AUTOMATED CONJECTURING: GENERATION, SELECTION AND EVALUATION

Ayush Agrawal[1,2,3†], **Vedant Shah**[1,2], **Navin Goyal**[3*], **Ankit Anand**[1,4*], **Aaron Courville**[1,2*]

[1]MILA - Quebec AI Institute, [2]Université de Montréal, [3]Microsoft Research, [4]McGill University

[*]Equal advising. [†]Correspondence to Ayush Agrawal: ayushagrawal1801@gmail.com

ABSTRACT

Deciding which conjectures are worth proving is a crucial part of mathematical discovery, and a system that learns mathematics from its axioms alone must make that decision without human guidance. Self-play provers such as Minimo learn a conjecturer alongside the prover and reward it for producing hard conjectures, but a conjecture that is harder to prove is not necessarily more useful for improving a prover: the learned conjecturer drifts toward long, artificial statements. We study conjecturing in a controlled, single-round setting in the Peano formal environment, across propositional logic, abelian groups, and natural-number multiplication, evaluating a set of conjectures by how much a fixed prover improves on held-out theorems after training on their proofs. We show that a far simpler generator does better: making small structural edits to statements the prover already knows yields conjectures that, at an equal training budget and under best-first search, improve the prover by 6 to 13 percentage points more than Minimo's learned conjectures. The gain holds even when conjectures are drawn at random from the generated pool. In contrast to the common practice of selecting the most difficult statements, we also propose Hard Neighbours (HN), a score that favors conjectures whose proof difficulty grows disproportionately relative to their structural distance from easier reference statements; HN gives the largest selection gains in propositional logic, but no selection method is best in every domain. In the current setting, how conjectures are generated matters more than how they are selected, and structural perturbation is the baseline that future conjecturers should be measured against. Apart from improvements in proving ability, we also study other attributes of generated proofs: the amount of search required and the length of the proofs.

1 INTRODUCTION

AI systems have advanced rapidly at the frontiers of mathematics, with recent developments including OpenAI's announcement of a proposed solution to the Navier-Stokes Millennium Prize problem (OpenAI, 2026a). However, mathematical discovery is not only about proving open problems; it is also about posing new ones. Therefore, autonomous mathematical discovery requires systems that can generate conjectures worth working on.

Conjecturing is a crucial part of mathematical discovery: Fermat's last theorem, the Riemann hypothesis, and Goldbach's conjecture each began as a judgment that a statement was worth pursuing, and shaped mathematics long before any proof appeared. A system that learns mathematics from its axioms alone must make such judgments itself, proposing statements and deciding which are worth the effort of proving; this is what separates a prover from a discoverer. Formal mathematics is the right test bed for this capability, because every statement and proof can be checked mechanically, so progress is measurable without human judgment in the loop.

Recent self-play systems learn to conjecture and prove together, using the prover's performance to guide subsequent conjecture generation (Poesia et al., 2024; Dong & Ma, 2025). Their objective is to generate statements that are hard but still provable for the current prover. As the prover improves, the conjecturer is encouraged to propose increasingly harder problems. However, this objective leads

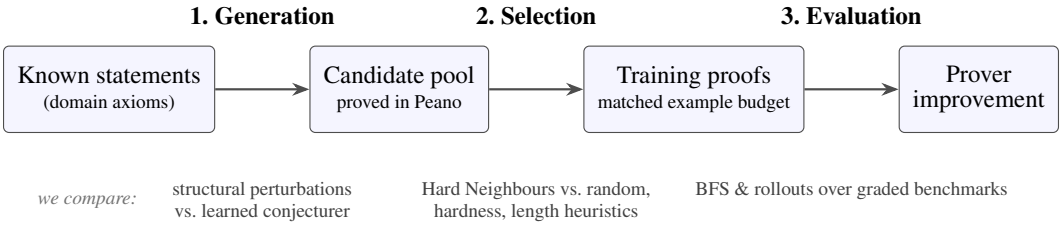


Figure 1: The three design decisions in an automated conjecturing pipeline.

to undesirable outcomes. Minimo (Poesia et al., 2024), one of such setups, mentions that conjecturer collapses into making longer and more complicated statements with almost no mathematical value. Similarly, STP (Dong & Ma, 2025) reports mode collapse toward algebraic manipulations and introduces both a diversity mechanism and a filter against artificially complicated goals. These observations expose a limitation of difficulty as a selection signal: making a statement harder for a prover does not necessarily make it more useful or interesting.

Consider a scenario where a high-school student has learned that $x^2 - 1 = (x - 1)(x + 1)$. What should they practice next in order to improve their understanding of the area? One could ask them to factor $4x^2 - 4$, which repeats the same identity with a constant factor, or $x^3 - 1$, which introduces a different factorization. Once they have practiced, how would one know whether the student has made progress? This presents three separate decisions. First, one must produce candidate exercises. Second, one must choose which exercises to assign. Third, one must assess whether practicing them helped the student solve other problems. We work in a theorem proving setup where our prover faces the same questions: where should its conjectures come from, out of them, which should it learn from, and how to measure its progress?

This motivates examining the three design choices for automated conjecturing: how candidate statements are *generated*, which of them are *selected* for the prover to learn from, and how we *evaluate* the progress (Figure 1). When generation and selection change together within a training loop, improvements in prover performance do not reveal which component was responsible. Conversely, increasing conjecture difficulty does not establish that the resulting training data is better for improving the prover. We therefore compare candidate sources and selection methods within a fixed pool, with base prover's improvement as our primary method of evaluation.

We work in a small, controlled setting: the Peano formal theorem proving environment (Poesia & Goodman, 2023), which enumerates legal actions at each proof state. Within this environment, we explore three distinct domains: propositional logic, natural-number multiplication, and abelian groups. Each training comparison starts from the same randomly initialized 25M parameter transformer model (Vaswani et al., 2017) (acting as a prover) and uses a single round of selection and finetuning. Our primary criterion is prover's improvement on difficulty-graded benchmarks after training on the selected proof traces, with matched training data samples. We evaluate using policy-guided best-first search (BFS) and policy rollouts, neither requiring a separately trained value function. This scale allows us to vary the design choices, and inspect generated statements and proofs.

Our candidate generator and selection score share a motivation: small changes to a known statement can produce a substantially more difficult candidates that could be of interest. For example, replacing one occurrence of A by B in the intuitionistically provable $((A \to A) \to A) \to A$ yields Peirce's law, $((A \to B) \to A) \to A$, which can be proven only in classical logic. The statement "every even integer greater than two is the sum of two positive integers" is trivial. Replace "positive integers" with "prime numbers", and we obtain Goldbach's conjecture: a small change turns a trivial observation into a much harder and genuinely interesting mathematical statement. Since the candidates are closer to the known statements, they are more likely to be hard for interesting reasons. These examples motivate both our generator and our selection score. We generate candidates by *structural perturbation*, editing the abstract syntax tree of a known statement, so that every candidate is a small edit away from something already proved (see Figure 2). We then select the candidates for which that small edit produced a large jump in proof-search effort. The *Hard Neighbours* (HN) score measures this jump relative to the edit distance from easier reference statements. As baselines we use random selection, absolute hardness, hardness per token, and shortest statements.

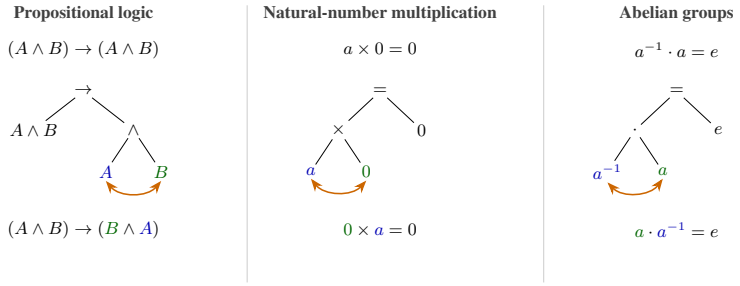


Figure 2: One perturbation across three domains: swapping two arguments in a statement's abstract syntax tree yields a new candidate statement. Colors track the swapped subterms. These are simple syntactic edits that need not preserve truth or increase difficulty; Appendix A lists all perturbations we did per domain.

Across the three domains, our comparisons yield three findings. First, the generation method has a large effect on the prover's improvement on the set benchmarks. Finetuning on proof traces from HN-selected perturbed statements outperforms finetuning on Minimo-generated statements in all three domains under best-first search, by 7 to 13 percentage points; random selection from the same pool is a competitive baseline, already improving over Minimo by 6 to 9 points (see Table 13). The advantage extends to benchmark levels where Minimo-trained provers solve no problems. This finding also holds under rollout evaluation in logic and groups, while Minimo achieves a slightly higher peak solve rate in natural-number multiplication. Second, selection yields additional training gains in logic and groups, but no criterion consistently improves both peak solve rate and coverage over random selection across all three domains. Selecting the hardest provable conjectures is not the strongest criterion in any domain under BFS. In a separate experiment, we inject benchmark theorems into the candidate pool and measure whether they are selected by the scorers. HN exceeds the random selection in all three domains, but selecting shortest statements is competitive in logic and arithmetic and selects the most benchmark theorems in groups. Third, evaluation choices affect the conclusions. A checkpoint's solve rate can decline while coverage across checkpoints continues to grow. Search also affects the returned proof quality: on matched solved subsets, rollout proofs are around two times as long as BFS proofs on average, and a concrete example contains repeated steps that leave the proof obligation unchanged (Table 14 in Appendix I.3).

Our contributions are: (1) A single-round study in Peano environment that examines conjecture generation, selection, and evaluation using approximately matched training samples. (Sections 4–6). (2) A structural-perturbation generator requiring no learned generation model, and the finding that its proved candidates provide better training signal than the Minimo pool under BFS in all three domains (Section 7). (3) The Hard Neighbours (HN) selection score, which combines structural similarity with relative proof-search difficulty to select prover-training data. HN gives the largest selection gains in propositional logic where it exceeds Random by around 5 percentage points of peak solve rate and 7.5 percentage points of coverage, though no selection method improves consistently across all three domains (Section 7). (4) Difficulty-graded benchmarks for three formal domains and an analysis of how search procedure and evaluation measure affect the conclusions (Sections 6–7).

2 Related Work

Computer programs that propose mathematical statements date to AM, Graffiti, and HR (Lenat, 1977; Fajtlowicz, 1988; Colton, 2002), followed by theory exploration (Johansson et al., 2011) and recently, by LLM based systems that evolve mathematical constructions using search (Novikov et al., 2025), mine literature for finding problems to solve (Zheng et al., 2026). Closest to our work are systems that generate statements to train provers: learned theorem generators (Wang & Deng, 2020), synthetic data at scale (Trinh et al., 2024) and self play systems that couple the conjecturer and the prover such as Minimo (Poesia et al., 2024), STP (Dong & Ma, 2025) or as as self-generated curriculum (Bailey et al., 2026). These systems typically reward conjectures for being hard but provable, and this difficulty based reward is hackable: conjectures drift towards longer and artificially complex statements. Moreover, generation, selection and evaluation are entangled where gains are hard to attribute to any single component. We study each stage of generation, selection and evaluation

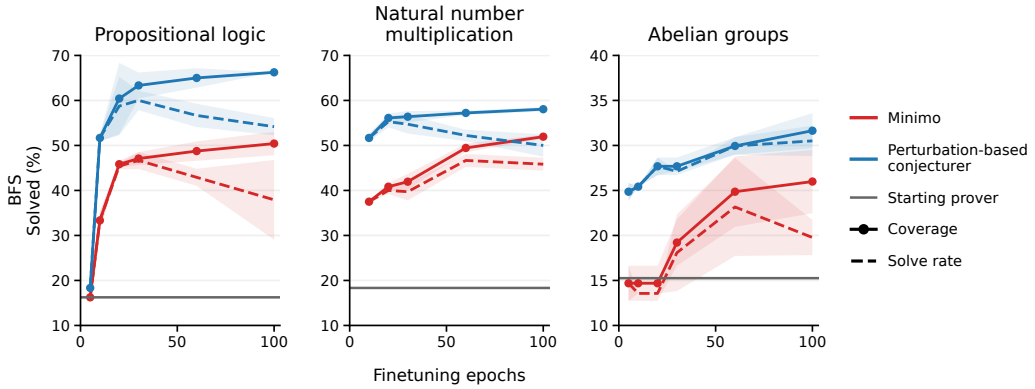


Figure 3: Training on samples from perturbation-based conjecturer yields higher BFS solve rates than training on Minimo-generated conjectures at every evaluated checkpoint in all three domains. Minimo's solve rate declines after epoch 30 in logic and epoch 60 in groups, while its coverage continues to rise: later checkpoints solve new problems while losing some previously solved ones. Blue curves show the perturbation-based conjecturer with HN selection. Dashed curves show solve rate; solid curves with circles show coverage. Shading denotes ± 1 sample SD across three seeds; solid gray lines show the starting prover's performance.

in the small scale setup, Peano (Poesia & Goodman, 2023) and ask whether a learned conjecturer beats a simple perturbation based generation baseline at matched training budget. Similarly, to avoid selecting complex statements, we use a selection score that selects candidates that are structurally close to the starting known statements but relatively harder to prove and compare with selection based on absolute hardness, and examine how the conclusions depend on the evaluation procedures as well.

3 Background: The Peano Environment

We work in Peano (Poesia & Goodman, 2023), a formal theorem proving environment. We consider three domains: propositional logic, natural-number multiplication, and abelian groups. Each domain is specified by a set of axioms and inference rules, listed in Appendix J. These domains expose different reasoning requirements, including logical rule composition and case analysis, equality rewriting, and induction. A proof state contains a goal and its local context. Peano enumerates the legal actions available in that state and computes the states resulting from their execution. In order to arrive at the proof, a learned policy has to select the right actions given the state until all proof states are closed.

Example. Consider proving that conjunction is commutative. The statement is written Peano as

```
[('A : prop) -> ('B : prop) -> (and 'A 'B) -> (and 'B 'A)]
```

A proof is a sequence of actions. After three introduction actions bind A, B, and the assumption $h : A \wedge B$, Peano enumerates the legal next steps: five in this state, such as extracting either conjunct of h or splitting the goal, and the policy chooses among them. Extracting A and B from h and then applying conjunction introduction closes the goal, for a total of nine actions. Each proof state paired with its chosen action is one policy training example, and the number of partial proofs a search expands before finding a proof is our measure of the statement's cost for the prover (Section 3). Appendix J.5 gives the full proof and its nine recorded actions.

Prover The prover is a randomly initialised 25M-parameter transformer following Poesia et al. (2024). Peano enumerates the legal actions at each proof state, and the model scores each state-action pair. During training, actions taken in recorded proofs receive label Y, while the other legal actions receive label N. At search time, we use the model's predicted probability of Y, normalized over its Y and N predictions, as the action's score, and normalize these scores across the legal actions to obtain the distribution used by proof search (Appendix C).

