of the statement, denoted by a period. After confirming the complete proof statement, I added it to the file.

Lastly, I saved import tactics in each Coq project directory. Instead of nesting directories, I represented their relative path with ‘.’ as a separator rather than ‘/’, which is useful for importing the context later.

4.1.2 Training Data

To create the training data, I added the goals for each proof and inserted the import context.

Inserting Goals

I proceeded by compiling the lines of the proof to a `.txt` file until reaching a proof statement that was associated with a proof in the JSON file from CoqGym. I then stepped through each tactic in each proof using CoqGym’s list of tactics. Next, I added the goal state as a Coq comment (`* Goal: ...*`) after each tactic, beginning with “**Proof.**” Notably, I only included foreground goals. If no goals were in focus,

Categories/Essentials/FactsTactics.v:

```
...
Definition equal_f : forall
  (A B : Type) (f g : A
    -> B), f = g -> forall x
    : A, f x = g x.
Proof.
intros A B f g H x.
destruct H; reflexivity.
Qed.

Definition f_equal : forall
  (A B : Type) (f : A ->
    B) (x y : A), x = y -> f
    x = f y.
Proof.
intros A B f x y H.
destruct H; reflexivity.
Qed.
...
```

Inserted Goals:

```
...
Definition equal_f : forall
  (A B : Type) (f g : A
    -> B), f = g -> forall x
    : A, f x = g x.
Proof.
(* Goal: forall (A B : Type)
  ) (f g : forall _ : A, B
  ) (_ : eq (forall _ : A,
    B) f g) (x : A), eq B (
    f x) (g x) *)
intros A B f g H x.
(* Goal: eq B (f x) (g x)
  *)
destruct H; reflexivity.
Qed.

Definition f_equal : forall
  (A B : Type) (f : A ->
    B) (x y : A), x = y -> f
    x = f y.
Proof.
(* Goal: forall (A B : Type)
  ) (f : forall _ : A, B)
  (x y : A) (_ : eq A x y)
  , eq B (f x) (f y) *)
intros A B f x y H.
(* Goal: eq B (f x) (f y)
  *)
destruct H; reflexivity.
Qed.
...
```

I included background goals as `(* BG Goal: ...*)`. Since tactics only apply to focused goals, it is logical to only include focused goals to ensure the model can learn them. Additionally, should there be more than one goal, I reversed the order in which the goals were included so that the most recently created goal would appear last in the list of focused goals. This modification arose after experimentation and seeing improved tactic generation from the base LLEMMA model when the goals were

reversed. Once the proof was completed, added the entire proof and goals after each tactic to the file

Inserting Import Context

To insert the import context, I first detected when modules were being imported by searching for the keywords “Require Import” or “Require Export”, which are followed by the list of modules to import. Sometimes, they may import modules explicitly referring to the directory with “From <larger module> import <module>”. In this case, I appended the larger module to the nested one with a period and searched for the module in the import context directory. In Coq, import modules are separated by periods, so I took the list of all available modules and performed the search for the relative path, made possible by the method of saving the import context described earlier. I then included the import context as a comment at the top of the file in the form `(* <module name>: <import context> *)`.

4.1.3 Query Context

Additionally, to create the query context, which the model uses to build the LLM query that contains the context necessary for a proof, I followed similar steps to generating the training data, but omitted the insertion of goals into the file. This omission is necessary to preserve context. The goals included in the Coq file can double the context size (or more), but I elected to keep the tactics in the proof context. The goals will come only when the theorem is actively proven.

4.2 Training

4.2.1 Getting Model

I first set up a Colab environment for training and pulled LLEMMA-7b from Hugging Face. I saved LLEMMA-7b to my Google drive to prevent pulling the model from the cloud each time, speeding up development. I could then mount the drive in Colab and load the model from the drive directly.

4.2.2 Preparation and Loss Masking

To prepare the data for training the auto-regressive model, I tokenized the dataset and shifted the labels by one, so that each token learns to generate the subsequent token. I then added a `<pad>` token at the end of labels so the dimensions matched.

However, the model does not need to learn to generate the next goal state, but rather just which tactics to apply given a goal state. In fact, attempting to force the model to learn to generate the next goal state resulted in notably poor performance when it came time to tactic generation. To mitigate this, I performed loss masking on any token within comments (notably, any goal state or imported context).

To ensure that the characters of the comment delimiters would always be tokenized together, I added `(*` and `*)` as special tokens to the tokenizer. Then, after resizing the model embeddings to fit the new special tokens, I stepped through all the tokens in the dataset until the beginning comment token `(*` was found. I then masked the loss of all tokens until the end comment token `*)` was found, setting all `token_ids` of the labels within comments to -100, which is an indication to Pytorch to ignore the cross-entropy loss at that token. Therefore, with this system in place, the model may still learn from the context of the goal states and imported modules but will not learn to generate them. This modification resulted in a substantial improvement between training and testing iterations of the fine-tuned model.

4.2.3 Data Chunking

Often the data files are much longer than the context of the model (4092 tokens), so the data must be truncated somewhere for the model to accept the input sequence on which to train. After tokenizing the entire file, I then split the data into equal-sized chunks of 4092 tokens. Some chunks, however, may be completely contained within comments (if the entire chunk was an imported module), so I omitted chunks fully within comments from train and test data to reduce unnecessary computation. 1,688 chunked files amounted to 104,902,656 tokens in the final dataset, including final padding tokens.

4.2.4 Training Runs

Using `Trainer` from the Transformers library [14], I trained the models on the prepared data. I used the Adam optimizer [6], specifically `paged_adamw_32bit` (for QLoRA), with a learning rate of 2E-5. Initially, I tried a learning rate of 2E-4 and found it to be too high, whereas a rate of 2E-5 seemed to stabilize the deviations in training loss.

