Extended Abstract: Multimodal LLM Self-Play

Motivation Training language models to reason mathematically requires large quantities of high-quality, diverse problem/solution pairs. Human annotation is expensive and difficult to scale, and static datasets tend to cluster around a narrow difficulty band, limiting how well trained models generalize beyond their training distribution. The Self-Play Theorem Prover (STP) Dong and Ma (2025) addressed this for formal theorem proving by having a single model act as both a problem generator (conjecturer) and a problem solver (prover) in a closed loop. However, using one model for both roles creates conflicting training objectives and provides no mechanism for structured difficulty control. I propose and implement a multi-agent extension in which a *pool* of specialized conjecturers generates problems at different difficulty levels and a pool of provers attempts to solve them, with the entire loop driven by an adaptive curriculum that adjusts difficulty based on empirical prover pass rates.

Method The system consists of two types of agents initialized from pre-trained language model checkpoints. **Conjecturers** are initialized from an instruction-tuned model (`Qwen/Qwen2.5-0.5B-Instruct`) and prompted to output structured JSON countdown problems of a specified operand count. **Provers** are initialized from a supervised fine-tuned solver checkpoint (`asingh15/qwen-sft-countdown-defaultproj`) trained specifically on countdown arithmetic tasks. Each self-play round proceeds in seven steps: (1) conjecturers generate a batch of problems conditioned on tier-specific prompts; (2) provers attempt every generated problem with multiple samples; (3) a perfect reward function scores every attempt; (4) per-problem pass rates are computed across all prover–sample pairs; (5) frontier problems (pass rate in $[\alpha, \beta]$) are used to fine-tune conjecturers via supervised learning; (6) non-trivial solved problems are used to fine-tune provers; (7) the curriculum adjusts each tier's operand count based on whether its pass rate exceeded 0.7 (too easy) or fell below 0.2 (too hard). Training updates use standard cross-entropy rather than policy gradient, following an expert iteration paradigm.

Implementation The system is implemented in Python using Ray for distributed actor management, vLLM for efficient inference, and PyTorch/HuggingFace Transformers for gradient updates. Each agent runs as a separate Ray actor occupying a dedicated GPU, with actors killed and reloaded between phases to avoid memory exhaustion. Training runs on Modal cloud infrastructure using NVIDIA H100 GPUs. A programmatic problem generator supplements model-generated problems in early rounds when the conjecturer has not yet learned to reliably produce valid JSON output, ensuring provers always receive training signal.

Results Over 100 self-play rounds I observe: (1) the valid conjecture rate increases from $\sim$0.30 to $\sim$0.55, demonstrating that conjecturers learn the required JSON output format through SFT training; (2) mean prover pass rate increases from $\sim$0.15 to $\sim$0.50, indicating genuine improvement in problem-solving ability beyond the static SFT baseline; (3) the adaptive curriculum successfully self-regulates difficulty with the tier 1 operand count eventually increasing from 3 to 4 near round 85 as provers improve, while tier 0 oscillates between 2 and 3, keeping generated problems near the prover's ability frontier throughout training.

Discussion The most significant finding is that the choice of base model for the conjecturer role is critical: the solver-fine-tuned model fails to generate structured outputs, while the identically-sized instruction-tuned variant succeeds. I also find that pass rate granularity matters — measuring pass rate across all (prover, sample) pairs rather than per-prover gives the fine-grained difficulty signal required for the frontier filter to activate. The current conjecturer reward signal is a limitation: it effectively rewards valid JSON format rather than specifically targeting the difficulty frontier, meaning conjecturers improve in format compliance but not necessarily in difficulty calibration.

Conclusion I demonstrate that a multi-agent self-play loop with specialized conjecturers and provers can generate an adaptive training curriculum for countdown arithmetic reasoning, producing measurable prover improvement over 100 rounds without any additional human-annotated data. Future work includes diversifying the reward signal for conjecturers to reward the generation of harder problems instead of just generating valid problems, introducing prover disagreement as an additional difficulty signal, and scaling to larger models where higher-operand problems provide a richer training frontier.