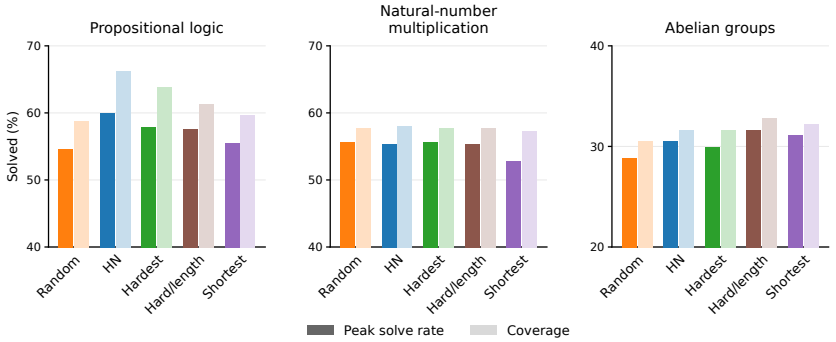


Figure 4: Selection gains vary by domain under best-first search at a 4,000-removal budget. Selection helps most in logic, where HN improves on Random by about 5 percentage points in peak solve rate and 8 points in coverage. In groups, Hard/length leads, exceeding Random's peak solve rate by about 3 points; in natural-number multiplication, no method exceeds Random's peak. Each pair shows peak solve rate (dark, left) and coverage (light, right) (Section 6.1)

Best-first search. Policy-guided best-first search (BFS) prioritizes partial proofs by their cumulative log-probability, following GPT-f (Polu & Sutskever, 2020). It maintains a priority queue of partial proofs; at each iteration it removes the highest-priority partial proof, explores its legal next actions, and adds the resulting unfinished proofs to the queue. We measure *search effort* by the number of partial proofs removed from the queue for expansion, or *queue removals*; the search budget limits this count. Search effort differs from *proof length*, which counts the actions in the returned proof. We use BFS without a learned value function so that differences in proving performance can be attributed to the training data rather than to the value model or search hyperparameters, as in MCTS-style provers (Lample et al., 2022; Poesia et al., 2024). Algorithm 1 in Appendix C gives the full procedure.

Policy rollouts. As a second search procedure, policy rollout search (Algorithm 2 in Appendix D) follows a single proof trajectory per attempt, choosing one legal action at a time for the leftmost open subgoal. Unlike BFS, it maintains no frontier of alternative paths and does not backtrack within an attempt. A greedy attempt always chooses the highest-scoring action; if it fails, subsequent attempts restart from the original statement and sample actions from the policy distribution.

Minimo as a learned conjecturer. Our learned-generation baseline is the Minimo conjecturer (Poesia et al., 2024), which learns to conjecture and prove through an iterative self-play loop in Peano, starting from randomly initialized parameters: the conjecturer generates well-formed statements through constrained decoding and type-directed synthesis, and learns to target hard but provable conjectures from proof-search feedback.

4 Conjecture Generation

Generation by Structural Perturbation. We generate candidates by editing the abstract syntax trees (ASTs) of seed statements, without training a conjecture generator. Appendix B lists the initial seeds for each domain. Each edit changes a selected subexpression or its variables. Figure 2 illustrates the idea across the three domains. In propositional logic, for example, exchanging the conjuncts in the conclusion of $(A \wedge B) \rightarrow (A \wedge B)$ produces $(A \wedge B) \rightarrow (B \wedge A)$. The operations include replacing or exchanging variables, introducing a fresh variable, changing an operator, adding or removing a unary operation, and inserting, removing, or replacing subexpressions. We instantiate these operations with the vocabulary of each domain: logical connectives and negation in propositional logic; addition, multiplication, and successor in arithmetic; and multiplication and inverse in groups. For the equational domains, edits can also add or remove equality hypotheses. These are candidate-generating operations, not inference rules: they need not preserve truth. For example, replacing conjunction by disjunction in $(A \wedge B) \rightarrow A$ produces $(A \vee B) \rightarrow A$, which is false. Appendix A specifies the operations for each domain. For each domain, we start with easiest set of problems (axioms) and then apply these edits to form more statements. The new statements become seeds and then this process is repeated for five times.

Generation by a learned conjecturer. Our learned-generation baseline uses the Minimo conjecturer introduced in Section 3. We collect candidates from Minimo across its five training iterations.

Filtering Candidates from either source are deduplicated, checked for well-formedness in Peano, and we simply discard candidates refuted by an oracle (Z3 (De Moura & Bjørner, 2008) for logic and multiplication, finite-model counterexample search for groups). The surviving candidates are then submitted to proof search, and only those statements that can be proved by base model are passed to next stage of conjecture selection. (Section 5). Appendix B gives the pool construction details and sizes.

5 Conjecture Selection

Given a large list of candidate conjectures which can be proved by a base model, it is unclear if all candidates are equally interesting and informative to selected as "conjectures" and can aid further in development of stronger provers. Recent work often favors statements that are hard but still provable for the current prover (Poesia et al., 2024; Dong & Ma, 2025). However, selecting conjectures by hardness alone is weak as it can favor long, mechanical statements, which may have little learning utility. To address this, we develop a novel notion of hardness, called as *Hard Neighbors* which brings out the concept of hardness relative to some reference statements. The idea of Hard Neighbours (HN), is to reward those difficult statements which are structurally and syntactically similar to easier reference statements(Figure 5) but are themselves very hard, i.e they require disproportionate search effort compared to reference statements. We further compare this selection strategy with the notions popular in literature like selecting shorter statements only, selecting hard statements only as well as selecting statements randomly out of generated pool.

Hard Neighbours (HN). Given a large set of statements from the perturbation pipeline, (from section 4, the goal of selection strategy is to determine the most useful conjectures for prover's improvement, out of this vast pool. For each candidate c, we perform the best first search and collect its proof (we have the proof for all candidates because we filter the candidates which can be proven by the base prover in the previous step). As a proxy for search effort, we define hardness (denoted as $b(c)$) as the number of queue removals used to prove the conjecture c. We further define log-hardness as $h(c) = \log(1 + b(c))$.

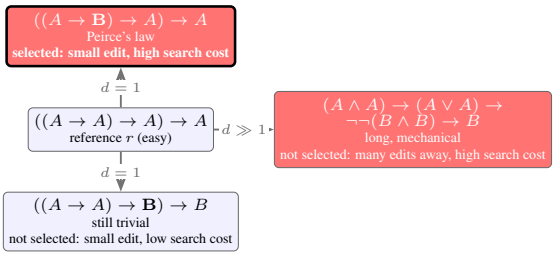


Figure 5: Hard Neighbours: small structural changes can produce large jumps in proof-search difficulty. Red marks costly statements; the thick border marks the HN selection.

We also define a distance metric which captures syntactic difference between any two statments. Let $d(a, b)$ be the Levenshtein distance between the token sequences of statements a and b, using unit-cost insertions, deletions, and substitutions. Lastly, a reference set is used to compute the utility or value of each candidate c. We initialize this reference set with domain's available proved axiom level statements at round t=0 and each subsequent round builds a new reference set as described below. At selection round t, let R_t be the reference set and $R_t^<(c) = \{r \in R_t : b(r) < b(c)\}$ be the references which are strictly easier than c as determined by the hardness metric $b(c)$. For candidate c with a non-empty reference set, we define

$$\mathrm{HN}(c; R_t) = \frac{1}{|R_t^<(c)|} \sum_{r \in R_t^<(c)} \frac{h(c) - h(r)}{\max\{d(c, r), 1\}}. \tag{1}$$

The score averages the log-hardness gap per unit token edit distance over all easier references, rewarding candidates which are substantially harder than the reference statement r but are closer in token Levenshtein distance space. Candidates with no easier reference are skipped for that round. In each round, we promote the m highest-scoring eligible candidates, and use them as the reference set for the next round. The union of promoted candidates over all rounds forms the HN candidate set.

Promotion stops when no candidate qualifies, the pool is exhausted, or the round limit is reached. The prover, candidate pool, and measured search costs remain fixed throughout this procedure. The starting axiom seeds are not included in the final pool and at every round, already promoted candidates are excluded from further selection. Appendix E gives the pseudocode, domain-specific settings, and implementation details.

Other selection measures. We compare HN with four alternatives applied to the same proved pool. *Random* selects candidates uniformly without replacement. *Hardest* ranks them by decreasing $b(c)$. *Hard/length* ranks them by decreasing $b(c)/\max\{\ell(c), 1\}$, where $\ell(c)$ is the statement's token count. *Shortest* ranks them by increasing $\ell(c)$. Statement length and HN distance use the same tokenization, described in Appendix E.

6 Conjecture Evaluation

We evaluate conjectures in two ways: their utility as training data for a prover, and recovery of known mathematical statements from a larger candidate pool. The first is our primary evaluation, while the second tests selection without finetuning a prover. The two evaluations are complementary. Prover improvement measures whether selected proofs help a specific prover on a specific benchmark; benchmark recovery measures whether a selector favors statements considered worth stating in the benchmark, independently of any training. Neither alone captures mathematical interest, so we report both.

6.1 Prover Improvement

Experimental design. Within each domain, all finetuning comparisons start from the same prover checkpoint and use the same domain axioms. We study a single round of data selection and prover finetuning, rather than repeatedly updating the conjecturer and prover in a bootstrapping loop. To compare candidate sources, we do supervised finetuning (SFT) on learned Minimo conjectures or a random selection from the structural-perturbation pool. To compare selection methods, we apply the procedures in Section 5 to the same proved perturbation pool.

Proof data and finetuning budgets. Each successful proof provides state-action training examples. At each visited state, the action taken in the proof is labeled Y, while the other legal actions are labeled N. Appendix E describes the example format and how we assemble the training datasets. Only the policy is trained; no separate value-function is trained. Within each domain, we match the target number of distinct labeled state-action examples across methods, instead of the number of selected conjectures. Longer proofs and states with more legal actions can contribute more examples, so equal numbers of conjectures would not necessarily provide equal amounts of training data. We set the budgets from the available Minimo training data: 1,800 examples for propositional logic, 3,020 for arithmetic, and 13,204 for abelian groups. We select complete proofs, stopping before the next proof would exceed the budget. We use three seeds for data selection. Appendix E gives the finetuning settings, Appendix E describes the example format and dataset construction, and Appendix E.1 reports the actual conjecture and example counts for each method and seed.

Benchmarks and evaluation search. For evaluation, we designed benchmarks consisting of level wise problems going from easier to harder problems. We construct these benchmarks using frontier models: GPT-5.6 (OpenAI, 2026b) and Claude Opus 4.8 (Anthropic, 2026) iteratively, and verify the correctness of the statements using solvers such as Z3 (more details in Appendix B). We evaluate on 80 propositional-logic problems across six levels, 120 arithmetic problems across five levels, and 59 abelian-group problems across five levels. Appendix F describes the levels and gives their problem counts in Table 12. The levels organize statements by their reasoning requirements. We report both overall and level-wise performance. We evaluate with both search procedures from Section 3. For best-first search, each problem has a budget of 4,000 queue removals and a 600-second time limit. For rollouts, each problem receives a greedy attempt followed by up to 16 sampled attempts at temperature 1, each limited to 60 recorded actions, under a 600-second time budget. A problem counts as solved if any attempt succeeds.

Performance measures. Our basic metric is the fraction of benchmark problems solved at a given finetuning checkpoint, averaged over the three seeds. We examine these checkpoint-wise curves as we train the model alongside two measures. *Peak solve rate* is the maximum over evaluated checkpoints, of the three-seed mean solve rate. *Coverage* is the fraction solved by at least one evaluated checkpoint within a seed, subsequently averaged over seeds. Coverage measures the breadth of solutions obtained over training.

6.2 Benchmark-Theorem Recovery

The training experiment measures whether selected proofs help a particular prover improve. Its outcome depends on what the prover already knows and how effectively it learns from those proofs; it does not directly measure the mathematical value of the selected statements. We therefore consider a second question: does a selection method prioritize known benchmark theorems when they appear among other proved conjectures? To answer the same, we construct an augmented pool containing proved perturbation candidates, benchmark theorems, and learned Minimo conjectures. All statements must be proved by the starting prover before they can be selected. We then apply the selection procedures from Section 5. Table 10 in Appendix E.2 gives the original proved source-pool sizes. For each method, we report the number and fraction of available benchmark theorems that it selects. We also report the number of selected Minimo conjectures. Thus, the experiment asks whether a selection method favors known benchmark theorems within the subset of statements already accessible to the starting prover.

7 Results and Analysis

Figure 3 compares prover learning curves for HN-selected perturbations and Minimo-generated conjectures. Table 13 in Appendix G reports peak solve rates and coverage for all methods under both search procedures.

Structural perturbations outperform learned conjecturing. Across all three domains, finetuning on HN-selected proved perturbations yields higher BFS solve rates than finetuning on Minimo-generated conjectures at matched training-example budgets (Figure 3). Peak solve rate improves by 13, 9, and 7 percentage points in propositional logic, multiplication, and abelian groups, respectively. The corresponding coverage gains are 16, 6, and 6 points (Table 13). The learning curves also show why checkpoint solve rate and coverage should be distinguished. For HN in multiplication, solve rate falls from about 55% at epoch 20 to 50% at epoch 100, while coverage increases from about 56% to 58%. Later checkpoints therefore solve some previously unsolved problems even as their overall solve rate declines. Minimo exhibits the same pattern in logic and groups (Figure 3). Under rollout evaluation, which combines a greedy attempt with up to 16 sampled attempts, HN-selected perturbations also outperform Minimo in both peak solve rate and coverage in logic and groups. Multiplication is the exception: Minimo leads HN by about 2 percentage points in peak solve rate (Figure 9; Table 13). Minimo rollout improvements in multiplication mainly come from sampled attempts. Minimo's greedy-only peak is lower than HN's. (Figure 25 in Appendix I).

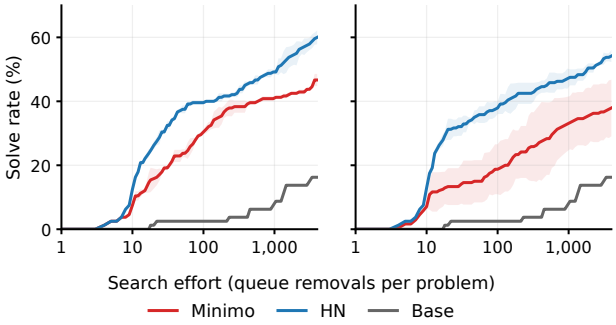


Figure 6: BFS solve rate versus search effort in propositional logic at epochs 30 (left) and 100 (right). Minimo and HN show three-seed means with ± 1 sample SD shading; Base (solid gray) is the starting prover baseline.

HN matches Minimo's solve rate with far less search. In propositional logic at epoch 100, HN reaches about 38% solve rate within 100 queue removals, whereas Minimo requires full 4000 removals to reach the same rate. At the full 4,000-removal budget, HN solves about 54% of problems, compared with Minimo's 38% (Figure 6). We see similar findings when effort is measured in policy action evaluations rather than queue removals. (Appendix H).

Selection is domain dependent. To separate the benefit of perturbation generation from that of selection, we first compare Random with Minimo. Under best-first search, Random improves peak solve rate over Minimo by about 8, 9, and 6 percentage points in propositional logic, multiplication, and abelian groups, respectively. The corresponding coverage gains are about 8, 6, and 4.5 points (Table 13). Comparing selectors on the same proved perturbation pool, best-first search reveals the largest selection gains in propositional logic (Figure 4). HN achieves the highest peak solve rate and coverage, exceeding Random by about 5 and 7.5 percentage points, respectively. Hardest also improves both metrics, but by smaller margins of about 3 and 5 points. In multiplication, no selector exceeds Random's peak solve rate, and the largest coverage gain, achieved by HN, is only about 0.3 points. In abelian groups, Hard/length achieves the highest peak solve rate and coverage, outperforming Random by about 3 and 2 points, respectively. Selection therefore improves performance in logic and groups, but the strongest selector differs between them. Selecting solely by absolute hardness is not the strongest approach in either domain.

HN leads benchmark recovery in logic and arithmetic. HN recovers 46% of the available benchmark theorems in propositional logic and about 30% in arithmetic (Table 11), compared with about 38% and 26% for Shortest. Selection by absolute hardness does not recover any benchmark theorems in logic and only 7.4% in arithmetic. In abelian groups, however, Shortest recovers about 67%, twice the 33% recovered by HN, Hardest, and Hard/length. HN and Shortest differ substantially in how frequently they select Minimo conjectures. For example, in logic, Shortest selects around 46% of the available Minimo conjectures, compared with only 1% for HN. Recovery is a sanity check, not a target: the pool contains thousands of uncurated proved conjectures, some of which may be more useful or more interesting than the benchmark theorems, so a selector that ranks a benchmark theorem below one of them is not necessarily wrong. We therefore read recovery rates as evidence of what each selector prefers rather than as a score to maximize.

