
LEARNING TO PLAN BY LOOKING BACK: HINDSIGHT HIERARCHIES FOR TRAINING REASONING MODELS

A PREPRINT

Lars Simon
Bundesdruckerei GmbH
Berlin, Germany
lars.simon@bdr.de

Holger Eble
Bundesdruckerei GmbH
Berlin, Germany
holger.eble@bdr.de

Manuel Radons
Bundesdruckerei GmbH
Berlin, Germany
manuel.radons@bdr.de

ABSTRACT

We introduce a self-improvement loop for reasoning models based on the following observation: Even when the difficulty of a problem exceeds the model’s current solving abilities, an additionally supplied solution might enable the model to extract useful solution ideas in hindsight. We operationalize this by jointly training the same model to exhibit the following three capabilities: predicting solution ideas from problems alone, reverse-engineering ideas from problems and known solutions, and solving problems using provided ideas. The loop alternates between reverse engineering such ideas from problems with supplied solutions and using these ideas as additional supervision for joint training of all three capabilities. We give a formal specification of our method and a concrete instantiation for interactive theorem proving in the Lean theorem prover; empirical evaluation remains future work.

1 Introduction

Our method is inspired by the following asymmetry in problem solving: It is, in general, harder to solve a problem from scratch than to reverse engineer the idea underlying a known solution. For a reasoning model, this presents a learning opportunity: even when a problem exceeds its current solving abilities, a supplied solution may enable it to extract a useful idea for solving that problem in hindsight. Such a reconstructed idea can then provide additional supervision for learning to develop a solution idea from the problem alone, before attempting to solve it. The opportunity is therefore to *learn from ideas that the model can already reverse engineer, but may not yet be able to discover unaided*.

We operationalize this observation by jointly training a model on three tasks. A *hindsight component* reverse engineers solution ideas from problems and known solutions. A *foresight component* predicts such ideas from problems alone. A *solver component* uses problems and provided ideas to construct solution candidates. All three components share a backbone network; for example, when their inputs and outputs can be represented as text, we may use a shared language model with role-specific prompt templates.

The centerpiece of our method is an alternating training loop. In each iteration, the current hindsight component reconstructs ideas for a batch of problems with supplied solutions. We then jointly train all three components using the resulting data together with previously collected examples. Concretely, given a problem x with solution y and reverse-engineered idea z , the foresight component is trained to predict z given x , the hindsight component is trained to predict z given x and y , and the solver component is trained to predict y given x and z . The next iteration uses the updated hindsight component to generate ideas. Thus, the capability responsible for producing additional supervision is itself an explicit training target. Figure 1 illustrates the central idea.

We express solution ideas at several levels of detail, forming a *hierarchy* from overarching strategies to detailed plans. Hindsight starts from a known solution, reconstructs a detailed plan, and compresses it into coarser ideas. Foresight proceeds in the opposite direction, predicting a coarse idea from the problem and progressively refining it. During inference, the solver uses the hierarchy predicted by the foresight component to construct a solution candidate. Figure 2 illustrates these directions and the use of hierarchies during inference. The ideas reverse-engineered by the hindsight component are intended to provide guidance that could plausibly be followed from the problem alone, rather than

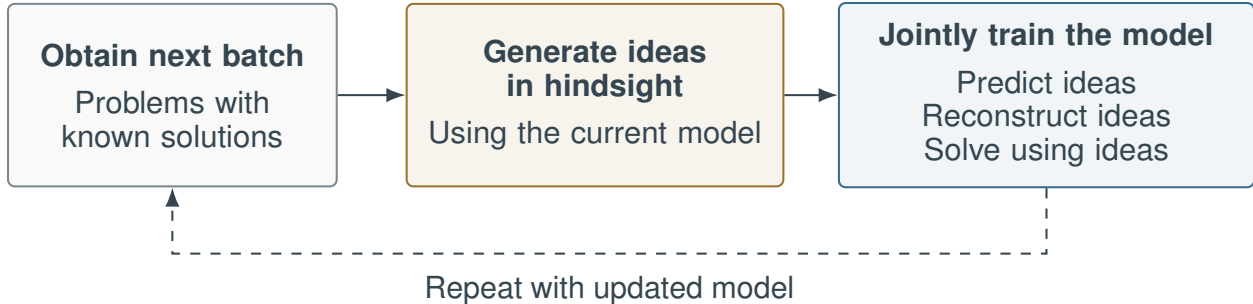


Figure 1: Simplified schematic of the training loop. Each iteration generates hindsight ideas for a new batch and jointly trains the same model to predict, reconstruct, and use ideas.

merely describe the supplied solution or reflect *hindsight bias*. In our Lean instantiation, we seek to avoid the latter through explicit instructions in the language model’s prompt templates.

One possible weakness of our method lies in the fact that the hindsight component is trained on its own outputs. While this might be partially mitigated by the fact that the three components share a backbone network, it would be desirable to select among reverse-engineered hierarchies in order to introduce an additional training signal for the hindsight component in the spirit of expert iteration [1]. We provide an optional refinement of our method which accomplishes this in the case of *verifiable problem domains*, i.e., when proposed solution candidates can be procedurally verified: For each problem and reference solution, we generate several candidate hierarchies in hindsight. The solver then repeatedly attempts the problem using each hierarchy, while the reference solution is withheld. We then compute a score for each hierarchy, based on its success ratio under the repeated solver attempts and, optionally, domain-specific quantities. We retain the highest-scoring hierarchy and an associated valid solution for training, provided the score meets an acceptance threshold.

We provide a formal specification of our method for a general problem setting, together with the optional refinement for verifiable problem domains. We also give a concrete instantiation for interactive theorem proving in the Lean theorem prover [2], where natural language proof ideas guide tactic generation during proof search. The method is implemented in an accompanying software project; empirical evaluation remains future work.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces the problem setting, and Section 4 specifies the general method. Section 5 presents the Lean instantiation, with implementation details in Appendix A. Section 6 discusses limitations and directions for empirical evaluation.

2 Related work

Language models for interactive theorem proving. Language models have been used for proof-step generation and guided proof search [3–5]. Further work incorporates premise retrieval [6] and integration with automated theorem provers [7]. Recent Lean provers also explore large-scale synthetic proof data and reinforcement learning from proof assistant feedback [8–11]. Our Lean instantiation builds on language-model-guided proof search, conditioning tactic generation on learned hierarchies of proof ideas.

Planning and hierarchical reasoning. Explicit planning and problem decomposition organize reasoning with language models into intermediate steps and subproblems [12–14], while tree-based search explores alternative reasoning paths [15]. In formal theorem proving, informal proofs can guide the construction of formal proof skeletons whose gaps are filled by an automated prover [16]. Modular and recursive approaches construct proofs through intermediate lemmas and subgoals [10, 17, 18]. Our hierarchies are intended to describe solution ideas at different levels of granularity, providing guidance for subsequent solving.

Iterative self-training and verifier-guided selection. Expert iteration uses a learned policy to guide search and then trains that policy to imitate the decisions produced by search [1], leading to an iterative improvement loop. Related self-training methods learn from selected model-generated solutions [19, 20]. In formal theorem proving, systems interleave proof search with training on discovered proofs [5, 21], while self-play approaches additionally generate new conjectures [22]. Learned correctness estimators, trained on final outcomes or individual reasoning steps, also support

candidate selection at inference time [23, 24]. In the case of verifiable problem domains, our method uses separate solving attempts to assess reverse-engineered hierarchies and selects training triples based on that assessment.

Retrospective supervision and hierarchical guidance. STaR iteratively trains on self-generated rationales, including rationales constructed retrospectively using correct-answer hints that are removed before fine-tuning [25]. On-policy self-distillation transfers next-token distributions conditioned on a reference solution to predictions made without access to that solution [26]. Lean-STaR uses a teacher language model to retrospectively annotate proof states and known next tactics with natural language thoughts. It then trains a model to generate thoughts and tactics, followed by expert iteration on its own successful proof trajectories [27]. DeepInsight uses an external language model to extract core techniques and proof sketches from reference proofs, and trains a model through progressive multi-stage supervised fine-tuning and hierarchy-aware policy optimization [28].

Our method makes retrospective generation of hierarchies (i.e., rationales at several levels of detail) an explicitly trained task. A shared backbone is *jointly* trained on three tasks: reverse engineering hierarchies from problems and reference solutions; predicting such hierarchies from problems alone; and solving problems using these hierarchies. Each retained training triple supervises all three tasks. The training loop alternates between generating training triples by reverse engineering hierarchies and jointly training on the above three tasks. Consequently, subsequent retrospective hierarchy generation uses the updated model, with the aim of improving the hierarchies available for further training. This differs from the forward generation of thoughts and tactics in Lean-STaR’s expert iterations, and from DeepInsight’s retrospective rationale generation using a teacher model: our loop repeatedly performs retrospective hierarchy generation with the same backbone that learns to predict and use the resulting hierarchies. Finally, in verifiable problem domains, we additionally assess the reverse-engineered hierarchies through separate solving attempts, in which the model receives the problem and hierarchy while the reference solution is withheld. We then select training triples based on this assessment, introducing an additional training signal through data selection, in the spirit of expert iteration [1].

3 Problem setting

Let $\mathcal{X}$ be a set of *problem instances* and let $\mathcal{Y}$ be a set of *candidate solutions*. Write $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{Y}$ for the relation specifying valid solutions. A problem instance can admit no solution, exactly one solution, or more than one solution. We do not initially assume that membership in $\mathcal{R}$ can be decided by an available algorithm.

The Problem. Given $x \in \mathcal{X}$, construct $y \in \mathcal{Y}$ such that $(x, y) \in \mathcal{R}$.

Training starts from a collection of supplied pairs $(x, y) \in \mathcal{X} \times \mathcal{Y}$, where y is a reference solution for x . The idealized description assumes that these are valid solutions, i.e., $(x, y) \in \mathcal{R}$. The training procedure can still be executed with imperfect reference solutions, but – without a verifier – the quality of these reference solutions is an assumption about the source of the data.

3.1 Verifiable problem domains

In a *verifiable problem domain*, we assume access to a deterministic verification procedure

$$\text{Verify}: \mathcal{X} \times \mathcal{Y} \rightarrow \{0, 1\},$$

which, on every input (x, y) we consider, terminates and returns 1 if and only if $(x, y) \in \mathcal{R}$, i.e., if and only if y is a valid solution for x . This allows us to repeatedly query Verify in our algorithm, see Section 4.