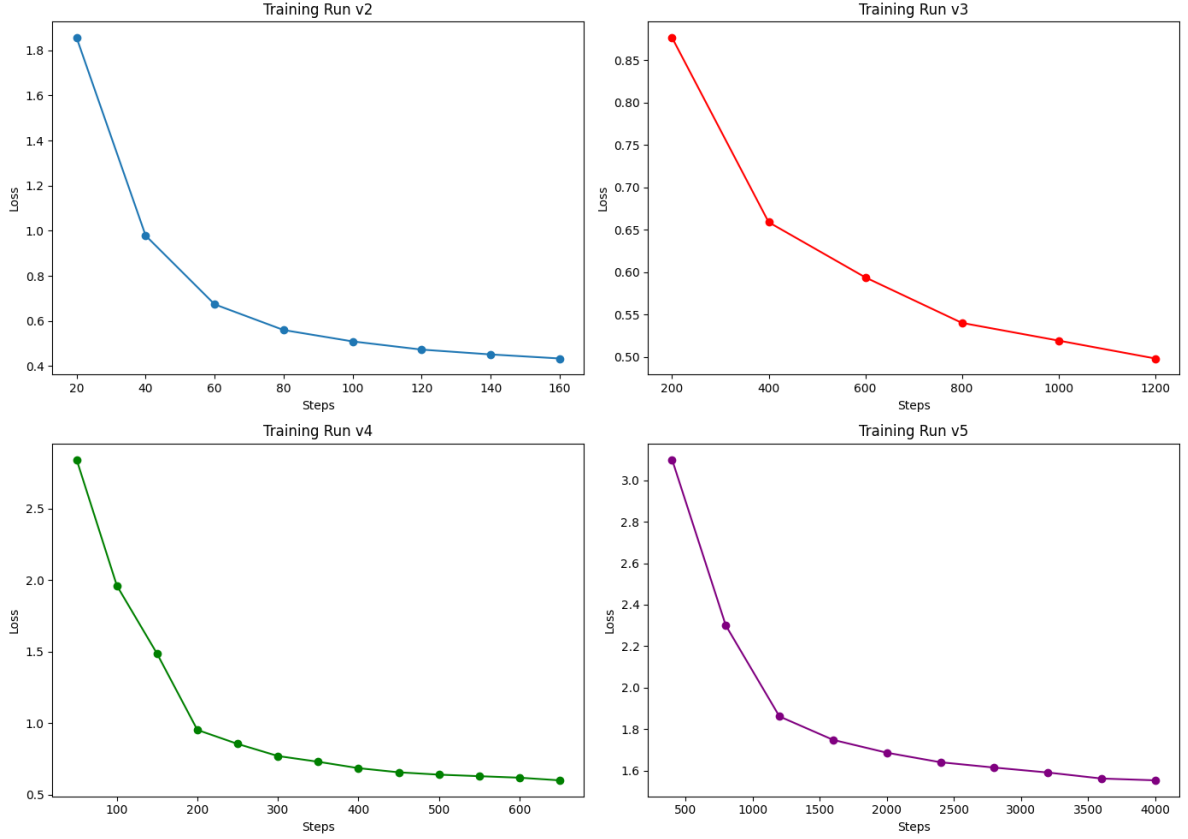


Figure 4.1: Training Steps vs. Val Loss on four training runs

To train on the entire dataset, as in v5, the model took 13 hours and 34 minutes (excluding evaluation and checkpointing) on 1xNVidia A100 to complete. After training the quantized model on the data, I merged the low ranked adapters with the base model and saved the model to Drive. For the final models, I uploaded them to Hugging Face for deployment.

4.3 Model Inference

Once a model was ready to be deployed, I created a private HuggingFace endpoint for the model, initializing it on GPUs (either 1xNVidia A100 or 4xNVidia T4) for faster inference. Unlike training, where the model was quantized, I used the full 32-bit floating point parameters for inference.

4.3.1 Custom Handler

Due to special additional features my approach requires, I abstained from using the premade handler and made my own `handler.py` file. First, I initialize the model and tokenizer on the GPU. Next, upon a call to the endpoint, I tokenize the input data, but because the query context is often longer than the model’s context size, I truncate input ids to fit within the context size. I also leave 75 additional empty tokens at the end so the model has space for generation, and stop generation after 75 tokens. This is enough room for any reasonable tactic generation any generated results longer than this are almost always invalid tactics.

Custom Stopping Criteria

Because I only sought for the next tactic to be generated, I added a function that checked if the previous generated token included a period, indicating the end of a tactic. It does this by decoding the most recently generated token after each step and stopping generation of that token if it includes a period. This approach worked well to speed up inference time; however, there was a case where the model tried to generate a valid tactic that included a period. This tactic, `destruct Archi.ptr64 eqn:SF`, is used to separately consider cases whether the target architecture uses 64-bit pointers (true) or not (false). These types of tactics are rare, and the benefit of using the custom stopping criteria likely outweighs the drawbacks of this type of scenario.

```

Theorem eval_addrstack:
  forall le ofs,
    exists v, eval_expr ge sp e m le (addrstack ofs) v /\
    Val.lessdef (Val.offset_ptr sp ofs) v.
Proof.
  intros.
  unfold addrstack.
  unfold eval_addressing; destruct Archi.ptr64 eqn:SF;

```

Highlighted part are the tokens not included due to the stopping criteria, resulting in an invalid tactic and error.

“Bad Words”

The `generate` function has an optional argument to prevent certain combinations of tokens from being generated, to avoid models outputting certain “bad words”. The original impetus of this is to prevent harmful outputs, but in this case I can leverage it in two ways.

I modified the `__call__` function to return a dictionary that includes the token ids of the output in addition to the generated text. In doing so, I could perform deeper analysis on the generation as well as include the previously generated token ids as “bad words” in the subsequent model inference.