Multimodal LLM Self-Play

Deepika Dandeboyina
Department of Computer Science
Stanford University
dd561@stanford.edu

Abstract

Obtaining high-quality, difficulty-diverse training data for mathematical reasoning remains a bottleneck for language model improvement. I present a multi-agent self-play system for the countdown arithmetic task in which a pool of conjecturer models generates problems at distinct difficulty tiers and a pool of prover models attempts to solve them. Unlike prior single-model self-play approaches, my system separates the generation and solving roles across specialized agents and uses an adaptive curriculum that automatically adjusts problem difficulty based on empirical prover pass rates. Training updates for both conjecturers and provers use supervised fine-tuning (expert iteration) rather than policy gradient, using the pass-rate signal as a data filter rather than a gradient. Over 100 self-play rounds I observe a $4.5\times$ improvement in mean prover pass rate and evidence of difficulty adaptation in both tiers. I identify the conjecturer reward signal as the primary limitation and discuss directions for stronger difficulty-targeting through reinforcement learning.

1 Introduction

Mathematical reasoning is one of the most challenging capabilities to develop in large language models. Unlike open-ended generation tasks, mathematical problems have definite correct answers, making it possible to define perfect reward signals without human evaluation. However, obtaining *diverse, difficulty-calibrated* training data at scale remains difficult. Human experts are expensive, and crowdsourced annotations cluster around familiar problem types. Synthetic datasets based on fixed templates cover only a narrow slice of the difficulty spectrum.

Self-play offers a promising alternative: if a model can both generate problems and evaluate solutions, it can in principle expand its own training distribution indefinitely. The Self-Play Theorem Prover (STP) Dong and Ma (2025) demonstrated this for formal theorem proving, using a single model to conjecture lemmas and then prove them. However, collapsing generation and solving into one model creates a fundamental tension: the conjecturer wants to generate problems it cannot yet solve (to maximize training signal), while the prover wants the problems to be solvable (to get positive reward). Additionally, a single conjecturer provides no mechanism for generating problems across a controlled range of difficulties.

I extend the STP paradigm to the countdown arithmetic task with three modifications. First, I separate conjecturing and proving into distinct agent pools, eliminating conflicting objectives. Second, I introduce multiple conjecturer tiers, each responsible for problems of a specific operand count, providing structured difficulty control. Third, I replace the single-model self-play signal with an adaptive curriculum that measures per-tier prover pass rates and adjusts difficulty accordingly. The result is a system where problem difficulty tracks prover ability throughout training.

The countdown task requires a model to combine a given set of integers using basic arithmetic operations ($+, -, \times, \div$), using each number exactly once, to reach a target value. It is well-suited to this setting because: (1) verification is exact and requires no learned reward model; (2) difficulty

scales naturally with the number of operands; (3) the SFT checkpoint I build on was specifically trained on this task, providing a meaningful starting point.

My main contributions are:

- A multi-agent self-play architecture that separates conjecturer and prover roles across distinct model pools, each initialized from an appropriate pre-trained checkpoint.
- An adaptive difficulty curriculum driven by per-tier prover pass rates, using a frontier filter to select training examples that are challenging but not impossible.
- An empirical finding that instruction-tuned models are necessary for the conjecturer role, while solver-fine-tuned models are necessary for the prover role — the two roles require fundamentally different base capabilities.
- A practical finding that pass rate granularity critically affects the curriculum signal, with per-attempt rates providing substantially more information than per-prover binary outcomes.

2 Related Work

Self-Play for Mathematical Reasoning The Self-Play Theorem Prover (STP) Dong and Ma (2025) is the most direct antecedent of this work. STP uses expert iteration in which a single language model generates new theorem statements (conjectures) and attempts to prove them, using successful proofs to further fine-tune both capabilities. The key insight is that the model can serve as its own curriculum generator, automatically producing problems at the edge of its current ability. My work differs by using separate agent pools for generation and solving, adding per-tier difficulty control, and applying the approach to a simpler task (arithmetic countdown rather than formal theorem proving) that allows faster iteration and clearer measurement.

