LEVER: ADAPTIVE COST-AWARE PROOF SEARCH OVER AND/OR GRAPHS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mathematicians value proofs for more than correctness: among correct proofs, simplicity, purity and the computational cost of finding them vary widely. Yet LLM-powered theorem provers largely search for *any* correct proof, and improve its quality only after it is found. We propose LEVER, a proof search algorithm that makes the objective over correct proofs programmable and optimizes it *during search*. LEVER scores partial proofs over an AND/OR proof graph, combining realized objective values with predictions for open subgoals, so the objective guides search before a proof is complete. The same mechanism optimizes computational cost, proof length, topical impurity, and even their weighted combinations, while the Lean kernel enforces correctness. On PutnamBench in Lean 4, under matched budgets, LEVER costs 34% less than a strong single-conversation agent while raising the solve rate from 80% to 96%. On reducing topical impurity, *i.e.*, how far a proof strays from its theorem's subject, it improves over post-hoc refactoring (42% reduction against 33%) at two-thirds of the cost and more reliably; on proof length, the metric refactoring is built for, it approaches refactoring. Varying the objective's weights traces a quality–cost trade-off curve, so the user can choose how much a better proof is worth. Overall, LEVER is a performant, cost-efficient and tunable proof search algorithm for navigating the space of correct proofs.

1 INTRODUCTION

Mathematicians have long valued proofs for more than their correctness. A proof may be shorter, more elementary, or reveal structure that another proof obscures. The four-color theorem, for example, was proved in 1976 and formally verified decades later, yet a substantially different proof fifty years on exposed new structure in planar graphs and led to a near-linear-time coloring algorithm (Appel & Haken, 1977; Gonthier, 2008; Inoue et al., 2026). More broadly, the mathematical literature has debated the importance of characteristics like purity, simplicity and explanation (Steiner, 1978; Detlefsen & Arana, 2011; Thiele, 2003; Tao, 2007). The challenge is, therefore, not only to find a correct proof, but to choose among many.

This observation is crucial in context of LLM-powered theorem provers that may generate correct proofs differing sharply in the computation needed to find them, their length, and the machinery they invoke. Yet current systems largely separate these concerns: proof search returns any verified proof (NearAI, 2026; MistralAI, 2026; Rögnvaldsson et al., 2026), sometimes with mechanisms to reduce inference cost, and proof quality is improved only afterwards through post-hoc refactoring (Ahuja et al., 2026; Lu et al., 2026; Fu et al., 2026). We take a different view: **the objective over the space of proofs should be programmable, with correctness as a constraint**. Optimizing such objectives during search can be both cheaper and more effective than optimizing it afterwards: a post-hoc method revisits a proof already paid for, and possibly inherits a trajectory (mathematical argument) chosen for a different objective.

The key idea of LEVER is to score a proof *before* it is finished. A partial proof has some goals proved and others still open. LEVER scores it as the realized objective of the proved parts plus a predicted objective for each open goal; at any time only the partial proof with the best score is expanded, so the objective steers search from the beginning rather than after the task is complete. We support this mechanism by representing partial proofs as an AND/OR graph: a theorem (OR node) is proved by any one of its alternative decompositions (AND nodes), and an argument needs all of its subgoals

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

(OR nodes). We demonstrate that objectives such as computational cost, proof length or topical impurity, or a weighted sum of them compose over the proof graphs, allowing us to optimize them. In this process, correctness is never compromised: the Lean kernel checks every decomposition, and the assembled final proof.

On PutnamBench (Tsoukalas et al., 2024), with the same model and budget allocation as a strong agent that generates proofs in a single conversation, LEVER raises the solve rate under a \$3 budget from 80.0% to 96.3% while reducing average cost from \$1.44 to \$0.95, a 34% reduction. The gain comes from keeping contexts local: LEVER reads less than half the input tokens of the single-conversation agent. We then optimize the proof itself, for length and *topical impurity*, which we operationalize following work on purity of method (Detlefsen & Arana, 2011; Martinot, 2024): a proof accumulates impurity when it depends on results outside the subject of its theorem. So, length is often amenable to *local* compression, whereas purity can depend on the argument chosen at the start. Consistent with this, LEVER approaches post-hoc refactoring on length, but outperforms both refactoring and direct instruction on impurity, at two-thirds of refactoring's cost. Finally, varying the weight on quality metrics traces explicit quality–cost trade-off curves. This turns theorem proving from searching for *any verified proof* into searching for a verified proof under a user-specified objective, using the *same* search algorithm.

Our contributions are:

- **Objective-guided search over partial proofs.** A unified algorithm to optimize computation and proof quality before a proof is complete by using values that combine realized costs with predictions for open subgoals, propagated through an AND/OR graph;
- **Cheaper and stronger proof search.** On PutnamBench, 34% lower cost and a 96.3% solve rate, against 80.0% for a strong single-conversation agent;
- **Proof quality during construction.** Search-time optimization outperforms refactoring on topical impurity at two-thirds of its cost, approaches it on length, and admits explicit quality–cost trade-off curves.

2 RELATED WORK

LLMs for theorem proving. LLM provers range from tactic generators inside tree search (Polu & Sutskever, 2020; Yang et al., 2023; Xin et al., 2024; 2025) to whole-proof generators refined with compiler feedback (First et al., 2023; Wang et al., 2025; Lin et al., 2025). Decomposition has since become central: an informal draft becomes a formal sketch whose subgoals are proved separately (Jiang et al., 2023; Wang et al., 2024a; Dong et al., 2024; Ren et al., 2025), and agentic systems such as HILBERT (Varambally et al., 2026) and Goedel-Architect (Chung et al., 2026) complete these, while others learn from millions of attempts (Chen et al., 2025; Achim et al., 2025; Hubert et al., 2025; Tsoukalas et al., 2026); LEAP (Kung et al., 2026) maintains an AND/OR graph but explores it depth-first; all these systems search for *any* correct proof. On cost, Rögnvaldsson et al. (2026) decide when to restart from a new decomposition, and Quarry (Zhang et al., 2026) ranks decompositions by predicted hammer solvability; in either case simple linear models with limited features from Lean structures make predictions. Proof quality is improved by refactoring a finished proof (Ahuja et al., 2026; Gu et al., 2025; Lu et al., 2026; Fu et al., 2026). LEVER instead lets an explicit objective over cost and proof quality steer the search using LLM-based value functions.

Value-guided search. AO* finds the cheapest solution graph of an AND/OR graph under an admissible heuristic (Martelli & Montanari, 1973; Nilsson, 2014), with extensions to cyclic graphs (Hansen & Zilberstein, 2001) and games (Allis et al., 1994). Learned values replaced hand-written heuristics in Monte Carlo tree search (Kocsis & Szepesvári, 2006; Silver et al., 2016; 2018; Guez et al., 2018) and HyperTree Proof Search (Lample et al., 2022); closest in spirit, Retro* (Chen et al., 2020) learns the cost-to-go of AND/OR trees for retrosynthesis. LLM search over partial solutions (Yao et al., 2023; Hao et al., 2023; Zhou et al., 2024; Snell et al., 2024) is typically guided by self-evaluation or process rewards that score correctness (Lightman et al., 2023; Wang et al., 2024b). LEVER differs in three ways: its expansions are stochastic, so a goal can be redrawn; its priors learn from the draws already paid for; and its objective composes the remaining computational cost, the concern of rational metareasoning and costly search (Russell & Wefald, 1991; Sabbata et al., 2025; Weitzman, 1979), with arbitrary proof metrics.

3 LEVER

3.1 BACKGROUND

AND/OR graphs. An AND/OR graph is a bipartite DAG $G = (V_{OR} \sqcup V_{AND}, E)$ whose edges alternate between OR and AND nodes. An OR node is a *goal*, a theorem or lemma to prove; an AND node is a *decomposition* of a goal into *subgoals*, whose validity an oracle (here the Lean kernel) checks. The root OR node r is the theorem to prove. An OR node is solved if *any* child is, and an AND node if *all* its children are; an AND node with no children, a goal proved without new lemmas, is solved.

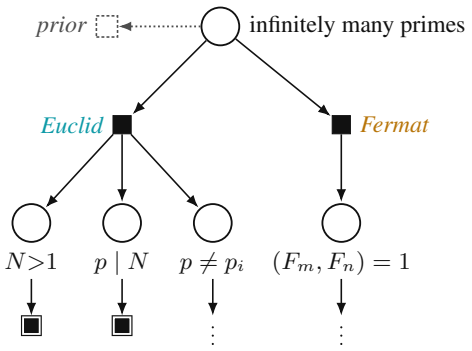


Figure 1: **An AND/OR graph.** Goals are circles, attempts squares; a goal may have several attempts (here Euclid's and Fermat's). Double border AND nodes are proved directly; dashed AND nodes are priors (§3.3).

Solution subgraphs. A *proof* or *solution subgraph* is a subgraph $S \subseteq G$ such that: (i) $r \in S$; (ii) for each OR node in S, exactly one AND child is picked; (iii) for every AND node in S, all its children are picked; (iv) the leaves in S are solved. Then each S maps to a unique proof written in Lean.

Costs. Every edge carries a nonnegative cost c. A decomposition needs to pay for all its subgoals whereas a goal takes the cheapest way to prove it, so values back up as:

$$V(n) = \sum_{d \in \mathrm{ch}(n)} c(n,d) + V(d) \quad (n \in V_{AND}), \qquad V(n) = \min_{d \in \mathrm{ch}(n)} c(n,d) + V(d) \quad (n \in V_{OR}),$$

with $V = 0$ at a solved leaf and $V = \infty$ at a false goal. $V(r)$ is then the cost of the cheapest solution subgraph, and the argmin at each OR node reconstructs it.

3.2 PROOF SEARCH

The problem of proof search can be modeled as an MDP with the following description:

- **States.** The graph built so far, with statuses and edge costs; fully observed.
- **Actions.** OR node(s) to expand, *i.e.*, which goals to decompose next or STOP if nothing to expand.
- **Transition.** transformation of the graph that adds decompositions (and the corresponding subgoals) to the graph; transitions are stochastic: expanding the same goal twice may not result in the same resulting graph.
- **Reward.** Score of $-s_t$ for the expansions at time step t, and $-V(r)$ at termination ($-\infty$ if unsolved). Episodes end when the policy decides to STOP or if the budget is exhausted.

Algorithm 1: LEVER. Each iteration selects goals along the current best solution subgraph, expands them, attaches a prior to each new subgoal, and backpropagates values, until selection returns nothing or the budget is spent.