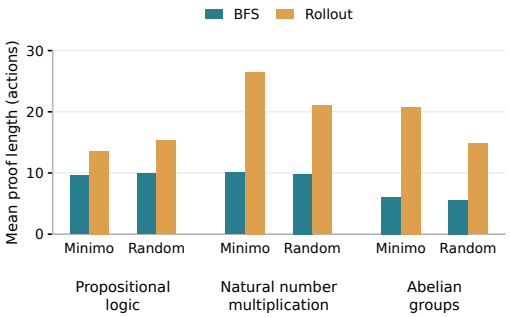


Figure 7: Rollout proofs are about $1.5\times$ as long as BFS proofs in logic and 2–$3\times$ as long in arithmetic and groups. Bars show mean lengths on matched solved problems (Appendix I.3).

Rollouts produce longer proofs on average. We compare BFS and rollout proof lengths on common solved problems at each seed and checkpoint (Figure 7). Mean rollout proof lengths are 1.3–1.6 times the BFS lengths in propositional logic, 2–2.6 times in arithmetic, and 2–3.4 times in abelian groups (Figure 27). This additional length is mostly redundant. For example, a 29-action rollout proof of contraposition contains nine repeated ex-falso applications, each taking two recorded actions without changing the proof obligation. Removing these repetitions yields exactly the corresponding 11-action BFS proof (Table 14 in Appendix I.3).

8 Conclusions and Limitations

Separating generation, selection, and evaluation, we found that in our Peano experiments the generator matters most: finetuning on structurally perturbed conjectures outperforms finetuning on the learned Minimo conjecturer under best-first search in all three domains, by 6 to 13 points in peak solve rate, even with random selection from the perturbation pool. In propositional logic, the perturbation-trained prover also matches Minimo's solve rate with about 40 times less search. Selection adds smaller gains: Hard Neighbours helps most in propositional logic, but no selection method leads in every domain, and selecting the hardest conjectures is never the best choice on its own. Evaluation choices affect these conclusions: Minimo performs better in natural-number multiplication under rollout evaluation, a checkpoint's solve rate can decline while coverage grows, and rollouts return longer proofs on matched solved problems. Together these findings make structural perturbation with random selection the baseline that new conjecturers should be measured against. Our study is limited to three elementary domains, a small policy, and one round of selection and finetuning from a fixed starting checkpoint. It is therefore not clear whether the findings would extend

to larger pretrained provers or subsequent conjecture–prove–train iterations. We measure usefulness through improvement on a finite benchmarks. HN similarly measures difficulty relative to a particular prover and reference set. Moreover, selection considers only candidates proved within the search budget. Assessing mathematical interest of unproved conjectures remains a broader challenge. Future work should develop better selection measures beyond proof difficulty, assess conjectures for which no proof has yet been found, and investigate whether selected conjectures can improve a learned generator over successive rounds. Extending these comparisons to larger provers involving LLMs and richer mathematics in Lean would be crucial next steps.

ACKNOWLEDGMENTS

Ayush Agrawal, Navin Goyal, and Aaron Courville acknowledge Renaissance Philanthropy's AI for Math Fund for supporting conjecturing research in our group. Ayush Agrawal also acknowledges support from the Fonds de recherche du Québec: this work was supported by a grant (2010362, https://doi.org/10.69777/2010362) from the Fonds de recherche du Québec. Ayush Agrawal and Navin Goyal thank Siddhartha Gadgil and Timothy Gowers for helpful discussions and feedback.

AUTHOR CONTRIBUTIONS

Ayush Agrawal led the project, framed the three design questions of generation, selection, and evaluation, conceived the main ideas of structural perturbations and Hard Neighbours, designed and ran all the experiments, and wrote and edited the paper. Vedant Shah took part in project discussions, performed sanity checks on the propositional-logic benchmark, and helped with paper formatting. Navin Goyal, Ankit Anand, and Aaron Courville served in advisory roles. Navin Goyal suggested the line of research, helped design the research questions for the Minimo and Peano setups and the difficulty-graded benchmarks, and gave feedback on the writing. Ankit Anand contributed to the design of the Hard Neighbours score and its ablations, directed the overall scope of the work, and helped write the paper. Aaron Courville gave feedback on experiment design decisions and helped write the paper's motivation.

REFERENCES

Anthropic. Claude opus 4.8 system card. Technical report, Anthropic, 2026. URL https://www-cdn.anthropic.com/0b4915911bb0d19eca5b5ee635c80fef830a37ea.pdf.

Luke Bailey, Kaiyue Wen, Kefan Dong, Tatsunori Hashimoto, and Tengyu Ma. Scaling self-play with self-guidance, 2026. URL https://arxiv.org/abs/2604.20209.

Simon Colton. Automated theory formation in pure mathematics. In *Distinguished dissertations*, 2002. URL https://api.semanticscholar.org/CorpusID:118130982.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Kefan Dong and Tengyu Ma. Stp: Self-play llm theorem provers with iterative conjecturing and proving, 2025. URL https://arxiv.org/abs/2502.00212.

Siemion Fajtlowicz. On conjectures of graffiti. *Discrete Math.*, 72(1-3):113–118, December 1988. ISSN 0012-365X. doi: 10.1016/0012-365X(88)90199-9. URL https://doi.org/10.1016/0012-365X(88)90199-9.

Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of automated reasoning*, 47(3):251–289, 2011.

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.

Douglas B. Lenat. Automated theory formation in mathematics. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'77, pp. 833–842, San Francisco, CA, USA, 1977. Morgan Kaufmann Publishers Inc.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. Finite time blowup for navier–stokes, 2026a. URL https://cdn.openai.com/pdf/32d9f210-8b73-45e0-91bc-82a30aef8a9a/navier-stokes.pdf.

OpenAI. Gpt-5.6 system card, 2026b. URL https://deploymentsafety.openai.com/gpt-5-6.

Gabriel Poesia and Noah D Goodman. Peano: learning formal mathematical reasoning. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 381(2251): 20220044, 2023.

Gabriel Poesia, David Broman, Nick Haber, and Noah D Goodman. Learning formal mathematics from intrinsic motivation. *Advances in Neural Information Processing Systems*, 37:43032–43057, 2024.

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

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Mingzhe Wang and Jia Deng. Learning to prove theorems by learning to generate theorems. *Advances in neural information processing systems*, 33:18146–18157, 2020.

Zeyu Zheng, Shengtong Zhang, Jeremy Avigad, Prasad Tetali, and Sean Welleck. The problem is the problem: Towards scalable mathematical discovery, 2026. URL https://arxiv.org/abs/2608.16977.

A PERTURBATION OPERATIONS

Each perturbation edits a statement's AST, rather than its proof. Applicable edits are enumerated at eligible nodes, sampled subject to the per-operator cap, and checked for statement well-formedness in Peano. None of these operations is assumed to preserve truth. In the descriptions below, a *local* edit affects the selected subtree, whereas a *global* edit affects the full statement body.

A.1 VARIABLE OPERATIONS

The following four operations are shared across the domains, using variables of the appropriate domain type.

- **Replace a variable** (`replace_var`). Replace one occurrence of a declared variable by another declared variable.
- **Exchange variables** (`swap_vars`). Exchange two declared variables that both occur in the selected subtree, leaving the rest of the statement unchanged.
- **Identify variables** (`rename_all`). Globally replace one declared variable by another and remove the redundant declaration. This identifies two variables; it is not merely an alpha-renaming.
- **Introduce a variable** (`add_var`). Replace a selected proposition in logic, or a selected domain term in arithmetic or groups, by a fresh variable and add its declaration.

A.2 PROPOSITIONAL LOGIC

The four variable operations and the eight operations below form the 12-operator propositional generator. Here P, Q are proposition-valued subexpressions, A, B are declared proposition variables, and $\circ \in \{\wedge, \vee, \leftrightarrow, \rightarrow\}$.

1. **Change connective** (`change_op`): replace the connective of $P \circ Q$ by a different member of the connective set, retaining both operands.
2. **Reverse implication** (`flip_arrow`): replace $P \rightarrow Q$ by $Q \rightarrow P$ when the two sides differ syntactically.
3. **Add negation** (`negate`): replace P by $\neg P$.
4. **Remove negation** (`unnegate`): replace $\neg P$ by P.
5. **Wrap** (`wrap`): replace P by $P \circ A$ or $A \circ P$.
6. **Unwrap** (`unwrap`): remove an immediate child that is a declared variable from a binary connective or implication, retaining its sibling in place of the parent.
7. **Delete a subtree** (`delete`): remove either child of a binary connective or implication and replace the parent by the other child. Unlike `unwrap`, the removed child need not be a variable.
8. **Replace by a binary expression** (`replace_with`): replace P by $A \circ B$, with $\circ \in \{\wedge, \vee, \leftrightarrow\}$. Here A and B may coincide; this operation does not introduce an implication.

A.3 NATURAL-NUMBER ARITHMETIC

The arithmetic generator also has 12 operators: the four variable operations and the eight below. Terms t, u have type `nat`; a, b are declared natural-number variables; P, Q are formulas; and s denotes successor. Equality is a formula constructor, not a replaceable arithmetic operator.

1. **Change arithmetic operator** (`change_op`): exchange $+$ and $\times$ at a binary term node, retaining its operands.
2. **Reverse implication** (`flip_arrow`): replace $P \rightarrow Q$ by $Q \rightarrow P$ when the sides differ syntactically.

3. **Add successor** (`negate`): replace t by $s(t)$. The implementation name is inherited from the propositional generator; this operation is not logical negation.

4. **Remove successor** (`unnegate`): replace $s(t)$ by t.

5. **Add an equality hypothesis** (`wrap`): replace P by $(a = b) \to P$, using two distinct declared variables.

6. **Remove a hypothesis** (`unwrap`): replace $P \to Q$ by Q.

7. **Delete a subtree** (`delete`): remove either operand of $t + u$ or $t \times u$, retaining its sibling, or remove either side of an implication and retain the other. Equality and successor nodes are not eligible parents for this operation.

8. **Replace by an arithmetic expression** (`replace_with`): replace a term by $a+b$ or $a \times b$, allowing $a = b$ as variable choices.

A.4 Abelian Groups

The group generator has 13 operators: the four variable operations and the nine below. Terms t, u and declared variables a, b have the group type; P, Q are formulas. There is no connective-change analogue for group terms, since the vocabulary has a single binary group operation.

1. **Reverse implication** (`flip_arrow`): replace $P \to Q$ by $Q \to P$ when the sides differ syntactically.

2. **Add inverse** (`inv_wrap`): replace t by t^{-1}.

3. **Remove inverse** (`inv_unwrap`): replace t^{-1} by t.

4. **Add an equality hypothesis** (`wrap`): replace P by $(a = b) \to P$, using two distinct declared variables.

5. **Remove a hypothesis** (`unwrap`): replace $P \to Q$ by Q.

6. **Delete a subtree** (`delete`): remove either factor of $t \cdot u$ and retain the other, or remove either side of an implication and retain the other. Equality and inverse nodes are not eligible parents.

7. **Replace by a product** (`replace_with`): replace a term by $a \cdot b$, allowing the two variables to coincide.

8. **Extend a product** (`op_wrap`): replace t by $t \cdot a$ or $a \cdot t$. Unlike replacement by a product, this retains t.

9. **Insert an inverse pair** (`inv_pair`): replace t by $t \cdot t^{-1}$ or $t^{-1} \cdot t$.

When an edit removes a variable's last occurrence through subtree removal, the generator removes the orphaned declaration where applicable. Variable identification also removes the redundant declaration, and the remaining declarations are renamed contiguously. Generated strings are deduplicated; no-op outcomes and statements identical to the current inputs are excluded.

B Conjecture Generation Details

Seed statements. Tables 1–3 list the seed statements for the three domains in mathematical notation, that are used for the starting perturbations round. The full domain theories and built-in equality rules are listed in Appendix J.

Table 1: Propositional-logic generation seeds. A, B, and C are propositions, and $\bot$ denotes false. All fourteen statements, including disjunction elimination (`or_e`), are used for generation. Identity is handled by Peano's $intro$ tactic.

Rule	Seed statement
Identity	$A \to A$
`and_el`	$(A \wedge B) \to A$
`and_er`	$(A \wedge B) \to B$
`and_i`	$A \to B \to (A \wedge B)$
`or_il`	$A \to (A \vee B)$
`or_ir`	$B \to (A \vee B)$
`iff_el`	$(A \leftrightarrow B) \to (A \to B)$
`iff_er`	$(A \leftrightarrow B) \to (B \to A)$
`not_e`	$\neg A \to A \to \bot$
`exfalso`	$\bot \to A$
`iff_i`	$(A \to B) \to (B \to A) \to (A \leftrightarrow B)$
`em`	$A \vee \neg A$
`not_i`	$(A \to \bot) \to \neg A$
`or_e`	$(A \vee B) \to (A \to C) \to (B \to C) \to C$

Table 2: Arithmetic generation seeds. $a, n, m \in \mathbb{N}$ and s denotes successor.

Identifier	Seed statement
`refl`	$a = a$
`plus_z`	$n + 0 = n$
`plus_s`	$n + s(m) = s(n + m)$
`times_z`	$n \times 0 = 0$
`times_s`	$n \times s(m) = n + (n \times m)$

Table 3: Abelian groups generation seeds. $a, b, c \in G$, e is the identity, and a^{-1} is the inverse of a.

Rule	Seed statement
`eq_refl`	$a = a$
`id_l`	$e \cdot a = a$
`inv_l`	$a^{-1} \cdot a = e$
`op_comm`	$a \cdot b = b \cdot a$
`op_assoc`	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$

Oracle filtering. Structural perturbations can produce false statements, so we filter candidates before spending compute on Peano proof search. This is done for efficiency reasons do that we don't waste compute on false statements. For propositional logic and arithmetic, we use Z3 (De Moura & Bjørner, 2008) to check whether the negation of a candidate is satisfiable. Propositional statements are encoded as Boolean formulas; arithmetic statements use non-negative integers with the usual addition, multiplication, and successor operations. An unsatisfiable negation establishes validity under this encoding. We implement domain-specific parsers for the supported Peano statement syntax. The parsed expressions are translated into Z3 formulas for propositional logic and arithmetic, or evaluated directly in finite abelian groups for the group domain. We retain only candidates with this verdict in the propositional and arithmetic perturbation pools, excluding counterexamples, unknown

results, and translation failures. For abelian groups, we implement a finite-model counterexample checker. It evaluates each candidate over cyclic groups $\mathbb{Z}_2, \ldots, \mathbb{Z}_9$ and the direct products $\mathbb{Z}_2^2$, $\mathbb{Z}_2 \times \mathbb{Z}_4$, $\mathbb{Z}_2^3$, and $\mathbb{Z}_3^2$. For each model, we enumerate variable assignments and reject a candidate if its hypotheses hold but its conclusion fails. We skip models requiring more than 500,000 group-variable assignments. This is a refutation filter: surviving candidates are not certified as theorems and must still be proved in Peano. Candidates that survive filtering are then eligible for Peano proof search. Only successful Peano searches supply the proof traces and search costs used for selection and training.

Statement deduplication. Across all three domains, we keep only one copy of each repeated statement within a candidate pool, even if it appears in several generation rounds. We standardize variable names before comparing statements, so copies that differ only in their variable names are treated as duplicates. These checks remove syntactic duplicates, obviously not all logically equivalent statements.