First, I prevented certain keywords that would allow the Solver to avoid completing proofs, such as “`Admitted.`” or “`Abort.`” Tactics like these allow a user, or the ATP, to leave a proof incomplete. Second, I added an optional argument `additional_bad_word_ids` that allows the server to pass in a set of ids that it will avoid generating. This is key to prevent the model from generating tactics that it has already generated at a given step in the tree, so that it does not attempt generating the same tactics each time. This strategy was preferable to returning multiple sequences from beam search with a higher temperature. The `temperature` parameter increased the randomness of each token being selected, increasing the “creativity”

of the model’s outputs. However, a higher temperature raised the likelihood of the model outputting incorrect tactics. So by maintaining a lower temperature and disallowing tactics that it has already attempted, the model is able to output more coherent tactics without repetitiveness.

4.4 Server

I wrote a FastAPI backend to serve as a bridge between the application and the language models, with endpoints for generating tactics and evaluating proof states. It uses Pydantic for request validation and FastAPI’s asynchronous capabilities so multiple endpoints can be called at once, and it can be deployed with Uvicorn as an ASGI server. The server is configured to allow cross-origin requests from the React frontend, allowing them to be run on the same machine. The Solver submits post requests to the server to vote on proposed proof steps, generate tactics with various models, and evaluate the states of the proof. There is a unique generation or eval endpoint for each model, so the solver can call different models by specifying the url.

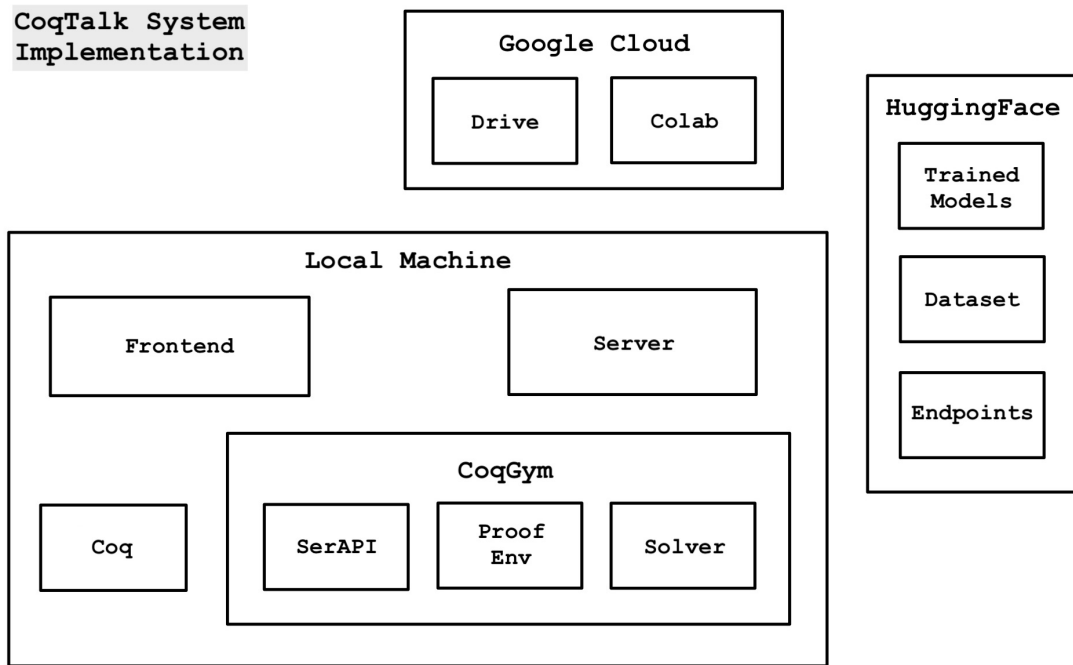


Figure 4.2: A diagram of the system and its interactions

4.5 Tactic Tree

To create the tree with which CoqTalk solves proofs, I created a `TreeNode` class to manage the state of a proof in the theorem-proving environment. Each node corresponds to a state in the proof process, characterized by a tactic applied at that state and the feedback from Coq.

- **feedback:** A dictionary containing feedback from the theorem prover after applying the tactic. This includes information about goals, errors, etc.
- **children:** A list of `TreeNode` instances that are the children of this node, representing subsequent states after applying further tactics.
- **parent:** A reference to the parent `TreeNode`, allowing navigation back up the

tree.

- **eval_score, cost, f_score**: Floats representing the value of reaching this node from the root, used in the search strategies A* or greedy search.
- **error**: An optional attribute to store any error that occurred when applying the tactic.
- **tactic**: A string representing the tactic applied at this node.

The methods of the class include `add_child`, `get_path`, and others for the solver to call and build proofs. The `TreeNode` class provides the building blocks of the proof tree, facilitating the exploration of different proof strategies.

4.6 Solver

The solver manages the proof, interacting with Coq, building the proof tree, and calling the server to generate tactics and evaluate proof states. It is defined in a modification of the `eval_env.py` file included in CoqGym, which creates the environment for interacting with Coq proofs. The solver is run by calling it with an associated JSON file corresponding to a single proof in a Coq file from CoqGym, and begins by setting certain parameters of the proof strategy: `proof_env` for interaction with Coq, model for generation, `eval_model`, `eval_strategy`, `search_strategy`, `root` node, and `current_node` which corresponds to the tactic path currently being explored. In the initialization of Solver, it generates the `base_proof_context` and `proof_statement`. The `base_proof_context` comes from the `query_context` directory, which contains all lines of the corresponding cleaned (free from comments, indentation, and extraneous whitespace as described in the Data section) Coq proof up until the declaration of the lemma or theorem. The `proof_statement` is the declaration of the theorem in question.