Input: root r, budget B
`Prior(`r`)`
while spent $< B$ **do**
  $T \leftarrow$ `Select(`r`)`
  if $T = \emptyset$ **then break**
  foreach $g \in T$ **do**
    $D \leftarrow$ `Expand(`g`)`
    foreach $d \in D$ **do** `Prior(`d`)`
    `Backpropagate(`g`)`
return *the best solution subgraph of* r

The objective of proof search is to maximize the sum of rewards at the end of the episode or minimize the cost objective: $V(r) + \sum_t s_t$. At any step, the spend so far is sunk and cannot be changed, so a good policy minimizes the *cost-to-go*: the expected *remaining* spend plus $V(r)$.

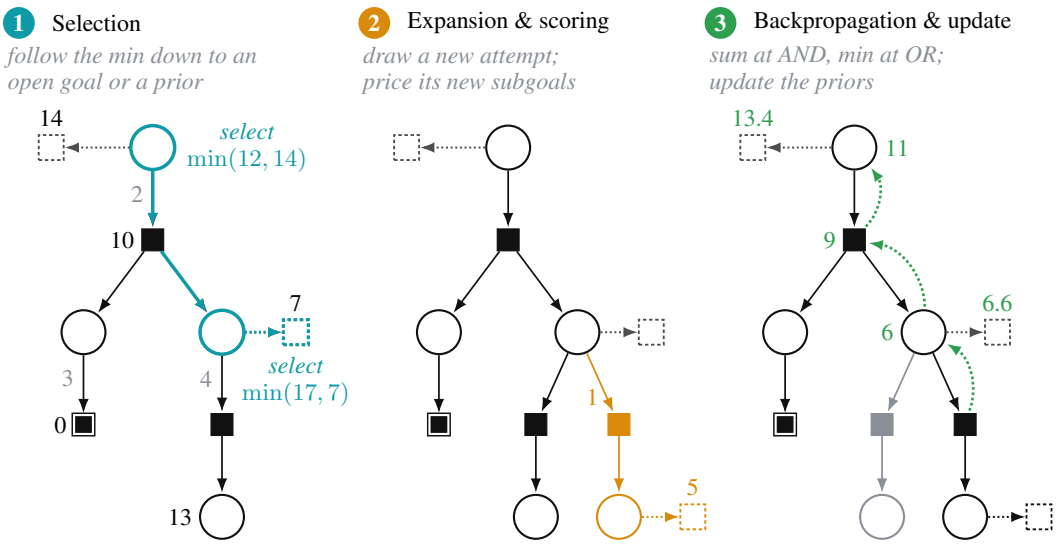


Figure 2: **One iteration of LEVER.** Grey numbers are edge costs and black numbers node values; later panels label only what their step changes. At the root's second subgoal the prior beats the realized attempt, so that interior goal is expanded again; the new attempt's value then passes up to the root, updating the priors on the way (Eq. 1, $n = 1$). The first attempt stays in the graph, paid for but no longer chosen (grey).

3.3 ADAPTIVE PROOF SEARCH WITH LEVER

Algorithm 1 shows how LEVER searches for proofs. The AO* algorithm (Nilsson, 2014) assumes a deterministic expansion function, but here the expander is an LLM: expanding a goal returns one decomposition drawn from an effectively unbounded set, and drawing again may return a cheaper one. Hence a goal that already has children can still be worth expanding, *i.e.*, any *interior* OR node is a candidate. To decide when, every OR node g carries one additional child, its *prior*: an AND node p that is never expanded, holding an estimate $\tilde{V}(g)$ of the cost of finishing g with a fresh decomposition. Each iteration has three steps (Figure 2):

1. Selection. Descend from the root along the current best solution subgraph, taking the $\arg\min$ child at an OR node and all children at an AND node. When the argmin is a prior, its goal is expanded; for a goal that already has children, this is a *redraw*. Values are noisy, so a small gap is no evidence that a fresh draw would be cheaper: a redraw must beat the incumbent by a factor δ. When selection returns nothing, the policy STOPs.

2. Expansion. An LLM expands each selected goal: it proposes a new decomposition (an AND node), the oracle checks it, and each subgoal it introduces is attached as a new OR node. The edge from the goal to the decomposition is scored with the cost of the expansion, and each new subgoal receives a prior from the value function (§4.4). Edges from a decomposition to its subgoals cost nothing: the subgoals are simply parsed from the compiled proof.

3. Backpropagation. New costs pass upward through the recursion of §3.1, updating every prior on the path. We model the log cost of a fresh decomposition of g as normal around an unknown mean μ_g with spread σ (draws of the same goal differ) and the value function's prediction $V_0(g)$ (§4.4) as off from μ_g by ρ. After k decompositions with costs $v_1, \ldots, v_k$, the posterior μ_g is normal with:

$$\hat{\mu}_g = \frac{n \log V_0(g) + \sum_{i=1}^{k} \log v_i}{n + k}, \qquad n = \sigma^2/\rho^2, \tag{1}$$

so the prior estimate is worth n draws and each realized draw, one. The value estimate is the posterior geometric mean, $\tilde{V}(g) = \exp(\hat{\mu}_g)$. Realized costs thus correct the value function's errors: this is how the search learns from what it has paid for. A failed attempt leaves no cost to learn from, and instead raises the estimate by one σ. Metrics on a linear scale use the same form without logarithm.

Objective. The steps above hold for any cost that composes over the algebra of §3.1, which metrics such as proof length do. The spend term $\sum_t s_t$ in the MDP objective of §3.2 does not: it accumulates over the whole search, including draws the final proof never uses. We therefore split it into what has been spent (sunk; cannot be optimized) and what remains (can be optimized). The remaining part does compose over the graph, and we call it $V_{\texttt{comp}}(n)$: the computational cost still needed to finish n, where realized edges add nothing and open goals add their prior's estimate. With this adjustment, any metric m, backed up as V_m with edge costs c_m, can be balanced against computational cost at a price of λ_m dollars per unit: $C(n) = V_{\texttt{comp}}(n) + \sum_m \lambda_m V_m(n)$. With every $\lambda_m = 0$, LEVER minimizes computational cost alone.

Exactness under exact priors. With exact priors, the backed-up values are exact: for any non-negative cost that adds along edges, $V(n)$ is the cost of the cheapest solution subgraph rooted at n (counting a subgoal once per use), and $V(n) = \infty$ exactly when none exists. We also share a formally verified proof of this in Lean (Appendix D). Real value functions are noisy, which the update of Equation 1 and the margin δ account for; their theoretical analysis is left for future work. When a subgoal is shared by several decompositions, paying for it once instead makes the cheapest solution subgraph NP-hard to find in general (Sahni, 1974); LEVER instead charges each shared subgoal to a single owner (Appendix A).

4 IMPLEMENTING LEVER WITH LEAN

4.1 SETUP

Model and Pricing. We use DeepSeek-V4-Flash-0731 (DeepSeek-AI et al., 2026) for every LLM call, in LEVER and in all baselines, to match the baseline's original setting. We price tokens at the official rates of \$0.22, \$0.007 and \$0.66 per million input, cached input and output tokens (DeepSeek-AI, 2026). Cost therefore depends heavily on the cache-hit rate. Served through multiple providers, API calls experience mean hit rate per call at 93.7% for NearAI (2026), whose single long conversation re-reads its own history, and 86.2% for LEVER, which starts many shorter conversations. To compare the searches rather than the caching, we fixed one 90% rate, between the two, before the evaluation runs. All costs in this paper are re-priced at this rate, and a problem counts as solved within a budget B only if its re-priced cost is at most B; Appendix C.3 reports the observed costs.

Data. We use PutnamBench (Tsoukalas et al., 2024), originally containing 672 competition problems formalized in Lean 4 (v4.31.0). Most are easy for a strong agent, while the hardest can take days of repeated attempts, too long for controlled comparisons. Hence, we rank the problems by what the published run of NearAI (2026) spent to solve them and take ranks 51–150, a slice that is hard but tractable; Appendix C.1 gives the full procedure and its biases. 20 of these calibrate the value functions (§4.4), and the other 80 form the evaluation set (§ 5).

4.2 AGENTIC SYSTEM DESIGN

We discuss salient aspects of LEVER's agent for goal expansion, leaving detailed descriptions in Appendix A.

Expanding a goal. Each expansion is one conversation of an LLM agent (Yao et al., 2022) with a Lean workspace, with tools for Lean compilation (Santos et al., 2025), Mathlib retrieval (Lu et al., 2025), Python, and file I/O; it ends by submitting a file or reporting the goal false. To *decompose* a goal, the agent writes a Lean proof that relies on open subgoals; small lemmas are proved in place and larger ones are left open. Each open helper becomes a subgoal, to be closed in a future iteration. It annotates dependencies among open subgoals with the `sorry_using` tactic of LeanArchitect (Zhu et al., 2026; Chung et al., 2026), so that subgoals can share work. To prove a goal *directly*, the agent writes a complete proof with no open subgoals; §5 states when this is enforced.

Context. Before the search, a planner writes an informal proof of the problem, following Varambally et al. (2026) and Chung et al. (2026). Every expansion carries this plan in its system prompt, and the planner's cost is included in every cost we report for LEVER. A subgoal's conversation starts from its parent's compacted output, or its full transcript if short, so the agent knows why the subgoal was introduced without carrying its parent's work (Appendix A). Every LLM inference call is made

Plan 1: slice the triangle and integrate 

```
lemma std_tri_volume : (MeasureTheory.volume
    (convexHull ℝ ···)).toReal = 1 / 2 := by
  ⋮
  rw [← intervalIntegral.integral_of_le ···]
  rw [intervalIntegral.integral_sub]
  · rw [intervalIntegral.integral_const, ···]
  · exact Continuous.intervalIntegrable ···
```

Plan 2: halve the unit square 

```
lemma std_tri_volume : (MeasureTheory.volume
    (convexHull ℝ ···)).toReal = 1 / 2 := by
  ⋮
  have hvol_union : MeasureTheory.volume
      (stdTri ∪ stdTri') = ··· := by
    have hia :=
      MeasureTheory.measure_union_add_inter ···
```

Figure 3: **Two proofs of one lemma: the unit triangle has area $\frac{1}{2}$.** The statement is about measure and convex hulls. Plan 1 slices the triangle and integrates. Its calculus references, such as `intervalIntegral.integral_sub`, are impure (underlined red): the statement never calls for integrals. Plan 2 halves the unit square. It argues in measure theory, and a reference such as `MeasureTheory.measure_union_add_inter` is pure (underlined green). Over the lemma and the helpers it uses, the metric flags 25 references in Plan 1's proof and 6 in Plan 2's. Excerpts from compiled proofs; dots mark elisions.

with 32K completion tokens at temperature 1.0; planning with high-effort reasoning and expansion with low-effort reasoning.