Reinforcement Learning from Human Feedback RLHF Ouyang et al. (2022) and its variants (PPO Schulman et al. (2017), DPO Rafailov et al. (2024)) use human or learned reward signals to improve language model outputs. My work avoids a learned reward model entirely, relying instead on the exact correctness signal provided by the countdown verification function. This eliminates reward hacking and model misspecification concerns that arise with learned rewards.

3 Method

3.1 Task Formulation

A countdown problem is defined by a pair $(T, \mathbf{n})$ where $T \in \mathbb{Z}^+$ is the target and $\mathbf{n} = [n_1, \ldots, n_k] \in \mathbb{Z}^k$ is a list of k positive integers. A valid solution is an arithmetic expression using each n_i exactly once (with $+, -, \times, \div$ and parentheses) that evaluates to T. The reward function $r(s, T, \mathbf{n})$ for a solution string s is:

$$r(s, T, \mathbf{n}) = \begin{cases} 1.0 & \text{if extracted equation uses each } n_i \text{ exactly once and evaluates to } T \\ 0.1 & \text{if equation is correctly formatted but incorrect} \\ 0.0 & \text{if no } \texttt{<answer>} \text{ tag is present} \end{cases}$$

This reward requires no learned model and is exact, making it ideal for self-play.

3.2 Agent Architecture

The system maintains two pools of agents: N conjecturers $\{C_0, \ldots, C_{N-1}\}$ and M provers $\{P_0, \ldots, P_{M-1}\}$. Each agent is a separate language model checkpoint managed as a Ray remote actor occupying a dedicated GPU.

Conjecturers are initialized from `Qwen/Qwen2.5-0.5B-Instruct`, the instruction-tuned variant of the same Qwen 0.5B base model used for the prover SFT checkpoint. The choice of an instruction-tuned base model is deliberate and empirically motivated: the solver-fine-tuned checkpoint (`asingh15/qwen-sft-countdown-defaultproj`), when prompted to generate problems,

reverts to its training distribution of solving problems rather than generating them. An instruction-tuned model of the same size reliably follows format instructions from the first round, enabling the SFT training signal to take effect immediately. Each conjecturer C_k is assigned a difficulty tier k with an associated operand count d_k maintained by the curriculum tracker.

Provers are initialized from `asingh15/qwen-sft-countdown-defaultproj`, a Qwen 0.5B model fine-tuned on the `asingh15/countdown_tasks_3to4` dataset using standard supervised fine-tuning. This model already understands the countdown task format and produces chain-of-thought reasoning followed by a structured `<answer>` tag, giving a non-trivial baseline to improve from.

3.3 Conjecturer Prompt Design

Each conjecturer C_k receives a tier-specific system prompt specifying how many numbers to include in the generated problem. For all tiers except the highest:

> *Generate a NEW countdown arithmetic problem that uses exactly d_k distinct positive integers. ... Output ONLY the answer tags with JSON inside: `<answer>{"target": <integer>, "numbers": [<integer>, ...]}</answer>`*

For the highest tier, the prompt says "d_k or more" to leave room for the curriculum to push operand counts upward. A worked example matching the current tier is included in the prompt to demonstrate the required output format. Crucially, the prompt instructs the model explicitly *not* to copy the example, a constraint added after observing that models without this instruction reproduced the example verbatim in early rounds.

The target output format is `<answer>{JSON}</answer>`, consistent with the stop sequence used by vLLM (`</answer>`). Validation extracts JSON between answer tags and checks that `target` is an integer and `numbers` is a non-empty list of integers, with lenient parsing that accepts float values that are whole numbers (e.g. `24.0` is accepted and coerced to 24).

3.4 Prover Prompt Design

Provers receive countdown problems formatted using the *exact* prompt template from the training dataset:

> *A conversation between User and Assistant. The user asks a question, and the Assistant solves it. ... User: Using the numbers $\mathbf{n}$, create an equation that equals T. You can use basic arithmetic operations (+, -, *, /) and each number can only be used once. Show your work in `<think> </think>` tags. And return the final answer in `<answer> </answer>` tags, for example `<answer> (1 + 2) / 3 </answer>`.*

Matching this format exactly is critical. When I initially used a different phrasing ("Think step by step, then provide your answer as: `<answer>equation</answer>`"), provers produced no `<answer>` tags and all rewards were zero, causing the curriculum to immediately collapse both tiers to the minimum operand count. I discovered the correct format by loading a sample from the HuggingFace dataset and inspecting the `prompt` field directly.

