DISCOVERING MULTI-AGENT LEARNING ALGORITHMS WITH LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Much of the advancement in Multi-Agent Reinforcement Learning (MARL) for imperfect-information games has historically depended on the manual, iterative refinement of algorithmic baselines. Recently, evolutionary coding agents powered by Large Language Models (LLMs) have emerged as powerful tools to automate this discovery process. In this work, we deploy one of such agentic frameworks, AlphaEvolve, to navigate the design spaces of two distinct game-theoretic paradigms: counterfactual regret minimization (CFR) and policy-space response oracles (PSRO). This automated search yielded two algorithms: Volatility-Adaptive Discounted (VAD-) CFR and Smoothed Hybrid Optimistic Regret (SHOR-) PSRO, which are consistently competitive with state-of-the-art human-designed baselines across an 18-game evaluation suite spanning Poker, Goofspiel, Liar's Dice, Blotto, and Battleship variants. However, because the LLM optimizes for fitness on a specific training set, it often constructs highly synergistic, complex mechanisms tailored to those environments. Through systematic train-only ablation criteria, we distill these discoveries into two minimal closed-form solvers: Warm-started Optimistic Predictive (WOP-)CFR and Projection Matching (PM-)PSRO. Ten independent search runs confirm that the distilled cores recur while the overfitted machinery differs across runs. Distillation isolates this stable algorithmic core, achieving superior held-out performance at drastically reduced structural complexity: WOP-CFR improves on scaled variants within family and maintains parity out of family, while PM-PSRO improves in both regimes and most strongly out of family.

1 INTRODUCTION

The field of Multi-Agent Reinforcement Learning has achieved remarkable milestones in recent years, reaching superhuman performance in domains ranging from Poker to real-time strategy games via game-theoretic regret minimization (Brown & Sandholm, 2019a) and population-based league training (Vinyals et al., 2019). The practical performance of these algorithms relies heavily on structural choices—such as how regret is discounted over time or how an equilibrium solution concept is derived. Historically, refining these choices has been a manual endeavor, relying on intuition and trial-and-error to navigate a vast combinatorial space of potential update rules, often defaulting to mathematically tractable heuristics (e.g., linear averaging or fixed discounting) that may not be optimal.

In this work, we propose to overcome this limitation by automating the design process itself with Large Language Models (LLMs). We apply AlphaEvolve (Novikov et al., 2025), a distributed evolutionary system powered by LLMs, to the domain of multi-agent learning. Unlike traditional hyperparameter optimization or genetic programming, AlphaEvolve leverages the code-generation capabilities of LLMs to perform semantic evolution. By treating the algorithm's source code as the genome, the system uses LLMs to act as intelligent genetic operators—performing mutation to rewrite logic, introduce new control flows, and inject novel symbolic operations. This allows to search beyond simple parameter tuning and discover new non-intuitive mechanisms for equilibrium finding. This automated search initially yielded two highly performant variants: Volatility-Adaptive Discounted (VAD-)CFR and Smoothed Hybrid Optimistic Regret (SHOR-)PSRO, both consistently competitive with state-of-the-art human-designed baselines across an 18-game suite covering five game families.

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

However, our analysis reveals that evolutionary agents can layer complex, specialized logic over foundational discoveries to maximize fitness on specific training games. For example, the namesake volatility-tracking in VAD-CFR operates in tight conjunction with regret boosting; while highly effective on training games, this complexity is unnecessary for generalization. To isolate the fundamental principles driving performance, we conduct systematic train-test split ablation studies. By distilling the algorithm down to its most important core components, we introduce two minimal, mathematically pure solvers: Warm-started Optimistic Predictive (WOP-)CFR and Projection Matching (PM-)PSRO. These distilled variants achieve superior generalization while reducing the heuristic complexity present in the raw discoveries.

Our main contributions are threefold: (1) We introduce modular, evolvable code decompositions for imperfect-information game solving (CFR and PSRO) that subsume established human-designed families as special cases, enabling evolutionary LLM search over symbolic update rules. (2) We demonstrate a general *propose-and-distill* methodology: evolutionary LLM search systematically discovers a stable mathematical core wrapped in training-overfitted heuristics (a phenomenon we verify across 10 independent trajectories), which we distill via train-only ablation criteria ($s_{\text{train}} < -2$ for CFR, $s_{\text{train}} < -0.05$ for PSRO). (3) We contribute WOP-CFR and PM-PSRO as minimal, closed-form equilibrium solvers that outperform the raw LLM discoveries on held-out games while linking PSRO meta-solvers to classical projection dynamics.

2 Game Theoretic Preliminaries

We formulate our problem within the framework of **Extensive-Form Games (EFGs)** with imperfect information, which models sequential interactions involving multiple agents and hidden information.

2.1 Extensive-Form Games and Exploitability

An N-player extensive-form game is $\Gamma = \langle \mathcal{N}, \mathcal{H}, \mathcal{Z}, \mathcal{A}, u, \mathcal{I} \rangle$, where $\mathcal{N} = \{1, \ldots, N\}$ denotes the set of players (Shoham & Leyton-Brown, 2008). $\mathcal{H}$ is the set of all possible histories (sequences of actions), where $\mathcal{Z} \subseteq \mathcal{H}$ represents terminal histories. For $i \in \mathcal{N}$, let $\mathcal{H}_i \subseteq H$ represent the subset of histories where player i acts. For any non-terminal history h, $\mathcal{A}(h)$ is the set of legal actions. The utility function $u_i : \mathcal{Z} \to \mathbb{R}$ assigns a payoff to player i at a terminal node. Crucially, imperfect information is modeled via **Information Sets** $I \in \mathcal{I}_i$. Specifically, $\mathcal{I}_i$ partitions the histories $\mathcal{H}_i$ such that player i cannot distinguish between histories $h, h' \in I$ (e.g., due to hidden cards). It is required that $\mathcal{A}(h) = \mathcal{A}(h')$ for all $h, h' \in I$, so for simplicity we denote the legal actions at I, $\mathcal{A}(I)$. A **strategy** (or policy) $\sigma_i(I)$ assigns a probability distribution over actions $a \in \mathcal{A}(I)$ for each information set. A strategy profile $\sigma = (\sigma_1, \ldots, \sigma_N)$ determines the expected utility $u_i(\sigma)$. A **Nash Equilibrium (NE)** is a strategy profile σ^* such that no player can increase their utility by deviating unilaterally: $u_i(\sigma_i^*, \sigma_{-i}^*) \geq u_i(\sigma_i', \sigma_{-i}^*) \quad \forall \sigma_i', \forall i \in \mathcal{N}$. To measure the performance of our evolved algorithms, we use **Exploitability**. The exploitability of a strategy profile σ is the average of the incentives for players to deviate to their Best Response (BR): $\text{Expl}(\sigma) = \frac{1}{N} \sum_{i \in \mathcal{N}} \left(\max_{\sigma_i'} u_i(\sigma_i', \sigma_{-i}) - u_i(\sigma) \right)$. In small or medium-sized games, we can compute this value exactly by traversing the full game tree.