Disproofs and soundness. The agent may report a goal false, with an informal reason (formal disproofs as in Lin et al. (2025) are not integrated for simplicity; these can wander into their own long trajectories). This ends the decomposition, and values are updated as in §3.3. When the root is solved, the solution subgraph is assembled into one Lean file and checked by the kernel. A problem counts as solved only if this file compiles using only the standard axioms, without `sorry`, `admit`, `axiom` or `native_decide`.

4.3 Cost Objectives

Correctness and computational cost. Correctness is enforced as a constraint: an unsolved goal costs ∞, so every objective is minimized over proofs that Lean type-checks. Computational cost, `comp`, is the dollars spent on LLM calls at the rates of §4.1. Spend already incurred cannot be recovered, so the search never optimizes it: each decision minimizes the *remaining* cost of finishing the proof, as estimated by the value function (§4.4), and every expansion replaces part of that estimate with what it actually cost (§3.3).

Proof length. We count the Lean tokens (`len`) a proof adds to its theorem statement, helper lemmas and definitions included and comments excluded, tokenized as in Gu et al. (2025). It is easy to measure but also easy to game, *e.g.*, a goal closed by one call to `nlinarith` is short while saying nothing about the mathematics behind it.

Topical impurity. (`imp`) instead measures the source of content in a proof. A proof is *pure* when it uses only what its theorem is about (Detlefsen & Arana, 2011). Purity is not computable in general, since a human must decide what a theorem's content is (Martinot, 2024), so we read the content off Mathlib's structure (Community, 2020). Every library name has a namespace and a directory, *e.g.*, `ZMod.wilsons_lemma` is about `ZMod` and lives in `Mathlib/NumberTheory`. A theorem's *subject* is the namespaces and directories of the names its statement uses. A referenced library theorem is *impure* when it falls outside the subject on both accounts; a generic lemma with no namespace, such as `mul_pos`, is impure only if it also lies outside a fixed set of infrastructure directories (*e.g.*, `Mathlib/Data/Nat`) that proofs of every topic use. Overall, we measure impurity as the number of impure references. See Figure 3 for an illustration of pure and impure proofs. We note here that this is a proxy for measuring mathematical (im)purity; further discussion and limitations of this metric are in Appendix B.

4.4 Value Functions

Predicting `comp` and `len`. For `comp` and `len`, an LLM reads the open goal and picks one of seven bins on a $\log_2$ scale ($0.016–$0.512 and 512–16,384 Lean tokens), and the goal is priced at its bin's

Method	Solved	Cost ($)	Tokens (M) In	Tokens (M) Out
NearAI	0.800	1.44	42.0	**0.36**
LEVER	**0.963**	**0.95**	**17.9**	0.62

Table 1: **Cost of proof search at a $3 budget per problem.** $N = 80$. Cost and tokens are per problem. Solved problems contribute the cost to finish; unsolved problems are charged the full budget with tokens spent up to the call when cut-off.

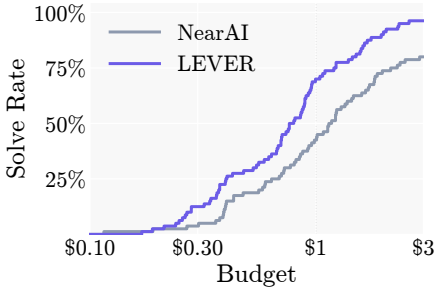


Figure 4: **Solve rate against budget.** A problem solved at cost c counts as solved at every budget $\geq c$.

geometric center. Further, for `len`, the estimate is 0.2 times the bin's center, deliberately underestimating length so that shorter proofs get explored. The prompt carries 40 in-context examples from offline runs on the calibration problems. Using an LLM as a value function prior is inexpensive: these calls make up only 0.89% of LEVER's spend, and is always included in the reported costs.

Predicting impurity. Impurity has a real floor, since a proof can cite nothing outside its subject, so the natural estimate for an open goal is 0. In practice this made search unstable: every open goal looked pure, so redrawing always beat keeping a decomposition, and graphs grew huge. We instead score every open goal at a constant 4 references, the 25th percentile of impurity over calibration goals.

Calibration. On the calibration problems only, we measure ρ, the value function's error against realized values, and σ, the spread of a goal's realized value across repeated draws: in log space for `comp` and `len`, and in references for `imp`. For `comp` and `len`, the redraw margin is one spread, $\delta = e^{\sigma}$. Appendix C.2 documents every constant and how it is measured.

5 EXPERIMENTS

We focus on two main questions: First, can LEVER optimize computational costs during proof search (§ 5.1)? Second, how effective is LEVER at recovering high quality proofs during search (§ 5.2)? All runs share the setup of § 4.1: one model, one Lean version and one price table, for LEVER and every baseline alike.

5.1 OPTIMIZING COMPUTATIONAL COST

Baseline. Our main baseline is the open-source agent of NearAI (2026), the top entry on the PutnamBench leaderboard at the time of writing. It is a single-conversation agent: one conversation holds the whole proof and every compiler message. We run the released agent with its documented 1M-token context and compaction, under the same model, prices and Lean version as LEVER (§4.1).

Setup. We compare LEVER with this agent. LEVER uses the `comp` value function (§4.4) during decomposition. In our exploration, we found that the LLM used here tends to over-decompose, even on simple problems. So, a goal five levels below the root is always proved directly, and LEVER uses its value function to force a direct proof for any goal priced under bin 5 (§4.4). Each problem has a budget of $3 at the re-priced rate (§4.1), and a problem not solved within it is charged the full $3. We evaluate on the 80 evaluation problems of §4.1.

Decomposition makes proof search cheaper. Table 1 compares the two systems at $3. LEVER solves 96.3% of the problems, against 80.0% for NearAI, and costs 34% less. This is not a trade-off between solving and saving: LEVER does better on *both*. Further, the failure charge is real spend: 94% of NearAI's failures spent the full $3 without producing a complete proof. Figure 4 shows that this holds at almost every budget: from $0.25 on, LEVER solves more problems than NearAI. The savings come from how LEVER manages context. NearAI writes fewer tokens than LEVER (357K against 621K per problem), but it carries the context of the *full* problem on every turn, and reads

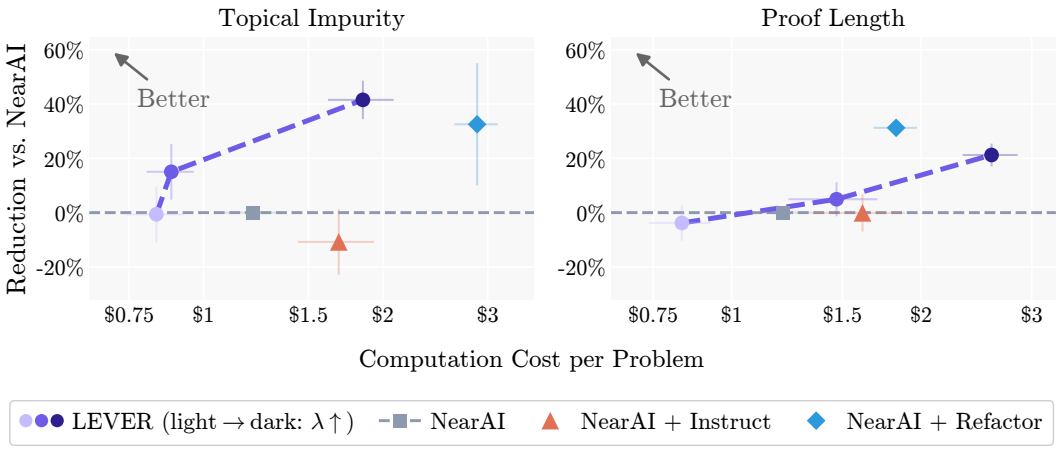


Figure 5: **Proof quality against computation cost.** Mean reduction of each metric relative to NearAI's proof of the same problem, against mean cost per problem, for topical impurity (left) and proof length (right), on the same 20 problems. LEVER's points are $\lambda = 0, 0.01, 0.1$ for impurity and $\lambda = 0, 10^{-4}, 10^{-3}$ for length, light to dark; $\lambda = 0$ is LEVER run with its \$3 budget and routing to direct attempts (§ 5.1); the $\lambda > 0$ runs have \$4.5 and no routing. A run that fails or exceeds \$4.5 is scored as NearAI's proof at \$4.5; + *Refactor* is also charged for NearAI's run. Error bars: ± 1 standard error over problems.

42.0M input tokens per problem. LEVER works on each subgoal in its own local context and reads 17.9M, less than half. At the cost each run actually incurred, with its own cache hits, LEVER still solves more problems than NearAI (0.963 vs. 0.863) and costs 16% less (Appendix C.3).

Identifying cheaper strategies. LEVER's algorithm uses the value function to *reject* decompositions whose subgoals sum up to be more expensive than the goal itself, and draw a new decomposition. On `putnam_1972_a3`, for example, LEVER rejects its first decomposition from the value predictions alone, before working on any of its subgoals, and proves the theorem through a second decomposition predicted to be cheaper (Appendix C.4.1). However, such cases are rare, and not every redraw leads to savings: every subgraph that is given up on represents a sunk cost that is deducted from the global budget.

5.2 PROOF QUALITY OPTIMIZATION

Setup. We optimize the two metrics of §4.3, topical impurity and proof length, on $N = 20$ problems drawn from successful runs of NearAI in §5.1. We write LEVER $^{\lambda}_{m}$ for LEVER minimizing the objective $C = \texttt{comp} + \lambda \cdot m$ with $m \in \{\texttt{imp}, \texttt{len}\}$. Further, LEVER $^{0}_{m}$ is always written as LEVER (*i.e.*the run from § 5.1). Unlike LEVER, which enforces no decomposition under a certain value, LEVER $^{\lambda}_{m}$ runs impose no such restriction to allow greater exploration during search. We run each metric m at two weights, and each run (including all baselines; see below) is with budget \$4.5 (50% higher than in § 5.1 to accommodate cost of quality optimization on top of correctness).