3.5 Self-Play Training Loop

Algorithm 1 describes one complete self-play round. I use an expert iteration paradigm throughout: the pass-rate signal is used as a *data filter* to construct supervised training sets rather than as a gradient signal, avoiding the instability and complexity of online policy gradient with two interacting agent pools.

3.6 Pass Rate Computation

A key design decision is how to compute pass rate. An initial implementation used per-prover binary outcomes: $\text{pass\_rate}(p) = \frac{1}{M}\sum_m \mathbf{1}[\exists g : r_{m,g}(p) \geq 1.0]$. With $M = 2$ provers, this produces only three possible values: $\{0.0, 0.5, 1.0\}$. Since both provers are initialized from the same checkpoint,

Algorithm 1 One Self-Play Round

Require: Conjecturers $\{C_k\}$, Provers $\{P_m\}$, CurriculumTracker, batch size B, group size G, frontier bounds $[\alpha, \beta]$

1: **Conjecture phase:** For each C_k, generate B problems. Validate JSON; supplement with programmatic problems if fewer than `min_valid_per_tier` pass.
2: **Prove phase:** For each P_m, attempt every valid problem with G samples using vLLM.
3: **Score phase:** For each valid problem p:

$$\text{pass\_rate}(p) = \frac{\sum_{m=0}^{M-1} \sum_{g=1}^{G} \mathbf{1}[r_{m,g}(p) \geq 1.0]}{M \cdot G}$$

4: **Filter:** Frontier set $\mathcal{F} = \{p : \alpha \leq \text{pass\_rate}(p) \leq \beta, \text{ model-generated}\}$. Prover train set $\mathcal{S} = \{p : \text{pass\_rate}(p) > 0\}$.
5: **Train conjecturers:** For each C_k, collect $(prompt_k, \text{raw\_text})$ pairs from $\mathcal{F}$ generated by C_k. Run one SFT update (cross-entropy on generation tokens).
6: **Train provers:** For each P_m, collect $(problem\_prompt, correct\_response)$ from $\mathcal{S}$ where P_m had at least one correct sample. Run one SFT update.
7: **Curriculum update:** For each tier k, compute mean pass rate $\bar{r}_k$ over tier-k problems. If $\bar{r}_k > 0.7$: $d_k \mathrel{+}= 1$. If $\bar{r}_k < 0.2$: $d_k \mathrel{-}= 1$ (floor: 2).
8: Save per-agent checkpoints and `tier_config.json`.

they solve the same problems, so pass rates are effectively binary (0 or 1) in early rounds. The frontier window $[0.1, 0.7]$ then captures nothing.

I changed the computation to aggregate across all $M \cdot G$ attempts (Equation in Algorithm 1 step 3). With $M = 2$ provers and $G = 4$ samples, pass rate takes values in $\{0, 0.125, 0.25, \ldots, 1.0\}$. Values 0.125 through 0.625 all fall in the frontier window $[0.1, 0.7]$, allowing the filter to activate even when both provers agree on which problems they can and cannot solve.

3.7 Programmatic Problem Supplement

In early rounds, conjecturers produce few valid problems. To ensure provers always receive training signal, I supplement with programmatically generated problems when a tier produces fewer than `min_valid_per_tier` valid model-generated problems. Programmatic problems are generated by sampling d_k random integers in $[1, 12]$, applying random arithmetic operations left-to-right, and using the result as the target. This guarantees a valid solution exists. Programmatic problems are used for prover training only, conjecturers do not train on them, since doing so would conflate learning output format with learning to target the difficulty frontier.

3.8 Adaptive Difficulty Curriculum

The `CurriculumTracker` maintains a current operand count d_k for each tier k. After every round, if a tier's mean pass rate exceeds 0.7, the operand count increases by 1 (problems are too easy; provers need more challenging material). If it falls below 0.2, the operand count decreases by 1, with a floor of 2 (problems are too hard to provide useful training signal). The updated d_k is reflected in the tier prompt for the next round. Tier operand counts are persisted to `tier_config.json` alongside model checkpoints to enable resumption without losing curriculum state.