Remark. By a slight abuse of notation, Verify refers to both a procedure and the set-theoretic map computed by this procedure, which is just the indicator function $\mathbf{1}_{\mathcal{R}}$ of $\mathcal{R}$ on $\mathcal{X} \times \mathcal{Y}$. Since $\mathbf{1}_{\mathcal{R}}$ is guaranteed to exist without further assumptions on the problem domain, the key requirement is access to a deterministic procedure for evaluating $\mathbf{1}_{\mathcal{R}}$ that terminates on all considered inputs.

Example: interactive theorem proving. We focus on interactive theorem provers that support step-by-step execution of proof commands (tactics), such as Lean. From a current proof state with at least one open goal a single tactic is provided to the interactive theorem prover, which either responds with a successor state or signals failure.

Let $\mathcal{S}$ denote the set of valid (i.e., accepted by the prover) proof states with at least one open goal and let $\mathcal{A}$ denote the set of tactics¹. For the sake of simplicity, we identify all valid proof states with no open goals and denote them as $\top$. Similarly, we identify all error states and denote them as $\perp$. Consequently, the set of all states is $\bar{\mathcal{S}} = \mathcal{S} \cup \{\top, \perp\}$.

¹We restrict $\mathcal{A}$ to tactics that do not admit unproved obligations or introduce disallowed axioms. In Lean, this excludes, for example, `sorry` and `admit`, including their use within other tactics.

The interactive theorem prover then provides a transition function²

$$\mathcal{T}: \bar{\mathcal{S}} \times \mathcal{A} \rightarrow \bar{\mathcal{S}}, \quad \mathcal{T}(\top, a) = \mathcal{T}(\perp, a) = \perp \text{ for all } a \in \mathcal{A},$$

where $\mathcal{T}(s, a)$ denotes the successor state when tactic $a \in \mathcal{A}$ is executed in state $s \in \bar{\mathcal{S}}$. For $s \in \mathcal{S}$, $a \in \mathcal{A}$, execution of tactic a in state s either results in an error (iff $\mathcal{T}(s, a) = \perp$) or produces a valid successor state; in the latter case, $\mathcal{T}(s, a) = \top$ iff no goals remain, otherwise $\mathcal{T}(s, a) \in \mathcal{S}$.

We now instantiate interactive theorem proving as a special case of verifiable problem domains: Letting

$$\mathcal{X} := \mathcal{S}, \quad \mathcal{Y} := \bigcup_{k=1}^{\infty} \mathcal{A}^k,$$

we define $\text{Verify}: \mathcal{X} \times \mathcal{Y} \rightarrow \{0, 1\}$ by imposing that, for $s_0 \in \mathcal{S}$, $(a_1, \dots, a_k) \in \mathcal{A}^k$, we have $\text{Verify}(s_0, (a_1, \dots, a_k)) = 1$ iff $s_k = \top$, where $s_{j+1} = \mathcal{T}(s_j, a_{j+1})$ for $j = 0, \dots, k-1$.

4 Method

Building on the key ideas introduced in Section 1, we now give a formal description of our method. We first describe the shared architecture and training loop, parameterized by a procedure for constructing hindsight training triples. We then specify two such procedures: one applicable to the general problem setting introduced in Section 3, and one tailored to the verifiable problem domains introduced in Section 3.1. We deliberately leave domain-specific choices open in this section; Section 5 provides a concrete instantiation of our method for interactive theorem proving in Lean.

4.1 Hierarchies

With $\mathcal{X}$ and $\mathcal{Y}$ as in Section 3, we fix a depth $n \in \mathbb{Z}_{\geq 0}$, and introduce *hierarchy sets* $\mathcal{Z}_0, \dots, \mathcal{Z}_n$. An element $z = (z_0, \dots, z_n) \in \mathcal{Z} := \mathcal{Z}_0 \times \dots \times \mathcal{Z}_n$ will be called a *hierarchy*. In practice, a hierarchy $z \in \mathcal{Z}$ for a problem $x \in \mathcal{X}$ is intended to represent a *semantically meaningful* idea, plan, or approach for constructing a solution $y \in \mathcal{Y}$ for x at different levels of granularity: From a detailed solution sketch z_0 to a coarse description of the overarching idea z_n . How the hierarchy sets are chosen and how the different levels of granularity are encouraged are problem-dependent; our choices in the special case of interactive theorem proving in Lean are detailed in Sections 5.1 and 5.3, respectively.

4.2 System components

The system, parametrized by a collection of parameters θ , consists of three components. While these components may have component-specific parameters, they share a backbone network which accounts for the majority of parameters (note that θ denotes the collection of all parameters: shared backbone and, possibly, component-specific parameters). The three components are

- a **solver component** S_θ which, given a problem $x \in \mathcal{X}$ and a hierarchy $z \in \mathcal{Z}$, produces a solution candidate $\hat{y} \in \mathcal{Y}$; we write $\hat{y} \sim S_\theta(\cdot \mid x, z)$,
- a **foresight component** F_θ which, given a problem $x \in \mathcal{X}$, iteratively predicts a hierarchy $\hat{z} \in \mathcal{Z}$ from coarsest to finest; we write $\hat{z} \sim F_\theta(\cdot \mid x)$. More precisely, F_θ is defined by a family $(F_{j,\theta})_{j=0,\dots,n}$ via $\hat{z}_n \sim F_{n,\theta}(\cdot \mid x)$ and $\hat{z}_j \sim F_{j,\theta}(\cdot \mid x, \hat{z}_{j+1})$ for all $j \in \{0, \dots, n-1\}$,
- a **hindsight component** H_θ which, given a problem $x \in \mathcal{X}$ with solution $y \in \mathcal{Y}$, iteratively reverse engineers a hierarchy $\tilde{z} \in \mathcal{Z}$ from finest to coarsest; we write $\tilde{z} \sim H_\theta(\cdot \mid x, y)$. More precisely, H_θ is defined by a family $(H_{j,\theta})_{j=0,\dots,n}$ via $\tilde{z}_0 \sim H_{0,\theta}(\cdot \mid x, y)$ and $\tilde{z}_j \sim H_{j,\theta}(\cdot \mid x, \tilde{z}_{j-1})$ for all $j \in \{1, \dots, n\}$.

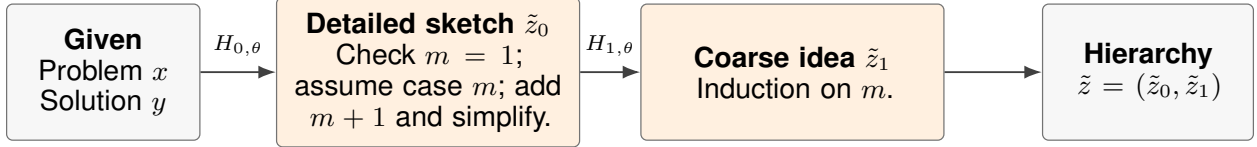
Figure 2 illustrates the opposite directions of hindsight and foresight, and how the solver uses the predicted hierarchy during inference, for a simple mathematical example.

In practice, the system may be implemented using a multi-headed neural network, or (in the case where elements of $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ can be represented by text) using a language model with role-specific prompt templates or special tokens indicating the three roles, e.g., $\langle \text{SOLVE} \rangle$, $\langle \text{FORE} \rangle$, and $\langle \text{HIND} \rangle$. Our choices in the special case of interactive theorem proving in Lean are detailed in Section 5.2.

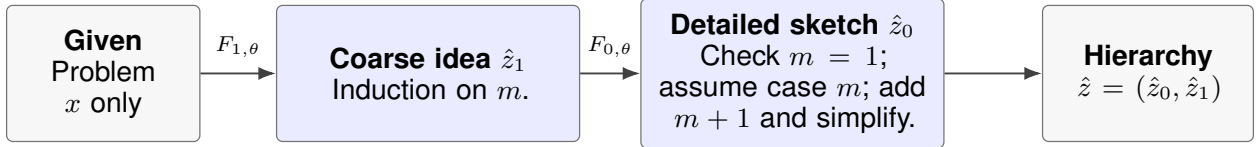
²As before, we assume access to a deterministic procedure for evaluating this function that terminates on all inputs we consider. In Lean, tactic execution need not terminate; even kernel type checking can fail to terminate on certain contrived inputs. As a practical fallback, we can impose timeouts in the implementation, treating a timeout as rejection.

Illustrative problem x : prove $\sum_{i=1}^m i = m(m+1)/2$ for $m \in \mathbb{Z}_{\geq 1}$.

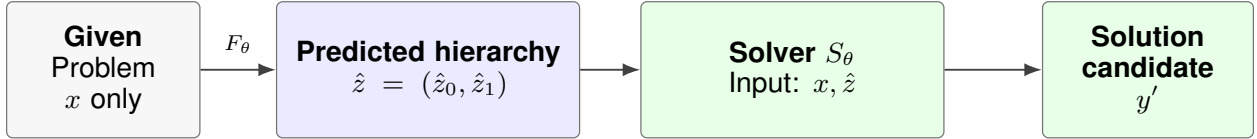
(a) Hindsight: reverse engineer detailed idea, then compress



(b) Foresight: predict coarse idea, then refine



(c) Inference: generate a hierarchy in foresight, then solve



All foresight and hindsight predictions are conditioned on x . Hindsight, foresight, and solving share model parameters θ .

Figure 2: Hierarchy generation in both directions (two levels, $n = 1$) and inference. The solver receives the full predicted hierarchy. The example texts are hand-written illustrations, not model outputs or empirical observations.

Note that conditioning the three components on x is necessary: For example, in the context of interactive theorem proving, assume the idea at the coarsest hierarchy level z_n is “induction on vector space dimension”. Of course, for $n \geq 1$, it is not possible for the foresight component to refine this to a detailed solution sketch z_0 without knowledge of the problem x to be solved.

In interactive theorem proving, the solver component S_θ denotes the entire search procedure, including both tactic generation and proof search. This procedure may return a failure outcome instead of a finite tactic sequence. Formally, we can accommodate this by adjoining the failure outcome to $\mathcal{Y}$ and declaring it invalid for every problem instance.

Inference. We describe the inference procedure of our system for fixed parameters θ : Given a problem instance $x \in \mathcal{X}$, we first generate a hierarchy $\hat{z} \sim F_\theta(\cdot \mid x)$ using the foresight component. Subsequently, we use the solver component to produce a solution candidate $y' \sim S_\theta(\cdot \mid x, \hat{z})$.