Baselines. Existing proof refactoring methods that target length, such as Lean Refactor (Lu et al., 2026) and ProofOptimizer (Gu et al., 2025) are not publicly available, while ImProver (Ahuja et al., 2026) targets a much older Lean version (v4.9) and model; none can be run in our controlled setup, with the same model, Lean version and budget. To our knowledge, no existing method optimizes topical impurity, and none optimizes proof quality *during search*. We therefore build baselines atop the strong agent from NearAI (2026). + *Instruct* steers the agent during its search: the start of its prompt states the metric and asks for a proof that minimizes it. + *Refactor* improves the proof after the search, as post-hoc refactoring does: it is handed NearAI's finished baseline proof (§ 5.1) and asked to improve the proof. Together they cover the two alternatives to optimizing during search: steering by instruction and repairing afterwards. See Appendix C.5 for details.

Reporting. We use the proofs delivered by NearAI from § 5.1 as a common anchor: we report relative reduction of metric m relative to this baseline proof, averaged over all problems. A run that

	$m = \texttt{imp}$		$m = \texttt{len}$	
Method	ΔImpurity	ΔLength	ΔLength	ΔImpurity
LEVER (λ=0)	$+1\%$	$+4\%$	$+4\%$	$+1\%$
LEVER $^{\lambda_1}_{m}$	-15%	$+8\%$	-5%	$+3\%$
LEVER $^{\lambda_2}_{m}$	-42%	$+10\%$	-21%	-18%

Table 2: **Optimizing one metric, measured on both.** Mean change of topical impurity and of proof length relative to NearAI's proof of the same problem; negative is better. λ_1, λ_2 are $0.01, 0.1$ when optimizing impurity and $10^{-4}, 10^{-3}$ when optimizing length. A run that fails or exceeds \$4.5 is scored as NearAI's proof.

fails or exceeds its budget is scored at the same metric as the baseline and charged at \$4.5. Finally, because + *Refactor* starts from the NearAI proof baseline, its cost is the original cost plus the cost to refactor.

Length optimization. We run LEVER $^{\lambda}_{\texttt{len}}$ for $\lambda \in \{0.0001, 0.001\}$ and plot the results in (Figure 5, right). The proof length objective is well optimized by refactoring: previous works have independently reported trained refactoring models shorten PutnamBench proofs by 57% (Gu et al., 2025) and competition proofs by over 70% (Lu et al., 2026), in their respective settings. Our + *Refactor* baseline agrees: it shortens NearAI's proofs by 31% at \$1.83 per problem. At $\lambda = 10^{-3}$, LEVER approaches + *Refactor*: its proofs are 21% shorter, though at a higher cost of \$2.59, indicating that searching for short proofs may be more challenging than searching for a correct proof first followed by refactoring.

Impurity optimization. We run LEVER $^{\lambda}_{\texttt{imp}}$ for $\lambda \in \{0.01, 0.1\}$ with results in Figure 5 (left). Unlike length, refactoring is not built for topical impurity because removing an out-of-subject reference usually means changing the argument rather than tidying it. Our + *Refactor* baseline reduces impurity by 33% at \$2.88 per problem. At $\lambda = 0.1$, LEVER's proofs are 42% purer, at \$1.85, about two-thirds of + *Refactor*'s cost. LEVER is also more reliable: the standard deviation of its per-problem reduction is 32 points, against 101 for + *Refactor*, whose rewrites can backfire badly (on `putnam_2007_a2`, from 29 out-of-subject references to 132). Instructing the agent at the start to look for pure proofs does not help either: + *Instruct*'s proofs are, on the contrary, 11% less pure than the baseline, indicating that the agent loses context of the search objective in its single long session. This indicates that, when a metric depends on the proof strategy itself, optimizing it during search and treating it locally, works best.

Trade-offs among metrics. As shown in Figure 5, LEVER allows tracing a cost-quality trade-off curve: as λ rises, proof quality improves and the search costs more. Since λ is a conversion rate between metrics, a user can choose how much quality is worth paying for, and move smoothly along this frontier. In contrast, any fixed policy, such as the baselines considered here, admits only single points in this space. We observe that LEVER's search also reveals relationships among various quality metrics (Table 2). Shortening a proof makes it purer: as proofs get shorter ($+4\% \to -21\%$), impurity reduces as well ($+1\% \to -18\%$). This is likely because as a proof is shortened, impure references are removed for free. However, the reverse does not hold: optimizing for impurity alone ($+1\% \to -42\%$) makes proofs longer ($+4\% \to +10\%$); this supports our observation that optimizing impurity requires a different strategy than what works for length reduction.

6 CONCLUSION

We presented LEVER, a proof search algorithm that treats correctness as a constraint and makes the objective over correct proofs programmable. By valuing partial proofs over an AND/OR graph, combining realized costs with predictions for open goals, LEVER optimizes that objective during search rather than repairing a proof afterwards. On PutnamBench, it is 34% cheaper than a strong single-conversation agent while solving more problems (96% against 80%). On topical impurity, search-time optimization removes more out-of-subject references than post-hoc refactoring at two-thirds of its cost, and on proof length it approaches refactoring. A single weight trades quality against cost, letting a user choose how much a better proof is worth.

AI USE STATEMENT

Beyond the LLMs that are the object of our experiments, we used generative AI tools to write code, run and monitor experiments, survey the literature, check ideas, draft proofs, and polish the writing. AI tools had no or minimal involvement in developing the main ideas of the paper, or in interpreting and iterating on experimental results. We have reviewed all AI-assisted work, and we take responsibility for the final content of this paper, including text, claims and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The code to reproduce the results of this paper will be shared upon acceptance.

REFERENCES

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: Imo-level automated theorem proving, 2025. URL https://arxiv.org/abs/2510.01346.

Riyaz Ahuja, Jeremy Avigad, Prasad Tetali, and Sean Welleck. Improver: Agent-based automated proof optimization, 2026. URL https://arxiv.org/abs/2410.04753.

L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-number search. *Artif. Intell.*, 66(1):91–124, March 1994. ISSN 0004-3702. doi: 10.1016/0004-3702(94)90004-3. URL https://doi.org/10.1016/0004-3702(94)90004-3.

Kenneth Appel and Wolfgang Haken. Every planar map is four colorable. Part I: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

Binghong Chen, Chengtao Li, Hanjun Dai, and Le Song. Retro*: Learning retrosynthetic planning with neural guided a* search. In *Presented at ICML 2020*, 2020.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, Shufa Wei, Yonghui Wu, Yuchen Wu, Yihang Xia, Huajian Xin, Fan Yang, Huaiyuan Ying, Hongyi Yuan, Zheng Yuan, Tianyang Zhan, Chi Zhang, Yue Zhang, Ge Zhang, Tianyun Zhao, Jianqiu Zhao, Yichi Zhou, and Thomas Hanwen Zhu. Seed-prover: Deep and broad reasoning for automated theorem proving, 2025. URL https://arxiv.org/abs/2507.23726.

Jui-Hui Chung, Ziyang Cai, Zihao Li, Qishuo Yin, Rohit Agarwal, Simon Park, Rodrigo Porto, Narutatsu Ri, Ziran Yang, Shange Tang, Xingyu Dang, Hongzhou Lin, Mengdi Wang, Danqi Chen, Chi Jin, Liam H Fowl, and Sanjeev Arora. Goedel-architect: Streamlining formal theorem proving with blueprint generation and refinement, 2026. URL https://arxiv.org/abs/2606.06468.

The Mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20, pp. 367–381. ACM, January 2020. doi: 10.1145/3372885.3373824. URL http://dx.doi.org/10.1145/3372885.3373824.

DeepSeek-AI. DeepSeek-V4-Pro GA Release. https://api-docs.deepseek.com/news/news260813, August 2026. Accessed: 2026-09-21.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin,

Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Michael Detlefsen and Andrew Arana. Purity of methods. *Philosophers*, 11, 2011.

Kefan Dong, Arvind V. Mahankali, and Tengyu Ma. Formal theorem proving by rewarding LLMs to decompose proofs hierarchically. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024. URL https://openreview.net/forum?id=D83tiHiNfF.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models, 2023. URL https://arxiv.org/abs/2303.04910.

Yiming Fu, Peixuan Liu, Zichen Wang, and Kun yuan. Proof-refactor: Refactoring generated formal proofs into modular artifacts, 2026. URL https://arxiv.org/abs/2606.03743.

Georges Gonthier. Formal proof—the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

Alex Gu, Bartosz Piotrowski, Fabian Gloeckle, Kaiyu Yang, and Aram H. Markosyan. Proofoptimizer: Training language models to simplify proofs without human demonstrations, 2025. URL https://arxiv.org/abs/2510.15700.

Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets, 2018. URL https://arxiv.org/abs/1802.04697.

Eric A. Hansen and Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1):35–62, 2001. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(01)00106-0. URL https://www.sciencedirect.com/science/article/pii/S0004370201001060.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.507. URL http://dx.doi.org/10.18653/v1/2023.emnlp-main.507.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, Nov 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09833-y. URL http://dx.doi.org/10.1038/s41586-025-09833-y.

Yuta Inoue, Ken-ichi Kawarabayashi, Atsuyuki Miyashita, Bojan Mohar, Carsten Thomassen, and Mikkel Thorup. The four color theorem with linearly many reducible configurations and near-linear time coloring, 2026.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=SMa9EAovKMC.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou (eds.), *Machine Learning: ECML 2006*, pp. 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-46056-5.

Po-Nien Kung, Linfeng Song, Dawsen Hwang, Jinsung Yoon, Chun-Liang Li, Simone Severini, Mirek Olšák, Edward Lockhart, Quoc V Le, Burak Gokturk, et al. Leap: Supercharging llms for formal mathematics with agentic frameworks. *arXiv preprint arXiv:2606.03303*, 2026.

Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving, 2022. URL https://arxiv.org/abs/2205.11491.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2025. URL https://arxiv.org/abs/2508.03613.

Jialin Lu, Kye Emond, Kaiyu Yang, Swarat Chaudhuri, Weiran Sun, and Wuyang Chen. Lean finder: Semantic search for mathlib that understands user intents. *arXiv preprint arXiv:2510.15940*, 2025.

Jialin Lu, Soonho Kong, Rodrigo Stehling, Kaiyu Yang, Zhangyang Wang, Weiran Sun, and Wuyang Chen. Lean refactor: Multi-objective controllable proof optimization via agentic strategy search, 2026. URL https://arxiv.org/abs/2605.20244.

Alberto Martelli and Ugo Montanari. Additive AND/OR Graphs. In *International Joint Conference on Artificial Intelligence*, pp. 1–11, 1973. URL https://mlanthology.org/ijcai/1973/martelli1973ijcai-additive/.