2.2 Counterfactual Regret Minimization (CFR)

CFR is an iterative algorithm that minimizes **counterfactual regret** (Zinkevich et al., 2007). It decomposes the global regret minimization problem into independent local regret minimization problems at each information set. Let $\pi^\sigma(h)$ be the probability of reaching history h under strategy profile σ. We define $\pi_{-i}^\sigma(h)$ as the contribution of all players *except* i (including chance) to reaching h. The **counterfactual value** of player i reaching information set I and playing action a is the expected utility given that player i played to reach I: $v_i(\sigma, I, a) = \sum_{h \in I} \pi_{-i}^\sigma(h) \sum_{z \in \mathcal{Z}, h \sqsubset z} \pi^\sigma(z \mid h, a) u_i(z)$. The **instantaneous counterfactual regret** for not playing action a at iteration t is the difference between the counterfactual value of action a and the expected value at I: $r_i^t(I, a) = v_i(\sigma^t, I, a) - \sum_{a' \in \mathcal{A}(I)} \sigma_i^t(I, a') v_i(\sigma^t, I, a')$. Standard CFR accumulates these values linearly over iterations T: $R_i^T(I, a) = \sum_{t=1}^T r_i^t(I, a)$. The current policy σ_i^{t+1} is derived from accumulated regret, typically using **Regret Matching (RM)**, which assigns probabilities proportional to positive regret: $\sigma_i^{t+1}(I, a) = \max(R_i^t(I, a), 0) / \sum_{a'} \max(R_i^t(I, a'), 0)$.

The strategy σ_i^t at any single iteration may not converge to NE. Instead, CFR outputs the **average strategy** $\bar{\sigma}_i^T$, computed by weighting the iteration strategy σ_i^t by the player's contribution to the reach probability $\pi_i^{\sigma^t}(I)$. Several variants modify these update rules. For example, **CFR+** (Tammelin, 2014) replaces the regret accumulation R_i^t with floor bounding $\max(R_i^{t-1} + r_i^t, 0)$ and uses linear averaging weights ($w_t = t$) for the average policy. Our work uses AlphaEvolve to search for optimal variations of these accumulation and derivation functions.

2.3 Policy Space Response Oracles (PSRO)

PSRO (Lanctot et al., 2017; Bighashdel et al., 2024) acts as a meta-solver that generalizes the Double Oracle algorithm (McMahan et al., 2003). It operates on a higher level of abstraction called the **Meta-Game** (or Empirical Game (Wellman et al., 2025)). PSRO maintains a population of policies $\Pi_i = \{\sigma_i^1, \ldots, \sigma_i^k\}$ for each player. The meta-game is represented by a payoff tensor M, where entries $M_i^{j_1 \cdots j_N} = u_i(\sigma_1^{j_1}, \ldots, \sigma_N^{j_N})$ correspond to the expected utility of player i when policies from the population are pitted against each other. At each epoch k, the algorithm performs three steps:

1. **Meta-Strategy Solver (MSS) at training-time:** A solver computes a meta-strategy ϕ_i (a probability distribution over the population Π_i). Common solvers include **Uniform** $\phi_i^{j_i} = 1/|\Pi_i|$ and **Nash** ϕ which is a Nash Equilibrium of the current meta-game M.
2. **Oracle (Best Response):** A new policy σ_i^{k+1} is trained via Reinforcement Learning (or exact solving) to be a Best Response to the opponent's meta-strategy: $\sigma_i^{k+1} \in \arg\max_{\sigma_i} \mathbb{E}_{\sigma_{-i} \sim \phi_{-i}} [u_i(\sigma_i, \sigma_{-i})]$. In this work, we utilize an exact oracle that computes the optimal best response to the meta-strategy, isolating the performance of the meta-solver from the variance of reinforcement learning.
3. **Expansion:** The new policy is added to the population $\Pi_i \leftarrow \Pi_i \cup \{\sigma_i^{k+1}\}$, and the payoff tensor M is updated. In this work, we calculate the exact expected payoff value for each entry $M_i^{j_1 \cdots j_N}$, thereby eliminating the stochastic noise associated with Monte Carlo sampling.

To quantify performance, exploitability must be calculated using a meta-strategy distribution over the current population. Standard PSRO typically relies on a single, fixed meta-solver (e.g., Uniform or Nash) for both the training-time oracle generation and the evaluation-time metric calculation. By exposing both the training and evaluation solvers as completely separate, evolvable classes, we allow AlphaEvolve to explore whether unified mathematical frameworks or decoupled, asymmetric schedules yield optimal learning dynamics.

3 Automating Algorithm Discovery for Multiagent Learning via LLMs

We utilize **AlphaEvolve** (Novikov et al., 2025), an evolutionary coding agent that leverages Large Language Models (LLMs) to automate the design of multi-agent learning algorithms. We apply this framework to two distinct paradigms: Regret Minimization (CFR) and Population-based Training (PSRO).

3.1 The AlphaEvolve Framework

AlphaEvolve combines LLM code generation with evolutionary selection. We initialize a population $\mathbf{P}$ with the standard implementation of the baseline algorithm (e.g., CFR or Uniform PSRO). In each generation, a parent $A \in \mathbf{P}$ is sampled by fitness; AlphaEvolve supports multi-objective scoring, so one objective is randomly selected per round and parents are sampled in favor of high values on that objective. The parent's source code is fed to an LLM (Gemini 2.5 Pro (Comanici et al., 2025)) with the prompt *"Modify the following code to improve fitness (reduce exploitability),"* yielding a candidate A'. The candidate is executed on a set of proxy games and scored by final negative exploitability; valid candidates are added to $\mathbf{P}$ and the loop repeats.

3.2 Evolving Regret Minimization Code

To discover novel variants of Counterfactual Regret Minimization (CFR) without constraining the evolutionary agent to predefined mathematical structures, we expose the algorithm's core update loop as the search space. Rather than prompting the LLM to write an entire solver from scratch, we extract the CFR framework into three distinct, evolvable Python classes with fixed function signatures in Listing 1 in the Appendix:

1. `RegretAccumulator`: Dictates how instantaneous counterfactual regrets are processed, discounted, or bounded before being added to the historical cumulative regret.
2. `PolicyFromRegretAccumulator`: Defines the projection mechanism (e.g., Regret Matching) used to derive the current iteration's strategy from the accumulated regrets.
3. `PolicyAccumulator`: Determines how the current iteration's policy is weighted and absorbed into the final average strategy used for evaluation.

This search space is expressive enough to encompass the entire family of known CFR variants as special cases; for instance, standard CFR uses linear regret accumulation for `RegretAccumulator` and Regret Matching for `PolicyFromRegretAccumulator`.