4.6.1 Executing Coq Code to Build Proofs

The proof environment in CoqGym interacts with Coq through SerAPI [1], a library that facilitates interaction via code with the Coq proof assistant. Coq tactics may be run by calling `proof_env.step(tactic)`, but the `proof_env` itself may only pursue one path of tactics at a time. Rather than creating multiple `proof_envs` for each tactic path, which can take longer depending on the import requirements of the Coq module, I use the `Undo.` tactic to step back up the tree and then step down to the target node. This is done with two helper functions `return_to_root` and `get_tactic_path(node)`, which step the proof environment to the target node.

4.6.2 Generation and Evaluation Inference

The Solver calls the server endpoints to generate and evaluate tactics and proof states. The `_call_model_generate` function sends the current proof context to the specified generation model, and the server returns a list of two possible tactics based on the proof context. Similarly, the `_call_model_eval` function evaluates the effectiveness of a tactic by sending the updated proof context, after applying the tactic, to the model specified by the `eval_model`. Depending on the `eval_strategy`, the model returns a score from 0.0 to 1.0 representing the tactic’s promise, and the Solver assigns it to the node. [A]

4.6.3 Search Strategies

The two search strategies implemented – A* and Greedy – differ in their approach to selecting from which nodes to generate tactics. The `choose_next_node` function steps the `proof_env` to the node the strategy dictates. Smaller section: A* prioritizes nodes based on a combination of the cost to reach that node (g-score) and an estimate of the cost to complete the proof from that node (the h-score), and stores the possible

nodes in a priority queue sorted by the sum of the two (f-score). The h-score of a node is the `node.eval_score`. After experimentation, I set 0.2 as the step cost, so the g-score ends up being $\text{<depth of node>} \times 0.2$. This step cost seemed to strike a balance between exploration and efficient pathfinding, allowing for a convergence to the solution. On the other hand, the Greedy strategy prioritizes nodes purely based on their immediate evaluation scores, leading to faster but potentially less optimal solutions. The selection of the node for the Greedy strategy is implemented in two ways: the first is just picking the node with the highest `eval_score` in the tree. The second way, the selection strategy, is to take all valid leaf nodes (nodes with no children) and query the evaluation model for which path was most promising, and choose that node. Because both require a call to the evaluation model – one strategy when the node is generated, and the other when the node is being chosen – I picked the one that performed better, which was the selection strategy. [A]

4.6.4 Building Queries

To build the generate query, I use the `get_generate_context` function to take the `base_proof_context`, add the `proof_statement`, and add the tactic path up to the current node. The function takes an optional argument to include the goal state after every tactic or just the last one. I experimented with both, and including only the most recent goal state worked better. To build the evaluate query, the `get_evaluate_context` function works differently depending on which evaluation strategy is being used. If it is the scoring strategy, I include the same context from the generate step and add instructions describing the task to GPT-4. If the evaluation strategy was the selection strategy, I included the `base_proof_context` and the proof states of each of the leaf nodes, numbered 1, 2, etc. I keep track of the index of each node upon the response from the server so that the corresponding node is selected properly.

4.6.5 Adding Nodes and Handling Errors

When a node is added, the proof environment executes the associated tactic, and the solver updates the proof tree. The node's children represent all possible continuations of the proof from that state. If a proof step succeeds, the solver marks the path leading to this node as successful, terminating the search early when a full proof is completed.

Error handling is critical within the Solver. If a generated tactic fails (due to an invalid proof step), the node is marked with an error, and its `eval_score` is set to `-Infinity` to prevent further exploration of this path. The Solver then undoes the tactic to backtrack and explore alternative paths. Additionally, should a node contain children whose tactic resulted in an error, it is added to the proof path as a comment in the form `(* Note: cannot use {child.tactic}, failed with error. *)`. After this modification, the model tended not to repeat errors at the same proof step.

4.6.6 Saving Outputs

Finally, the implementation includes serialization capabilities with `to_json` function, allowing the proof tree and solver state to be saved as a JSON object. This facilitates debugging, testing, and replication of results, as the entire state of the solver is captured, and also allows the proof tree to be examined in the frontend.

4.7 Frontend

To visualize the progression of proof trees, I developed a React-based web application with Chakra UI using TypeScript. The application fetches and processes a proof tree from a JSON file, handling special cases like infinite evaluation scores. It allows a user to navigate through different tactics and their outcomes, displaying the current proof state and available tactic options at each step.

While developing the frontend, I created three versions of the React web app in the

iteration process. The first version is very simple, and can only explore one possible tactic path. It calls GPT-3 on the backend to generate two potential tactics, and then again calls GPT-3 to vote on which one to pursue. It generates and evaluates one step at a time only when the user requests it.

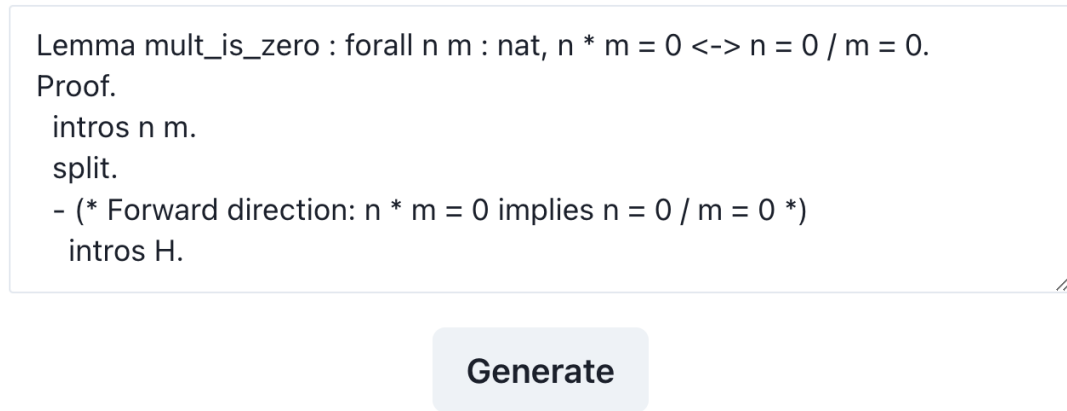


Figure 4.3: Frontend v0: an example proof, mult-is-zero, is shown.