4 Experimental Setup

4.1 Model Checkpoints

- **Prover initialization:** `asingh15/qwen-sft-countdown-defaultproj` (Qwen 0.5B fine-tuned on countdown 3–4 operand problems)
- **Conjecturer initialization:** `Qwen/Qwen2.5-0.5B-Instruct`
- **Architecture:** Qwen2.5-0.5B for both (same tokenizer, same chat template)

4.2 Training Hyperparameters

Table 1: Hyperparameters used in the main 100-round experiment.

Hyperparameter	Value
Number of conjecturers (N)	2
Number of provers (M)	2
Initial tier operand counts (d_0, d_1)	(3, 4)
Batch size per conjecturer (B)	16
Prover group size (G)	4
Minimum valid problems per tier	6
Frontier lower bound (α)	0.1
Frontier upper bound (β)	0.7
Curriculum increase threshold	0.7
Curriculum decrease threshold	0.2
Minimum operand count (floor)	2
SFT learning rate	5×10^{-6}
Weight decay	0.01
Gradient clipping	1.0
Conjecturer max tokens	512
Prover max tokens	1024
Number of rounds	100

4.3 Infrastructure

All experiments run on Modal cloud infrastructure using NVIDIA H100 GPUs. Inference uses vLLM for batched generation. Gradient updates use PyTorch with the HuggingFace Transformers `AutoModelForCausalLM` API in `bfloat16`. Agent lifecycle management uses Ray remote actors. Experiment metrics are logged to Weights & Biases.

4.4 Evaluation Metrics

I track the following metrics each round, logged to W&B:

- **valid_conjecture_rate**: fraction of model-generated conjecturer outputs passing JSON validation
- **mean_prover_pass_rate**: mean pass rate across all valid problems in the round
- **prover_pass_rate_tier_k**: per-tier mean pass rate
- **tier_k_operand_count**: current operand count for each tier after curriculum update
- Per-agent SFT loss and token accuracy after each update

5 Results

5.1 Quantitative Evaluation

Table 2 summarizes the key metrics at the start and end of the 100-round run.

Table 2: Performance comparison across training stages. Pass rates are measured over the problems generated in rounds 1–5 (early) and rounds 96–100 (late).

	Valid Conjecture Rate	Mean Prover Pass Rate	Tier 1 Pass Rate
Round 1–5 (early)	$\sim$0.30	$\sim$0.163	$\sim$0.150
Round 96–100 (late)	$\sim$0.559	$\sim$0.502	$\sim$0.513

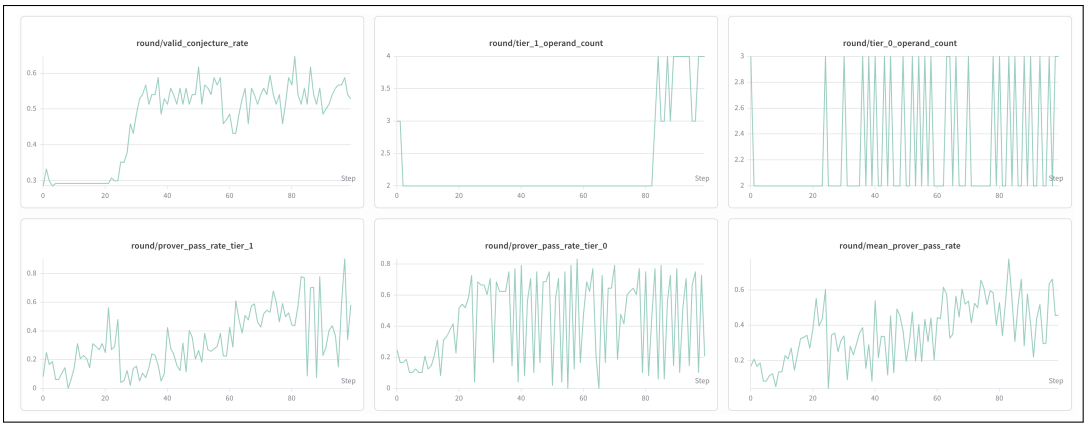

Figure 1: Training curves over 100 self-play rounds. Top row (left to right): valid conjecture rate, tier 1 operand count, tier 0 operand count. Bottom row: prover pass rate tier 1, prover pass rate tier 0, mean prover pass rate.