3.3 Evolving Meta-Strategy Solvers Code

For PSRO, we evolve python classes `TrainMetaStrategySolver` for generating the mixed strategies used during the oracle training phase, and `EvalMetaStrategySolver` for computing a strategy profile over population to report exploitability. This interface (Listing 2 in the Appendix) supports the representation of all standard baselines as special cases: for example, standard double oracle algorithm solves for a Nash equilibrium in the `get_meta_strategy` method for both solver classes. By exposing the training and evaluation steps as two completely separate, evolvable Python classes, we grant the LLM the maximum degree of freedom to discover dynamic, asymmetric schedules between training and evaluation.

3.4 Meta-Training Objective

We manually select a set $\mathcal{G}_{train}$ of training games for AlphaEvolve to compute $|\mathcal{G}_{train}| + 1$ fitness scores. These are the negative exploitability $-\text{Expl}(A(g)_K)$ of the final strategy profile after K iterations for each $g \in \mathcal{G}_{train}$, as well as their average $-\frac{1}{|\mathcal{G}_{train}|}\sum_{g\in\mathcal{G}_{train}} \text{Expl}(A(g)_K)$, where $A(g)_K$ denotes the strategy produced by algorithm A on game g at iteration K. The reported algorithms are selected based on their average scores.

4 Experimental Evaluation

4.1 Experimental Setup

To test the robustness and generalizability of the algorithms we discover, we adopted a rigorous evaluation protocol involving two distinct sets of games. The algorithm's architecture and hyperparameters were developed and tuned on a *Training Set* $\mathcal{G}_{train}$ of four games. For both CFR and PSRO discoveries, we choose this set to be 3-player Kuhn Poker, 2-player Leduc Poker, 4-card Goofspiel, and 5-sided Liars Dice. Subsequently, the fixed algorithm was evaluated on a *Test Set* $\mathcal{G}_{test}$. In the main body of this paper, we present results on four larger and more difficult games as a representative subset of test games: 4-player Kuhn Poker, 3-player Leduc Poker, 5-card Goofspiel, and 6-sided Liar's Dice. The results of a full-sweep of four training games and fourteen test games are provided in Appendix 7.4. We utilized the OpenSpiel (Lanctot et al., 2019) framework for all experiments. For all experiments we use a uniform policy as the initial point. Our implementation of AlphaEvolve is backboned by Gemini 2.5 pro (Comanici et al., 2025). All evaluated algorithms are fully deterministic and metrics are computed exactly by exhaustively traversing the game-tree.

4.2 Ablation Methodology

AlphaEvolve may exhibit a tendency to over-engineer solutions, interleaving profound algorithmic discoveries with symbolic complexity. To rigorously isolate the true engines of performance from

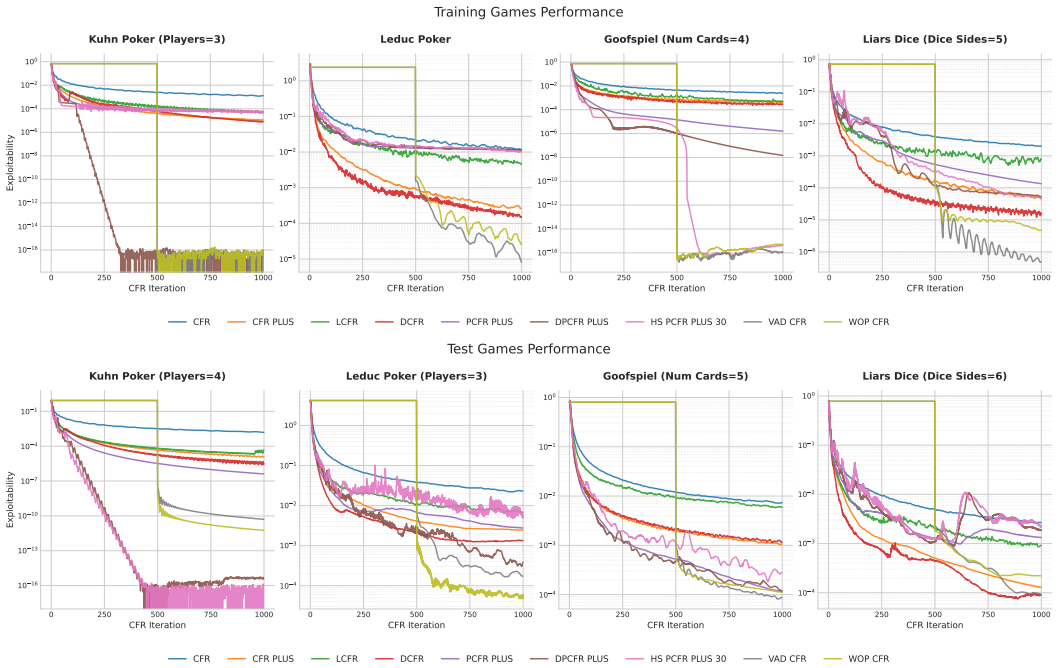

Figure 1: **CFR variants performances.**

these over-specialized heuristics, we introduce a comprehensive algorithmic auditing pipeline via train-test split ablations. We adopt the Interquartile Mean (IQM) (Agarwal et al., 2021; Oh et al., 2025), a robust statistical aggregator that is highly resistant to outlier domains. For a given set of games $\mathcal{G}$, let $\text{Expl}_{\text{raw}}^{(g)}$ and $\text{Expl}_{\text{abl}}^{(g)}$ denote the final exploitability of the raw LLM algorithm and its ablated counterpart on game $g \in \mathcal{G}$, respectively. We first define the *log-improvement score* s_g for each game: $s_g = \log_{10}\left(\text{Expl}_{\text{raw}}^{(g)}\right) - \log_{10}\left(\text{Expl}_{\text{abl}}^{(g)}\right)$. Under this formulation, a positive score $s_g > 0$ indicates that the ablated (simplified) algorithm achieves lower exploitability than the raw discovery, identifying the removed component as an over-parameterized generalization penalty. Conversely, a negative score $s_g < 0$ indicates a degradation in performance. To aggregate these scores across the diverse game tasks, the IQM computes the mean of the central 50% of the scores, systematically trimming the top 25% and bottom 25% of extreme variations. To isolate generalization effects, we report the IQM across three distinct splits: Train IQM (on $\mathcal{G}_{train}$) serves as our primary diagnostic for overfitting, Test IQM (on $\mathcal{G}_{test}$) acts as our ultimate metric for generalization, and Total IQM provides a global measure of algorithmic robustness. Because 11 of the 14 held-out games scale the training families while 3 (Colonel Blotto, Battleship 2×2, Battleship 3×2) are structurally distinct out-of-family games, we further decompose Test IQM into within-family and out-of-family splits. Distillation is governed by explicit ablation thresholds evaluated strictly on training games: a component is selected for the distilled core if and only if its individual ablation collapses training fit ($s_{\text{train}} < -2$ for CFR, $s_{\text{train}} < -0.05$ for PSRO), discarding components with marginal impact as tuning heuristics.