The second version of the frontend explored the idea of navigating through a tree, still requiring user input at each step. It contained the following collapsible options:

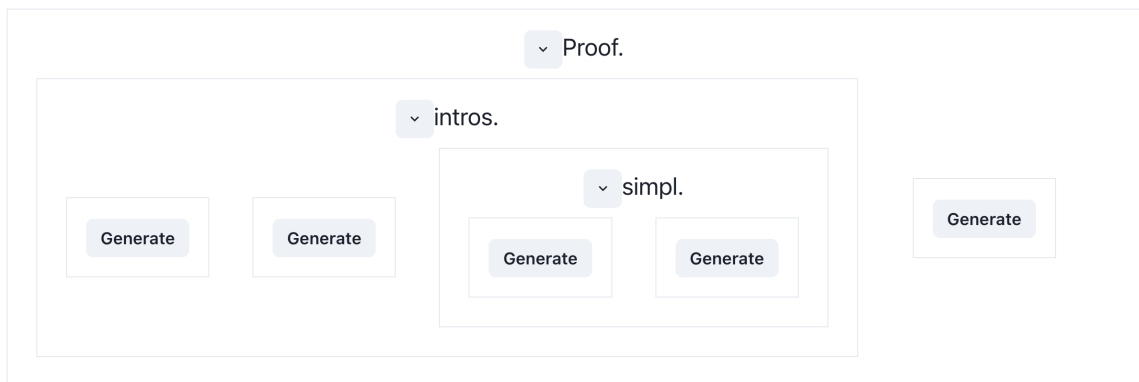


Figure 4.4: Frontend v1: Introduction of tree navigation. An example proof, mult-is-zero, is shown.

The third and final version is cleaner, allowing the user to easily jump around the tree. Each row represents a node in the proof tree, with the ability to select the next

tactic to explore. Contrary to earlier versions, I elected to have the final version not interact with the backend to generate tactics, but rather solely display the result of the automated theorem prover.

PROOF TEXT	TACTIC OPTIONS	PROOF STATE
Proof.	<code>intros [f Hf] [g Hg].</code> <code>intros f g.</code>	<code>forall (_ : @reduces X Y p q) (_ : @reduces Y Z q r), @reduces X Z p r</code>
<code>intros [f Hf] [g Hg].</code>	<code>exists (fun x => g (f x)); intros x.</code> <code>exists (fun x => g (f x)).</code>	<code>@reduces X Z p r</code>
<code>exists (fun x => g (f x)); intros x.</code>	<code>split.</code> <code>apply Hg.</code>	<code>iff (p x) (r (g (f x)))</code>
<code>split.</code>	<code>- intros H; apply Hg; apply Hf; assumption.</code> <code>intros H.</code>	<code>forall _ : r (g (f x)), p x, forall _ : p x, r (g (f x))</code>
<code>- intros H; apply Hg; apply Hf; assumption.</code>	<code>- intros H; apply Hf; apply Hg; assumption.</code>	<i>BG Goals:</i> <code>forall _ : r (g (f x)), p x</code>
<code>- intros H; apply Hf; apply Hg; assumption.</code>		No goals remaining.

Figure 4.5: Frontend v2: a straightforward system of analyzing proof trees. LLEMMA-base using A* search generated this complete proof of reduces-transitive from coq-library-undecidability/Problems/Reduction.v

4.8 Evaluation

The main function serves as the operational core of the evaluation environment. Upon initialization, it parses command line arguments to configure the proof file, output directory, search strategy, and generative model to be used. The function then enters a loop where it iteratively processes each proof in the provided Coq file. For each proof, the function initializes the proof environment and employs the Solver to generate and apply tactics based on the selected search and evaluation strategy, in a loop until `max_new_tactics` is reached, the proof succeeded, or it failed. Successful proofs are logged and saved to the designated output directory. Occasionally, SerAPI raises exceptions if tactics create loops that reach the maximum recursion limit, so I run the proof in a try-except loop that logs it as a failure and continues onto the next proof. To run the evaluations, I set up an `eval_runner.py` script that runs evaluations on a

list of JSON files from the CoqGym validation set of proof files, calling my evaluation environment using subprocesses in Python.

Inferences that occupied the entire model’s context took much longer to run than those that did not, so due to these computational concerns, I ran the evaluation on a combination of randomly selected proofs and proofs whose corresponding `query_context` was shorter than 4096 tokens. This allowed me to evaluate the theorem-prover on more proofs but likely biased for simpler proofs outside the context of much larger Coq projects.

In total, evaluating the models took 2,168 minutes on 1xNVidia A100 and 3,346 minutes on 4xNVidia T4, amounting to a negligible difference in the stock price.

I experimented with multiple state evaluator models: Base Llemma, Fine-tuned LLEMMA, GPT-3 and GPT-4. The best results were from GPT-4, and the worst from the LLEMMA-based models. This is likely due to the nature of the evaluation task being very different from its training data, whereas GPT-4 is better able to understand the specific instructions.

Chapter 5

Results

To generate tactics, I employed three models: LLEMMA base, LLEMMA fine-tuned, and GPT-4. I evaluated these models on 572 proofs from the CoqGym validation dataset, allowing a maximum of 25 queries to generate tactics, and using GPT-4 as the state evaluator. This resulted in 5,437 tactic generation queries across all models. For LLEMMA base and LLEMMA fine-tuned, I evaluated each with an A* search strategy or a greedy search strategy.

5.1 Quantitative Results

I first evaluated the models on a common subset of 72 proofs to more fairly compare approaches. Though the sample size is relatively low, it still functions as a baseline demonstration of each model’s relative capabilities.