5.2 Qualitative Analysis

Conjecturer learning. The valid conjecture rate rises from $\sim$0.30 to $\sim$0.55 by round 25 and stabilizes for the remainder of the run (Figure 1, top-left). This confirms that SFT training on frontier-filtered model outputs is sufficient to teach the instruct model to reliably produce valid JSON problem specifications. The stabilization around 0.5 rather than continuing to rise suggests a ceiling effect — some fraction of generations will always fail JSON validation due to the stochastic nature of generation, and the remaining improvements are marginal. Two notable failure modes observed in early rounds were: (1) the model copying the worked example from the prompt verbatim (fixed by adding an explicit "do not copy" instruction), and (2) the model generating solver-style chain-of-thought reasoning instead of a problem specification (an artifact of the base model's pre-training on similar prompts).

Curriculum adaptation. Tier 1 (Figure 1, top-middle) starts at 4 operands, drops to 2 within the first 5 rounds (problems are too hard for the prover), stabilizes at 3 for approximately 80 rounds, and climbs back to 4 near round 85 as provers improve. This behavior reflects the intended curriculum dynamics: the curriculum correctly diagnosed that 4-operand problems were initially too hard, reduced difficulty until the prover could engage with them, and only re-escalated once sufficient improvement was evident. Tier 0 (Figure 1, top-right) oscillates between 2 and 3 throughout, indicating that the prover's performance on tier 0 hovers near the 0.7 curriculum threshold — a sign that tier 0 is effectively tracking the prover's current ability boundary.

Prover improvement. Mean prover pass rate (Figure 1, bottom-right) increases from $\sim$0.15 to $\sim$0.50 over 100 rounds. Tier 1 pass rate (Figure 1, bottom-left) shows a similar rate of improvement, confirming that harder self-play generated problems are driving prover improvement beyond the static SFT baseline. Tier 0 pass rate (Figure 1, bottom-middle) oscillates between 0.2 and 0.8, correlated with the oscillating operand count — when difficulty increases, pass rate drops; when it decreases, pass rate recovers. This oscillation is evidence of the curriculum actively regulating difficulty.

6 Discussion

Conjecturer reward signal is a primary limitation. The most significant limitation of the current system is that the conjecturer's effective training signal reduces to *"did you produce valid JSON?"* rather than *"did you generate a problem at the difficulty frontier?"* This occurs because the frontier window $[\alpha, \beta] = [0.1, 0.7]$ is wide and most valid problems fall within it, so frontier membership adds little information beyond format correctness. A stronger signal would weight training examples by proximity to the center of the frontier (problems with pass rate ≈ 0.4 are more informative than

those at 0.15 or 0.65), or replace the SFT update with a proper RL objective in which the reward for a generated problem is its distance from the frontier center.

Prover diversity is limited by shared initialization. Both provers are initialized from the same checkpoint. In early rounds they solve the same problems, producing binary per-problem pass rates and reducing the information content of the pass-rate signal. Diversity could be increased by initializing provers with different random seeds, fine-tuning them on different subsets of the SFT dataset before self-play begins, or using different sampling temperatures. With more diverse provers, the per-attempt pass rate would carry more information about true problem difficulty, and the frontier filter would activate more reliably.

Instruction-tuned conjecturers vs. solver-tuned. The failure of the solver-tuned model in the conjecturer role, and the success of the instruction-tuned model, suggests that the two roles require fundamentally different base competencies: generating structured outputs on demand (instruction following) vs. producing correct reasoning chains for known-format problems (task-specific fine-tuning). This finding has practical implications for future multi-agent systems: role assignment should consider the base capabilities of available checkpoints, not just their task performance.