Robin Martinot. Ontological purity for formal proofs. *Review of Symbolic Logic*, 17(2):395–434, 2024. doi: 10.1017/s1755020323000333.

MistralAI. Leanstral. https://github.com/mistralai/LeanstralSafeVerify/blob/main/LeanstralReport.pdf, Jul 2026.

NearAI. putnambench-deepseek: An autonomous DeepSeek-driven agent that solved PutnamBench in Lean 4. https://github.com/nearai/putnambench-deepseek, 2026. Accessed 14 September 2026.

Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020. URL https://arxiv.org/abs/2009.03393.

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

Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 49(1):361–395, 1991. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(91)90015-C. URL https://www.sciencedirect.com/science/article/pii/000437029190015C.

Kári Rögnvaldsson, Chenhao Sun, Jasper Dekoninck, and Martin Vechev. Optimizing the cost-quality tradeoff of agentic theorem provers in lean, 2026. URL https://arxiv.org/abs/2606.04883.

C. Nicolò De Sabbata, Theodore R. Sumers, Badr AlKhamissi, Antoine Bosselut, and Thomas L. Griffiths. Rational metareasoning for large language models, 2025. URL https://arxiv.org/abs/2410.05563.

Sartaj Sahni. Computationally related problems. *SIAM Journal on Computing*, 3(4):262–279, 1974. doi: 10.1137/0203021.

Marco Dos Santos, Hugues de Saxcé, Haiming Wang, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. Kimina lean server: A high-performance lean server for large-scale verification, 2025. URL https://arxiv.org/abs/2504.21230.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL http://dx.doi.org/10.1038/nature16961.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, Dec 2018. ISSN 1095-9203. doi: 10.1126/science.aar6404. URL http://dx.doi.org/10.1126/science.aar6404.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Mark Steiner. Mathematical explanation. *Philosophical Studies: An International Journal for Philosophy in the Analytic Tradition*, 34(2):135–151, 1978.

Terence Tao. What is good mathematics? *Bulletin of the American Mathematical Society*, 44(4): 623–634, 2007.

Rüdiger Thiele. Hilbert's twenty-fourth problem. *The American mathematical monthly*, 110(1): 1–24, 2003.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=ChKCF75Ocd.

George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco J. R. Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, Lei Yu, Aja Huang, Miklós Z. Horváth, Andrew Ferraiuolo, Henryk Michalewski, Edward Lockhart, Codrut Grosu, Thomas Hubert, Matej Balog, Pushmeet Kohli, and Swarat Chaudhuri. Advancing mathematics research with ai-driven formal proof search, 2026. URL https://arxiv.org/abs/2605.22763.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning, 2026. URL https://arxiv.org/abs/2509.22819.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. LEGO-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=3f5PALef5B.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning, 2025. URL https://arxiv.org/abs/2504.11354.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439. Association for Computational Linguistics, 2024b. doi: 10.18653/v1/2024.acl-long.510. URL http://dx.doi.org/10.18653/v1/2024.acl-long.510.

Martin L. Weitzman. Optimal search for the best alternative. *Econometrica*, 47(3):641–654, 1979. URL https://onlinelibrary.wiley.com/doi/abs/0012-9682(197905)47:3<641:OSFTBA>2.0.CO;2-1.

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

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Kai Shen. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving, 2025. URL https://arxiv.org/abs/2502.03438.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models, 2023. URL https://arxiv.org/abs/2306.15626.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems 36*, NeurIPS 2023, pp. 11809–11822. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023. doi: 10.52202/075280-0517. URL http://dx.doi.org/10.52202/075280-0517.

Ning Zhang, Nongyu Di, Zenan Li, Yuan Yao, and Xiaoxing Ma. Planning to hammer: Difficulty-aware decomposition for automating rocq proofs, 2026. URL https://arxiv.org/abs/2606.17981.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2024. URL https://arxiv.org/abs/2310.04406.

Thomas Zhu, Pietro Monticone, Jeremy Avigad, and Sean Welleck. Leanarchitect: Automating blueprint generation for humans and ai, 2026. URL https://arxiv.org/abs/2601.22554.

A LEVER: SYSTEM DESIGN

Planner. Before the search, one planner conversation per problem writes an informal proof. It has `search_mathlib` and `python`, runs at high reasoning effort, and is capped at $0.20. The plan is written once per problem and reused by every LEVER run on that problem. Every expansion carries it in its system prompt, and is told that the plan is a guide, which the task in hand supersedes where the two disagree.

Tools. An expansion edits the goal's Lean file in a buffer (`view`, `replace`, `replace_all`, `delete_lines`), and any edit can compile the file in the same step. It can also check a Lean snippet without touching the buffer (`lean_check`), search Mathlib (`search_mathlib`), run Python (`python`), compile the file against every submission rule (`lean_compile`), and submit the last file that compiled (`submit_proof`). A subgoal's conversation may instead report its goal false (`disprove`).

Decomposing a goal. The agent writes the proof of the target in the file it is given, and may add two kinds of helper lemmas above it. *Closed* helpers are small steps proved in place. *Open* helpers are large steps left as `sorry_using [...]`, where the list names the declarations in the file that the helper's own proof will use; each becomes a subgoal, proved in a later conversation. Every declaration it adds carries a LeanArchitect blueprint attribute, an informal statement and proof in words, so a subgoal's conversation knows what its lemma is for. A file is accepted only if its header, the declarations it was handed and the target's signature are unchanged; the target is its last declaration, with no `sorry` in its body; every helper lemma is either proved or open with exactly `sorry_using [...]` as its body; and it uses none of `native_decide`, `axiom`, `admit`, or search tactics ending in `?`.

Shared subgoals. A subgoal's identity is its Lean file with the blueprint annotations stripped: the lemma, together with the open helpers it depends on through `sorry_using`. Two decompositions that introduce the same lemma, or a subgoal whose proof uses a sibling it depends on, therefore reach the same OR node, which is proved once. A proof should then pay for a shared subgoal once, not once per use as the recursion of §3.1 does, and under that accounting finding the cheapest solution subgraph is NP-hard (Sahni, 1974).

LEVER instead charges each shared subgoal to one goal, the one whose decomposition first introduced it (or the next, if that decomposition dies). Every alternative decomposition of that goal pays for the subgoal, not only the one that introduced it, so the goal compares its alternatives on what each would cost; decompositions of other goals use the subgoal free, since it is proved once for all of them. Ownership depends only on the order in which nodes were created, not on which alternatives are chosen, so values cannot oscillate; the cost is that a shared subgoal's price drops out of the root's value when its owner chooses an alternative that does not use it.

Proving directly. The direct arm uses the same plan, context and tools, but may leave no helper open.

Context inheritance. A subgoal's conversation starts from its parent's: from the full transcript if the parent's conversation is under 20K tokens, and otherwise from the parent's compacted output. Any conversation that reaches 128K tokens is likewise replaced by its compacted output.

Redraws. When a goal is expanded again, its new conversation is told what the earlier attempts found: which of their helpers were proved, which were reported false and why, which were never proved, and what the value function priced the rest at.

Disproofs and failures. A disproof is a claim about the goal rather than the attempt: the goal is closed as false, its value becomes ∞, and later decompositions of its parent are told the statement is false and why. An attempt that ends without an accepted file is a failure. Its spend is charged, and its goal's estimate rises by one σ (§3.3).

Parallelism and budget. Each iteration expands up to 25 selected goals concurrently, one attempt each. The spend of every LLM call, whether expansion, value function or planner, is charged to the problem's budget as it is made, and running conversations read that spend between calls, so the search stops once the budget is reached rather than after the attempts in flight return.

Prompts. Listings 1–3 give, verbatim, the planner's prompt, the system prompt of every expansion, and the prompt of a decomposition. <statement> stands for the goal's Lean file, <target> for the declaration to prove, and <plan> for the planner's output. A subgoal's conversation, which continues from its parent's, opens instead with "The buffer now holds <target>, a lemma you left open above, as its own file, shown below. Your task is to write the proof of <target> in it." The direct arm's prompt has the same structure, but requires every new helper to be proved.

```
Your task is to write a natural language proof of the goal `<target>` found in
the Lean 4 file below.

## File

```lean4
<statement>
```

Your final submission should be the complete mathematical argument that proves
`<target>`, written in natural language prose.

Instructions

- **Pick a strategy that is easy to formalize**: A later stage will write
  your plan out in Lean 4 against Mathlib, and that is what your plan is judged
  on. So the approach to submit is the one that can be implemented using
  Mathlib machinery that already exists. Every approach may need results
  Mathlib lacks: for each candidate, list what is missing, and pick the one
  whose missing pieces are smallest and most elementary.

- **Survey Mathlib before you commit to an approach**: Use `search_mathlib` to
  investigate what exists in Mathlib, and explicitly name the Mathlib declarations
  that your argument depends on.

- **No Lean proofs, but do name Lean results**: Do NOT write proof tactics or
  segments of a Lean proof. However, you should name the Mathlib declarations
  your steps rest on. Every name you give must be one a search actually
  returned.

- **Verify each step**: Make sure that each step in your argument is logically
  sound and follows from the previous steps. The full proof that you submit should
  be complete and correct, and should not rely on any unproven claims or assumptions.
  Each intermediate claim must be true as a Lean statement using Mathlib, not just
  informally.

- **A missing theory is a reason to change route**: If a step needs a general
  theorem Mathlib does not have, do NOT plan to build that theorem. Look for an
  argument that uses the specific hypotheses you have to prove only the special
  case you need. Build a missing result only when no route avoids it, and then
  build the SMALLEST version your argument uses.

Tools

You have access to the following tools:

* `search_mathlib`: Use this tool to search for Mathlib declarations that may be
  useful in your proof. Submit queries that describe a Mathlib proposition or
  make a guess at the declaration name (e.g., "the product of two continuous
  functions is continuous" or "Continuous.mul"). Do not describe the problem
  you are solving itself; the index does not hold the answer to the problem.

* `python`: Runs a Python 3 snippet and returns what it printed. The standard
  library, numpy, sympy and scipy are available. Use it to check a claim on
  small cases, or confirm a value before you build the argument on it.
  Use this tool sparingly.

* `submit_plan`: Submits your plan and finishes the task.

Output Format

Your output should contain one section as follows:

```
# Plan

A step-by-step outline of the argument that proves the statement, each step
```
```

````
justified. Name the Mathlib results the steps rest on. When a step has no
Mathlib result behind it, say so there, and say what you would build it from.