Perturbation pool construction. We start from seed statements mentioned above (refer B) for all the three domains. For each input statement, we enumerate applicable edits and sample generations. We repeat this process for five rounds, using the preceding round's well-formed candidates as inputs to the next round. We cap expanded inputs at 5,000 from the fourth round in propositional logic and the third round in arithmetic and groups, when subsampling is necessary. Intermediate seeds need not be proved. We remove repeated statements within and across rounds as described in Appendix B. The five rounds produce $2,667,475$, $556,312$, and $1,027,027$ saved candidate records in logic, arithmetic, and groups, respectively. These totals sum the per-round outputs before cross-round deduplication and final filtering; they are not counts of globally distinct, well-formed statements.

Table 4 reports the candidate counts for each of the five perturbation rounds. These counts include repeated statements across rounds and candidates later discarded by the well-formedness and oracle checks.

Before proof collection, candidates must pass Peano's well-formedness check and the oracle filtering procedure described above. To limit proof-search cost, we cap each (round, primary seed ancestor) stratum at 200 candidates in logic and 1,500 in groups, sampling with a fixed seed. The primary ancestor is the alphabetically first seed among a candidate's recorded seed ancestors. We attempt all filtered arithmetic candidates. These filters and caps precede the selection methods studied below and are shared across methods using the same perturbation pool. Only candidates subsequently proved in Peano provide the proof traces used for training (Section 3).

Minimo pool construction. We collect Minimo conjectures from saved self-play runs (Section 3). We use generations across five iterations having batches of 200 statements each. From the collected statements, we remove repeated statements (Appendix B), and check well-formedness in Peano. This go through the same oracle filtering step for efficiency purposes above. We submit all remaining candidates to best-first search with the starting prover, regardless of whether they were proved during the original Minimo run. Only statements proved by this search supply training proofs.

Table 5 summarizes the candidate pools. "Filtered" counts candidates remaining after removing duplicate statements, checking well-formedness in Peano, and applying the Z3 checks for logic and arithmetic or finite-group counterexample checks for groups, as described above. "Attempted" counts candidates submitted to Peano proof search after the sampling step. "Proved" counts successful proof searches.

This gives 5,530 proved perturbations in propositional logic, 8,584 in arithmetic, and 1,427 in abelian groups, compared with 162, 148, and 592 proved Minimo conjectures, respectively. These proved statements form the pools from which training data are selected.

C BEST-FIRST PROOF SEARCH

Policy scoring Details. For each legal action a at state s, we give the model the input:

```
S: <proof state>
A: <legal action>
```

Table 4: Saved perturbation candidates generated each round, before deduplication and filtering.

Round	Propositional logic	Arithmetic	Abelian groups
1	1,468	134	195
2	52,201	2,842	7,313
3	1,871,928	57,703	224,591
4	366,115	218,118	379,802
5	375,763	277,515	415,126
Sum	2,667,475	556,312	1,027,027

Table 5: Candidate pools before training-data selection. Filtered counts are after deduplication and filters described in Section B. Attempted candidates are the ones sampled for the proof search.

Domain	Source	Filtered	Attempted	Proved
Propositional logic	Perturbations	795,258	10,861	5,530
	Minimo	184	184	162
Arithmetic	Perturbations	19,356	19,356	8,584
	Minimo	783	783	148
Abelian groups	Perturbations	52,266	17,071	1,427
	Minimo	776	776	592

???

The proof state contains the local context and current goal. During training, this input is followed by Y for the action taken in the recorded proof, or N for another legal action.

Let $z_Y(s, a)$ and $z_N(s, a)$ denote the model's next-character logits for these two labels. We compute an action score

$$q(s, a) = \frac{\exp(z_Y(s, a))}{\exp(z_Y(s, a)) + \exp(z_N(s, a))}.$$

We then normalize across the legal actions $\mathcal{A}(s)$:

$$\pi(a \mid s) = \frac{q(s, a)}{\sum_{a' \in \mathcal{A}(s)} q(s, a')}.$$

Greedy rollouts choose the action with the largest probability, while sampled rollouts draw from this distribution at temperature 1. BFS assigns each partial proof the cumulative priority

$$g = \sum_t \log \pi(a_t \mid s_t)$$

and expands the highest-priority partial proof next. This priority is computed from the policy probabilities, not from a separate learned value function.

Algorithm 1 maintains a frontier of partial proofs, each containing an ordered list of open subgoals and its action history. Here $\|$ denotes sequence concatenation. The optional action cap is disabled when $A_{\max} = 0$. The pseudocode shows the pop-budget logic; searches also stop at the configured time limit.

D Policy Rollout Search

Algorithm 2 first runs a greedy trajectory, then up to n sampled trajectories, stopping at the first success. Each attempt starts from a fresh copy of the initial state. The queue contains only the open subgoals of the current trajectory, not alternative partial proofs. Executing an action replaces the focused subgoal with its resulting subgoals; the attempt succeeds when none remain. Here $\pi(a \mid s)$

Algorithm 1 Policy-guided best-first proof search

Require: policy π, initial state s_0, pop budget B
Require: optional action cap $A_{\max}$
Ensure: a solving action sequence, or UNSOLVED

frontier $\leftarrow$ max-priority queue keyed by g
push (g=0, *subgoals*=$[s_0]$, *acts*=$[\,]$) onto *frontier*
pops $\leftarrow 0$
while *frontier* $\neq \emptyset$ **and** *pops* $< B$ **do**
  (g, *subgoals*, *acts*) $\leftarrow$ POPMAX(*frontier*)
  pops $\leftarrow$ *pops* $+ 1$
  legal $\leftarrow$ LEGALACTIONS(*subgoals*[0])
  if *legal* $= \emptyset$ **or** ($A_{\max}$>0 **and** |*legal*|>$A_{\max}$) **then**
    continue
  end if
  scores $\leftarrow \pi$(*subgoals*[0], *legal*)
  $\log \pi \leftarrow \log\big($*scores*$/\sum$ *scores*$\big)$
  for each action $a_i \in$ *legal* **do**
    spawned $\leftarrow$ EXPAND(*subgoals*[0], a_i)
    subgoals$' \leftarrow$ *spawned* $\|$ *subgoals*[1:]
    if *subgoals*$' = \emptyset$ **then**
      return *acts* $\|\, [a_i]$
    end if
    push ($g + \log \pi_i$, *subgoals*$'$, *acts* $\|\, [a_i]$) onto *frontier*
  end for
end while
return UNSOLVED

is the policy distribution over legal actions, and $\|$ denotes sequence concatenation. Sampled actions use $\pi_\tau(a \mid s) \propto \pi(a \mid s)^{1/\tau}$.

The reported evaluations use $n = 16$, $D = 60$, $\tau = 1$, and a single 600-second time budget shared by all attempts. Deadline checks occur between attempts and at the beginning of each action step; an in-progress enumeration, scoring, or execution operation can therefore exceed the time limit. The pseudocode omits exception handling: the implementation abandons an attempt if action enumeration fails, and skips an action whose execution fails, choosing again among the remaining actions at that state. The optional legal-action cap is disabled in these evaluations.

E CONJECTURE SELECTION DETAILS

Statement representation and scores. The selection tokenizer separates each parenthesis and square bracket from the surrounding text, then splits on whitespace. Variable names, operators, type names, and delimiters therefore contribute to both the token distance and the statement length. These tokens are distinct from the raw character IDs used by the prover's language model. Distances are computed with unit-cost Levenshtein distance. Hardness values are saved best-first queue-removal counts from the starting-prover collection; unsolved candidates are simply not selected. The HN implementation and stores the log-hardness values and pairwise ratios. Hard/length instead uses the raw count $b(c)$ divided by token length.

Promotion procedure. Algorithm 3 summarizes the HN selection procedure using the score defined in Equation 1.

Settings for prover improvement. Table 6 reports the HN traversals supplying the finetuning datasets. The propositional run uses 13 solved seed statements as references, Arithmetic and groups each use five references. All three traversals stop because no remaining candidate has an easier reference in the latest batch, before reaching their round cap. Promoted counts exclude the initial seeds and precede subsampling to the training-example budget. These settings describe the prover-improvement experiment.

Algorithm 2 Policy rollout search

Require: policy π, initial state s_0, sampled attempts n, action limit D
Require: temperature τ, shared time budget T
Ensure: a solving action sequence, or UNSOLVED

deadline $\leftarrow$ TIME $+ T$
result $\leftarrow$ ROLLOUT$(s_0, \text{GREEDY}, D, \tau, \textit{deadline})$
if *result* is solved **then**
  return *result*
end if
for $j = 1, \ldots, n$ **do**
  if TIME $>$ *deadline* **then**
    break
  end if
  result $\leftarrow$ ROLLOUT$(s_0, \text{SAMPLE}, D, \tau, \textit{deadline})$
  if *result* is solved **then**
    return *result*
  end if
end for
return UNSOLVED

function ROLLOUT$(s_0, \textit{mode}, D, \tau, \textit{deadline})$
  queue $\leftarrow [\text{CLONE}(s_0)]$; *acts* $\leftarrow [\,]$
  for *step* $= 1, \ldots, D$ **do**
    if TIME $>$ *deadline* **then**
      return TIMEOUT
    end if
    if *queue* $= \emptyset$ **then**
      return $(\text{SOLVED}, \textit{acts})$
    end if
    $s \leftarrow$ *queue*$[0]$; *legal* $\leftarrow$ LEGALACTIONS(s)
    if *legal* $= \emptyset$ **then**
      return DEAD
    end if
    if *mode* $=$ GREEDY **then**
      $a \leftarrow \arg\max_{a' \in \textit{legal}} \pi(a' \mid s)$
    else
      sample $a \sim \pi_\tau(\cdot \mid s)$ over *legal*
    end if
    spawned $\leftarrow$ EXPAND(s, a)
    queue $\leftarrow$ *spawned* $\|$ *queue*$[1{:}]$
    acts $\leftarrow$ *acts* $\| [a]$
  end for
  if *queue* $= \emptyset$ **then**
    return $(\text{SOLVED}, \textit{acts})$
  end if
  return MAXDEPTH
end function

Table 6: HN promotion settings and resulting candidate sets for prover improvement. Rounds counts completed promotion rounds.

Domain	Seeds	Batch cap	Round cap	Rounds	Promoted
Propositional logic	13	20	30	7	140
Arithmetic	5	20	30	6	114
Abelian groups	5	100	60	6	546

Algorithm 3 Hard Neighbours selection on a fixed proved pool

Require: proved candidates $\mathcal{C}$, proved seed references R_1
Require: search costs b, token distance d, batch cap m, round cap T
Ensure: promoted candidate set $\mathcal{P}$
$\mathcal{U} \leftarrow \mathcal{C} \setminus R_1$; $\mathcal{P} \leftarrow \emptyset$; $R \leftarrow R_1$
for $t = 1, \ldots, T$ **do**
  $\mathcal{E} \leftarrow \{c \in \mathcal{U} : \exists r \in R,\ b(r) < b(c)\}$
  if $\mathcal{E} = \emptyset$ **then**
    break
  end if
  Compute $\mathrm{HN}(c; R)$ for each $c \in \mathcal{E}$ using Equation 1
  $\mathcal{B} \leftarrow$ the $\min\{m, |\mathcal{E}|\}$ highest-scoring candidates
  $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{B}$; $\mathcal{U} \leftarrow \mathcal{U} \setminus \mathcal{B}$
  $R \leftarrow \mathcal{B}$
end for
return $\mathcal{P}$

From candidate sets to training datasets. Each successful proof contributes labeled state–action training examples. At each state visited in the proof, we pair the state with every legal action. The action taken in the recorded proof receives label Y, and the alternatives receive label N. Each example has the form:

```
S: <proof state>
A: <legal action>
???<label>
```

The proof state contains the local context and current goal. An N label indicates that the action was not taken in the recorded proof, not that it is invalid or cannot lead to another proof.

We combine the examples from selected proofs and deduplicate using exact string match. We include all examples from each proof and stop before adding the next proof would exceed the training-example budget.

Random takes proved perturbations in random order. HN samples the conjectures from the pool of the conjectures selected by its promotion procedure. Hardest, Hard/length, and Shortest follow their respective rankings. We shuffle the resulting training examples before finetuning, using three seeds.

Finetuning setup. All finetuning runs start from the same initial prover checkpoint. We train for 100 epochs using AdamW with a learning rate of 10^{-4}, weight decay of 0.01, and batches of 64 training examples. We reshuffle the examples at the start of each epoch. The objective is next-character cross-entropy over the complete labeled state–action strings, including the Y/N labels, with padding excluded from the loss.

Compute resources. All experiments were conducted using a pool of 12 NVIDIA A100 GPUs, each with 80 GB of GPU memory. Proof search was run with six parallel workers per GPU across all experiments.

E.1 Training Dataset Sizes

Tables 7–9 report the final datasets used for prover finetuning, separately for data-selection and shuffling seeds 42, 43, and 44. Conjecture counts refer to whole statements retained after budget matching. Training examples are distinct policy-example strings after deduplication within each dataset; these counts are not proof lengths or numbers of gradient updates. The example-count targets are 1,800 in propositional logic, 3,020 in arithmetic, and 13,204 in groups.

Minimo is the learned-generation baseline; the other five methods select from the perturbation pool. All three seeds are shown. The tables describe the prover-improvement experiment.

Table 7: Propositional-logic finetuning datasets. The target is 1,800 training examples. Conj. denotes selected conjectures.

	Seed 42		Seed 43		Seed 44	
Method	Conj.	Examples	Conj.	Examples	Conj.	Examples
Minimo	162	1,800	162	1,800	162	1,800
Random	95	1,786	98	1,797	92	1,777
HN	66	1,783	69	1,792	67	1,785
Hardest	62	1,775	62	1,775	62	1,775
Hard/length	61	1,779	61	1,779	61	1,779
Shortest	149	1,799	149	1,799	149	1,799

Table 8: Natural-number multiplication finetuning datasets. The target is 3,020 training examples. Conj. denotes selected conjectures.

	Seed 42		Seed 43		Seed 44	
Method	Conj.	Examples	Conj.	Examples	Conj.	Examples
Minimo	148	3,020	148	3,020	148	3,020
Random	159	3,009	156	3,013	164	3,009
HN	96	2,996	96	2,996	94	2,996
Hardest	96	2,991	96	2,991	96	2,991
Hard/length	98	3,014	98	3,014	98	3,014
Shortest	298	3,016	298	3,016	298	3,016

Table 9: Abelian-group finetuning datasets. The target is 13,204 training examples. Conj. denotes selected conjectures.

	Seed 42		Seed 43		Seed 44	
Method	Conj.	Examples	Conj.	Examples	Conj.	Examples
Minimo	592	13,204	592	13,204	592	13,204
Random	556	13,195	542	13,183	558	13,189
HN	475	13,179	473	13,182	473	13,180
Hardest	466	13,199	466	13,199	466	13,199
Hard/length	466	13,194	466	13,194	466	13,194
Shortest	562	13,174	562	13,174	562	13,174

E.2 BENCHMARK-THEOREM RECOVERY POOLS

Table 10: Original proved perturbation and Minimo source-pool sizes for benchmark-theorem recovery. Gold counts evaluated benchmark theorems proved by the starting prover.

Domain	Perturbation	Gold	Minimo	Total
Propositional logic	5,530	13	162	5,705
Arithmetic	8,584	22	148	8,754
Abelian groups	1,427	9	592	2,028

Table 11: Base-prover benchmark-theorem recovery using the HN settings from the corresponding prover-improvement experiment. Entries give the percentage of available Gold or Minimo statements selected, rounded to one decimal place. Random gives the expected percentage under uniform sampling without replacement.