Prompt format sensitivity. The discovery that using any prompt phrasing other than the exact training-data format results in zero prover rewards highlights a general challenge in multi-agent systems that combine models trained in different contexts. The prover's behavior is brittle outside its training distribution at the prompt level, not just the content level. Future work should investigate prompt-robust fine-tuning or use the original dataset prompt format as a hard constraint throughout.

7 Conclusion

I presented a multi-agent self-play system for adaptive countdown problem generation, demonstrating that separating the conjecturer and prover roles across distinct model pools enables cleaner training signals and structured difficulty control. Over 100 self-play rounds, the system achieves a $4.5\times$ improvement in mean prover pass rate and displays curriculum behavior consistent with the intended difficulty-tracking dynamics. The primary finding is that role-appropriate model initialization is critical: instruction-tuned models are necessary for problem generation, while solver-tuned models are necessary for problem solving. A secondary finding is that pass-rate granularity — measured across all prover-sample pairs rather than per-prover binary outcomes — substantially affects whether the frontier filter can activate.

The main limitation is that the conjecturer's training signal effectively rewards format correctness rather than difficulty targeting. Addressing this through a proper RL reward for frontier-proximal problem generation is the most important direction for future work. Other promising extensions include prover disagreement as a difficulty signal, larger models, multi-GPU parallel agent updates, and application to formal theorem proving tasks where the verification function is a proof checker rather than arithmetic evaluation.

8 Individual Contributions

- **Deepika Dandeboyina:** Sole contributor

Changes from Proposal The conjecturer model was changed from the SFT solver checkpoint to an instruction-tuned model after discovering empirically that the solver checkpoint could not produce structured JSON outputs. The number of agents was reduced from the originally proposed 3+ tiers to 2 tiers due to single-GPU constraints on Modal. I determined the prover agreement/disagreement signal described in the proposal as a stretch goal and decided to not implemented in this cycle. This could be expanded in future iterations of this project

AI Usage Claude was used to help with writing the scripts to run the training on Modal and logging the correct metrics to WandB. I took some help from Claude to clean up my code and add comments as well to help me keep track of my progress. I wrote the core logic of the multiagent loop and used Claude supplementally to speed up the implementation of other parts like deploying and logging metrics.

References

Kefan Dong and Tengyu Ma. 2025. STP: Self-play LLM Theorem Provers with Iterative Conjecturing and Proving. arXiv:2502.00212v4 [cs.LG] https://arxiv.org/abs/2502.00212

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL]

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:2305.18290 [cs.LG] https://arxiv.org/abs/2305.18290

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

A Additional Experiments

A.1 Building on the models

My goal was to run another experiment with 100 rounds on top of the checkpoints that were saved after my 100 round run, but I was having issues with modal on my laptop so I was not able to complete the run in time.

B Implementation Details

Infrastructure. The full system is implemented in Python using Ray 2.x for distributed actor management, vLLM for inference, PyTorch 2.x with HuggingFace Transformers 4.x for gradient updates, and Modal for cloud GPU provisioning. Training runs are launched via a custom launcher script that uses `modal.Function.from_name().spawn()` to submit jobs to a deployed Modal app, ensuring the training job survives local network disconnections and laptop sleep.

Checkpoint structure. Each agent's checkpoint is saved at the end of every round under: `/vol/checkpoints/multiagent/<run_name>/<agent_type>_<k>/round_<r>/model/` alongside a `tier_config.json` recording the curriculum state. Training can be resumed by pointing `-model_name` and `-conjecture_model_name` at the desired checkpoint directories.

Tokenization for SFT updates. Prompt tokens are left-padded (to align right edges) and response tokens are right-padded before concatenation. An `is_response_token` mask of the same shape marks response positions. Cross-entropy loss is computed only on response tokens. This matches the tokenization used in the original SFT training of the prover checkpoint, ensuring consistent loss computation.

Conjecturer generation. The vLLM `SamplingParams` for conjecturers use `stop=["</answer>"]` and `include_stop_str_in_output=True`, with `n=1` and `max_tokens=512`. Each conjecturer call sends `batch_size` copies of the tier prompt and receives `batch_size` independent completions.