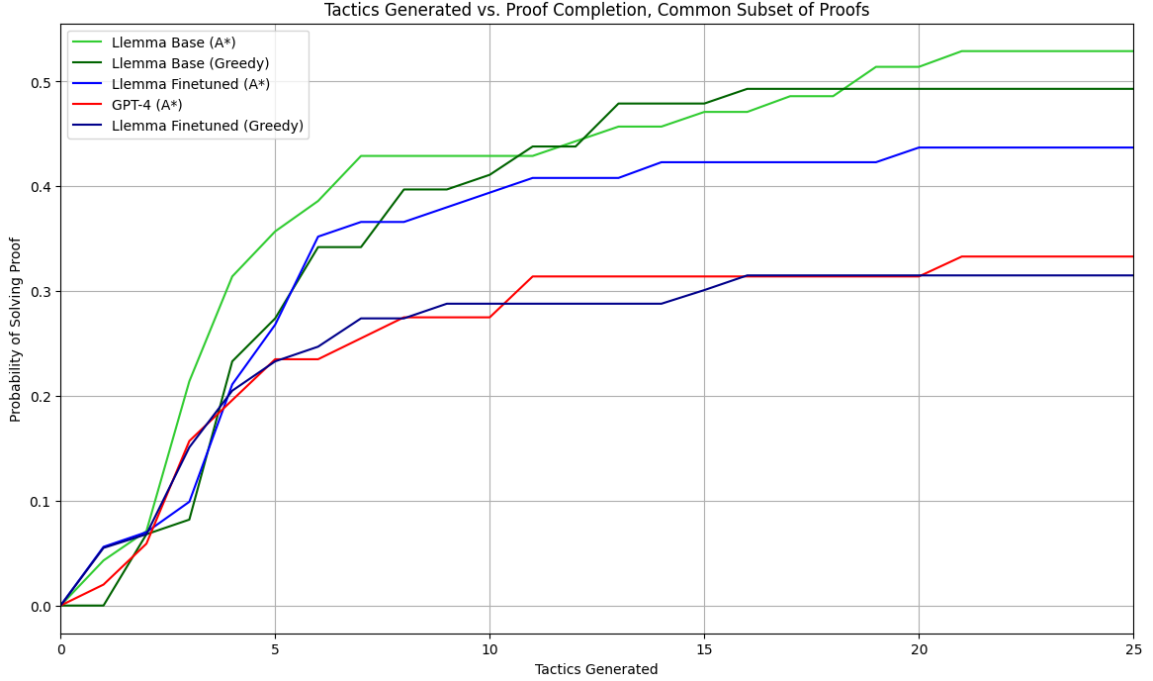


Figure 5.1: Tactics Generated vs. Proof Completion, Common Subset of Proofs

The results in 5.1 indicated that the A* search model outperforms the greedy model as tactics generated increases. This means that while the greedy search strategies rival the A* model earlier on, as more tactics are generated, the A* model sees a higher rate of success, providing evidence for it being the better search strategy.

Table 5.1: Summary of Proof Results on Common Subset

Model	Search Strategy	Proofs Attempted	Success Rate (%)
Llemma Base	A*	69	(53.62%)
Llemma Fine-tuned	Greedy	72	(31.94%)
Llemma Fine-tuned	A*	70	(44.29%)
GPT-4	A*	71	(35.90%)
Llemma Base	Greedy	72	(50.00%)

Note: Small differences in proofs attempted due to errors during evaluation (such as inference endpoints failing).

To get a sense of how the models would perform against a larger set of the proofs

in the validation set, I also graph the performance of each model on all the proofs it was used to solve. The results in 5.2 emphasize the superiority of A* as a search strategy. Once again, early on, the greedy models keep pace with the A* strategy, but as is shown more clearly against a larger set of proofs, models using A* outperform those using a greedy strategy over time.

The effectiveness of a smarter search strategy is encouraging because it underscores the role of sophisticated decision-making processes in the realm of automated theorem proving. The integration of a Tree of Thoughts (ToT) framework with the A* search strategy exemplifies an enhancement in navigating proof paths efficiently. In particular, the superior performance of the A* strategy across a broader dataset confirms its effectiveness in managing the complexities of proof exploration.

This result aligns with the conceptual goals of ToT – searching through a structured decision tree to identify the most promising proof strategies optimally. As LLMs for theorem-proving continue to evolve, their integration into a framework like ToT will empower them to handle more complex proofs, pushing the boundaries of what automated systems can achieve in formal verification.

Table 5.2: Summary of All Proof Results

Model	Search Strategy	Proofs Attempted	Success Rate (%)
LLEMMA Base	A*	69	(53.62%)
LLEMMA Fine-tuned	Greedy	259	(30.89%)
Llemma Fine-tuned	A*	166	(56.02%)
GPT-4	A*	78	(35.90%)
LLEMMA Base	Greedy	73	(49.32%)

On the larger set of proofs, the fine-tuned model outperforms the other models. This is an interesting and potentially promising result. This result supports the notion that the base model may benefit from not only fine-tuning but learning the relationships between tactics and the state of the proof. However, as it was attempted on a larger set of proofs – where the base model’s set was smaller – it may have seen