	Logic		Arithmetic		Groups	
Method	Gold	Minimo	Gold	Minimo	Gold	Minimo
HN	46.2	1.2	29.6	4.2	33.3	3.4
Hardest	0.0	1.9	7.4	3.5	33.3	1.4
Hard/length	7.7	5.6	22.2	3.5	33.3	2.6
Shortest	38.5	46.3	25.9	34.5	66.7	39.9
Random (expected)	2.5	2.5	1.3	1.3	22.3	22.3

F Evaluation Benchmarks

Table 12 describes the reasoning involved at each benchmark level, using the same level numbers as the results figures. In logic, the levels distinguish basic rule use, constructive reasoning, and classical reasoning. Arithmetic separates rewriting from induction, while group problems vary in rewrite depth, inverse manipulation, and use of equality hypotheses.

Table 12: Benchmark levels used to evaluate prover improvement. The benchmarks contain 80 problems in propositional logic, 120 in natural-number multiplication, and 59 in abelian groups.

Level	Problems	Main reasoning requirements
Propositional logic		
L1	4	Basic implication and negation
L2	13	Short compositions of logical rules and equivalences
L3	29	Multi-step constructive reasoning and case analysis
L4	14	Classical reasoning and double-negation elimination
L5	11	Composite classical equivalences and transformations
L6	9	Nested equivalences and multi-premise reasoning
Natural-number multiplication		
L1	20	Short rewrite chains and congruence
L2	30	Equality chains and substitution using hypotheses
L3	30	One-variable induction
L4	25	Longer composite rewrites without induction
L5	15	Multi-variable induction and general arithmetic laws
Abelian groups		
L1	16	Short associativity, commutativity, and identity rewrites
L2	15	Inverse manipulation and inverse identities
L3	10	Cancellation and reasoning from equality hypotheses
L4	10	Multi-variable rearrangements and inverse combinations
L5	8	Longer composite identities and equation solving

Table 13: Overall solve rates and coverage (%) for every method under both search procedures. Random outperforms Minimo in all three domains under best-first search; under rollouts, Minimo leads only in arithmetic. The best selection method depends on the domain, search procedure, and metric. No selection method exceeds Random's peak solve rate in arithmetic under BFS. Peak is the maximum over evaluated checkpoints of the three-seed mean; coverage is the percentage of problems solved at any checkpoint within each seed, averaged over seeds (Section 6.1). Base is the starting prover evaluated once. Values are rounded to one decimal place. Bold marks the column maximum within each search block, including ties. Rollouts use a greedy attempt followed by up to 16 sampled attempts.

	Logic (80)		Arithmetic (120)		Groups (59)	
Method	Peak	Coverage	Peak	Coverage	Peak	Coverage
Best-first search						
Base	16.3	–	18.3	–	15.3	–
Minimo	46.7	50.4	46.7	51.9	23.2	26.0
Random	54.6	58.8	**55.6**	57.8	28.8	30.5
HN	**60.0**	**66.3**	55.3	**58.1**	30.5	31.6
Hardest	57.9	63.8	**55.6**	57.8	29.9	31.6
Hard/length	57.5	61.3	55.3	57.8	**31.6**	**32.8**
Shortest	55.4	59.6	52.8	57.2	31.1	32.2
Policy rollout search						
Base	0.0	–	20.8	–	13.6	–
Minimo	32.9	37.9	**50.3**	**60.8**	21.5	31.1
Random	45.8	50.0	47.5	54.4	22.6	31.6
HN	48.3	55.8	48.1	55.3	**28.8**	**33.9**
Hardest	**50.4**	54.6	46.7	56.9	27.7	**33.9**
Hard/length	47.9	55.4	48.1	55.3	23.7	30.5
Shortest	50.0	**56.3**	48.9	53.3	26.6	31.6

G Overall Solve Rates and Coverage

Table 13 provides the numerical summaries corresponding to the peak and coverage bars. For each trained method, peak is the maximum over evaluated checkpoints of the mean solve rate across sampling seeds 42, 43, and 44. Coverage is computed by taking the union of solved problems across checkpoints within each seed, then averaging the resulting solve rates across seeds. It is not the solve rate of a single checkpoint. Peak epochs may differ between methods and search procedures; these are retrospective summaries, not validation-selected checkpoint results.

The evaluated epochs are $\{5, 10, 20, 30, 60, 100\}$ for propositional logic and abelian groups, and $\{10, 20, 30, 60, 100\}$ for arithmetic. Base denotes the starting prover, evaluated once per domain and search procedure: its entry in the Peak column is its solve rate, not a checkpoint maximum or a three-seed average. Coverage is not applicable to Base and is shown as a dash. Minimo is the learned-generation baseline; the other methods select from the perturbation pool.

G.1 Level-Wise BFS and Overall Rollout Results

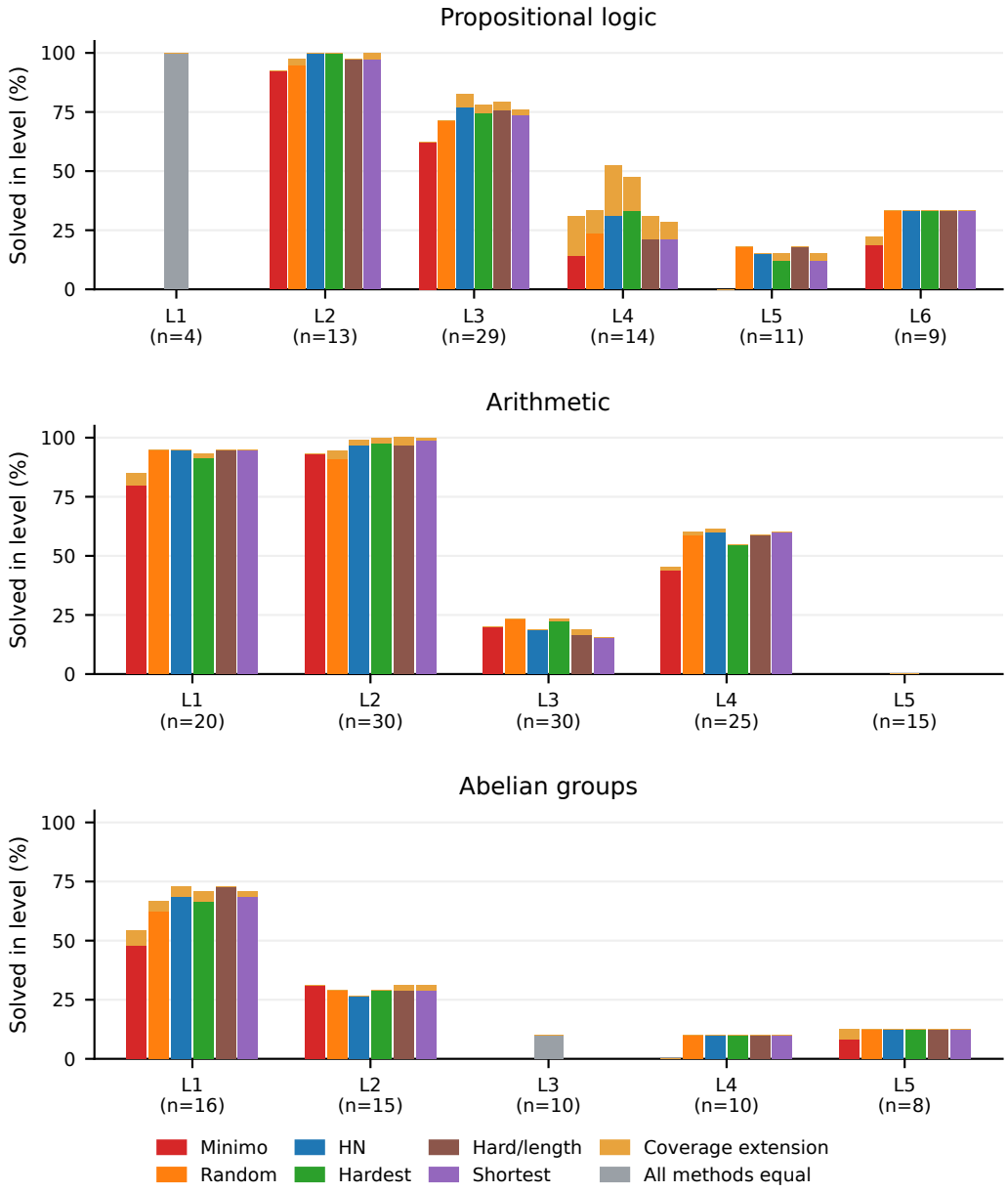


Figure 8: Level-wise best-first-search performance. Each panel show propositional logic, arithmetic, and abelian groups; n is the number of problems in each level. Colored bars show the maximum over checkpoints of the three-seed mean solve rate within that level. Gold extensions indicate coverage, computed by taking the union across checkpoints within each seed and then averaging. A gray bar represents all methods have the same performance; a missing bar has value zero.

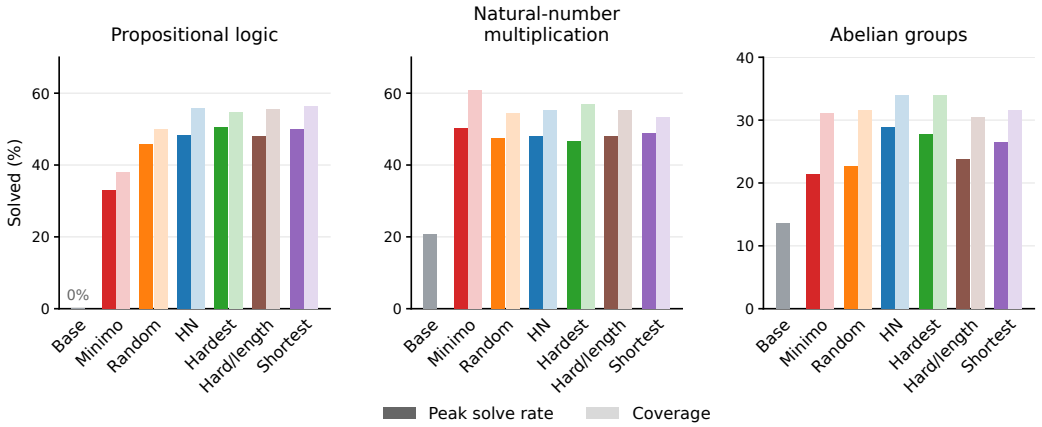


Figure 9: Overall rollout performance, including Base, Minimo, and all five perturbation selection methods. Each problem receives a greedy attempt followed by up to 16 sampled attempts. For trained methods, paired bars show peak solve rate (dark, left) and coverage (light, right): the maximum checkpoint-wise three-seed mean and mean within-seed coverage across checkpoints, respectively. Base is a single gray bar showing the starting prover's solve rate.

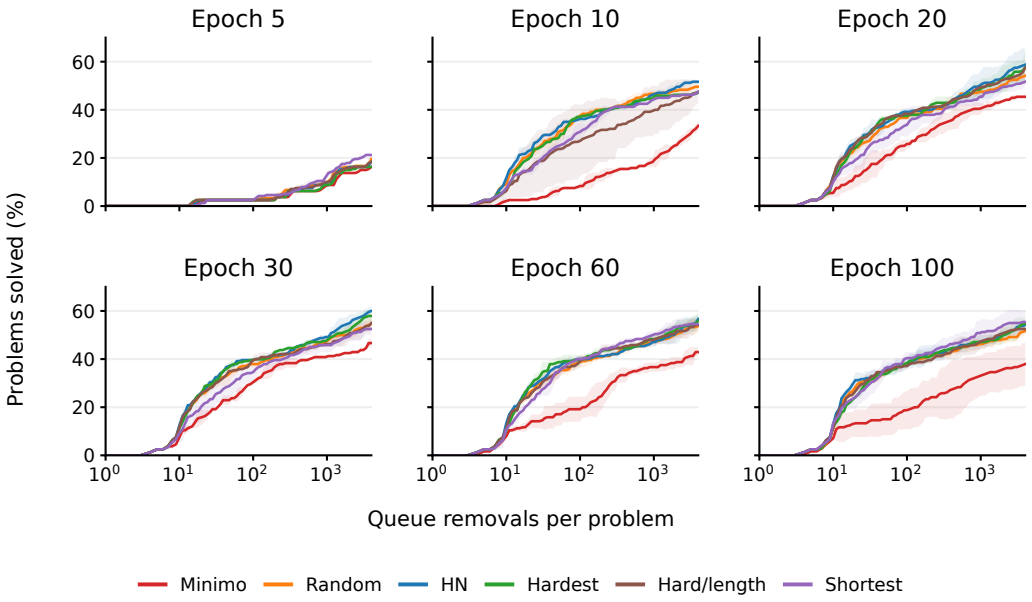


Figure 10: Propositional logic: best-first solve rate versus queue removals per problem, at all six evaluated checkpoints. Lines show three-seed means and shading shows ± 1 sample SD.

H Search-Effort Curves

The following figures show all six training methods at every evaluated checkpoint, separately for the 80 propositional-logic, 120 arithmetic, and 59 abelian-group problems. Lines are means across sampling seeds 42, 43, and 44; shaded bands indicate ± 1 sample standard deviation.

For the cost-based curves, a problem contributes at threshold x only if its saved search succeeded with recorded cost at most x. These are post-hoc summaries of the completed searches. Best-first queue removals count partial proofs taken from the frontier. An action evaluation counts one state–action score: multiple action scores can be computed in a batch. Rollout evaluation counts include all attempts preceding the first success. These cost units are not interchangeable and do not establish equal computational budgets between search procedures. All curves retain the original per-problem time limit.

H.1 Best-First Search: Queue Removals

Figures 10–12 vary the queue-removal threshold up to 4,000, on logarithmic horizontal axes. At the right endpoint, each curve reaches that checkpoint's reported best-first solve rate.

H.2 Best-First Search: Action Evaluations

Figures 13–15 use the total number of policy action evaluations as the horizontal axis. This accounts for the varying number of legal actions scored at each visited state. For each domain, the plotted evaluation range is shared with the rollout evaluation curves and spans every recorded successful search.

H.3 Rollouts: Action Evaluations

Figures 16–18 count action evaluations over the greedy attempt and all subsequent sampled attempts through the first success. The horizontal axes are logarithmic. The 600-second shared deadline and the limit of 60 actions per attempt still apply.

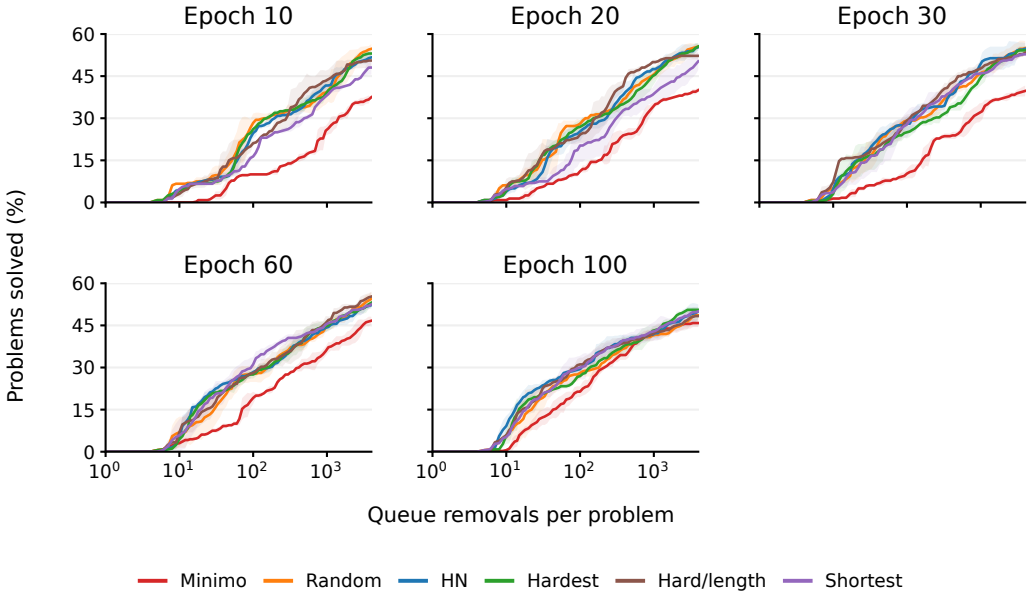


Figure 11: Arithmetic: best-first solve rate versus queue removals per problem, at all five evaluated checkpoints. Lines show three-seed means and shading shows ±1 sample SD.