4.3 Experimental Evaluation: Discovering and Distilling State-of-the-Art CFR

The evolutionary search over the CFR design space yielded a remarkably sophisticated algorithm we term Volatility-Adaptive Discounted (VAD-)CFR, whose source code is provided in Listing 3. We benchmarked evolved VAD-CFR against a suite of state-of-the-art regret minimization algorithms: standard CFR (Zinkevich et al., 2007), CFR+ (Tammelin, 2014), Linear CFR (LCFR), Discounted CFR (DCFR) (Brown & Sandholm, 2019b), Predictive CFR+ (PCFR+) (Farina et al., 2021), Discounted Predictive CFR+ (DPCFR+) (Xu et al., 2024b), and a Hyperparameter Schedule-powered PCFR variant HS-PCFR+(30) (Zhang et al., 2026). Performance was quantified using exploitability (measured on a logarithmic scale) over a fixed horizon of $K = 1000$ iterations. All algorithms

Table 1: **Ablation of VAD-CFR Components and Distillation across Splits.** Results are reported as IQM Log-Improvement Scores (s_g) relative to the raw discovery. Negative scores indicate performance degradation. Test IQM is decomposed into within-family scaling (11 games) and out-of-family transfer (3 games).

Variant	Total IQM	Train IQM	Test IQM (All)	Test (Within)	Test (Out-of-Family)
WOP-CFR (Distilled)	**-0.073**	**-0.802**	**+0.119**	**+0.120**	**-0.014**
Individual VAD-CFR Ablations:					
`no_volatility`	-0.120	-0.842	-0.029	-0.009	**+0.461**
`no_boost`	-0.275	-0.954	-0.204	-0.162	-0.384
`no_prediction`	-1.584	**-7.029**	-0.905	-1.190	-1.422
`no_asym_discount`	-1.637	-4.192	-1.065	-1.108	-0.981
`no_warmstart`	**-2.733**	-5.256	**-2.003**	**-3.702**	-0.098

are evaluated deterministically using exact best responses and full game-tree traversal, rendering evaluation cross-seed variance zero by construction. We evaluate search stochasticity across 10 independent trajectories in Appendix 7.7 and itemize search compute in Appendix 7.9. We use CFR+ as the seed program. The prompt we use is shown in Listing 7 in the Appendix.

As shown in Figure 1, VAD-CFR ranks among the top performers across both the training set and the test set. For 3-player Kuhn Poker, VAD-CFR achieves significantly lower exploitability than all baselines. For Leduc Poker and 4-card Goofspiel, the algorithm maintains a steeper convergence slope compared to DPCFR+ and other state-of-the-art variants. In 5-sided Liars Dice, VAD-CFR exhibits robust performance, effectively managing the larger state space through its adaptive discounting and boosting mechanisms. Results illustrated in test games highlight its generalization. In 3-player Leduc Poker, VAD-CFR reaches exploitability levels below 10^{-3} while most baselines plateau at higher levels. In 6-sided Liar's Dice, VAD-CFR continues to match established baselines like DCFR, suggesting that its evolved mechanisms for managing regret scaling and noise are highly effective across different tasks. Overall, VAD-CFR demonstrates efficient convergence rate and generalization across a broad variety of domains: as shown in Figure 3 in the Appendix, **compared with baselines, VAD-CFR ranks in the top three on every one of the 18 games, and achieves the lowest exploitability on 15 of them**.

4.3.1 Identifying Foundational Mechanics

The raw VAD-CFR discovery (Listing 3) operates as a highly synergistic ecosystem of update rules. To isolate the true engines of performance from the LLM's over-specialized heuristics, we conducted individual feature ablations using our IQM log-improvement score (s_g). The details of all ablations are in Appendix 7.2.1.

Crucially, removing secondary features such as EWMA volatility tracking (`no_volatility`, $s_{\text{train}} = -0.842, s_{\text{test}} = -0.029$) or instantaneous regret boosting (`no_boost`, $s_{\text{train}} = -0.954, s_{\text{test}} = -0.204$) degrades training fit while having negligible effect on held-out generalization (Table 1). In fact, removing volatility tracking actually improves out-of-family transfer ($s_{\text{out}} = +0.461$), directly indicating that it is an overfitted training artifact rather than a generalizable principle. This indicates that the LLM evolved a tightly coupled system where these secondary features matter for in-distribution synergy but contribute little to generalization. However, while this synergistic complexity maximizes fitness on the training domains, our audit reveals that the true capacity for generalization rests entirely on three foundational structural shifts. Removing any of these causes a catastrophic collapse in exploitability:

- **Hard Warm-Start ($s_{\text{train}} = -5.256, s_{\text{test}} = -2.003$):** The most critical feature discovered by the LLM is a strict 500-iteration suspension of all policy averaging. By enforcing a hard delay, it ensures that early-stage, high-variance exploration noise does not pollute the final cumulative strategy profile.
- **Extreme Asymmetric Discounting ($s_{\text{train}} = -4.192, s_{\text{test}} = -1.065$):** While standard DCFR establishes the value of decoupling positive and negative regret discounting, the LLM pushes this to a mathematical extreme. It assigns a negative exponent ($\beta = -0.1$) for negative regrets. Because the discount multiplier is driven by a negative exponent, it rapidly decays toward zero,

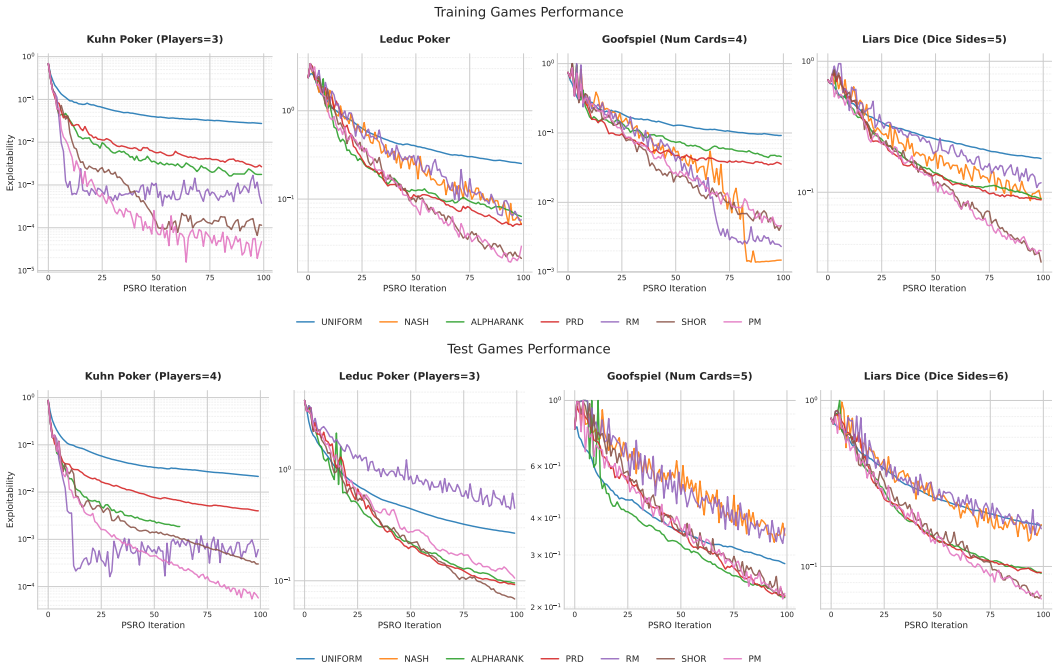

Figure 2: **PSRO variants performances.**

acting as an almost instantaneous "forget-gate" that prevents regret lock-in significantly faster than human-tuned DCFR baselines.