```

## Recommended Workflow

1. Work out how the statement is proved mathematically, and note the two or
three approaches that could work. Do not commit to one yet.
2. Investigate using `search_mathlib` which approach would be the easiest to
formalize in Lean.
3. Work the chosen argument out in full, with each step justified.
4. Verify that each step is sound, then submit it with `submit_plan`.
````

Listing 1: The planner's prompt.

````
Your goal is to help formalize the solution to the following problem in Lean 4:

```lean4
<statement>
```

Here is the originally constructed plan to solve this problem:

<plan>

Use the above plan to guide your responses for the task shown below.

NOTES:
— The task may have deviated from the plan; in this case the task description
supersedes the plan.
— If something in the plan is false or unclear, you should work out a fix, ignore
the plan and complete the task using your own reasoning. The plan is only a guide,
not a rule.

Your response to the task below should help formalize the solution to the above
problem as easily as possible.
````

Listing 2: The system prompt of every expansion.

````
Your task is to write the proof of `<target>` in the Lean 4 file below.

The proof may use helper declarations above `<target>`, of two kinds:

— Closed helpers: new theorems/lemmas that you prove here. These represent small
pieces of the argument that are used in the main proof body of `<target>`. You should
prove these here.

— Open helpers: new theorems/lemmas that you leave open using the `sorry_using` tactic.
These represent large pieces of the argument that are used in the main proof body of
`<target>`. These are proved in a later session, so they are expensive. The hardest
open helper you introduce should be much easier than `<target>`. If it is not, an AI
grader will reject your submission. You should be able to judge the quality of your
decomposition by how much work you do in the current file.

## File

```lean4
<statement>
```

## What you submit

The file as it stands when `lean_compile` last accepted it. All of these should
hold, else the file is rejected:

— The header, all declarations above `<target>` and `<target>`'s signature are left
exactly as given (including the blueprint attributes). You may insert new
declarations above `<target>`; do not edit or delete the ones already there.

— `<target>` is the last declaration in the file. Its body contains no `sorry`
or `sorry_using`. If you use `sorry` or `sorry_using` in the main proof body, it
will be rejected but return the open goal(s) as feedback.

— All helper declarations you add and `<target>` carry a blueprint attribute:
````
````

````
```lean4
@[blueprint
  (statement := /-- what it says, in words -/)
  (proof := /-- the mathematical argument, worked out in detail -/)]
```

- Each `lemma` and `theorem` helper is either proved in full (closed), or left open
with exactly `:= by sorry_using [...]` as its whole body, where `[...]` lists other
declarations in the file that the helper will use in its own proof (even
transitively). Do not include Mathlib names or tactics. `[]` if none. When in doubt
whether to add a declaration from this file in the list of dependencies, include it,
aiming for recall.

- All other helpers (like `def` or `abbrev`) should have a complete body and also
carry the blueprint attribute.

- These tactics are not allowed anywhere in the file: `native_decide`, `axiom`,
`admit`, or search tactics ending in `?`.

- Names may collide with a global namespace you cannot see. Rename if told to.


## How to proceed:

- Work out a mathematical argument that can be easily formalized in lean. Ensure
each step in your argument is sound and can be proven in Lean. Consider different
formalization strategies. Your goal is to pick one that is the simplest to formalize
in Lean.

- Search before you commit to an approach. Investigate the Mathlib machinery that
you can reuse instead of doing it yourself. If you are in doubt about a
declaration, use the `search_mathlib` tool to look it up.

- Write the complete proof in the main body, introducing open/closed helpers as
necessary. You should prefer closed helpers and reserve open helpers for only the
hard parts of the proof that are still much easier than `<target>`. If `<target>` can
be proved in this session, prove it HERE.

- Edit the file, compile it, and fix what the compiler reports.

- Never state that the task is too complex or too long to finish. Solve the task
in an iterative manner guided by the compiler feedback.

- When to stop. Use `submit_proof` to finish. It submits the file your last
`lean_compile` ACCEPTED. So if your final revision failed, fix the errors and
compile it successfully before finishing.


## Tools

The file above is open in an editing buffer. You always change it in place:

- `view`/`replace`/`replace_all`/`delete_lines` — use these for interfacing
  with the buffer. Every edit takes `compile: true`, which compiles the file
  right after the edit, exactly as `lean_compile` would, and reports the result
  in the same reply. Set it on the edit that finishes a revision.

- `lean_check` — compiles Lean 4 code you pass to it. This is to be used as a probe
for quick checks of Lean code. Do NOT submit your proof to this tool; use
`lean_compile` for that. Use `lean_check` sparingly. This tool does NOT write to
the buffer.

- `python` — runs a Python 3 snippet and returns what it printed. The standard
library only. Use it to check a claim on small cases, or confirm a value before
you formalise it. Use this tool sparingly.

- `search_mathlib` — describe a proposition, or guess a name ("the product of two
continuous functions is continuous", "Continuous.mul"). You should look for known
mathematical facts that are likely to be present in Mathlib.

- `lean_compile` — checks the file as it now stands against every requirement
above. Reports PASSED, or the failures. Proofs with `sorry` are considered failures but
reveal the goal state where `sorry` occurs, which may be useful for the next attempt.
Run this on your own RARELY. Prefer running compilation after edit as described above.

- `submit_proof` — submits the file your last `lean_compile` accepted, and ends
the task.
````

```
IMPORTANT: every response costs a turn, and a turn re—sends the whole
conversation, so put as many tool calls as you can into one response. Calls run in
the order you list them: a later call sees an earlier one's effect (an edit with
`compile: true`, then a `view` of the result, in ONE response) but not its result.
Batch probes together
(`lean_check`, `search_mathlib`, `python`, `view`); make a call on its own only when
its arguments depend on a result you have not seen yet.

Batching tools together is BEST practice and you should practice it as much as possible.
```

Listing 3: The prompt of a decomposition, as the first expansion of a goal.

B TOPICAL IMPURITY

Implementation. Listing 4 computes a proof's impurity. Its input `uses` comes from `#lever_uses`, a Lean command that elaborates the proof file and reports, for each declaration in order, its kind, whether it rests on `sorry`, the constants in its statement (`type`), the library constants in explicit argument positions of its proof (`mathlib_refs`), and the file's own declarations its proof references (`own_refs`). The last declaration is the theorem being proved. `TABLE` maps every Mathlib name to its directory, three levels deep, and whether it is a theorem.

```python
ROOT = "<root>"   # the concept of a generic lemma, which belongs to no subject
LOGIC = {"Eq", "Ne", "HEq", "And", "Or", "Iff", "Not", "Exists", "True", "False",
         "Classical", "congrArg", "congrFun", "cast", "id"}
INFRASTRUCTURE = {"Init", "Batteries", "Mathlib/Data/Nat", "Mathlib/Logic/Basic",
                  "Mathlib/Order/Basic", "Mathlib/Order/Defs",
                  "Mathlib/Algebra/BigOperators", "Mathlib/Algebra/Group",
                  "Mathlib/Algebra/GroupWithZero", "Mathlib/Algebra/NeZero",
                  "Mathlib/Algebra/Order", "Mathlib/Algebra/Ring"}

def concept(name):    # the namespace; a bare definition is its own concept
    head, _, rest = name.partition(".")
    if rest:
        return ROOT if head in ("Left", "Right") else head
    return ROOT if TABLE[name].is_theorem else name

def machinery(name):  # tactics, meta code, logical connectives, instances
    where = TABLE[name].location
    return ((name.startswith("inst") and name[4:5].isupper())
            or where.startswith(("Mathlib/Tactic", "Mathlib/Util", "Mathlib/Lean"))
            or where.split("/")[0] in ("Lean", "Aesop", "Qq", "Plausible")
            or concept(name) in LOGIC)

def subject(uses):    # what the statement is about, on both readings
    stated = list(uses.values())[—1]["type"]
    named = [n for n in stated if n not in uses and n in TABLE and not machinery(n)]
    return ({concept(n) for n in named} | {ROOT},
            {TABLE[n].location for n in named})

def through(uses, helpers, seen):  # library names in the helpers' statements
    out = []
    for h in helpers:
        if h in seen:
            continue
        seen.add(h)
        stated = uses[h]["type"]
        out += [n for n in stated if n not in uses]
        out += through(uses, [n for n in stated if n in uses], seen)
    return out

def references(uses):  # once per proved theorem that reaches for a name
    out = []
    for decl in uses.values():
        if decl["kind"] != "theorem" or decl["sorry"]:
            continue
        out += decl["mathlib_refs"] + through(uses, decl["own_refs"], set())
    return out
```

```
def impurity(uses):
    concepts, locations = subject(uses)
    count = 0
    for n in references(uses):
        if n not in TABLE or not TABLE[n].is_theorem or machinery(n):
            continue
        where = TABLE[n].location
        if concept(n) == ROOT:
            pure = where in INFRASTRUCTURE or where in locations
        else:
            pure = concept(n) in concepts or where in locations
        count += not pure
    return count
```

Listing 4: Topical impurity of a compiled proof, condensed from our implementation (logging and error handling removed).

Example. `putnam_1964_b4` asks how many regions n great circles cut a sphere into. NearAI's proof cites 44 impure theorems, from topology, convexity and group actions (*e.g.*, `isOpen_lt`, `convex_iff_forall_pos`). LEVER's proof at $\lambda = 0.1$ cites 8, mostly from the linear algebra of the circles' planes (*e.g.*`Submodule.finrank_sup_add_finrank_inf_eq`).

Can the metric be gamed? A proof could lower its count by re-proving an out-of-subject library lemma as a local helper: the helper's statement names only definitions, which are not charged, and its proof may cite nothing impure. We checked for this on the 20 problems of §5.2, counting the helpers whose statements name something outside the subject. At $\lambda = 0.1$, LEVER writes more helpers than at $\lambda = 0$ (36.1 against 25.9 per proof), but only slightly more off-subject ones (11.2 against 10.1; NearAI: 14.1). Charging every off-subject helper as one more impure reference lowers LEVER's mean impurity reduction at $\lambda = 0.1$ by under one point.

Limitations. Topical impurity is a proxy for purity of method. It takes Mathlib's organization as a stand-in for mathematical subject, and namespace and directory boundaries do not always match a mathematician's; the infrastructure set is a judgment, measured on one pilot corpus. It counts references rather than weighing them, so an essential detour through another field counts the same as a routine lemma from it. And, as above, it charges a local re-proof of an out-of-subject lemma only through the library theorems that re-proof cites.