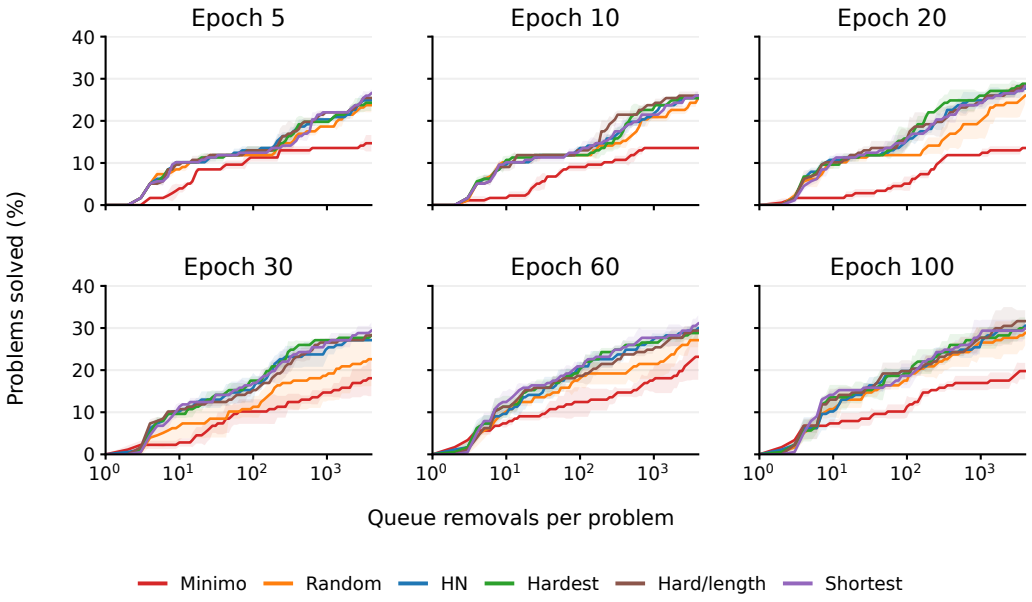


Figure 12: Abelian groups: best-first solve rate versus queue removals per problem, at all six evaluated checkpoints. Lines show three-seed means and shading shows ±1 sample SD.

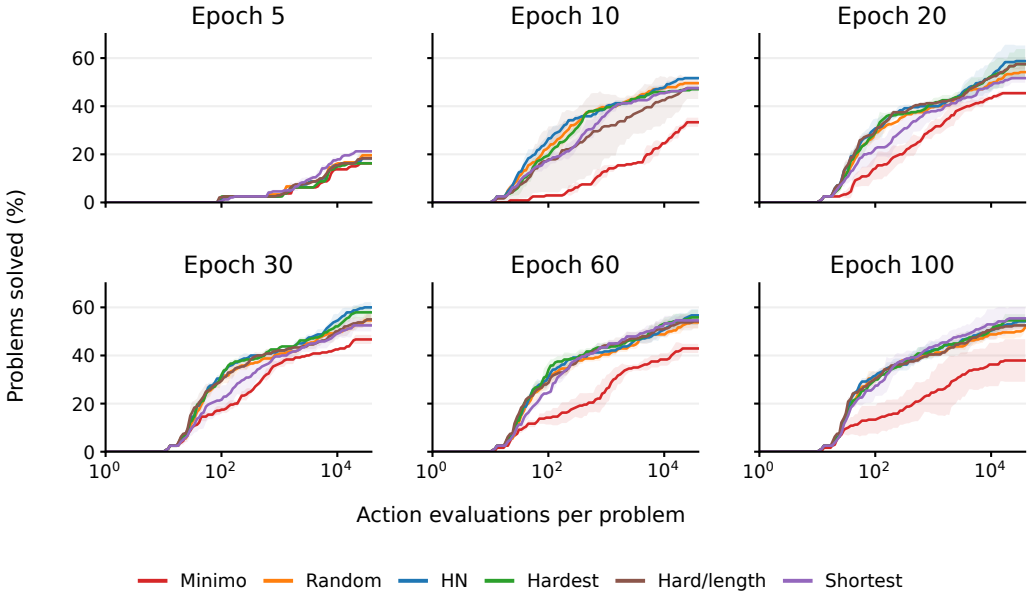


Figure 13: Propositional logic: best-first solve rate versus policy action evaluations per problem. Lines and bands show the three-seed mean ± 1 sample SD.

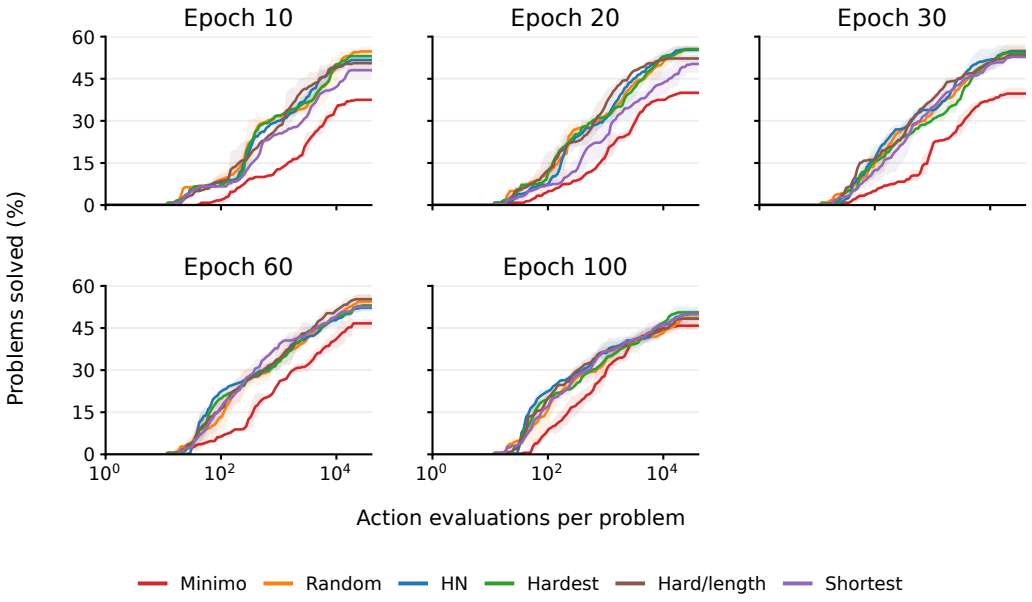


Figure 14: Arithmetic: best-first solve rate versus policy action evaluations per problem. Lines and bands show the three-seed mean ± 1 sample SD.

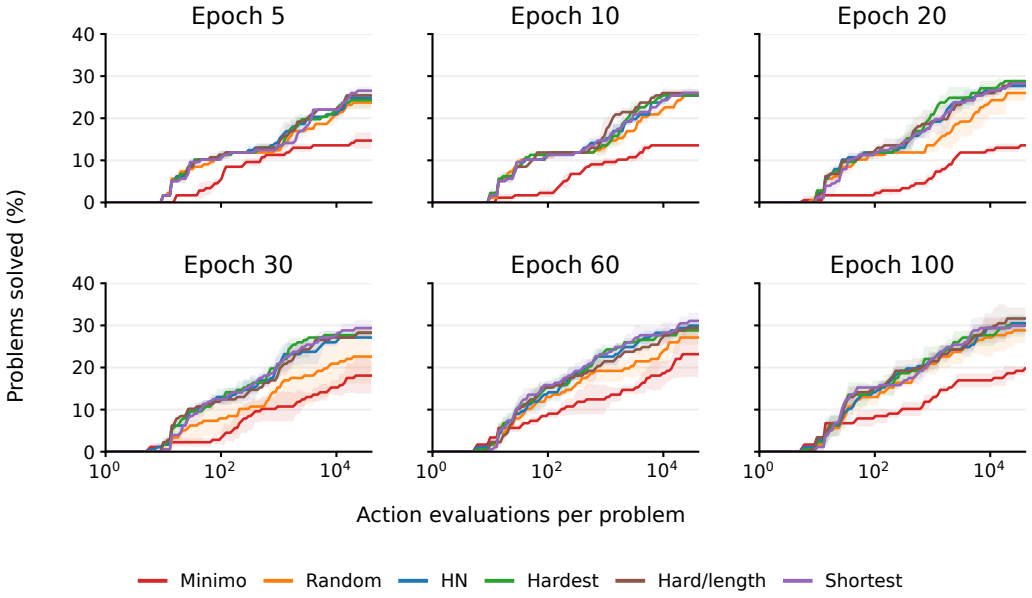


Figure 15: Abelian groups: best-first solve rate versus policy action evaluations per problem. Lines and bands show the three-seed mean ± 1 sample SD.

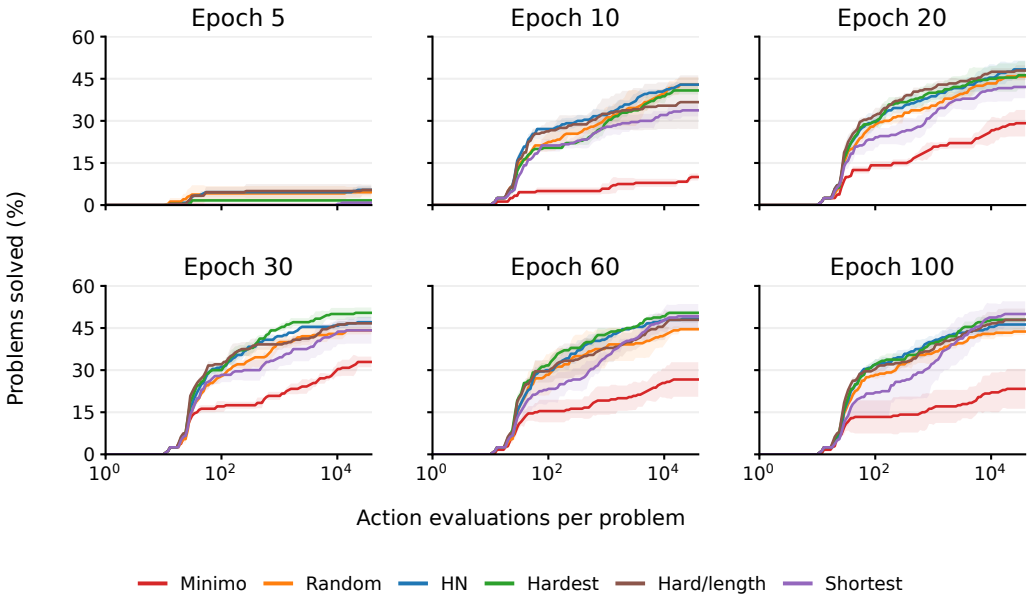


Figure 16: Propositional logic: rollout solve rate versus cumulative policy action evaluations per problem. Lines show three-seed means; shading shows ± 1 sample SD.

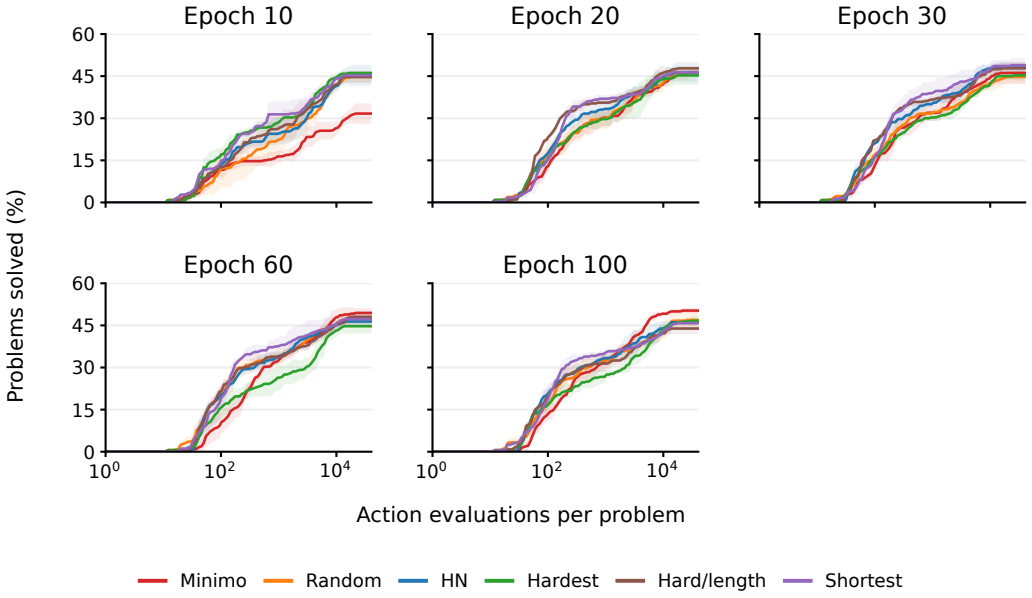


Figure 17: Arithmetic: rollout solve rate versus cumulative policy action evaluations per problem. Lines show three-seed means; shading shows ± 1 sample SD.

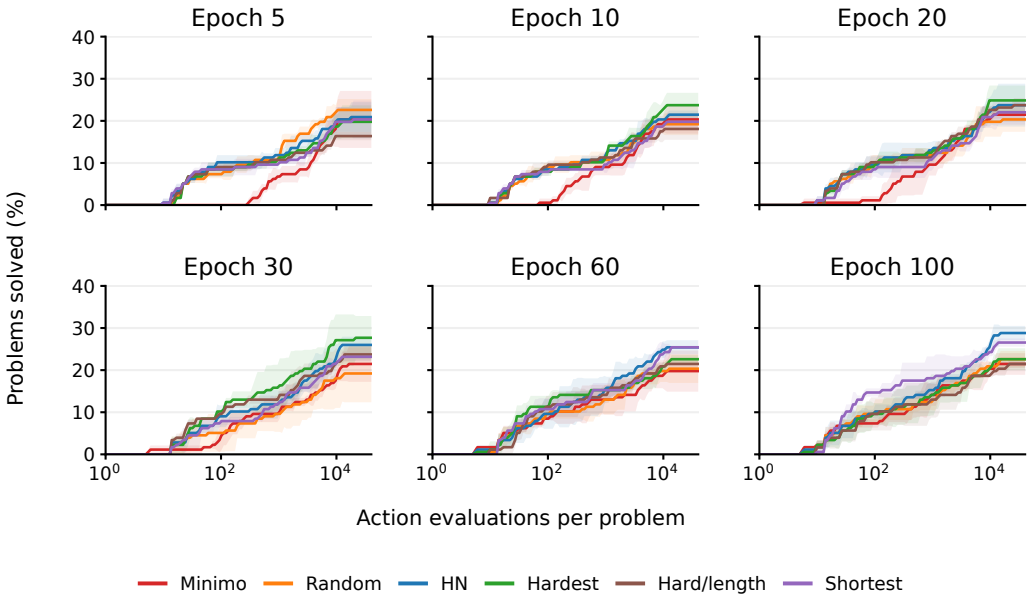


Figure 18: Abelian groups: rollout solve rate versus cumulative policy action evaluations per problem. Lines show three-seed means; shading shows ± 1 sample SD.