- **Optimistic Prediction** ($s_{\textbf{train}} = -7.029, s_{\textbf{test}} = -0.905$)**:** VAD-CFR abandons standard linear Regret Matching, instead deriving the policy using a non-linear prediction with an exponent of 1.5 and a decaying optimism factor. This allows the policy to "anticipate" opponent shifts while aggressively correcting major strategic deficits.

4.3.2 Distillation: Warm-started Optimistic Predictive (WOP-) CFR

While single-feature ablations destabilize the solver's synergy, simultaneously stripping away all secondary heuristics reveals a highly distilled algorithmic core. By preserving only the three foundational components identified in the audit, we produce Warm-started Optimistic Predictive (WOP-) CFR (Listing 5). Unlike the raw discovery, WOP-CFR can be defined purely through three formal mathematical modifications to the standard CFR framework. First, it applies the discovered extreme asymmetric discounting to the accumulated regret $R_i^t(I,a) = d_t^{(\pm)} \cdot R_i^{t-1}(I,a) + r_i^t(I,a)$, where $d_t^{(+)} = \frac{t^{1.5}}{t^{1.5}+1}$ if $R_i^{t-1}(I,a) \geq 0$ and $d_t^{(-)} = \frac{t^{-0.1}}{t^{-0.1}+1}$ otherwise. Second, it derives the current policy using a non-linear, optimistic prediction for the next target: $\sigma_i^{t+1}(I,a) \propto \max(0,\ d_t^{(\pm)} \cdot R_i^t(I,a) + \frac{1}{1+t/100} \cdot r_i^t(I,a))^{1.5}$, where $d_t^{(\pm)}$ now depends on the sign of $R_i^t(I,a)$ rather than $R_i^{t-1}(I,a)$. Finally, it enforces the strict hard warm-start, suspending average policy accumulation entirely for the first 500 iterations, after which it accumulates using a static polynomial weight $w_t = t^2$. By discarding the LLM's overfitted training heuristics and structurally enforcing these three core mechanics, this distilled variant achieves a **+0.119 Test IQM** improvement over the raw VAD-CFR discovery. As shown in Figure 3 in the Appendix, **compared with baselines, WOP-CFR ranks in the top three on every one of the 18 games, and achieves the lowest exploitability on 14 of them**. In certain games like 3-player Leduc or 4-card limited information Goofspiel, it even outperforms VAD-CFR by non-negligible margins. When extending WOP-CFR to $K = 10{,}000$ iterations ($10\times$ the reported horizon) on the hardest test games, exploitability continues to decrease monotonically by an additional order of magnitude with no divergence or plateau (Appendix 7.8).

4.4 Experimental Evaluation: Discovering and Distilling State-of-the-Art PSRO

Next, we evaluated the performance of the evolved Smoothed Hybrid Optimistic Regret (SHOR)-PSRO algorithm, comparing its ability to reduce exploitability against standard meta-solver baselines. We use the exploitability at the $K = 100$-th PSRO iteration as the metric. We employ an *exact best response oracle* via value iteration, where at each state it distributes probability mass uniformly among actions with the same optimal values. We benchmarked SHOR-PSRO against standard established meta-solver baselines: Uniform, Nash equilibrium via linear program for 2-player games, AlphaRank (Muller et al., 2020), Projected Replicator Dynamics (PRD) (Lanctot et al., 2017), and Regret Matching (RM) executed for 10^4 steps per PSRO iteration. We use Uniform as the initial program for both solver classes. The prompt we use is shown in Listing 9 in the Appendix.

As shown in Figure 2, SHOR-PSRO ranks among the top performers across both the training set and the test set. In simpler domains like 3-player Kuhn Poker, SHOR-PSRO achieves exploitability levels ($< 10^{-3}$) significantly faster than PRD or RM. In 3-player Leduc Poker, the meta-game landscape becomes significantly more chaotic due to multi-agent dynamics. Despite this, SHOR-PSRO consistently matches or outperforms the best-performing baselines. In the most demanding test case, Liar's Dice (6 sides), SHOR-PSRO also demonstrates a clear advantage.

4.4.1 Identifying Foundational Mechanics

The raw SHOR-PSRO algorithm (Listing 4) contains several highly synergistic mechanisms, most notably an optimistic regret matching+ (ORM+) engine, a hybrid blending module that actively interpolates between the ORM+ output and a smoothed best pure strategy, and asymmetric solvers that utilize distinct logic for training versus evaluation.

To isolate the core drivers of performance, we conducted a systematic train-test split ablation study (with exhaustive configurations detailed in Appendix 7.2.3). As shown in Table 2, the audit exposes the fragility of the LLM's asymmetric design and blending schedules. First, the result identifies the ORM$^+$ loop as the undeniable foundational engine. Removing the ORM$^+$ loop entirely to rely solely on the smoothed pure strategy (`no_blending_pure`) causes a catastrophic collapse in both training and testing environments ($s_{\text{train}} = -2.292, s_{\text{test}} = -1.268$). In contrast, disabling the hybrid pure-strategy blending (`no_blending_orm`, $s_{\text{train}} = -0.069$) and removing the dynamic scheduling (`no_annealing`, $s_{\text{train}} = -0.104$) have negligible effects on training fit, indicating these components do not even drive in-distribution performance. Test IQM corroborates this diagnosis on held-out games: `no_blending_pure` reduces Test IQM to -1.268, while removing the hybrid blending and dynamic scheduling each independently improves Test IQM (+0.020, +0.016), confirming they are overfitted artifacts rather than generalization drivers.