C Experimental Details

C.1 Problem Selection

We rank PutnamBench with the published run of NearAI (2026), which solves all 672 problems and records every attempt's tokens and wall-clock time; Each problem is ranked by the token cost of its cheapest solving attempt, most expensive first, priced at a fixed rate table of \$0.15, \$0.003 and \$0.60 per million input, cached input and output tokens (prices from NearAI (2026) originally; these rates are no longer available). Most problems are easy for this agent: below rank 150, the median problem is solved in 0.3 hours (Table 3). The hardest are impractical for controlled comparisons that run every method on every problem: in the top 50, 20 problems needed more than one attempt and one took 5.5 days. We therefore take ranks 51–150, hard for the agent but solved in a median of 1.5 hours, and split them at random into 20 calibration and 80 evaluation problems.

Ranks	Problems	Wall-clock, all attempts (h)
1–50	50	9.6 (max 132)
51–150	100	1.5 (max 33)
151–672	520	0.3 (max 9)

Table 3: **PutnamBench by the cost of the published NearAI run.** Wall-clock: median and maximum over problems of the time summed over all attempts.

C.2 VALUE FUNCTIONS

Table 4 lists the constants of §3.3 for each metric.

Metric	Prediction	Scale	σ	ρ	n	δ
`comp`	LLM, 7 bins	$\log_2$, \$0.016–\$0.512	0.615	1.08	0.33	1.85
`len`	LLM, 7 bins	$\log_2$, 512–16,384 tokens	0.408	0.72	0.32	1.50
`imp`	constant ($= 4$)	linear	6.79	31.2	0.05	1.72

Table 4: **Value functions.** ρ: prediction error on held-out calibration goals (log scale for `comp` and `len`, references for `imp`); σ: variation between draws of the same goal; $n = \sigma^2/\rho^2$: the prediction's weight in draws; δ: the factor a redraw must promise.

Bins and prompt. For `comp`, the seven bins are: under \$0.016, factor-of-two bins from \$0.016 to \$0.512, and over \$0.512; a goal is priced at its bin's geometric center, and the open top bin at \$0.72. For `len`, they are: under 512 tokens, factor-of-two bins up to 16,384, and over 16,384. The value function may also answer that the goal is false, which prices it at ∞. We use the same model as the prover, DeepSeek-V4-Flash, at low reasoning effort, prompted with the goal's Lean statement and 40 worked examples: goals from runs on the calibration problems, each labelled with the bin its realized value fell in. `comp` labels are priced at the 90% cache rate of §4.1, and `len` labels count tokens as the metric does.

Prediction error ρ. We split the calibration problems by problem: the worked examples come from 12 of them for `comp` and 11 for `len`, and ρ is the root-mean-square of $\log(\text{predicted}/\text{realized})$ over 64 goals of the remaining 8 or 7. For `comp`, 34% of predictions fall in the right bin and 75% within one bin, and predictions are low by a factor of 1.53 on average. For `len`, 40% fall in the right bin and 89% within one, with no systematic bias. For `imp`, ρ is the root-mean-square impurity of the 125 solved goals of the 18 solved calibration problems, *i.e.*, the error of predicting 0.

Spread between draws σ. Measuring σ requires drawing the same goal repeatedly: on 162 goals from a held-out set, each redrawn 6 times, gives $\sigma = 0.615 \pm 0.039$ in log dollars and 0.408 in log tokens.

Redraw margin δ and scaling. For `comp` and `len`, a redraw must beat the incumbent by one spread, $\delta = e^{\sigma}$. For `imp`, e^{σ} in references is meaningless ($e^{6.79} \approx 890$), so we read the spread off the same draws proportionally: the median ratio of a goal's standard deviation to its mean is 0.72, giving $\delta = 1.72$. The `len` estimate is 0.2 times the bin's center, so the search expects shorter proofs than typical and keeps looking for them; the constant 4 for `imp` is the 25th percentile of impurity over the calibration goals (§4.4).

C.3 CACHE-HIT RATES AND ACTUAL COST

All costs in the main text are re-priced at one common cache-hit rate of 90% (§4.1). Here we report the rates we observed and the costs they imply.

Method	Per call	Per token
NearAI	93.7%	96.2%
LEVER	86.2%	91.4%

Table 5: **Observed cache-hit rates** on the 80 evaluation problems. Per call: the mean, over calls, of the fraction of each prompt read from cache. Per token: cached input tokens over all input tokens.

Why the rates differ. Calls went through OpenRouter and InferX, which spread calls over several providers. Further, NearAI carries a single long conversation whereas LEVER works on each goal in its own smaller context, reducing cache hit rate.

Costs under other pricings. Table 6 repeats Table 1 under each pricing. At any common rate from 90% to 97%, LEVER solves more problems than NearAI and costs 29–34% less. Priced at its own

	NearAI		LEVER	
Pricing	Solved	Cost ($)	Solved	Cost ($)
90%, common (Table 1)	0.800	1.44	0.963	0.95
93%, common	0.825	1.28	0.963	0.85
95%, common	0.875	1.15	0.963	0.78
97%, common	0.888	1.00	0.963	0.71
Own rate per call	0.850	1.24	0.950	1.09
Actual	0.863	1.07	0.963	0.90

Table 6: **Table 1 under other pricings.** $N = 80$, budget $3. Common: every method at the same rate. Own rate per call: each method at its per-call rate from Table 5. Actual: every call at the cache hits it got. Under each pricing, the $3 rule of Table 1 is applied to each run's calls in order.

rates, LEVER still solves more and costs less: 12% less at its per-call rate, and 16% less at actual cost.

C.4 CASE STUDIES

C.4.1 REJECTING A COSTLY DECOMPOSITION: `putnam_1972_a3`

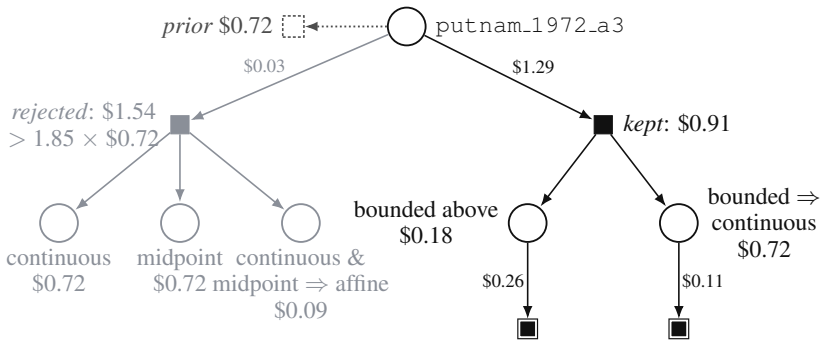


Figure 6: **LEVER on `putnam_1972_a3`.** Edge labels: what each attempt cost. Subgoal labels: the value function's price. The first decomposition (grey) is rejected before any of its subgoals is attempted.

The problem asks for every function on $[0, 1]$ that preserves Cesàro limits (Figure 6). The root is priced at $0.72. The first decomposition leaves three subgoals priced at $1.54 in total, more than $\delta = 1.85$ times the root's prior, so the search draws the root again without attempting any of them. The second leaves two subgoals priced at $0.91; it is kept, both are proved, and the run costs $1.75.

C.5 QUALITY BASELINES

Both baselines run the agent of NearAI (2026) unchanged, with the model, prices and Lean version of §4.1, and append an objective to the end of its system prompt. So that they are asked for exactly what LEVER optimizes, the objective is LEVER's own text, word for word: what LEVER tells a goal it redraws for that metric. Listing 5 shows the impurity objective, where `<concepts>` is the statement's subject, read off by the same probe as the metric (Appendix B). The length objective has the same form: it defines a Lean token as the metric counts it, and asks for the shortest complete proof.

+ Instruct solves the problem from its statement, with the objective in its system prompt, and is capped at $4.5.

+ Refactor starts from NearAI's finished proof of the problem (§5.1), which replaces the statement in the agent's workspace, and is asked to rewrite it for the objective while keeping it a proof of the

same theorem. Its objective adds what LEVER tells a redraw about the attempt in hand: for impurity, how many times the proof leaves the subject and which lemmas do, grouped by field; for length, the proof's token count. The run is capped at $4.5 minus what NearAI's run cost, so that the pipeline as a whole has the same budget as every other method, and its reported cost includes NearAI's.

```
# Optimization Objective

`<problem>` is stated in terms of:
```
<concepts>
```
Those are its *subjects*. A proof stays inside the subject when every library fact
it applies is about one of them, or is general enough to belong to general Mathlib
infrastructure; it leaves the subject when it uses declaration(s) belonging to some
other field.

HINTS:
    — Revise the proof to keep it inside the subject the theorem is stated in. A proof
    leaves it when it reaches for a declaration in some other field;
    — If a helper leaves the subject, you should restate the helper so that it stays within the subject;
    — Change any open helper that you believe cannot be closed completely without leaving the subject;
    — You may have to consider a new formalization strategy for the full proof to stay within the subject.
```

Listing 5: The impurity objective, appended to the baseline agent's system prompt.