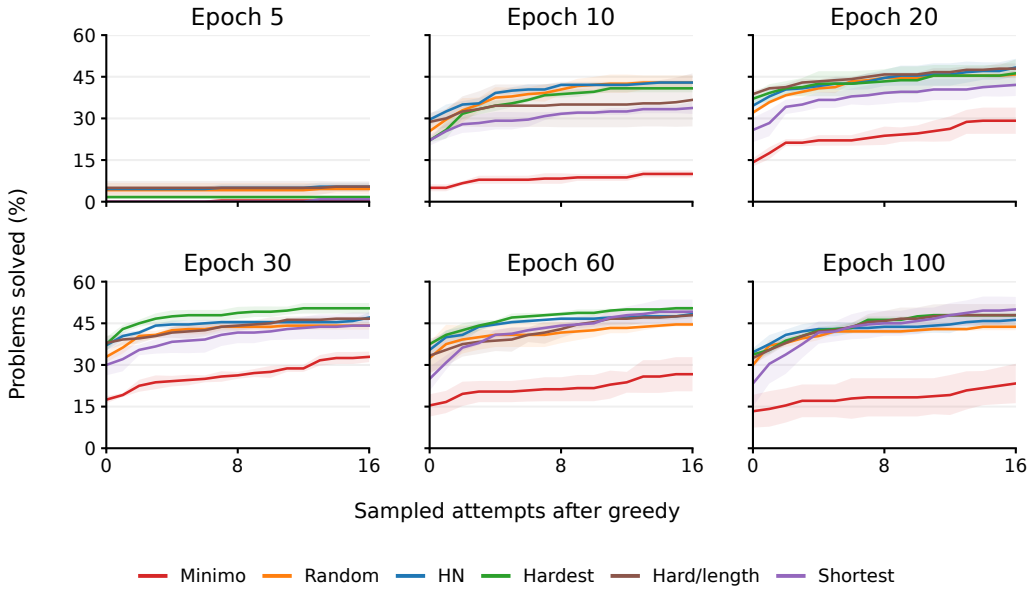


Figure 19: Propositional logic: solve rate after a greedy attempt plus up to k sampled attempts. Lines show three-seed means; shading shows ± 1 sample SD.

H.4 Rollouts: Number of Attempts

Figures 19–21 show success after the greedy attempt and up to k sampled attempts, for $k = 0, \ldots, 16$. The point at $k = 0$ is greedy-only performance; $k = 16$ reaches the reported rollout solve rate. Searches stop at the first success or when the shared time budget expires. These curves describe the saved greedy-first protocol, not unrestricted sampling-only pass@k.

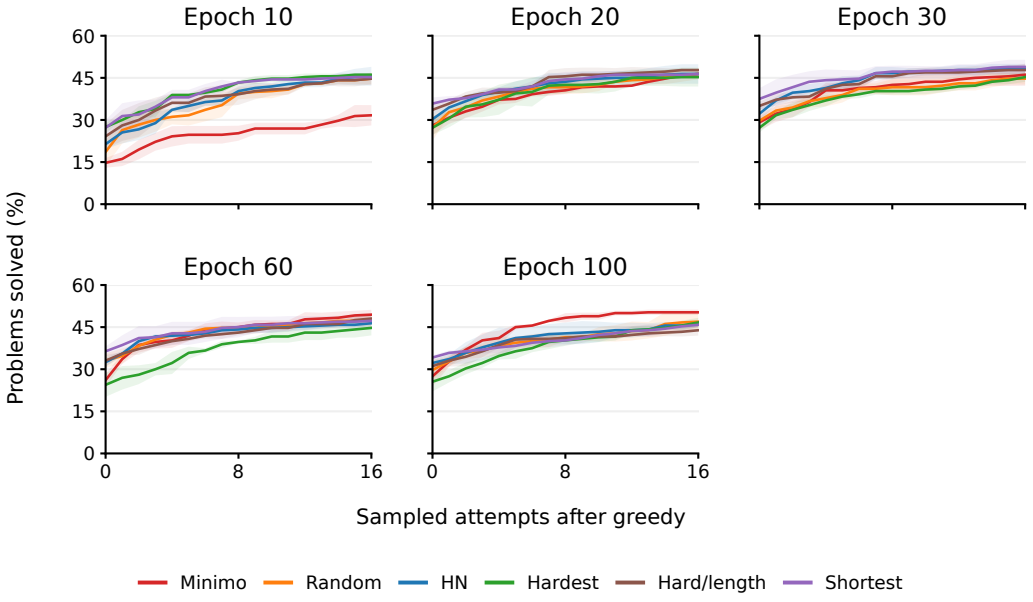


Figure 20: Arithmetic: solve rate after a greedy attempt plus up to k sampled attempts. Lines show three-seed means; shading shows ± 1 sample SD.

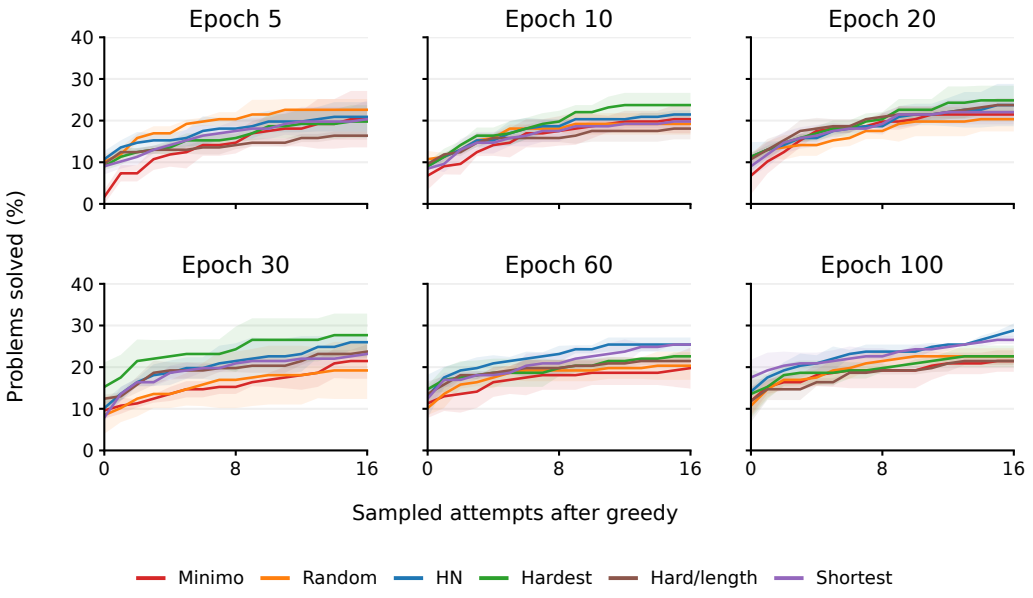


Figure 21: Abelian groups: solve rate after a greedy attempt plus up to k sampled attempts. Lines show three-seed means; shading shows ± 1 sample SD.

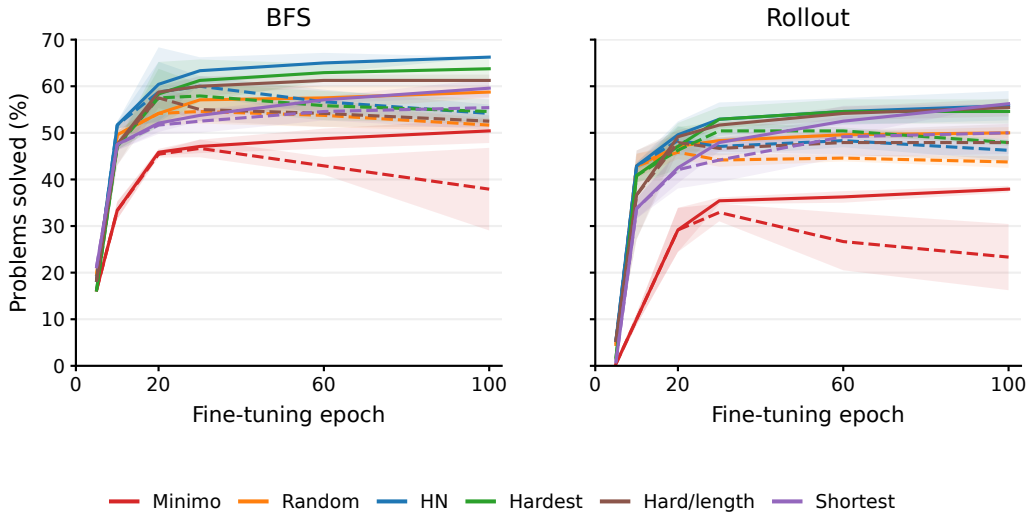


Figure 22: Propositional-logic learning curves for all methods: best-first search (left) and rollouts (right). Dashed lines show solve rate, solid lines show coverage, and shading shows ± 1 sample SD.

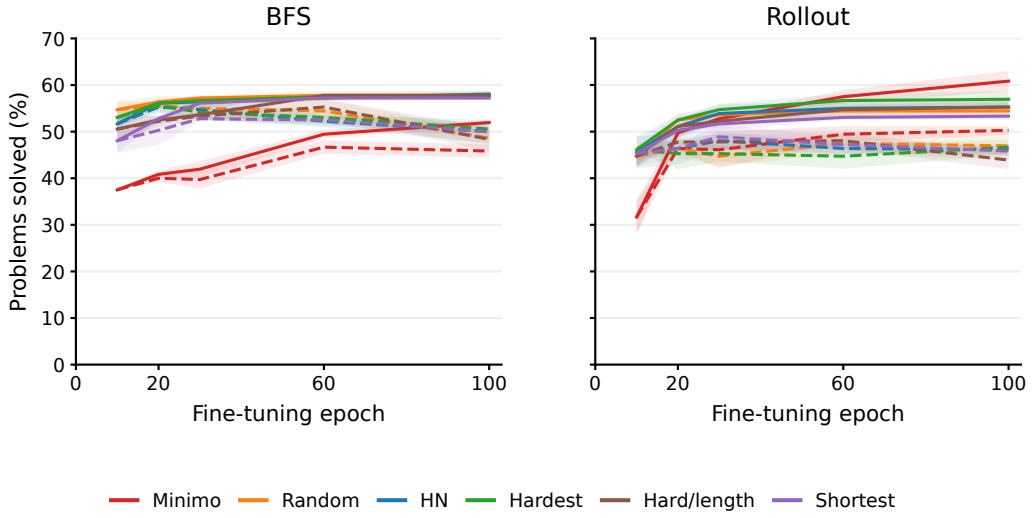


Figure 23: Arithmetic learning curves for all methods: best-first search (left) and rollouts (right). Dashed lines show solve rate, solid lines show coverage, and shading shows ± 1 sample SD.

I Additional Learning and Selection Diagnostics

I.1 All-Method Learning Curves

Figures 22–24 show all six methods under both search procedures. Dashed curves are checkpoint solve rates. Solid curves are coverage up to that checkpoint, computed by forming unions within each seed before averaging. Both metrics use three-seed means with ± 1 sample-SD shading. Coverage is not the ability of a single checkpoint.

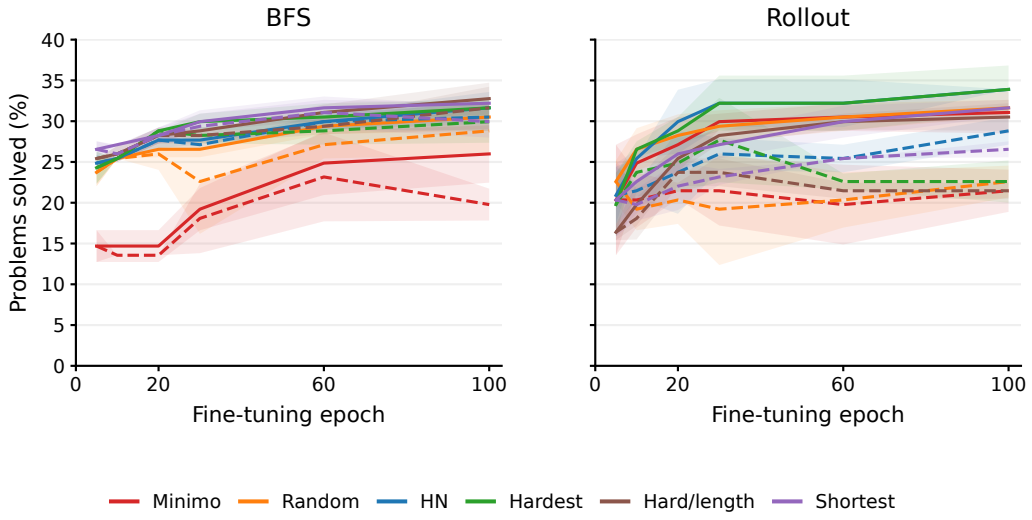


Figure 24: Abelian-group learning curves for all methods: best-first search (left) and rollouts (right). Dashed lines show solve rate, solid lines show coverage, and shading shows ± 1 sample SD.

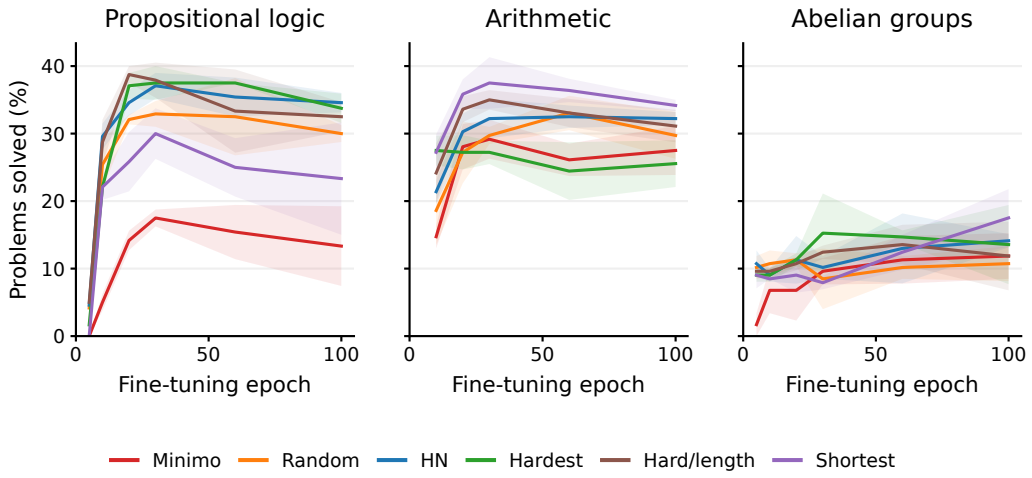


Figure 25: Greedy-only solve rates throughout finetuning in the three domains. Lines are means across three sampling seeds; shading indicates ± 1 sample SD. No sampled attempts contribute to these curves.

I.2 Greedy and Level-Wise Rollout Results

Figure 25 isolates the first, greedy attempt of the rollout protocol. Figure 26 complements the main-text BFS level-wise analysis with rollout results, retaining the same peak and coverage definitions.