In the *general problem setting*, when no verifier is available, we simply return y' .

In a *verifiable problem domain*, we evaluate $\text{Verify}(x, y')$ to check whether y' solves x ; in case of success we return y' , else we signal failure. At test time, we may increase success probability (at the cost of additional compute) by running a nested loop: we sample multiple hierarchies in an outer loop and, for each hierarchy, sample multiple solution candidates in an inner loop; verification is performed immediately after each candidate is generated, and the procedure terminates as soon as a solution for x is found (or a predetermined budget is exhausted).

4.3 Ingredients for training the system

We require four more ingredients before describing the full training loop, namely a *seed dataset*, a way to *obtain additional data*, a way to *construct training triples in hindsight*, and a *training operator* for training the three components on datasets consisting of triples in $\mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$. Here we give an abstract description of these four ingredients and refer to Section 5 for our concrete choices in the special case of interactive theorem proving in Lean. The procedures introduced in this subsection are not assumed to be deterministic; they may be randomized or approximate.

Seed dataset. We assume access to a *seed dataset* D_0 of triples in $\mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$. For each entry (x, y, z) of D_0 , y is a reference solution for x and z is intended to be a hierarchy for constructing y from x in the abstract sense explained

in Section 4.1; in verifiable problem domains, we moreover require that $\text{Verify}(x, y) = 1$. In practice, most available datasets consist of pairs in $\mathcal{X} \times \mathcal{Y}$, making it necessary to augment data points of the form (x, y) with *seed hierarchies*. How the latter can be obtained is problem-dependent; in the case where elements of $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ can be represented by text, an existing language model can be used to reverse engineer seed hierarchies for such data points. The seed dataset may also be empty, e.g., when a pretrained model already provides a suitable initialization for the three components.

Obtaining additional data. The full training loop requires additional data beyond the seed dataset D_0 , sourced from an external dataset or generated synthetically. Crucially, we only need data points of the form $(x, y) \in \mathcal{X} \times \mathcal{Y}$, where y is a reference solution for x , since the hindsight component will be used to reverse engineer hierarchies for such data points. In verifiable problem domains, we additionally require that $\text{Verify}(x, y) = 1$ for every such data point.

We assume access to a procedure which, in iteration t , returns a batch $\text{SampleBatch}(t)$ of such pairs. This procedure may depend on, e.g., the current model parameters θ , previously collected data, external datasets, synthetic data generators, and resource budgets; such dependencies are suppressed in the notation.

In verifiable problem domains, one optional refinement is to enforce that $\text{SampleBatch}(t)$ has prescribed proportions of problems from different difficulty groups. These difficulty groups can be defined by observed success under different inference budgets, using the system with the current parameters θ and a specified evaluation protocol. This includes “difficult, but doable” problems that are only solved under a large budget but not under an ordinary budget; problems not solved under any tested budget may form a separate difficulty group.

Constructing training triples in hindsight. We assume access to a procedure ConstructTriple which, given a pair (x, y) with reference solution y for x (satisfying $\text{Verify}(x, y) = 1$ in verifiable problem domains), returns either a triple $(x, \bar{y}, z)$ or a distinguished symbol `fail`, indicating that no suitable training triple was obtained:

$$\text{ConstructTriple}(x, y; \theta) \in (\{x\} \times \mathcal{Y} \times \mathcal{Z}) \cup \{\text{fail}\}$$

Here z is a hierarchy generated by the current hindsight component $H_\theta(\cdot | x, y)$ and $\bar{y}$ is a reference solution for x which, in general, can be different from y . In verifiable problem domains, we additionally require that $\text{Verify}(x, \bar{y}) = 1$.

The procedure ConstructTriple encapsulates the main difference between the two versions of our method for the general problem setting and for verifiable problem domains: In the former, we simply query the current hindsight component, obtaining $z \sim H_\theta(\cdot | x, y)$, and then return (x, y, z) . However, in verifiable problem domains, access to the procedure Verify allows us to assess the “usefulness” of various generated hierarchies for the solver component S_θ ; consequently, by selecting hierarchies based on their estimated usefulness, we introduce an additional training signal, similar in spirit to expert iteration [1]. This signal enters entirely through data selection; the training objective remains unchanged. We describe these two choices for ConstructTriple in detail in Sections 4.5 and 4.6, respectively.

Training operator. We assume access to a training procedure Train , which, given the current parameters θ and a dataset D of triples in $\mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$, returns updated parameters $\theta_{\text{new}} = \text{Train}(D; \theta)$. The procedure will be used on datasets D with entries of the form (x, y, z) , where y is a reference solution for x and, ideally, realizes the hierarchy z . Consequently, the training objective of the procedure is chosen so that $S_\theta(\cdot | x, z)$, $F_\theta(\cdot | x)$, and $H_\theta(\cdot | x, y)$ are encouraged to predict y , z , and z respectively. As the three components share a backbone network, each of the individual training signals affects all three components. On empty datasets, Train leaves the parameters unchanged. Our concrete choice of Train in the case of interactive theorem proving in Lean is described in Section 5.4.

4.4 The full training loop

We start by training the system once on the seed dataset, setting $\theta \leftarrow \text{Train}(D_0; \theta_{\text{init}})$, where θ_{init} denotes the initial parameters. Subsequently, we execute a loop that alternates between obtaining a batch $\text{SampleBatch}(t)$ of pairs in $\mathcal{X} \times \mathcal{Y}$, constructing training triples in $\mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$ from these pairs in hindsight, and training the system on a dataset composed of these newly constructed triples and previously collected data. Figure 3 illustrates the training loop including the two choices for ConstructTriple from Sections 4.5 and 4.6; Algorithm 1 gives the pseudocode.

Specifically, in iteration $t = 1, \dots, T$, we obtain a batch $B \leftarrow \text{SampleBatch}(t)$ and initialize an empty dataset D_t . For each entry (x, y) of B , we invoke $\text{ConstructTriple}(x, y; \theta)$ and append the returned triple to D_t , unless the procedure returns `fail`. The parameters θ remain fixed throughout this construction phase. After processing the batch, we form a training dataset D from the new dataset D_t and previously collected data from $D_0, \dots, D_{t-1}$. This is accomplished using a *replay policy* which specifies how new and previously collected entries are mixed. Subsequently, we update the parameters via $\theta \leftarrow \text{Train}(D; \theta)$.

In practice, the procedures ConstructTriple and Train may depend on the iteration t . This dependence on t is suppressed in the notation used throughout this section.

(a) Full training loop

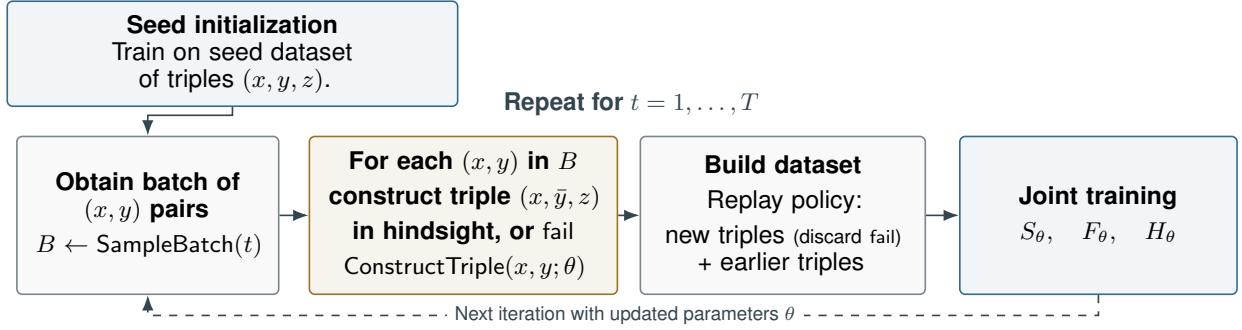
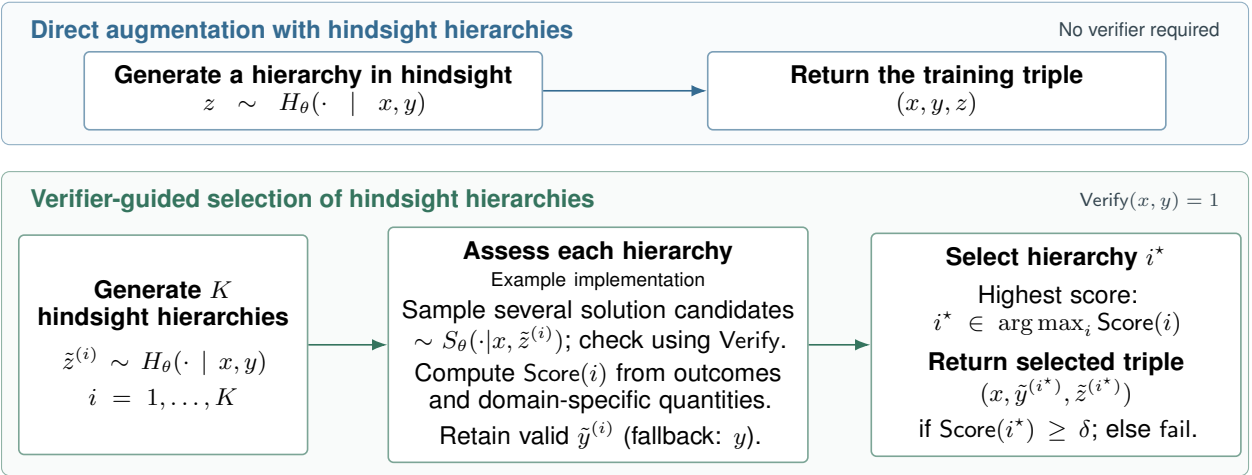
(b) Two choices for $\text{ConstructTriple}(x, y; \theta)$ 

Figure 3: Schematic for the full training loop and the two choices for ConstructTriple from Sections 4.5 and 4.6. The assessment box illustrates one possible way to score candidate hierarchies and retain a valid solution.