D THEORETICAL JUSTIFICATION FOR LEVER

Listing 6 proves the exactness claim of §3.3 in Lean 4, without Mathlib. If V satisfies the backup equations on an acyclic AND/OR graph with nonnegative additive costs, then at every node n, $V(n) = k$ exactly when k is the least cost of a solution subgraph rooted at n, and $V(n) = \infty$ exactly when none exists (`lemma1_exact_values`). A solution's cost sums edge costs over its tree unfolding, so a shared subgoal counts once per use, and a prior with an exact estimate enters as a childless AND node whose edge costs that estimate. Costs are natural numbers in the formalization.

```
/—!
# LEVER, Lemma 1 (Exact values)

"For any nonnegative cost that adds along edges, the backed—up value `V(r)` is
exactly the cost of the cheapest solution subgraph of the graph built so far."

Formalization (core Lean 4, no Mathlib):
* Costs are natural numbers (nonnegative by construction); `∞` is `none : Option Nat`.
* The graph is a finite—branching AND/OR DAG; acyclicity is witnessed by a `rank`
  that strictly decreases along every edge.
* A solution subgraph is defined inductively; its cost adds edge costs along every
  occurrence (the tree unfolding), which is what the backup of §3.1 computes.
* `V` is *any* function satisfying the backup equations of §3.1. We prove it equals
  the optimal solution cost at every node (so the backup is also unique), and that a
  backed—up `∞` means no solution subgraph exists.
—/

namespace Lever

inductive Kind where
  | or
  | and
  deriving DecidableEq

structure Graph (N : Type) where
  kind    : N → Kind
  ch      : N → List N
  cost    : N → N → Nat
  rank    : N → Nat
  acyclic : ∀ n d, d ∈ ch n → rank d < rank n

/—! ### Extended naturals `Option Nat` (`none = ∞`) —/

def addE (a : Nat) : Option Nat → Option Nat
  | none   => none
  | some b => some (a + b)
```

```
/-- Binary sum on `Option Nat`; `∞` absorbs. -/
def add2 : Option Nat → Option Nat → Option Nat
  | some a, some b => some (a + b)
  | _, _           => none

/-- Binary min on `Option Nat`; `∞` is the identity. -/
def min2 : Option Nat → Option Nat → Option Nat
  | none,   r      => r
  | some a, none   => some a
  | some a, some b => some (min a b)

/-- Sum; empty sum is `0`: a childless AND node is solved. -/
def sumE : List (Option Nat) → Option Nat
  | []      => some 0
  | x :: xs => add2 x (sumE xs)

/-- Minimum; empty min is `∞`: a childless OR node is unsolved. -/
def minE : List (Option Nat) → Option Nat
  | []      => none
  | x :: xs => min2 x (minE xs)

def sumN : List Nat → Nat
  | []      => 0
  | x :: xs => x + sumN xs

/-! ### Solution subgraphs and backed-up values -/

/-- `Sol G n k`: a solution subgraph rooted at `n` exists with total cost `k`. -/
inductive Sol {N : Type} (G : Graph N) : N → Nat → Prop where
  | or  {n d : N} {k : Nat} :
      G.kind n = .or → d ∈ G.ch n → Sol G d k → Sol G n (G.cost n d + k)
  | and {n : N} (k : N → Nat) :
      G.kind n = .and → (∀ d ∈ G.ch n, Sol G d (k d)) →
      Sol G n (sumN ((G.ch n).map fun d => G.cost n d + k d))

/-- The backup equations of §3.1. -/
structure IsBackup {N : Type} (G : Graph N) (V : N → Option Nat) : Prop where
  or_eq  : ∀ n, G.kind n = .or →
    V n = minE ((G.ch n).map fun d => addE (G.cost n d) (V d))
  and_eq : ∀ n, G.kind n = .and →
    V n = sumE ((G.ch n).map fun d => addE (G.cost n d) (V d))

/-! ### Auxiliary lemmas -/

theorem minE_mem : ∀ (l : List (Option Nat)) (k : Nat), minE l = some k → some k ∈ l
  | [], k, h => by simp [minE] at h
  | x :: xs, k, h => by
    cases x with
    | none => simp only [minE, min2] at h; exact List.mem_cons_of_mem _ (minE_mem xs k h)
    | some a =>
      cases hr : minE xs with
      | none => simp [minE, min2, hr] at h; simp [h]
      | some b =>
        simp only [minE, min2, hr, Option.some.injEq] at h
        by_cases hab : a ≤ b
        · simp only [Nat.min_def, if_pos hab] at h; simp [h]
        · simp only [Nat.min_def, if_neg hab] at h; subst h
          exact List.mem_cons_of_mem _ (minE_mem xs b hr)

theorem minE_le : ∀ (l : List (Option Nat)) (a : Nat), some a ∈ l →
    ∃ k, minE l = some k ∧ k ≤ a
  | [], a, h => by simp at h
  | x :: xs, a, h => by
    rcases List.mem_cons.mp h with h | h
    · subst h
      cases hr : minE xs with
      | none => exact ⟨a, by simp [minE, min2, hr], Nat.le_refl _⟩
      | some b => exact ⟨min a b, by simp [minE, min2, hr], Nat.min_le_left _ _⟩
    · obtain ⟨k, hk, hka⟩ := minE_le xs a h
      cases x with
      | none => exact ⟨k, by simp [minE, min2, hk], hka⟩
      | some c =>
        exact ⟨min c k, by simp [minE, min2, hk],
          Nat.le_trans (Nat.min_le_right _ _) hka⟩

theorem sumE_some_all : ∀ (l : List (Option Nat)) (k : Nat), sumE l = some k →
    ∀ x ∈ l, ∃ j, x = some j
```

```
  | [], _, _, x, hx => by simp at hx
  | y :: ys, k, h, x, hx => by
    cases y with
    | none => simp [sumE, add2] at h
    | some a =>
      cases hr : sumE ys with
      | none => simp [sumE, add2, hr] at h
      | some b =>
        rcases List.mem_cons.mp hx with hx | hx
        · exact ⟨a, hx⟩
        · exact sumE_some_all ys b hr x hx

theorem sumE_map_of_all {N : Type} (c : N → Nat) (V : N → Option Nat) :
    ∀ l : List N, (∀ d ∈ l, ∃ j, V d = some j) →
    sumE (l.map fun d => addE (c d) (V d)) = some (sumN (l.map fun d => c d + (V d).getD 0))
  | [], _ => rfl
  | d :: ds, h => by
    obtain ⟨j, hj⟩ := h d (List.mem_cons_self ..)
    have ih := sumE_map_of_all c V ds (fun e he => h e (List.mem_cons_of_mem _ he))
    rw [List.map_cons, List.map_cons, sumE, sumN, ih, hj]; rfl

theorem sumN_mono {N : Type} (f g : N → Nat) :
    ∀ l : List N, (∀ d ∈ l, f d ≤ g d) → sumN (l.map f) ≤ sumN (l.map g)
  | [], _ => Nat.le_refl _
  | d :: ds, h => by
    simp only [List.map, sumN]
    exact Nat.add_le_add (h d (List.mem_cons_self ..))
      (sumN_mono f g ds (fun e he => h e (List.mem_cons_of_mem _ he)))

/-! ### Main results -/

section
variable {N : Type} {G : Graph N} {V : N → Option Nat}

/-- Soundness: a finite backed-up value is realized by some solution subgraph. -/
theorem backup_sound (hV : IsBackup G V) :
    ∀ (r : Nat) (n : N), G.rank n < r → ∀ k, V n = some k → Sol G n k
  | 0, _, hr, _, _ => absurd hr (Nat.not_lt_zero _)
  | r + 1, n, hr, k, hk => by
    have ih : ∀ d ∈ G.ch n, ∀ j, V d = some j → Sol G d j := fun d hd =>
      backup_sound hV r d (Nat.lt_of_lt_of_le (G.acyclic n d hd) (Nat.le_of_lt_succ hr))
    cases hkind : G.kind n with
    | or =>
      rw [hV.or_eq n hkind] at hk
      obtain ⟨d, hd, hdk⟩ := List.mem_map.mp (minE_mem _ _ hk)
      cases hVd : V d with
      | none => simp [hVd, addE] at hdk
      | some j =>
        simp only [hVd, addE, Option.some.injEq] at hdk
        subst hdk
        exact Sol.or hkind hd (ih d hd j hVd)
    | and =>
      rw [hV.and_eq n hkind] at hk
      have hall : ∀ d ∈ G.ch n, ∃ j, V d = some j := by
        intro d hd
        obtain ⟨x, hx⟩ := sumE_some_all _ k hk _ (List.mem_map_of_mem hd)
        cases hVd : V d with
        | none => simp [hVd, addE] at hx
        | some j => exact ⟨j, rfl⟩
      rw [sumE_map_of_all (G.cost n) V _ hall, Option.some.injEq] at hk
      subst hk
      refine Sol.and (fun d => (V d).getD 0) hkind (fun d hd => ?_)
      obtain ⟨j, hj⟩ := hall d hd
      exact ih d hd _ (by simp [hj])

/-- Optimality: every solution subgraph costs at least the backed-up value
(in particular the backed-up value is finite whenever a solution exists). -/
theorem backup_le (hV : IsBackup G V) {n : N} {k : Nat} (h : Sol G n k) :
    ∃ v, V n = some v ∧ v ≤ k := by
  induction h with
  | @or n d k hkind hd _ ih =>
    obtain ⟨v, hv, hvk⟩ := ih
    obtain ⟨m, hm, hmk⟩ := minE_le ((G.ch n).map fun d => addE (G.cost n d) (V d))
      (G.cost n d + v) (List.mem_map.mpr ⟨d, hd, by simp [hv, addE]⟩)
    exact ⟨m, (hV.or_eq n hkind).trans hm, Nat.le_trans hmk (Nat.add_le_add_left hvk _)⟩
  | @and n k hkind _ ih =>
    have hall : ∀ d ∈ G.ch n, ∃ j, V d = some j := fun d hd =>
```

```
      let ⟨v, hv, _⟩ := ih d hd; ⟨v, hv⟩
    refine ⟨_, (hV.and_eq n hkind).trans (sumE_map_of_all (G.cost n) V _ hall), ?_⟩
    apply sumN_mono
    intro d hd
    obtain ⟨v, hv, hvk⟩ := ih d hd
    simpa [hv] using Nat.add_le_add_left hvk (G.cost n d)

/-- **Lemma 1 (Exact values).** If `V` satisfies the backup equations on an acyclic
AND/OR graph with nonnegative additive costs, then at every node `n` (in particular the
root `r`): `V n = some k` iff `k` is the minimum cost of a solution subgraph rooted at `n`,
and `V n = ∞` iff no solution subgraph rooted at `n` exists. -/
theorem lemma1_exact_values (hV : IsBackup G V) (n : N) :
    (∀ k, V n = some k ↔ (Sol G n k ∧ ∀ k', Sol G n k' → k ≤ k')) ∧
    (V n = none ↔ ∀ k, ¬ Sol G n k) := by
  have sound := fun k => backup_sound hV (G.rank n + 1) n (Nat.lt_succ_self _) k
  refine ⟨fun k => ⟨fun hk => ⟨sound k hk, fun k' h' => ?_⟩, fun ⟨hs, hmin⟩ => ?_⟩,
          ⟨fun hn k hs => ?_, fun hno => ?_⟩⟩
  · obtain ⟨v, hv, hvk⟩ := backup_le hV h'
    rw [hk, Option.some.injEq] at hv; exact hv ▷ hvk
  · obtain ⟨v, hv, hvk⟩ := backup_le hV hs
    have := hmin v (sound v hv)
    rw [hv, Nat.le_antisymm hvk this]
  · obtain ⟨v, hv, _⟩ := backup_le hV hs; rw [hn] at hv; cases hv
  · cases hVn : V n with
    | none => rfl
    | some v => exact absurd (sound v hVn) (hno v)

end
end Lever
```

Listing 6: Exact values, verified in Lean 4 (`Lemma1.lean`).