I.3 Returned Proof Lengths

Figure 27 compares BFS and rollout proof lengths for Minimo, Random, and HN across three seeds and checkpoints at epochs $\{10, 20, 30, 60, 100\}$. For each method, both searches use the same model checkpoint. At each seed and epoch, we compare all methods on the same set of problems: those

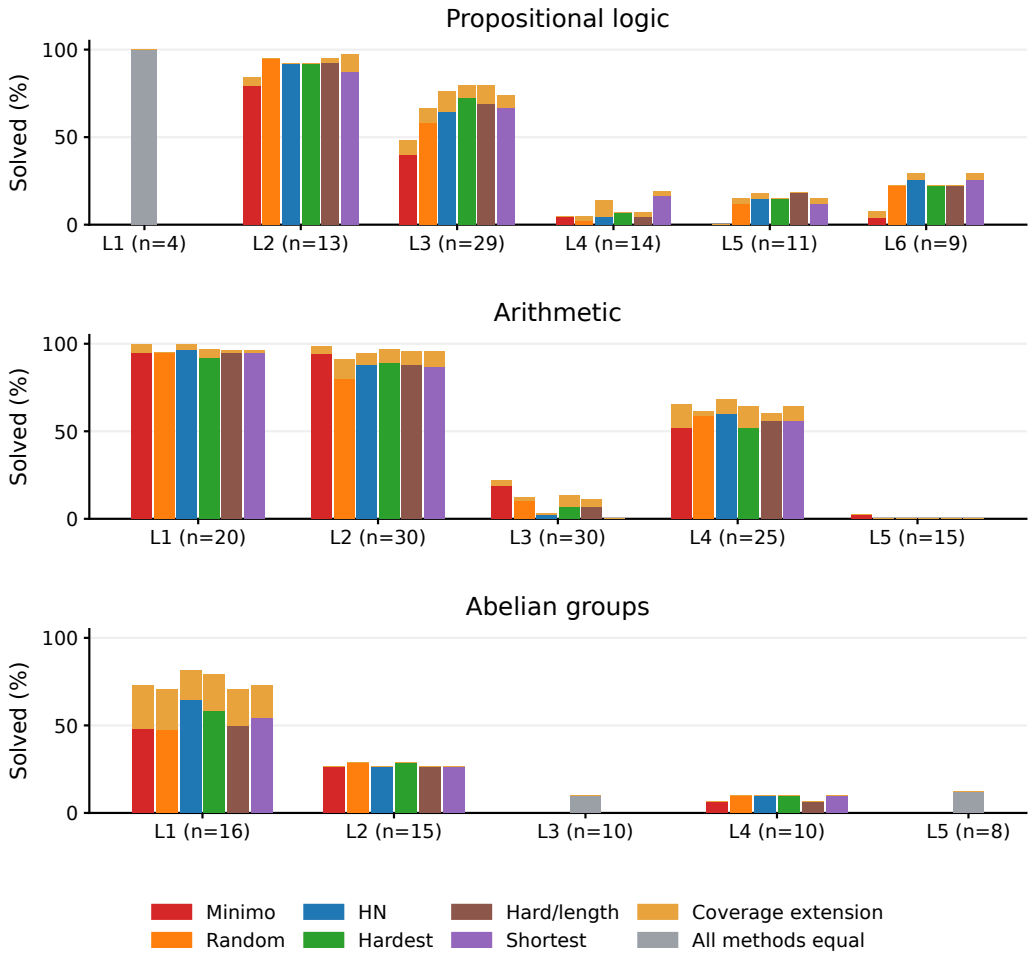


Figure 26: Level-wise rollout performance. Colored bars show the peak checkpoint-wise three-seed mean within each level; gold bars indicate mean within-seed coverage.

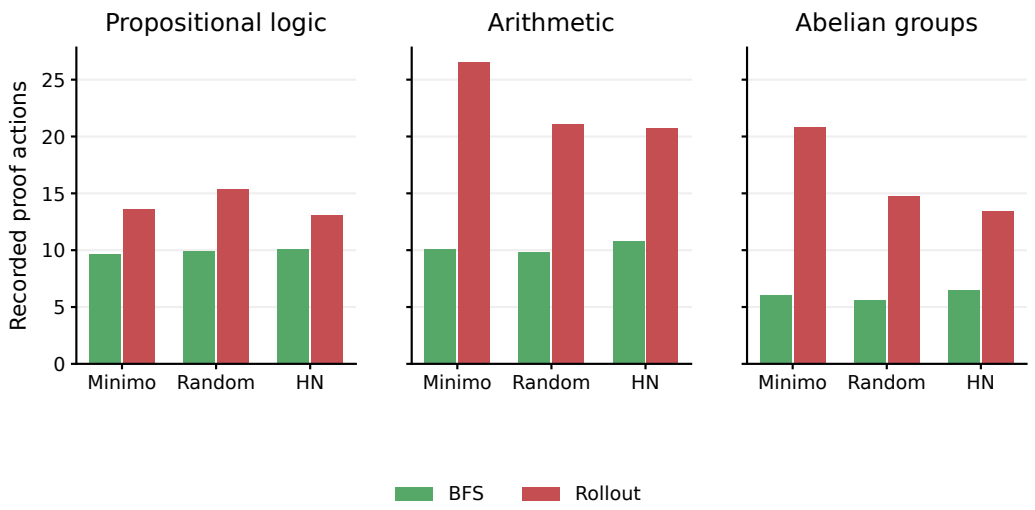


Figure 27: Mean returned proof lengths for BFS and rollout on problems solved by all three displayed methods under both searches, matched within each seed and checkpoint.

solved by all three methods under both searches. For each method and search procedure, we compute the mean proof length on this set, then average equally across the 15 seed–epoch pairs.

A redundant successful rollout. For contraposition, $(A \to B) \to (\neg B \to \neg A)$ (benchmark problem `L3_4`), the Minimo seed-42 model at epoch 100 returns an 11-action BFS proof and a 29-action successful sampled rollout. Table 14 shows the recorded actions from the same checkpoint. The first seven and final four actions are identical. Between them, the rollout repeats the pair `a exfalso, => false.` nine times.

BFS actions	Rollout actions	Recorded action
1–4	1–4	`intro.` (four times)
5	5	`a not_i`
6	6	`=> [x -> false].`
7	7	`intro.`
—	8–25	`a exfalso` (repeat this pair nine times) `=> false.`
8	26	`c x1`
9	27	`=> (x1 x3).`
10	28	`c not_e`
11	29	`=> (not_e x0 x2 p).`

Table 14: Contraposition proved by BFS (11 actions) and a successful sampled rollout (29 actions). The middle block occurs only in the rollout and contributes 18 redundant actions. Counts refer to the returned successful traces, not search effort or failed attempts.

The first four introductions name the propositions `x` (A) and `x0` (B), and the assumptions `x1`: $A \to B$ and `x2`: $\neg B$. Negation introduction then reduces the goal $\neg A$ to proving `false` under an additional assumption `x3`: A. In the trace, `a` chooses backward application, `c` chooses forward construction, and `=>` selects an instantiation.

With the goal already `false`, each repeated pair applies ex falso backward with conclusion `false`, replacing the obligation to prove false by the same obligation. The logical context and goal are unchanged after each pair; no contradiction has yet been derived. The final four actions derive `p`: B by applying `x1` to `x3`, then combine `p` with `x2`: $\neg B$ to derive false and close the proof.

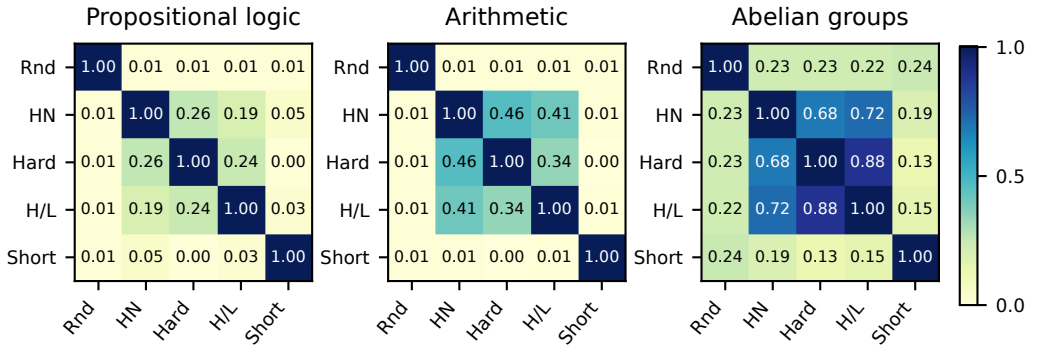


Figure 28: Mean within-seed Jaccard overlap of selected conjecture sets. Rnd denotes Random, Hard denotes Hardest, H/L denotes Hard/length, and Short denotes Shortest.

Removing the 18 repeated actions yields exactly the saved BFS trace. This example illustrates concrete redundancy, not its prevalence or an explanation of the arithmetic rollout advantage.

I.4 Selected-Candidate Overlap

Figure 28 reports Jaccard overlap, $J(S_1, S_2) = |S_1 \cap S_2|/|S_1 \cup S_2|$, between the final selected conjecture sets. We compute each overlap within a sampling seed and then average across seeds 42, 43, and 44; we do not pool the seed sets before computing the ratio. Candidate sets are recovered from their saved training examples and their sizes checked against selection metadata. These are statement-set overlaps, not proof-step overlaps or measures of semantic equivalence. Minimo is omitted because it uses a different source pool.

J DOMAIN AXIOMS AND A WORKED PROOF

The following listings give the declarations and search annotations used in the three Peano theory files. They specify the domain vocabulary and inference rules, not the smaller sets of statements used to seed conjecture generation. In Peano, `[X -> Y]` denotes a function type, representing implication when `X` and `Y` are propositions. A named argument, as in `[('x : T) -> ...]`, binds a variable of type `T`.

The `#forward` and `#backward` annotations govern how rules are used in search; they are not additional logical axioms. Forward rules construct facts from the context, while backward rules reduce the current goal to subgoals. Pattern annotations restrict forward instantiation; `infer` and `subgoal` specify argument roles in backward use. Introduction of variables and hypotheses, and closure of goals by existing facts, are provided by the environment.

J.1 PROPOSITIONAL LOGIC

The configured premise set consists of the thirteen named rules below, from `and_i` through `em`. The excluded-middle rule makes this a classical propositional theory.

```
prop : type.
false : prop.

not : [prop -> prop].
and : [prop -> prop -> prop].
or : [prop -> prop -> prop].
iff : [prop -> prop -> prop].

#backward and_i.
and_i : [('P : prop) -> ('Q : prop) -> 'P -> 'Q -> (and 'P 'Q)].
#forward and_el ('_ : (and 'P 'Q)).
and_el : [('P : prop) -> ('Q : prop) -> (and 'P 'Q) -> 'P].
#forward and_er ('_ : (and 'P 'Q)).
and_er : [('P : prop) -> ('Q : prop) -> (and 'P 'Q) -> 'Q].

#backward or_il.
or_il : [('P : prop) -> ('Q : prop) -> 'P -> (or 'P 'Q)].
#backward or_ir.
or_ir : [('P : prop) -> ('Q : prop) -> 'Q -> (or 'P 'Q)].
#backward or_e infer infer infer infer subgoal subgoal.
or_e : [('P : prop) -> ('Q : prop) -> ('R : prop) ->
   (or 'P 'Q) -> ['P -> 'R] -> ['Q -> 'R] -> 'R].

#backward not_i.
not_i : [('P : prop) -> ['P -> false] -> (not 'P)].
not_e : [('P : prop) -> (not 'P) -> 'P -> false].
#backward exfalso.
exfalso : [false -> ('P : prop) -> 'P].
#backward iff_i.
iff_i : [('P : prop) -> ('Q : prop) -> ['P -> 'Q] ->
    ['Q -> 'P] -> (iff 'P 'Q)].
#forward iff_el ('_ : (iff 'P 'Q)).
iff_el : [('P : prop) -> ('Q : prop) ->
          (iff 'P 'Q) -> ['P -> 'Q]].
#forward iff_er ('_ : (iff 'P 'Q)).
iff_er : [('P : prop) -> ('Q : prop) ->
          (iff 'P 'Q) -> ['Q -> 'P]].
#forward em.
em : [('P : prop) -> (or 'P (not 'P))].
```

J.2 NATURAL-NUMBER ARITHMETIC WITH MULTIPLICATION

Here `z` is zero and `s` is successor. Addition and multiplication recurse on their second argument. The rule `nat_ind` supplies induction over natural numbers. Search uses the five named rules below together with the three built-in equality rules described in Appendix J.4.

```
= : [('t : type) -> 't -> 't -> prop].
nat : type.
z : nat.
s : [nat -> nat].
+ : [nat -> nat -> nat].
* : [nat -> nat -> nat].

+_z : [('n : nat) -> (= (+ 'n z) 'n)].
+_s : [('n : nat) -> ('m : nat) ->
  (= (+ 'n (s 'm)) (s (+ 'n 'm)))].
*_z : [('n : nat) -> (= (* 'n z) z)].
*_s : [('n : nat) -> ('m : nat) ->
  (= (* 'n (s 'm)) (+ 'n (* 'n 'm)))].

nat_ind : [('p : [nat -> prop]) -> ('p z) ->
      [('n : nat) -> ('p 'n) -> ('p (s 'n))] ->
      [('n : nat) -> ('p 'n)]].

#backward nat_ind.
#forward +_z ((+ 'n z) : nat).
#forward +_s ((+ 'n (s 'm)) : nat).
#forward *_z ((* 'n z) : nat).
#forward *_s ((* 'n (s 'm)) : nat).
```

J.3 ABELIAN GROUPS

Here `op`, `id`, and `inv` denote the group operation, identity, and inverse. The axioms provide associativity, commutativity, a left identity, and a left inverse. Search uses these four rules together with the three built-in equality rules.

```
= : [('t : type) -> 't -> 't -> prop].
G : type.
op : [G -> G -> G].
id : G.

#forward op_assoc ((op (op 'a 'b) 'c) : G).
op_assoc : [('a : G) -> ('b : G) -> ('c : G) ->
       (= (op (op 'a 'b) 'c) (op 'a (op 'b 'c)))].

#forward op_comm ((op 'a 'b) : G).
op_comm : [('a : G) -> ('b : G) ->
      (= (op 'a 'b) (op 'b 'a))].

#forward id_l.
id_l : [('a : G) -> (= (op id 'a) 'a)].

inv : [G -> G].
#forward inv_l.
inv_l : [('a : G) -> (= (op (inv 'a) 'a) id)].
```

J.4 BUILT-IN EQUALITY RULES

Arithmetic and groups additionally enable `eq_refl`, `eq_symm`, and `rewrite`. These are implemented in Peano rather than declared in the theory files. Reflexivity constructs $t = t$ for a well-typed term t; symmetry obtains $u = t$ from $t = u$; rewriting substitutes equal terms in a proposition. These operations provide equality reasoning without separate domain-specific axioms for transitivity or congruence.

J.5 FULL PROOF OF CONJUNCTION COMMUTATIVITY

The example from Section 3 has statement

```
[('A : prop) -> ('B : prop) -> (and 'A 'B) -> (and 'B 'A)]
```

Introduce arbitrary propositions A and B and assume $h : A \wedge B$. Conjunction elimination gives $u : A$ and $v : B$. Conjunction introduction, with the arguments in the opposite order, then gives $B \wedge A$:

$$u := \text{and_el}(A, B, h), \qquad v := \text{and_er}(A, B, h), \qquad \text{and_i}(B, A, v, u) : B \wedge A.$$

In the recorded trace, Peano names A, B, and h as `x`, `x0`, and `x1`, respectively. After the three introductions, the context is $\{x : \mathsf{prop}, x0 : \mathsf{prop}, x1 : x \wedge x0\}$ and the goal is $x0 \wedge x$. The five available rule choices are `a and_i`, `c and_el`, `c and_er`, `a exfalso`, and `c em`.

Table 15 lists all nine actions of the proof. The prefix `c` selects a forward construction and `a` selects a backward application. Each is followed by a separate instantiation action, prefixed by `=>`. Selecting a rule alone does not change the logical context or goal. At the final action, the two required conjuncts are already in the context, so no open goals remain. Thus the trace has nine recorded actions but only three uses of conjunction rules after the introductions.

Table 15: Complete nine-action trace for conjunction commutativity. The trace was verified in Peano.

Step	Recorded action	Effect
1	`intro.`	Declare $x : \mathsf{prop}$.
2	`intro.`	Declare $x0 : \mathsf{prop}$.
3	`intro.`	Assume $x1 : x \wedge x0$.
4	`c and_el`	Select left-conjunct elimination.
5	`=> (and_el x x0 x1).`	Add a proof of x to the context.
6	`c and_er`	Select right-conjunct elimination.
7	`=> (and_er x x0 x1).`	Add a proof of $x0$ to the context.
8	`a and_i`	Select conjunction introduction.
9	`=> .`	Close $x0 \wedge x$ using the two extracted facts.