4.4.2 Distillation: Projection Matching (PM-)PSRO

To isolate the true engine of performance, we bypass the LLM's entangled heuristics and distill the solver down to its mathematical core: **Projection Matching (PM-)PSRO** (in Listing 6). By stripping away annealing schedules, hybrid blending, and complex evaluation logic, PM is defined strictly by three essential mechanics. First, PM replaces the regret signal in regret matching with a tangent-projected utility: at each inner step t, given expected utility $u_i^t \in \mathbb{R}^{|\Pi_i|}$, we project it onto the tangent space of the simplex $\mathcal{T}\Delta$: $g_i^t = \mathcal{P}_{\mathcal{T}\Delta}(u_i^t) = (I - \frac{1}{|\Pi_i|}\mathbf{1}\mathbf{1}^\top)u_i^t$, i.e., u_i^t centered by its mean. This shares the geometric operation of continuous-time Projection Dynamics (Lahkar & Sandholm, 2008). Second, PM scales its inner iteration budget linearly with the empirical-game size. Third, PM uses a unified time-averaged solver for training and evaluation (Table 2). Centering utilities evaluates each policy relative to the population mean; paired with RM's positive bounding, the solver instantly forgets below-average strategies, providing geometric immunity to payoff outliers. Combining this with dynamic iteration budgeting yields a **+0.059 Test IQM** log-improvement over the raw SHOR-PSRO discovery. As shown in Figure 4 in the Appendix, **compared with baselines, both SHOR-PSRO and PM-PSRO rank in the top three on every one of the 18 games, and achieve the lowest exploitability on 12 of them**. We further validate these findings on 1000 randomly sampled N-player A-action constant-sum normal-form games per (N, A) configuration (see Appendix 7.5), where SHOR-PSRO and PM-PSRO dominate all baselines across every (N, A) tested.

Decomposing held-out Test IQM into within-family scaling and out-of-family transfer (Colonel Blotto, Battleship 2×2, Battleship 3×2; Table 1 and Appendix Table 2), WOP-CFR achieves $+0.120$ within-family and -0.014 out-of-family (parity at vastly reduced structural complexity). For PM-PSRO, distillation gains are nearly an order of magnitude larger out-of-family ($+0.514$) than within-family ($+0.062$), providing direct evidence that stripping overfitted training machinery aids cross-domain transfer. Extending PM-PSRO to $K = 300$ iterations ($3\times$ the reported horizon) likewise confirms strict monotonic exploitability descent (Appendix 7.8).

5 Related Work

Significant research effort has been dedicated to improving the convergence speed of CFR through specific weighting schemes and regret targets, yielding notable human-designed variants including CFR+ (Tammelin, 2014), strategy-based warm starting (Brown & Sandholm, 2016), Discounted CFR (DCFR) (Brown & Sandholm, 2019b), and Predictive CFR+ (PCFR+) (Farina et al., 2021). Similarly, Policy Space Response Oracles (PSRO) (Bighashdel et al., 2024) generalizes the Double Oracle algorithm (McMahan et al., 2003) by iteratively expanding a policy population via best responses, though practical convergence and population quality remain open challenges (Bighashdel et al., 2024; Zhang & Sandholm, 2024).

Research on automating algorithm design spans both neural meta-learning (e.g., meta-RL (Xu et al., 2018; Oh et al., 2025) and learned optimizers (Metz et al., 2019)) and symbolic discovery (e.g., AutoML-Zero (Real et al., 2020), evolved optimizers (Chen et al., 2023), and evolved RL rules (Co-Reyes et al., 2021; Lu et al., 2024)). Our contribution differs along two axes: we evolve structured algorithmic skeletons with multiple interacting components rather than isolated update rules, and we introduce a formal train-test ablation methodology to distill raw discoveries into minimal solvers — whereas prior work either ships the raw artifact (Lu et al., 2024; Co-Reyes et al., 2021) or applies one-off manual simplification (Chen et al., 2023).

Within multi-agent learning, prior automated design approaches (Feng et al., 2021; Xu et al., 2022; 2024a; Sychrovsk'y et al., 2024; Li et al., 2025) often operated within narrowly constrained spaces or relied on opaque neural parameterizations. Our work builds on AlphaEvolve (Novikov et al., 2025), utilizing LLMs to perform semantic mutation on interpretable code and bridging expressive neural meta-learning with symbolic discovery (Georgiev et al., 2025; Nagda et al., 2025).

6 Conclusion

We automate the design of multi-agent learning algorithms in two paradigms — CFR and PSRO. Through a propose-and-distill methodology, we distill the raw discoveries VAD-CFR and SHOR-PSRO into minimal closed-form solvers, WOP-CFR and PM-PSRO, that improve generalization across an 18-game suite with substantially reduced complexity. This demonstrates that evolutionary LLM search is most effective as a proposal mechanism whose outputs require a generalization audit to isolate stable algorithmic cores.

Limitations. (i) The two algorithms emerge from single evolutionary trajectories; while ten independent runs confirm that their distilled cores recur (Appendix 7.7), all searches share one LLM backbone and prompt design. (ii) Held-out evaluation is dominated by within-family scaling; out of family, PM-PSRO shows large transfer gains while WOP-CFR achieves parity (Section 4.4). (iii) While distillation follows objective train-only ablation criteria, expressing surviving code in closed-form mathematics is done manually. (iv) We show empirical convergence at $10\times$ and $3\times$ horizons (Appendix 7.8) with partial connections to DCFR and double-oracle guarantees, but formal rates remain open. (v) We evaluate with exact oracles to isolate algorithmic updates from function approximation noise; extending to deep CFR and neural PSRO is the natural next step. (vi) Exact exploitability limits evaluation to games under 350K states; scaling to larger domains remains open.

AI use statement

Generative AI is the object of study in this work, not merely an authoring aid, so we distinguish three uses. **(1) AI as the research method.** All algorithms reported here were proposed by an LLM. We

used AlphaEvolve (Novikov et al., 2025) with a Gemini 2.5 Pro backbone (Comanici et al., 2025) as an evolutionary code-generation operator: the LLM mutated the source code of candidate solvers, and candidates were selected by an automated fitness evaluation that involves no LLM. The exact prompts are reproduced in Appendix 7.6, the evolvable code skeletons in Appendix 7.1, and the full search cost — prompts, generations, token volumes, monetary cost at list prices, and CPU-hours — in Appendix 7.9. The subsequent audit-and-distillation phase makes no LLM calls: it consists of deterministic ablation runs evaluated by exact game-tree traversal. **(2) Verification of AI-generated artifacts.** Every algorithm we report was independently re-implemented or re-executed and re-evaluated by the authors outside the search loop, on games the search never saw. The complete source of all four algorithms (VAD-CFR, WOP-CFR, SHOR-PSRO, PM-PSRO) is included in Appendix 7.2 so that readers can inspect exactly what the LLM produced and what we retained. The mathematical descriptions in Section 4 were written by the authors from the generated code and checked against it line by line. The authors take full responsibility for the correctness of all reported algorithms, claims, and results. **(3) AI assistance in writing.** We used LLM-based tools for language editing and for LaTeX formatting assistance. They were not used to generate experimental results, to produce or interpret figures and tables, or to conduct the literature survey; all citations were verified by the authors against the original sources.