4.5 ConstructTriple in the general problem setting: direct augmentation with hindsight hierarchies

In the general problem setting, see Section 3, we construct a training triple directly from a supplied pair $(x, y) \in \mathcal{X} \times \mathcal{Y}$, where y is a reference solution for x . Concretely, we sample a hierarchy z in hindsight using the current parameters θ , and then augment the pair (x, y) with z :

$\text{ConstructTriple}(x, y; \theta) :$
sample $z \sim H_\theta(\cdot \mid x, y)$
return (x, y, z)

We refer to this as *direct augmentation with hindsight hierarchies*. Note that this can also be used in a verifiable problem domain when verifier-guided selection (see Section 4.6) is not desired.

4.6 ConstructTriple in verifiable problem domains: verifier-guided selection of hindsight hierarchies

We now describe our choice of ConstructTriple in verifiable problem domains, see Section 3.1. In this setting, the supplied pairs $(x, y) \in \mathcal{X} \times \mathcal{Y}$ are assumed to satisfy $\text{Verify}(x, y) = 1$. A priori, a hierarchy z sampled from $H_\theta(\cdot \mid x, y)$ need not be *useful*, in the sense that the solver component, given (x, z) , is likely to discover a solution for x . We therefore sample several hindsight-derived hierarchies and select among them using an estimate of their usefulness, together with a penalty encouraging the intended levels of granularity. We refer to this as *verifier-guided selection of hindsight hierarchies*. Algorithm 2 summarizes the procedure. We describe its ingredients and selection rule below.

Assessing usefulness and retaining a reference solution. We assume access to a procedure which, given $(x, y, z) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$ with $\text{Verify}(x, y) = 1$, jointly returns a nonnegative usefulness estimate $\text{Utility}(x, y, z; \theta)$ and a valid

Algorithm 1: The full training loop.

Given : Iteration count $T \in \mathbb{Z}_{\geq 1}$; procedures `SampleBatch`, `ConstructTriple`, and `Train`; replay policy
Input : Seed dataset D_0 ; initial parameters θ_{init}
Output : Trained parameters θ

$\theta \leftarrow \text{Train}(D_0; \theta_{\text{init}})$
for $t = 1, \dots, T$ **do**
 $B \leftarrow \text{SampleBatch}(t)$
 Initialize an empty dataset D_t
 foreach *entry* (x, y) *of* B **do**
 $r \leftarrow \text{ConstructTriple}(x, y; \theta)$ // triple with a hindsight hierarchy, or fail
 if $r \neq \text{fail}$ **then**
 Append the triple r to D_t
 Form a training dataset D from D_t and $D_0, \dots, D_{t-1}$ according to the replay policy
 $\theta \leftarrow \text{Train}(D; \theta)$
return θ

solution `DiscoveredSolution`($x, y, z; \theta$) for x , i.e.,

$$\text{Verify}(x, \text{DiscoveredSolution}(x, y, z; \theta)) = 1.$$

Larger values of Utility are intended to indicate higher estimated usefulness of the hierarchy z for the solver, given problem x . The solution returned by `DiscoveredSolution` need not equal y ; this is intentional: if the hierarchy z is useful for the solver, then $S_\theta(\cdot | x, z)$ is likely to discover *some* valid solution for x , which need not be equal to or even “similar to” y . Accordingly, `DiscoveredSolution`($x, y, z; \theta$) is intended to be a valid solution for x that the solver is likely to discover under the hierarchy z . We write $(\text{Utility}, \text{DiscoveredSolution})(x, y, z; \theta)$ for the output of this joint procedure.

A natural implementation of this joint procedure samples several solution candidates from $S_\theta(\cdot | x, z)$ and checks them using `Verify`. The outcome of this process provides a usefulness estimate, based on the success ratio and optionally domain-specific quantities, while a valid solution found in this process can be retained as `DiscoveredSolution`($x, y, z; \theta$). If none is found, we retain y instead. Our concrete choice for the joint procedure in the case of interactive theorem proving in Lean is described in Section 5.3.

Encouraging different levels of granularity. To encourage the generated hierarchies to represent ideas at different levels of granularity (as opposed to, for example, listing several ideas at the same level of detail), we assume access to a procedure, which, given $z \in \mathcal{Z}$, returns a non-negative real number $\text{Penalty}(z)$, which penalizes departure from this requirement. Such a penalty can encourage compression (between hierarchy levels), but does not by itself establish that a hierarchy is semantically meaningful. Our concrete choice of Penalty in the case of interactive theorem proving in Lean is described in Section 5.3.

Selecting a training triple. Given $(x, y) \in \mathcal{X} \times \mathcal{Y}$ with $\text{Verify}(x, y) = 1$, we sample $K \in \mathbb{Z}_{\geq 1}$ candidate hierarchies $\tilde{z}^{(i)} \sim H_\theta(\cdot | x, y)$, where $i = 1, \dots, K$. Each candidate hierarchy is assessed by jointly obtaining a usefulness estimate and a valid solution for x (recall that `DiscoveredSolution` always has the valid solution y available as fallback),

$$(u^{(i)}, \tilde{y}^{(i)}) \leftarrow (\text{Utility}, \text{DiscoveredSolution})(x, y, \tilde{z}^{(i)}; \theta).$$

We combine the usefulness estimate with the penalty to obtain a score; subsequently, we choose a hierarchy maximizing the score (ties are broken by candidate order):

$$\text{Score}(i) = u^{(i)} - \lambda \cdot \text{Penalty}(\tilde{z}^{(i)}), \quad \lambda > 0, \quad i^* \in \arg \max_{i \in \{1, \dots, K\}} \text{Score}(i).$$

If $\text{Score}(i^*) \geq \delta$ for a prescribed threshold $\delta > 0$, we let `ConstructTriple` return the triple $(x, \tilde{y}^{(i^*)}, \tilde{z}^{(i^*)})$, else we return fail. Algorithm 2 summarizes this specification of `ConstructTriple`.

5 A concrete Lean instantiation

To instantiate our method in the special case of interactive theorem proving (see Section 3.1), we provide concrete realizations of the abstract concepts introduced in Section 4. We use Lean and the verifier-guided choice for `ConstructTriple` from Section 4.6, as summarized in Algorithm 2. We use a single language model as the shared backbone of all three system components, with role-specific prompt templates. We specify the Lean instantiation of our method below; Appendix A provides concrete implementation details, including hyperparameter settings and prompt templates.

Algorithm 2: ConstructTriple in verifiable problem domains: verifier-guided selection of hindsight hierarchies.

Given : Hindsight component H_θ , candidate count $K \in \mathbb{Z}_{\geq 1}$; joint procedure (Utility, DiscoveredSolution); penalty Penalty; hyperparameters $\lambda, \delta > 0$

Input : Pair (x, y) with $\text{Verify}(x, y) = 1$; parameters θ , held fixed throughout this call

Output : One training triple in $\{x\} \times \mathcal{Y} \times \mathcal{Z}$ or fail

```

for  $i = 1, \dots, K$  do
  Sample  $\tilde{z}^{(i)} \sim H_\theta(\cdot \mid x, y)$ 
   $(u^{(i)}, \tilde{y}^{(i)}) \leftarrow (\text{Utility}, \text{DiscoveredSolution})(x, y, \tilde{z}^{(i)}; \theta)$ 
  Score( $i$ )  $\leftarrow u^{(i)} - \lambda \cdot \text{Penalty}(\tilde{z}^{(i)})$ 
 $i^* \leftarrow \arg \max_{i \in \{1, \dots, K\}} \text{Score}(i)$  // ties broken by candidate order
if  $\text{Score}(i^*) \geq \delta$  then
  return  $(x, \tilde{y}^{(i^*)}, \tilde{z}^{(i^*)})$ 
return fail

```

5.1 Lean instantiation: hierarchies and data

We first specify the hierarchy representation and then describe how to construct the seed dataset D_0 of triples and the procedure SampleBatch for obtaining additional batches of pairs in the context of interactive theorem proving in Lean.

Hierarchies. Formally, $\mathcal{Z}_0 = \dots = \mathcal{Z}_n$ is realized as the set of strings over an appropriate alphabet, and we impose no further constraints. However, we aim to ensure during training that the hierarchies represent semantically meaningful proof ideas at different levels of granularity. This motivates the construction of the seed dataset described below and the choice of Penalty, see Section 5.3. The concrete prompt templates in Appendix A.3 use two levels ($n = 1$): a detailed plan z_0 and a coarse proof idea z_1 ; neither is required to consist of executable Lean commands.

Seed dataset. Starting from a dataset of pairs $(x, y) = (s_0, (a_1, \dots, a_k)) \in \mathcal{X} \times \mathcal{Y}$ with $\text{Verify}(x, y) = 1$, we use a *teacher language model* to reverse engineer natural language seed hierarchies for these pairs; the prompt template for this teacher includes instructions to avoid *hindsight bias*. We then augment the pairs (x, y) with the computed hierarchies, yielding the seed dataset D_0 .

Importantly, the teacher is only used for the generation of the seed dataset, i.e., does not appear in the subsequent training loop, where hindsight hierarchies are exclusively generated by the hindsight component. Our concrete choice for the initial dataset of pairs and for the teacher language model is described in Appendix A.2.

Alternatively, if our chosen language model has been pretrained on technical material and already produces useful hindsight hierarchies for some problems, D_0 may be empty, avoiding the need for a teacher to generate seed hierarchies.

Obtaining additional data. We assume access to a supply of pairs $(x, y) \in \mathcal{X} \times \mathcal{Y}$ with $\text{Verify}(x, y) = 1$; in practice, these are either sourced from an external dataset or generated synthetically. Ordinary sampling from such a supply, or generating pairs on demand, suffices to obtain a procedure SampleBatch satisfying the specifications from Section 4.3. As an optional refinement, one may prescribe proportions of problems from different difficulty groups (relative to the current parameters θ of the system) as described in Section 4.3. For example, there may be groups consisting of problems that are solved under an ordinary proof search budget, problems that are solved under a generous (but not under an ordinary) proof search budget, and problems that are not solved under any of the tested proof search budgets. Our concrete choice of SampleBatch is given in Appendix A.2.

5.2 Lean instantiation: system components

All three components are implemented using a single underlying language model with shared parameters, and we distinguish between components using role-specific prompt templates. We implement the foresight component F_θ and the hindsight component H_θ via iterative calls to the language model that mirror the setup in Section 4.2. The solver component S_θ executes a best-first search, using the language model as a tactic oracle. We give the details below and refer to Appendix A.3 for the concrete prompt templates.