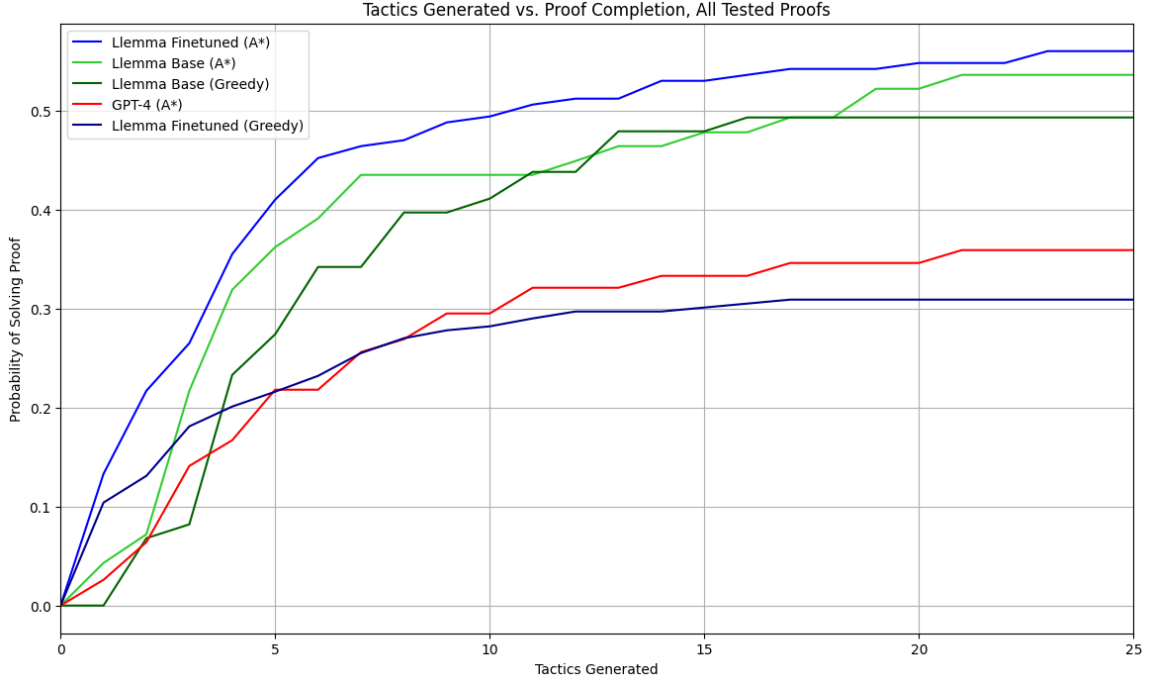


Figure 5.2: Tactics Generated vs. Proof Completion, All Tested Proofs

more proofs that were easier to solve than the LLEMMA base model, so it is not conclusive.

One result that was particularly interesting is how much worse the fine-tuned model employing greedy search performed relative to the other models. This indicates that for some reason, the fine-tuning assisted the model when the search was forced to be more exploratory, yet hurt the model when it pursued tactic paths with a more narrow search focus. It is not obvious why, and is a question for future work.

Table 5.3: Tactic Generation Statistics

Model	Tactics Generated	Errors	Legal Tactics %	Proofs Attempted
Llemma Base	714	189	72.69	142
Llemma Fine-tuned	3845	1607	58.21	425
GPT-4	878	417	52.51	78

Another result that supports this project’s main aims is the superiority of the specialized models over the general purpose GPT-4 in this style of automated theorem

proving. Additionally, GPT-4 produced legal tactics at an the lowest rate (52.51%), however the next lowest rate was still close with the fine-tuned model (58.21%). The LLEMMA base model generated legal tactics at the highest rate (72.69%), evidence of the efficacy of pretraining a model on a large corpus of Coq code. However, the fine-tuned model’s low rate of legal tactic generation was surprising, but not necessarily concerning, as it still proved its effectiveness in automated theorem proving.

5.2 Qualitative Results

One key goal of this project, in the inclusion of proper query context, was to facilitate the model’s ability to call relevant theorems or lemmas from context. Here, in `zorns-lemma/Proj1SigInjective.v`, we see the ability of the fine-tuned model to apply a lemma that was just proven to another proof. This is one of a number of promising examples of the potential for LLMs to understand and apply contextual information in proofs.

zorns-
lemma/Proj1SigInjective.v:

```
Require Import
  ProofIrrelevance.

Lemma subset_eq_compatT:
  forall (U:Type) (P:U->
    Prop) (x y:U)
    (p:P x) (q:P y), x = y ->
      exist P x p = exist P
        y q.
Proof.
  intros.
  destruct H.
  f_equal.
  apply proof_irrelevance.
Qed.

Lemma proj1_sig_injective:
  forall {A:Type} (P:A->
    Prop)
    (a1 a2:{x:A | P x}),
    proj1_sig a1 =
    proj1_sig a2 -> a1 =
    a2.
Proof.
  intros.
  destruct a1.
  destruct a2.
  simpl in H.
  apply subset_eq_compatT;
    trivial.
Qed.
```

Original proof file.

Generated proof for proj1-sig-
injective:

```
Require Import
  ProofIrrelevance.

Lemma subset_eq_compatT:
  forall (U:Type) (P:U->
    Prop) (x y:U)
    (p:P x) (q:P y), x = y ->
      exist P x p = exist P
        y q.
Proof.
  intros.
  destruct H.
  f_equal.
  apply proof_irrelevance.
Qed.

Lemma proj1_sig_injective:
  forall {A:Type} (P:A->
    Prop)
    (a1 a2:{x:A | P x}),
    proj1_sig a1 =
    proj1_sig a2 -> a1 =
    a2.
Proof.
  intros.
  destruct a1 as [x1 p1].
  destruct a2 as [x2 p2];
    simpl in *.
  apply subset_eq_compatT
    with (P:=P) (x:=x1)
    (y:=x2) (p:=p1) (q
      :=p2); assumption.
Qed.
```

Generated proof, applying a lemma from context.

This style of integration, wherein the LLM recalls and utilizes elements from its immediate textual context, is crucial for the advancement of automated theorem proving. The example above shows the model's ability both understand the structure

and requirements of Coq proofs and apply recently proven lemmas in new contexts effectively. This demonstrates the model’s contextual awareness, and further LLMs’ potential to function as an autonomous aid in more complex theorem proving scenarios.

As a whole, these results are encouraging as they suggest that with further refinement, LLMs can become even more adept at handling the nuances of proof construction, making them valuable tools in the domain of formal verification and beyond.