Ethics statement

This work studies equilibrium-finding algorithms for abstract benchmark games (poker variants, Goofspiel, Liar's Dice, Colonel Blotto, and Battleship) implemented in OpenSpiel (Lanctot et al., 2019). It involves no human subjects, no personal or user-generated data, and no newly released dataset. The immediate application domain — computing approximate Nash equilibria in small imperfect-information games — carries no direct foreseeable harm beyond the generic dual-use consideration that stronger game-solving methods can be applied to competitive or adversarial settings such as online gambling; our games are standard research benchmarks many orders of magnitude smaller than deployed systems. One consideration specific to LLM-driven discovery deserves mention: automatically generated algorithms can be adopted on the strength of benchmark numbers without their mechanisms being understood, and a central point of this paper is that raw LLM discoveries contain machinery that is tuned to the training distribution and does not transfer. We therefore regard the generalization audit we describe as a responsible-use recommendation as much as a methodological one. The compute consumed by our searches is itemized in Appendix 7.9; all candidate evaluation was CPU-only.

Reproducibility statement

We have aimed to make every stage of the pipeline reproducible. The search stage is specified by the evolvable code skeletons and fixed function signatures (Appendix 7.1), the verbatim prompts (Appendix 7.6), the training games and fitness definition (Section 3), and the LLM backbone and search budget (Appendix 7.9). The artifacts themselves are given as complete source code for all four algorithms (Appendix 7.2), so the reported solvers can be re-run without re-running any search. The evaluation stage is deterministic: all solvers are deterministic programs and exploitability is computed exactly by full game-tree traversal on 18 public OpenSpiel games whose exact configurations, sizes, and parameters are tabulated in Appendix 7.3, so our numbers carry no seed variance. The audit stage is specified by the explicit retention rule and the IQM log-improvement score defined in Section 4.2, together with a precise description of each ablated variant (Appendices 7.2.1 and 7.2.3); applying the stated rule to the reported ablation tables reproduces our distillation choices exactly. Finally, Appendix 7.7 reports ten additional independent search runs so that readers can judge which findings are properties of the search rather than of one trajectory.

References

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In *Thirty-Fifth Conference on Neural Information Processing Systems*, volume 34, pp. 29304–29320, 2021.

Ariyan Bighashdel, Yongzhao Wang, Stephen McAleer, Rahul Savani, and Frans A Oliehoek. Policy space response oracles: A survey. In *Thirty-Third International Joint Conference on Artificial Intelligence (IJCAI-24) Survey Track*, 2024.

Emile Borel. The theory of play and integral equations with skew symmetric kernels. *Econometrica: journal of the Econometric Society*, pp. 97–100, 1953.

Noam Brown and Tuomas Sandholm. Strategy-based warm starting for regret minimization in games. In *Thirtieth AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456): 885–890, 2019a. doi: 10.1126/science.aay2400.

Noam Brown and Tuomas Sandholm. Solving imperfect-information games via discounted regret minimization. In *Thirty-Third AAAI Conference on Artificial Intelligence*, volume 33, pp. 1829–1836, 2019b.

Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. In *Thirty-Sixth International Conference on Machine Learning*, pp. 793–802, 2019.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, et al. Symbolic discovery of optimization algorithms. In *Thirty-Seventh International Conference on neural information processing systems*, pp. 49205–49233, 2023.

John D Co-Reyes, Yingjie Miao, Daiyi Peng, Esteban Real, Quoc V Le, Sergey Levine, Honglak Lee, and Aleksandra Faust. Evolving reinforcement learning algorithms. In *Ninth International Conference on Learning Representations*, 2021.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Gabriele Farina, Chun Kai Ling, Fei Fang, and Tuomas Sandholm. Correlation in extensive-form games: Saddle-point formulation and benchmarks. In *Thirty-third International Conference on Neural Information Processing Systems*, volume 32, 2019.

Gabriele Farina, Christian Kroer, and Tuomas Sandholm. Faster game solving via predictive blackwell approachability: Connecting regret matching and mirror descent. In *Thirty-Fifth AAAI Conference on Artificial Intelligence*, volume 35, pp. 5363–5371, 2021.

Xidong Feng, Oliver Slumbers, Ziyu Wan, Bo Liu, Stephen Marcus McAleer, Ying Wen, Jun Wang, and Yaodong Yang. Neural auto-curricula in two-player zero-sum games. In *Thirty-Fifth International Conference on Neural Information Processing Systems*, 2021.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

Harold W. Kuhn. Simplified two-person poker. *Contributions to the Theory of Games*, pp. 97–103, 1950.

Ratul Lahkar and William H Sandholm. The projection dynamic and the geometry of population games. *Games and Economic Behavior*, 64(2):565–590, 2008.

Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Péréat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. In *Thirty-First International Conference on Neural Information Processing Systems*, pp. 4190–4203, 2017.

Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. Openspiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2019.

Hanyu Li, Dongchen Li, and Xiaotie Deng. Discovering expert-level nash equilibrium algorithms with large language models. *arXiv preprint arXiv:2508.11874*, 2025.

Viliam Lisỳ, Marc Lanctot, and Michael H Bowling. Online monte-carlo counterfactual regret minimization for search in imperfect information games. In *Fourteenth International Conference on Autonomous Agents and Multiagent Systems*, pp. 27–36, 2015.

Chris Lu, Samuel Holt, Claudio Fanconi, Alex J Chan, Jakob Foerster, Mihaela van der Schaar, and Robert T Lange. Discovering preference optimization algorithms with and for large language models. In *Thirty-Sixth Conference on Neural Information Processing Systems*, volume 37, pp. 86528–86573, 2024.

Stephen Mcaleer, John Lanier, Roy Fox, and Pierre Baldi. Pipeline psro: A scalable approach for finding approximate nash equilibria in large games. In *Thirty-Fifth Neural Information Processing Systems*, volume 34, pp. 20238–20248. Curran Associates, Inc., 2020.

H Brendan McMahan, Geoffrey J Gordon, and Avrim Blum. Planning in the presence of cost functions controlled by an adversary. In *Twentieth International Conference on Machine Learning*, pp. 536–543, 2003.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, C Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *Thirty-Sixth International Conference on Machine Learning*, pp. 4556–4565, 2019.

Paul Muller, Shayegan Omidshafiei, Mark Rowland, Karl Tuyls, Julien Perolat, Siqi Liu, Daniel Hennes, Luke Marris, Marc Lanctot, Edward Hughes, et al. A generalized training approach for multiagent learning. In *Ninth International Conference on Learning Representations*, 2020.

Ansh Nagda, Prabhakar Raghavan, and Abhradeep Thakurta. Reinforced generation of combinatorial structures: Hardness of approximation. *arXiv preprint arXiv:2509.18057*, 2025.

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.

Junhyuk Oh, Greg Farquhar, Iurii Kemaev, Dan A Calian, Matteo Hessel, Luisa Zintgraf, Satinder Singh, Hado Van Hasselt, and David Silver. Discovering state-of-the-art reinforcement learning algorithms. *Nature*, pp. 1–2, 2025.

Esteban Real, Chen Liang, David So, and Quoc V Le. Automl-zero: Evolving machine learning algorithms from scratch. In *Thirty-Seventh International Conference on Machine Learning*, pp. 8007–8019, 2020.