Foresight component. For F_θ we first query the language model with a text representation of the proof state $s_0 \in \mathcal{X} = \mathcal{S}$ and a prompt requesting a coarse proof idea. We then iteratively refine the idea by querying the language

model with the same text representation of s_0 , with the idea output in the previous iteration, and with a prompt requesting a more detailed idea.

Hindsight component. The hindsight component H_θ is implemented analogously: In the first step, given $(x, y) = (s_0, (a_1, \dots, a_k)) \in \mathcal{X} \times \mathcal{Y}$ with $\text{Verify}(x, y) = 1$, we query the language model with a text representation of s_0 , with a text representation of the tactic sequence $(a_1, \dots, a_k)$, and with a prompt requesting a reverse-engineered detailed proof idea. Optionally, if the context window of the language model allows, we may also include the intermediate proof states $s_1, \dots, s_{k-1}$ in the prompt, which can be obtained by querying the transition function $\mathcal{T}$ of the interactive theorem prover. We then iteratively compress the idea by querying the language model with the same text representation of s_0 , with the idea output in the previous iteration, and with a prompt requesting a coarser idea. All prompts for the hindsight component contain instructions to avoid hindsight bias and to produce proof ideas that could plausibly be followed from the initial state, rather than merely describe the supplied proof.

Solver component. Given a problem $s_0 \in \mathcal{X} = \mathcal{S}$ and a hierarchy $z \in \mathcal{Z}$, the solver component S_θ performs best-first proof search starting from s_0 , using the shared language model as a tactic oracle, see, e.g., [6]. When expanding a proof state s , we query the language model with text representations of the initial proof state s_0 , the hierarchy z , and the current proof state s , requesting a next tactic. The hierarchy z and the initial state s_0 remain fixed throughout the search; they are nevertheless included in every prompt to the language model (together with the state at the current node in the proof search graph): the purpose of the hierarchy is to guide tactic generation, and the initial state provides context for the hierarchy. Our configuration of best-first proof search, including the priority rule, tactic generation settings, and search budgets, is specified in Appendix A.4.

Unlike an executed tactic, a hierarchy receives no immediate feedback from Lean. During proof search, we therefore treat the hierarchy as guidance that may be incomplete or partly wrong. Our solver prompt encourages the language model to use the hierarchy where it fits the current proof state, while allowing it to depart from the suggested plan when necessary. The concrete prompt template is provided in Appendix A.3.

Inference. For fixed parameters θ , we follow the inference procedure described in Section 4.2: Given a problem $s_0 \in \mathcal{X} = \mathcal{S}$, we first generate a hierarchy $\hat{z} \sim F_\theta(\cdot \mid s_0)$ using the foresight component, and then invoke the solver component $S_\theta(\cdot \mid s_0, \hat{z})$. We refer to this procedure as an *arm*. In order to explore several hierarchies, we may execute several arms sequentially, each with a newly sampled hierarchy and a fresh proof search graph rooted at s_0 . If an arm finds no solution, we proceed to the next arm. This procedure stops as soon as a valid solution is found, or signals failure when all arms have been exhausted. Implementation details, such as the concrete number of arms, together with their search budgets, are provided in Appendix A.5, while Figure 4 illustrates this procedure.

5.3 Lean instantiation: ConstructTriple

We now specify concrete choices for the joint procedure (Utility, DiscoveredSolution) and the penalty Penalty, which, together with Algorithm 2, determine ConstructTriple in our Lean instantiation. These choices are not essential to the general method; we give them to provide a concrete instantiation.

Joint procedure (Utility, DiscoveredSolution). Usefulness can be assessed, for example, using the success ratio of repeated attempts under a prescribed proof search budget, optionally incorporating quantities such as consumed search resources and the lengths of discovered proofs. Our concrete choice below combines the success ratio with a preference for shorter proofs:

Given $(x, y, z) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$ with $\text{Verify}(x, y) = 1$, we assess the usefulness of the hierarchy z for finding a solution for x by executing $N \in \mathbb{Z}_{\geq 1}$ runs of the solver component $S_\theta(\cdot \mid x, z)$ under a prescribed proof search budget. Each run starts with a fresh proof search graph rooted at x , with the hierarchy z and the parameters θ held fixed. A run is successful if it returns a valid solution for x .

If none of these N runs succeeds, we set

$$(\text{Utility}, \text{DiscoveredSolution})(x, y, z; \theta) \leftarrow (0, y).$$

Otherwise, denote the number of successful runs by $M \in \{1, \dots, N\}$ and their returned solutions, in attempt order, by $y_i \in \mathcal{A}^{k_i}$, $i = 1, \dots, M$. Note that, for $i \neq j$, the finite tactic sequences y_i and y_j need not be distinct. We then choose a solution that minimizes proof length, i.e., number of tactics, among the discovered solutions and compute a usefulness estimate based on proof success ratio $r = M/N$ and average proof length $\ell = (k_1 + \dots + k_M)/M$.

More precisely, with $m \in \arg \min_{i \in \{1, \dots, M\}} k_i$ (breaking ties by attempt order) and a hyperparameter $\eta > 0$, we let

$$(\text{Utility}, \text{DiscoveredSolution})(x, y, z; \theta) \leftarrow (r + \eta/\ell, y_m).$$

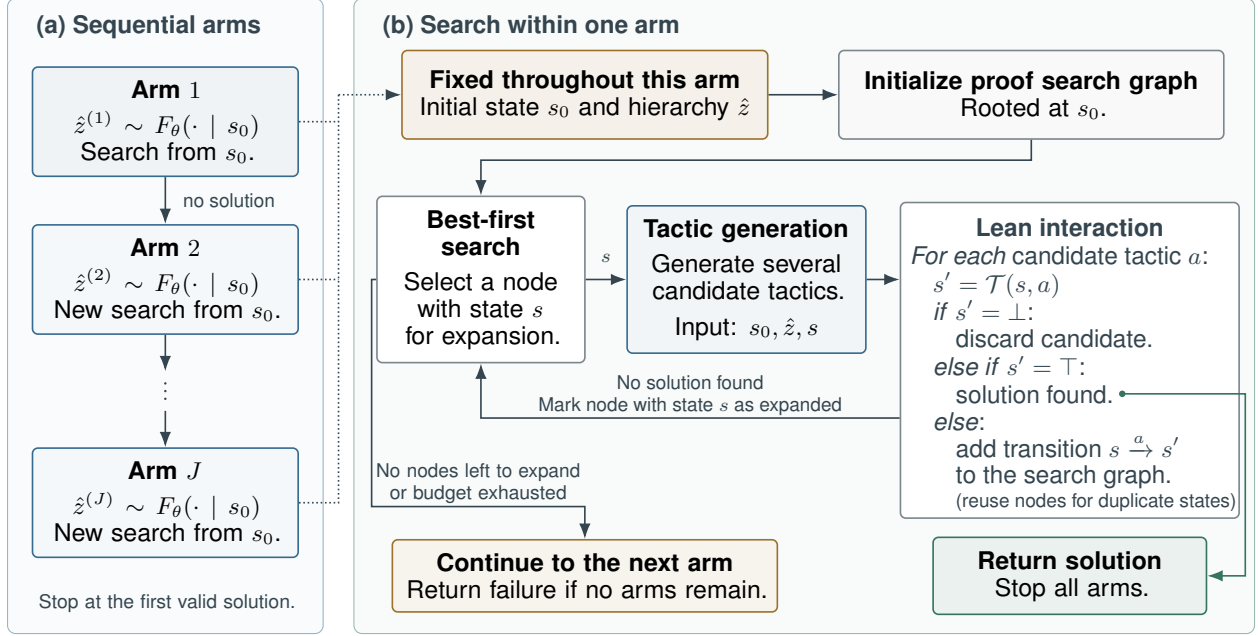


Figure 4: Hierarchy-guided Lean inference with J sequential arms. Each arm samples a hierarchy from F_θ and subsequently invokes the solver with this fixed hierarchy and a fresh proof search graph rooted at the initial state s_0 . An unsuccessful arm is followed by the next and a valid solution stops the entire procedure; we signal failure if all J arms have been exhausted. The right panel illustrates tactic generation and Lean interaction within one arm.

Penalty for hierarchies. In order to encourage the generation of hierarchies that represent proof ideas at different levels of granularity, we use a penalty that encourages compression between successive hierarchy levels. Concretely, with hyperparameters $\alpha_1, \dots, \alpha_n \in (0, 1)$ and the usual convention for empty sums (the depth n may be 0), we define

$$\text{Penalty}(z) = \sum_{j=1}^n \text{ReLU}(\text{len}(z_j) - \alpha_j \text{len}(z_{j-1})),$$

for $z = (z_0, \dots, z_n) \in \mathcal{Z}_0 \times \dots \times \mathcal{Z}_n$, where $\text{len}(z_j)$ denotes the number of characters in the string z_j . This choice of Penalty encourages compression, but does not by itself establish that the hierarchy is semantically meaningful.

Remark. With these choices, a hierarchy for which no solver run succeeds has Utility zero and hence a non-positive Score in the context of Algorithm 2. While the fallback y for DiscoveredSolution makes the joint procedure well-defined in this case, Algorithm 2 will not return the corresponding triple, since the hierarchy’s Score lies below the positive acceptance threshold δ . Consequently, every triple returned by Algorithm 2 contains a valid solution discovered by the solver under the triple’s hierarchy.

5.4 Lean instantiation: training operator Train

We implement Train by supervised fine-tuning of the shared language model underlying the three system components. We first describe how training triples are converted into prompt-response pairs, and then specify the composition of mini-batches and the training objective.

Prompt-response pairs. Given a dataset D of triples $(x, y, z) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$ with $\text{Verify}(x, y) = 1$, we proceed as follows. If D is empty, the parameters θ remain unchanged. Otherwise, let (x, y, z) be an entry of D and write $y = (a_1, \dots, a_k)$, $s_0 = x$, and $s_{t+1} = \mathcal{T}(s_t, a_{t+1})$ for $t = 0, \dots, k-1$. In accordance with the description of the system components in Section 5.2, the triple (x, y, z) yields a total of $2n + k + 2$ prompt-response pairs, namely

- $n + 1$ prompt-response pairs for the foresight component: for predicting z_n from x and, for $j = 0, \dots, n-1$, predicting z_j from (x, z_{j+1}) ,
- $n + 1$ prompt-response pairs for the hindsight component: for predicting z_0 from (x, y) and, for $j = 1, \dots, n$, predicting z_j from (x, z_{j-1}) ,

- k prompt-response pairs for the solver component: for predicting a_{t+1} from (x, z, s_t) for $t = 0, \dots, k-1$,

where all occurring prompts use the prompt templates from Appendix A.3. Note that, for the solver component, these pairs are intended to train the language model for tactic generation within best-first proof search.