Chapter 6

Conclusions and Future Work

6.1 Future Work

The completion of the project raised several areas for future improvement. Future iterations might consider a deep integration with the token generation process of the LLM, by streaming the model output and running the Coq code as it generates, only stopping a generation once a tactic fails.

Other modifications might employ vector-based semantic search for data retrieval, enhancing the system’s ability to determine relevant content for inclusion, and helping mitigate the problem of small context sizes.

Regarding context sizes, future work will increase the context size of these LLMs, which can improve the model’s ability to recall more definitions or theorems. Additionally, as the baseline capabilities of these models improve, the ability of ATPs that employ LLMs to mimic or even surpass a human user will only improve.

6.2 Discussion

This project accomplished the overarching goals it set out to, demonstrating the efficacy of combining specialized LLMs with ToT reasoning techniques for Coq proofs

and providing compelling evidence that this is an area that should and will continue to be explored.

By leveraging the strengths of the LLEMMA model, initially pretrained on a vast corpus including Coq code, and fine-tuning it using modern techniques on a custom dataset, this project built a applicable tool to adjust to the nuances of Coq’s logical structure. GPT-4’s role as a state evaluator underscores the capacity of state-of-the-art language models to contribute meaningful insights during the proof verification process.

The use of the Tree of Thoughts (ToT) reasoning framework to guide the exploration and evaluation of proof strategies has been particularly impactful. This approach enabled the efficient navigation through the complex decision space of a proof, evident from the superior performance of the A* search strategy over the greedy approach in the evaluation. These results validate the effectiveness of integrating a thoughtful, model-based search strategy and the potential for such technologies to push the field forward.

This project also addressed critical aspects of context management and query optimization, ensuring that each proof is approached with the context needed to understand its goals and how to prove them. Additionally, the multifaceted system architecture facilitated interactions between the theorem-prover, the models, and the Coq environment.

The quantitative and qualitative results obtained confirm that the enhancements made to the theorem proving process through this approach are viable and advantageous. This project suggests that further exploration and refinement of these techniques could lead to far more sophisticated automated theorem proving systems. The results emphasize the importance of the evolution of smart automated theorem proving with ML techniques, opening avenues for better systems in the future.

Appendix A

Prompts

```
INSTRUCT_ZERO_TO_ONE_PROMPT = """...
{base_proof_context}
-----
Above is the context for a Coq proof. You are an evaluation
  model, and you score the proof from 0 to 1, where closer to
    0 means it's far from the end of the proof, and closer to
    1 means it is on the right track and close to being proven.
  For example, if the proof seems to not be making progress,
    give it a low score. Here is the current state of the
  proof:
---
{proof_statement}
{path}
---
Output, followed by a period and nothing else, the score of
  this proof from 0 to 1.
"""
```

```

EVAL_SELECTION_PROMPT = """...
{base_proof_context}
-----

Above is the context for a Coq proof. You are an evaluation
    model, and you select which option is closest to completing
        the proof. Here are the options:
---

{choices}

Output, followed by a period and nothing else, the number of
    the option you select.
"""

choice_format = "\n#{option_number}:\n{path}\n---"
choices_formatted = ""
for index, (_, node) in enumerate(self.open_set.queue, start=1):
    node_path = "\n".join(node.get_path(only_include_last_goals=True))
    choices_formatted += choice_format.format(index, node_path)
proof_file_txt = EVAL_SELECTION_PROMPT.format(base_proof_context,
        proof_statement, choices_formatted)

```

Bibliography

- [1] E. J. G. Arias. `ejgallego/coq-serapi`, Apr. 2024. `original-date: 2016-05-26T04:22:15Z`.
- [2] Z. Azerbayev, H. Schoelkopf, K. Paster, M. D. Santos, S. McAleer, A. Q. Jiang, J. Deng, S. Biderman, and S. Welleck. Llemma: An Open Language Model For Mathematics, Nov. 2023. `arXiv:2310.10631 [cs]`.
- [3] L. Blaauwbroek, J. Urban, and H. Geuvers. Tactic Learning and Proving for the Coq Proof Assistant. pages 138–124. `arXiv:2003.09140 [cs]`.
- [4] L. Blaauwbroek, J. Urban, and H. Geuvers. The Tactician. In C. Benz Müller and B. Miller, editors, *Intelligent Computer Mathematics*, Lecture Notes in Computer Science, pages 271–277, Cham, 2020. Springer International Publishing.
- [5] E. First, M. Rabe, T. Ringer, and Y. Brun. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 1229–1241, New York, NY, USA, Nov. 2023. Association for Computing Machinery.
- [6] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization, Jan. 2017. `arXiv:1412.6980 [cs]`.

- [7] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. CompCert - A Formally Verified Optimizing Compiler.
- [8] H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, K. Cobbe, and I. Sutskever. Let’s Verify Step by Step.
- [9] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan,

J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O’Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. d. A. B. Peres, M. Petrov, H. P. d. O. Pinto, Michael, Pokorny, M. Pokrass, V. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. J. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph. GPT-4 Technical Report, Dec. 2023. arXiv:2303.08774 [cs].

- [10] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code Llama: Open Foundation Models for Code, Jan. 2024. arXiv:2308.12950 [cs].

- [11] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2020, pages 1–10, New York, NY, USA, June 2020. Association for Computing Machinery.
- [12] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, Oct. 2023. arXiv:2303.11366 [cs].
- [13] A. Thakur, G. Tsoukalas, Y. Wen, J. Xin, and S. Chaudhuri. An In-Context Learning Agent for Formal Theorem-Proving, Feb. 2024. arXiv:2310.04353 [cs].
- [14] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. Transformers: State-of-the-Art Natural Language Processing, Oct. 2020. Pages: 38–45 original-date: 2018-10-29T13:56:00Z.
- [15] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models, Dec. 2023. arXiv:2305.10601 [cs].
- [16] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing Reasoning and Acting in Language Models, Mar. 2023. arXiv:2210.03629 [cs].