Shuffling and balanced batching. For nonempty D , we group the constructed prompt-response pairs by component and, for foresight and hindsight, by hierarchy level. Thus, we have one solver group and, for each $j = 0, \dots, n$, one foresight group and one hindsight group, for a total of $2n + 3$ groups. Since a proof with k tactics contributes k solver pairs, longer proofs contribute more entries within the solver group. We fix positive integers b_S, b_F, b_H . Each mini-batch contains b_S pairs from the solver group, b_F pairs from each foresight group, and b_H pairs from each hindsight group. Thus, each mini-batch contains a total of

$$b = b_S + (n + 1)(b_F + b_H)$$

prompt-response pairs. Here, a mini-batch denotes the full collection of prompt-response pairs used for one optimizer update. Consequently,

$$\beta_S = \frac{b_S}{b}, \quad \beta_F = \frac{(n + 1)b_F}{b}, \quad \beta_H = \frac{(n + 1)b_H}{b}$$

are the fractions of pairs in each mini-batch belonging to the solver, foresight, and hindsight components, respectively.

Within each of the $2n + 3$ groups, we shuffle the entries and traverse the resulting sequence in order; this shuffling happens independently between groups. To construct a mini-batch, we take the next prescribed number of entries from each group, see b_S, b_F, b_H above, continuing from where the preceding mini-batch left off. Whenever a group is exhausted, we reshuffle that group and continue from the beginning of its new order, without resetting the other groups. This also applies if a group is exhausted while a mini-batch is being constructed.

Training objective. Write p_θ for the conditional next-token probabilities of the shared language model. For a tokenized prompt $u = (u_1, \dots, u_m)$ and a tokenized target response $v = (v_1, \dots, v_L)$, including its end-of-response marker but excluding padding, we use the loss

$$\ell_\theta(u, v) = -\frac{1}{L} \sum_{q=1}^L \log p_\theta(v_q \mid u, v_1, \dots, v_{q-1}).$$

Note that $L \geq 1$, since v includes its end-of-response marker. The normalization by L was included, since hierarchy descriptions can be substantially longer than individual Lean tactics.

For nonempty D , let $\mathcal{L}_S(\theta; D)$ denote the arithmetic mean of ℓ_θ over all solver prompt-response pairs constructed from D . Similarly, for $j = 0, \dots, n$, let $\mathcal{L}_{F,j}(\theta; D)$ and $\mathcal{L}_{H,j}(\theta; D)$ denote the arithmetic mean of ℓ_θ over all foresight and hindsight prompt-response pairs at hierarchy level j , respectively. These averages count repeated entries separately. We train the shared language model underlying the three components using mini-batch optimization of the objective

$$\mathcal{L}(\theta; D) = \beta_S \mathcal{L}_S(\theta; D) + \frac{\beta_F}{n + 1} \sum_{j=0}^n \mathcal{L}_{F,j}(\theta; D) + \frac{\beta_H}{n + 1} \sum_{j=0}^n \mathcal{L}_{H,j}(\theta; D),$$

where $\beta_S, \beta_F, \beta_H$ are the fractions for the three components per mini-batch introduced in the preceding paragraph.

The procedure Train is then implemented by iterating over mini-batches and executing one optimizer step per mini-batch, where the loss used for an optimizer step is the arithmetic mean of ℓ_θ over the prompt-response pairs in the mini-batch. Concrete batch counts, optimizer settings, and training duration are specified in Appendix A.8.

6 Discussion

From hindsight to foresight. Our central hypothesis is that reverse engineering the idea underlying a known solution is, in general, easier than discovering it from the problem alone. We operationalize this hypothesis through a training loop that alternates between reverse engineering ideas from known solutions (hindsight) and training the model using these ideas. When reverse engineering solution ideas, it is crucial to avoid hindsight bias. In our implementation, we seek to mitigate this risk through corresponding instructions in the language model’s prompt templates. Nevertheless, it is a priori not clear that solution ideas derived in hindsight are suitable targets for foresight prediction. This highlights the need for a thorough experimental evaluation of the proposed method.

The case of verifiable problem domains. In verifiable problem domains we have access to a verifier that either certifies or rejects proposed solutions. We exploit this by refining the hindsight idea generation in the alternating training loop: Given a problem and a reference solution, we generate several ideas in hindsight and ask the model to solve the problem using the generated ideas. We then use the verifier and domain-specific quantities to evaluate the proposed solutions and subsequently score the usefulness of the generated ideas. The highest-scoring idea is retained (provided a certain threshold is met), while the other ideas are discarded. This introduces an additional training signal in the spirit of expert iteration. However, a high score does not by itself establish that the retained idea helped the model solve the problem. For example, when not orchestrating the training loop carefully, the model might learn to ignore the proposed ideas and generate solutions based on the problem alone. This once again highlights the need for a thorough evaluation of the proposed method, including ablation studies assessing whether selecting hindsight-generated ideas based on score improves model performance.

Future empirical validation. Future work should experimentally evaluate the proposed method. The evaluation should cover both the Lean instantiation and the version without a verifier, and include comparisons with standard supervised fine-tuning under comparable data and computational budgets, as well as thorough ablation studies.

Use of AI tools

The method and its original rigorous formal specification (i.e., the content of Sections 3, 4, and 5) were developed without AI assistance. ChatGPT, using multiple model versions, assisted with the preparation of this manuscript. This assistance included drafting and revising prose from author-provided technical material, as well as searching for and discussing related literature. It also included assistance with generating figures, L^AT_EX preparation, and typesetting.

AI assistance was also used for implementation and testing in the associated software project. This version of the manuscript reports no empirical evaluation. The human authors retain full responsibility for the manuscript, including its scientific claims, mathematical content, references, and all material prepared with AI assistance.

References

- [1] T. Anthony, Z. Tian, and D. Barber, “Thinking fast and slow with deep learning and tree search,” in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017.
- [2] L. de Moura and S. 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*, p. 625–635, Springer-Verlag, 2021.
- [3] S. Polu and I. Sutskever, “Generative language modeling for automated theorem proving,” 2020.
- [4] J. M. Han, J. Rute, Y. Wu, E. Ayers, and S. Polu, “Proof artifact co-training for theorem proving with language models,” in *International Conference on Learning Representations*, 2022.
- [5] G. Lample, T. Lacroix, M.-A. Lachaux, A. Rodriguez, A. Hayat, T. Lavril, G. Ebner, and X. Martinet, “Hypertree proof search for neural theorem proving,” in *Advances in Neural Information Processing Systems*, vol. 35, pp. 26337–26349, Curran Associates, Inc., 2022.
- [6] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar, “Leandojo: Theorem proving with retrieval-augmented language models,” in *Advances in Neural Information Processing Systems*, vol. 36, pp. 21573–21612, Curran Associates, Inc., 2023.
- [7] A. Q. Jiang, W. Li, S. Tworowski, K. Czechowski, T. Odrzygóźdź, P. Miłoś, Y. Wu, and M. Jamnik, “Thor: Wielding hammers to integrate language models and automated theorem provers,” in *Advances in Neural Information Processing Systems*, vol. 35, pp. 8360–8373, Curran Associates, Inc., 2022.
- [8] H. Xin, D. Guo, Z. Shao, Z. Ren, Q. Zhu, B. Liu, C. Ruan, W. Li, and X. Liang, “Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data,” 2024.
- [9] H. Xin, Z. Ren, J. Song, Z. Shao, W. Zhao, H. Wang, B. Liu, L. Zhang, X. Lu, Q. Du, W. Gao, H. Zhang, Q. Zhu, D. Yang, Z. Gou, Z. Wu, F. Luo, and C. Ruan, “Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search,” in *International Conference on Learning Representations*, vol. 2025, pp. 72274–72303, 2025.
- [10] Z. Z. Ren, Z. Shao, J. Song, H. Xin, H. Wang, W. Zhao, L. Zhang, Z. Fu, Q. Zhu, D. Yang, Z. F. Wu, Z. Gou, S. Ma, H. Tang, Y. Liu, W. Gao, D. Guo, and C. Ruan, “Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition,” 2025.

- [11] Y. Lin, S. Tang, B. Lyu, J. Wu, H. Lin, K. Yang, J. Li, M. Xia, D. Chen, S. Arora, and C. Jin, “Goedel-prover: A frontier model for open-source automated theorem proving,” in *Second Conference on Language Modeling*, 2025.
- [12] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, and E.-P. Lim, “Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Toronto, Canada), pp. 2609–2634, Association for Computational Linguistics, July 2023.
- [13] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. V. Le, and E. H. Chi, “Least-to-most prompting enables complex reasoning in large language models,” in *The Eleventh International Conference on Learning Representations*, 2023.
- [14] T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal, “Decomposed prompting: A modular approach for solving complex tasks,” in *The Eleventh International Conference on Learning Representations*, 2023.
- [15] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, “Tree of thoughts: Deliberate problem solving with large language models,” in *Advances in Neural Information Processing Systems*, vol. 36, pp. 11809–11822, Curran Associates, Inc., 2023.
- [16] A. Q. Jiang, S. Welleck, J. P. Zhou, T. Lacroix, J. Liu, W. Li, M. Jamnik, G. Lample, and Y. Wu, “Draft, sketch, and prove: Guiding formal theorem provers with informal proofs,” in *The Eleventh International Conference on Learning Representations*, 2023.
- [17] H. Wang, H. Xin, C. Zheng, Z. Liu, Q. Cao, Y. Huang, J. Xiong, H. Shi, E. Xie, J. Yin, Z. Li, and X. Liang, “LEGO-prover: Neural theorem proving with growing libraries,” in *The Twelfth International Conference on Learning Representations*, 2024.
- [18] H. Wang, H. Xin, Z. Liu, W. Li, Y. Huang, J. Lu, Z. Yang, J. Tang, J. Yin, Z. Li, and X. Liang, “Proving theorems recursively,” in *Advances in Neural Information Processing Systems*, vol. 37, pp. 86720–86748, Curran Associates, Inc., 2024.
- [19] J. Huang, S. Gu, L. Hou, Y. Wu, X. Wang, H. Yu, and J. Han, “Large language models can self-improve,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, (Singapore), pp. 1051–1068, Association for Computational Linguistics, Dec. 2023.
- [20] A. Singh, J. D. Co-Reyes, R. Agarwal, A. Anand, P. Patil, X. Garcia, P. J. Liu, J. Harrison, J. Lee, K. Xu, A. T. Parisi, A. Kumar, A. A. Alemi, A. Rizkowsky, A. Nova, B. Adlam, B. Bohnet, G. F. Elsayed, H. Sedghi, I. Mordatch, I. Simpson, I. Gur, J. Snoek, J. Pennington, J. Hron, K. Kenealy, K. Swersky, K. Mahajan, L. A. Culp, L. Xiao, M. Bileschi, N. Constant, R. Novak, R. Liu, T. Warkentin, Y. Bansal, E. Dyer, B. Neyshabur, J. Sohl-Dickstein, and N. Fiedel, “Beyond human data: Scaling self-training for problem-solving with language models,” *Transactions on Machine Learning Research*, 2024. Expert Certification.
- [21] S. Polu, J. M. Han, K. Zheng, M. Baksys, I. Babuschkin, and I. Sutskever, “Formal mathematics statement curriculum learning,” in *The Eleventh International Conference on Learning Representations*, 2023.
- [22] K. Dong and T. Ma, “STP: Self-play LLM theorem provers with iterative conjecturing and proving,” in *Proceedings of the 42nd International Conference on Machine Learning*, vol. 267 of *Proceedings of Machine Learning Research*, pp. 14114–14136, PMLR, 2025.
- [23] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, “Training verifiers to solve math word problems,” 2021.
- [24] H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K. Cobbe, “Let’s verify step by step,” in *International Conference on Learning Representations*, vol. 2024, pp. 39578–39601, 2024.
- [25] E. Zelikman, Y. Wu, J. Mu, and N. Goodman, “Star: Bootstrapping reasoning with reasoning,” in *Advances in Neural Information Processing Systems*, vol. 35, pp. 15476–15488, Curran Associates, Inc., 2022.
- [26] S. Zhao, Z. Xie, M. Liu, J. Huang, G. Pang, F. Chen, and A. Grover, “Self-distilled reasoner: On-policy self-distillation for large language models,” in *Proceedings of the 43rd International Conference on Machine Learning*, vol. 306 of *Proceedings of Machine Learning Research*, pp. 162433–162448, PMLR, 2026.
- [27] H. Lin, Z. Sun, S. Welleck, and Y. Yang, “Lean-star: Learning to interleave thinking and proving,” in *International Conference on Learning Representations*, vol. 2025, pp. 66041–66062, 2025.
- [28] Y. Li, H. Shi, B. Deng, W. Wang, M. Ruan, H. Hou, Z. Dai, S. Gao, C. Wang, S. Qiu, and L. Song, “Learning to reason with insight for informal theorem proving,” 2026.

- [29] H. Ying, S. Zhang, L. Li, Z. Zhou, Y. Shao, Z. Fei, Y. Ma, J. Hong, K. Liu, Z. Wang, Y. Wang, Z. Wu, S. Li, F. Zhou, H. Liu, S. Zhang, W. Zhang, H. Yan, X. Qiu, J. Wang, K. Chen, and D. Lin, “Internlm-math: Open math large language models toward verifiable reasoning,” 2024.
- [30] The mathlib Community, “The Lean Mathematical Library,” in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, (New Orleans, LA, USA), ACM, Jan. 2020.

A Appendix

This appendix provides the details for our implementation of the Lean instantiation of the proposed method.

A.1 Language model and environment

Shared model	internlm/internlm2-math-plus-7b [29]
Weights/tokenizer revision	2565e424cc6f83358e7a5aa19b639205fcdd173d
Weights; context	bfloat16; 8192 tokens
Hierarchy depth	$n = 1$
Theorem-proving environment	Lean 4.10.0 [2]; LeanDojo 2.1.3 [6]
Runtime	Python 3.11.9; x86_64 Linux
Reference inference packages	vLLM 0.4.1; PyTorch 2.2.1; Transformers 4.40.1

All trainable parameters are shared; prompts distinguish components.

A.2 Data

External data and seed split. We extract named theorems from mathlib [30], revision

a719ba5c3115d47b68bf0497a9dd1bcbb21ea663.

Following the random splitting procedure of LeanDojo [6], we partition the extracted theorems into training, validation, and test sets, reserving 2,000 theorems each for validation and testing (random seed 3407). Only the training partition is used to construct the seed dataset and obtain subsequent training batches. From the training partition, we retain only complete tactic proofs that, when executed in Lean, close all goals and reproduce the proof states recorded in the dataset. Proofs with missing tactics are excluded. This yields 50,523 pairs (x, y) with $\text{Verify}(x, y) = 1$. We randomly permute these pairs using seed 20260523, assign the first 15,000 to seed hierarchy generation, and reserve the remaining 35,523 for subsequent batches. Both subsets are used for training; neither is an evaluation set.

Constructing the seed dataset: teacher hierarchies. The teacher language model, gpt-5.4-mini (reasoning effort high, verbosity low), receives the initial state, complete tactic sequence, and theorem metadata, excluding intermediate states. Its prompt (Appendix A.3) requests a detailed plan z_0 , a coarse idea z_1 , and a short anti-hindsight-bias check. We form D_0 by augmenting the 15,000 pairs assigned to seed hierarchy generation with their corresponding hierarchies (z_0, z_1) . Later hierarchies intended for training come exclusively from the current hindsight component.

Choice of SampleBatch. We propose processing the reserved pool in the order induced by the random permutation above, taking successive batches of up to 2,048 entries. After processing the entire pool, we revisit pairs for which ConstructTriple returned fail; we allow at most two such retry passes, as specified in Appendix A.6. This uses neither synthetic data nor difficulty groups (Section 4.3). The associated replay policy and training duration are specified in Appendices A.7 and A.8.

A.3 Role-specific prompt templates

For $n = 1$ (Section 5.1), z_0 is labelled DETAILED PLAN and z_1 COARSE IDEA; angle brackets mark inserted text. The teacher language model produces both levels per response; each shared-model foresight or hindsight call produces one.

Teacher language model. The teacher receives the initial state, verified tactics, and theorem metadata, without intermediate states. Its JSON response must contain exactly the three string fields below. Only the first two are hierarchy targets; anti_hindsight_check records an explanation.

You are helping create training data for a Lean theorem-proving system.

For each verified Lean theorem proof, write two forward-usable proof ideas that describe the same proof strategy at different levels of detail:

1. a detailed proof plan, and
2. a shorter, more abstract proof idea.

You will see:

1. the original Lean 4 proof state before any tactic is run, and
2. a verified tactic proof that solves the theorem.

Your task is to infer the mathematical proof idea behind the verified proof and rewrite it as guidance that could plausibly have been used from the original state.

Critical anti-hindsight-bias rule:

- The verified proof is evidence for inferring the idea, not something to copy.
- Do not write a chronological replay of the tactic proof.
- Do not say "the proof does X", "the verified proof uses X", "after seeing the proof", or similar hindsight-only language.
- Do not rely on information that is only visible after running later tactics, unless you phrase it as a forward-usable mathematical reason already suggested by the initial state.
- Tactic names may be mentioned only when they express a reusable strategy visible from the initial state, such as rewriting a definition, splitting cases, applying induction, using a known hypothesis, or discharging arithmetic.

Return only valid JSON with this exact shape:

```
```json
{
 "z0_detailed_plan": "A forward-usable detailed proof plan. It may mention the
 useful definitions, hypotheses, transformations, and proof obligations, but it must
 not be a tactic-by-tactic replay.",
 "z1_coarse_idea": "A short reusable proof idea at a higher level of abstraction.",
 "anti_hindsight_check": "One sentence explaining why the plan could plausibly be
 chosen from the initial state alone."
}
```
```

Requirements for `z0\_detailed\_plan`:

- 2 to 6 sentences.
- Explain why the strategy should work mathematically.
- Mention relevant hypotheses, definitions, lemmas, or theorems when they are visible from the initial state or are natural standard tools for the theorem's mathematical context.
- Avoid copying the tactic sequence.

Requirements for `z1\_coarse\_idea`:

- 1 sentence.
- More abstract and shorter than `z0\_detailed\_plan`.
- Reusable across similar theorem states.

Initial Lean 4 state:

```
```lean
<INITIAL_STATE>
```
```

Verified tactic proof:

```
```lean
<TACTIC_PROOF>
```
```

Theorem metadata:

```
```text
theorem_full_name: <THEOREM_FULL_NAME>
file_path: <FILE_PATH>
proof_step_count: <PROOF_STEP_COUNT>
```
```

The anti-hindsight-bias instruction encourages guidance usable from the initial state.

Shared-model message format. Foresight, hindsight, and tactic generation use the same underlying language model, initialized from internlm/internlm2-math-plus-7b (Appendix A.1) and jointly fine-tuned as described in Section 5.4. Their prompts use the following chat wrapper, matching the chosen model’s chat format, without a system message:

```
<|im_start|>user
<USER MESSAGE><|im_end|>
<|im_start|>assistant
```

Each template below specifies a complete user message for the chat wrapper above. Responses end with `<|im_end|>`. Displayed line breaks within prose sentences serve typesetting only.

Foresight: coarse idea, $F_{1,\theta}$. Request z_1 from the initial state:

```
My initial LEAN 4 state is:
'''lean
<INITIAL STATE>
'''

Please write a concise proof idea that could help prove the theorem
from this state.
Write only the requested proof idea, with no extra commentary.
COARSE IDEA:
```

Foresight: detailed plan, $F_{0,\theta}$. Request z_0 from the initial state and the preceding coarse idea:

```
My initial LEAN 4 state is:
'''lean
<INITIAL STATE>
'''

Here is the previous proof idea:
COARSE IDEA:
<COARSE IDEA>

Please refine the previous proof idea into a more detailed proof idea
that could help prove the theorem from this state.
Write only the requested proof idea, with no extra commentary.
DETAILED PLAN:
```

Hindsight: detailed plan, $H_{0,\theta}$. Request z_0 from the initial state and the reference tactic sequence, with tactics separated by newlines:

```
My initial LEAN 4 state is:
'''lean
<INITIAL STATE>
'''

Here is the verified Lean 4 tactic proof:
'''lean
<TACTIC SEQUENCE>
'''

Reverse-engineer a detailed proof idea from this verified proof.
Avoid hindsight bias: write a forward-usable idea that could plausibly
have been followed from the original state, not an idea that can only
be obtained from knowledge of the complete proof.
Write only the requested proof idea, with no extra commentary.
REVERSE ENGINEERED DETAILED PROOF IDEA:
```

Hindsight: coarse idea, $H_{1,\theta}$. Request z_1 from the initial state and the preceding detailed plan:

My initial LEAN 4 state is:
 ```lean  
 <INITIAL STATE>  
 ```

Here is the previous, more detailed proof idea:
 DETAILED PLAN:
 <DETAILED PLAN>

Summarize the previous proof idea into a coarser forward-usable proof idea. Preserve the strategic reason it could help from the initial state.
 Write only the requested proof idea, with no extra commentary.
 SUMMARIZED IDEA:

Solver: next tactic. Request a tactic from the initial state, the hierarchy, and the current proof state. The prompt explicitly allows the hierarchy to be imperfect:

My initial LEAN 4 state is:
 ```lean  
 <INITIAL STATE>  
 ```

Here is a suggested proof idea at several levels of detail. It may be useful, incomplete, or partly wrong. Use it as guidance where it fits the current LEAN 4 state, but choose the next tactic based on the current state.

COARSE IDEA:
 <COARSE IDEA>

DETAILED PLAN:
 <DETAILED PLAN>

My current LEAN 4 state is:
 ```lean  
 <CURRENT STATE>  
 ```

Please predict a possible tactic to help me prove the theorem.

Only the current state changes during search; the initial state and hierarchy remain fixed. The foresight, hindsight, and solver templates above also define the prompts of the supervised prompt-response pairs described in Section 5.4.

Null arm. The optional null arm (Appendix A.5) makes no foresight call. It uses the solver template with both hierarchy placeholders replaced by No proof idea provided., yielding the following user message:

My initial LEAN 4 state is:

```
'''lean
<INITIAL STATE>
'''
```

Here is a suggested proof idea at several levels of detail. It may be useful, incomplete, or partly wrong. Use it as guidance where it fits the current LEAN 4 state, but choose the next tactic based on the current state.

COARSE IDEA:

No proof idea provided.

DETAILED PLAN:

No proof idea provided.

My current LEAN 4 state is:

```
'''lean
<CURRENT STATE>
'''
```

Please predict a possible tactic to help me prove the theorem.

A.4 Best-first proof search configuration

The following settings apply to inference; hierarchy assessment during training uses smaller limits, see Appendix A.6.

Tactic decoding	Deterministic beam search; 32 responses; ≤ 256 new tokens each; length penalty 0; stop <code>< im_end ></code> .
Per-attempt search limits	100 node expansions; depth 100; 32 tactics per expanded state; nominal time 600 seconds.
LeanDojo tactic timeout	300 seconds.

The parser removes tactic wrappers and rejects empty outputs; duplicate tactic texts are removed before execution. While beam search generates candidate tactics for the selected proof state, the priority rule below determines which node in the proof search graph to expand next.

Search priority. A newly created node in the proof search graph is assigned the priority

$$P(a_1, \dots, a_d) = \sum_{t=1}^d \frac{g_t}{c_t},$$

where $(a_1, \dots, a_d)$ is the sequence of tactics along the path by which the node was first reached from the root. For each $t = 1, \dots, d$, $c_t \geq 1$ is the number of tokens in the response generated by the language model from which tactic a_t was extracted, and g_t is the sum of their conditional log probabilities under the language model. Each token probability is conditioned on the solver prompt used at that step and all preceding tokens of that response. These quantities refer to the generated response before removing tactic wrappers; prompt tokens are not counted. The root node has priority zero.

State reuse and termination. At each expansion, select the unexpanded node in the queue with the highest priority, breaking ties by insertion order. For the selected node, execute the candidate tactics generated by the language model in Lean in decreasing order of their step scores, where a step score is the average conditional log probability over the tokens of the corresponding generated response. Each tactic is applied independently to the same proof state associated with the selected node. Discard failed tactics; if a tactic completes the proof, return the discovered solution without executing the remaining candidate tactics. If a tactic produces a valid successor state with open goals, record the corresponding state and transition in the proof search graph: If the state has occurred before, use the corresponding existing node (without updating its depth and priority), otherwise create a new node, compute its search priority, and insert it into the queue; in either case, add an edge to this node from the selected node, labelled by the tactic. If none of the candidate tactics completes a proof, mark the selected node as expanded. Expanded nodes which receive a new incoming edge retain their expanded status. Nodes at the depth limit are marked as expanded without generating candidate tactics. Treat a tactic timeout as a failed tactic and continue with the remaining candidates. Return failure when no nodes remain to expand or the overall search budget is exhausted.

A.5 Inference: multiple arms

For the sequential procedure in Section 5.2 (see also Figure 4), we choose $J = 2$ hierarchy-conditioned arms. Each generates a new hierarchy and subsequently invokes the solver component with the full budget from Appendix A.4. The parameters θ of the shared language model stay fixed and we stop at the first valid solution.

Foresight and hindsight use the same per-level decoding:

Sampling	One response; temperature 0.7; nucleus threshold 0.95.
Length and termination	≤ 512 new tokens; stop <code>< im_end ></code> .
Foresight seed schedule	First arm: 20260820, 20260821; second: 20260822, 20260823, in generation order.

Optional null arm. We may optionally prepend a *null arm* to the hierarchy-conditioned arms described in Section 5.2: Such an arm skips foresight hierarchy generation and calls the solver component with the problem x and the hierarchy $(z_0, z_1) = (\text{No proof idea provided.}, \text{No proof idea provided.})$; the corresponding prompt template for the shared language model can be found in Appendix A.3. It uses the same budget and proof search configuration as regular hierarchy-conditioned arms, see Appendix A.4.

A.6 Hierarchy selection and iteration settings

We propose the following configuration for implementing the Lean instantiation of ConstructTriple, see Section 5.3, and the full training loop. These settings specify a concrete starting point for evaluation; they have not been selected through empirical comparison.

Candidates per pair	$K = 4$ hindsight hierarchies, each from separate $H_{0,\theta}$ then $H_{1,\theta}$ calls.
Candidate decoding	Temperature 0.7; nucleus threshold 0.95; ≤ 512 new tokens per level; fresh pseudo-random seed per call, recorded with the hierarchy.
Assessment	$N = 1$ solver run per hierarchy; 20 node expansions; nominal time 120 seconds; LeanDojo tactic timeout 30 seconds. Other settings as in Appendix A.4.
Selection coefficients	$\eta = 1, \alpha_1 = \frac{1}{2}, \lambda = 10^{-3}, \delta = \frac{1}{2}$.
Pass limit	One complete pass through the 35,523 reserved pairs, followed by at most two retry passes.

Deterministic tactic beam search makes repeated runs with the same hierarchy redundant; the general definition (Section 5.3) allows $N > 1$, which makes sense when using non-deterministic decoding strategies for tactic generation or other search randomization. To limit the cost of hierarchy assessment during the full training loop, we use smaller proof search budgets than during inference (see Appendix A.4). All candidates use the same assessment limits.

Success with k tactics gives usefulness $1 + 1/k$; failure gives zero. We measure the penalty in characters of the decoded hierarchy strings.

Passes and retries. We iterate over queues of pairs, processing their entries in batches of up to 2,048, and initialize the first queue with the reserved pairs in the order from Appendix A.2. Within each pass through a queue, we append pairs for which ConstructTriple returns fail to a separate retry queue, preserving their order. After the current queue has been processed in full, the constructed retry queue becomes the queue for the next pass. Pairs yielding an accepted triple leave the queue permanently; their triples remain available for replay. A pair also enters the retry queue when a solution was found during assessment but no candidate hierarchy attained the acceptance threshold.

We stop if the retry queue is empty, a complete pass through a queue produced no accepted triple, or three passes have been completed.

The initial pass comprises 17 batches of 2,048 and one of 707 entries; each retry pass has at most 18 batches. Thus, in Algorithm 1, take $T = 54$ and let $\text{SampleBatch}(t)$ return an empty batch after one of the above stopping conditions is met; the resulting remaining iterations make no parameter updates and may be omitted in implementation.

A.7 Replay policy

For the proposed schedule, let m_t denote the number of accepted triples in D_t . If $m_t > 0$, sample m_t entries uniformly without replacement from the concatenation of $D_0, \dots, D_{t-1}$ and concatenate them with all entries of D_t to form D . Thus, D contains $2m_t$ triples: equal numbers of new and replayed entries. The history contains at least the 15,000 seed triples from D_0 , while $m_t \leq 2,048$, so this sampling is always possible in the concrete configuration. If $m_t = 0$, choose D empty; Train then leaves the parameters unchanged.

A.8 Implementation details for Train

We use the five groups and objective of Section 5.4 with $n = 1$ and the prompt templates in Appendix A.3. The following are proposed settings that remain to be evaluated empirically.

Let $|D|$ count dataset entries and let $N_S(D)$ denote the number of solver prompt-response pairs constructed from D . For nonempty D , perform

$$U(D) = \max \left\{ \left\lceil \frac{|D|}{2} \right\rceil, \left\lceil \frac{N_S(D)}{8} \right\rceil \right\}$$

optimizer updates, traversing every group at least once (since $b_S = 8, b_F = b_H = 2$, see below). For our seed dataset, $N_S(D_0) = 42,397$ and $U(D_0) = 7500$.

Mini-batch counts	$b_S = 8, b_F = b_H = 2$ per level; $b = 16$.
Optimizer	AdamW; all shared parameters; otherwise default settings.
Peak learning rate	$5 \cdot 10^{-6}$ for seed training; 10^{-6} subsequently.
Schedule	$\lfloor U(D)/15 \rfloor$ linear warmup steps, then linear decay to zero.
Weight decay; gradient norm clip	0; 1.
Arithmetic; memory	bfloat16; gradient checkpointing.
Shuffle/training seeds	20260609 for seed training; $20260609 + t$ in iteration t .
Maximum prompt+response length	8192 tokens.

Each call starts from the current parameters with a fresh optimizer and freshly shuffled groups. Empty D leaves the parameters unchanged. Tokenize prompts and responses separately without adding implicit special tokens, then concatenate them; responses include `<|im_end|>`. Every pair must fit within the context limit; no truncation is applied.