Sheldon M Ross. Goofspiel—the game of pure strategy. *Journal of Applied Probability*, 8(3): 621–625, 1971.

Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, New York, NY, USA, 2008.

Finnegan Southey, Michael Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and Chris Rayner. Bayes' bluff: opponent modelling in poker. In *Twenty-First Conference on Uncertainty in Artificial Intelligence*, pp. 550–558, 2005.

David Sychrovsk'y, Michal Šustr, Elnaz Davoodi, Michael Bowling, Marc Lanctot, and Martin Schmid. Learning to not regret. In *Thirty-Eighth AAAI Conference on Artificial Intelligence*, 2024.

Oskari Tammelin. Solving large imperfect information games using cfr+. *arXiv preprint arXiv:1407.5042*, 2014.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782): 350–354, 2019.

Michael P Wellman, Karl Tuyls, and Amy Greenwald. Empirical game theoretic analysis: A survey. *Journal of Artificial Intelligence Research*, 82:1017–1076, 2025.

Hang Xu, Kai Li, Haobo Fu, Qiang Fu, and Junliang Xing. Autocfr: Learning to design counterfactual regret minimization algorithms. In *Thirty-Sixth AAAI Conference on Artificial Intelligence*, pp. 5244–5251, 2022.

Hang Xu, Kai Li, Haobo Fu, Qiang Fu, Junliang Xing, and Jian Cheng. Dynamic discounted counterfactual regret minimization. In *Twelfth International Conference on Learning Representations*, 2024a.

Hang Xu, Kai Li, Bingyun Liu, Haobo Fu, Qiang Fu, Junliang Xing, and Jian Cheng. Minimizing weighted counterfactual regret with optimistic online mirror descent. In *Thirty-Third International Joint Conference on Artificial Intelligence*, pp. 5272–5280, 2024b.

Zhongwen Xu, Hado van Hasselt, and David Silver. Meta-gradient reinforcement learning. In *Thirty-Second International Conference on Neural Information Processing Systems*, volume 31, 2018.

Brian Hu Zhang and Tuomas Sandholm. Exponential lower bounds on the double oracle algorithm in zero-sum games. In *Thirty-Third International Joint Conference on Artificial Intelligence*, pp. 3032–3039, 2024.

Naifeng Zhang, Stephen McAleer, and Tuomas Sandholm. Faster game solving via hyperparameter schedules. In *Fortieth AAAI Conference on Artificial Intelligence*, 2026.

Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Twenty-First International Conference on Neural Information Processing Systems*, pp. 1729–1736, 2007.

7 APPENDIX

7.1 CODE SKELETON

Listing 1: The Python CFR code skeleton used as the search space for AlphaEvolve. The highlighted methods `update_accumulate_regret`, `get_updated_current_policy`, and `update_accumulate_policy` represent the evolvable components of the CFR algorithm.

```
class RegretAccumulator:
  """A class that updates cumulative regret at an information set."""

  def update_accumulate_regret(self, info_state_node, iteration_number, cfr_regrets):
    """
    Args:
      info_state_node: Data structure with cumulative_regret and cumulative_policy.
      iteration_number: Current CFR iteration.
      cfr_regrets: Counterfactual regrets (not yet added to node).
    Returns:
      Updated cumulative regret dictionary for each action.
    """
    ...


class PolicyFromRegretAccumulator:
  """A class that derives the current policy from regret."""

  def get_updated_current_policy(self, info_state_node, iteration_number, cfr_regrets,
     ↪ previous_policy):
    """
    Args:
      info_state_node: Data structure with cumulative_regret.
      iteration_number: Current CFR iteration.
      cfr_regrets: Counterfactual regrets (already added to node).
      previous_policy: Previous policy at this info set.
    Returns:
      Updated current policy dictionary.
    """
    ...


class PolicyAccumulator:
  """A class that updates the average policy during tree traversal."""

  def update_accumulate_policy(self, info_state_node, iteration_number, info_state_policy,
     ↪ cfr_regrets, reach_prob, counterfactual_reach_prob):
    """
    Args:
      info_state_node: Data structure with cumulative_policy.
      iteration_number: Current CFR iteration.
      info_state_policy: Current policy at this info set.
      cfr_regrets: Counterfactual regrets (already added to node).
      reach_prob: Probability of reaching current history (player's contribution).
      counterfactual_reach_prob: Probability of reaching current history (opponents'
     ↪ contribution).
    Returns:
      Updated cumulative policy dictionary.
    """
    ...
```

Listing 2: The Python PSRO code skeleton used as the search space for AlphaEvolve. The highlighted methods `TrainMetaStrategySolver` and `EvalMetaStrategySolver` represent the evolvable components of the PSRO algorithm.

```
class TrainMetaStrategySolver:
  """Returns meta strategies to train against in PSRO."""

  def get_meta_strategy(self, game, policy_sets, meta_games):
    """Returns meta strategies to train against in policy-space response oracles.

    Args:
      game: The pyspiel game object.
      policy_sets: A list of lists of policies, one list per player.
        policy_sets[p][i] is player p's i-th policy. len(policy_sets[p]) ==
        meta_games[0].shape[p].
      meta_games: A list of n-dimensional numpy arrays, one per player. Each
        array has shape (num_strats_p0, num_strats_p1, ..., num_strats_pn-1) and
```

```
        meta_games[p][i0, i1, ..., in-1] is the payoff of player p when player k
        chooses strategy ik.

    Returns:
      A list of mixed-strategies, one for each player. Each mixed strategy is
      a list of non-negative weights (not necessarily normalized). It is used
      to train best response against.
    """
    ...


class EvalMetaStrategySolver:
  """Returns meta strategies for evaluation in PSRO."""

  def get_meta_strategy(self, game, policy_sets, meta_games):
    """Returns meta strategies for evaluation in policy-space response oracles.

    Args:
      game: The pyspiel game object.
      policy_sets: A list of lists of policies, one list per player.
        policy_sets[p][i] is player p's i-th policy. len(policy_sets[p]) ==
        meta_games[0].shape[p].
      meta_games: A list of n-dimensional numpy arrays, one per player. Each
        array has shape (num_strats_p0, num_strats_p1, ..., num_strats_pn-1) and
        meta_games[p][i0, i1, ..., in-1] is the payoff of player p when player k
        chooses strategy ik.

    Returns:
      A list of mixed-strategies, one for each player. Each mixed strategy is
      a list of non-negative weights (not necessarily normalized). It is used
      for evaluation of the current PSRO policies. E.g., computing
      exploitability.
    """
    ...
```

7.2 SOURCE CODE OF DISCOVERED ALGORITHMS

Listing 3: VAD-CFR

```
class RegretAccumulator:
  """A class that updates cumulative regret using Adaptive Discounting
  with separate discounting for positive and negative regrets, and instantaneous regret
    ↪ boosting.
  """

  @staticmethod
  def _calculate_adaptive_params(
      iteration_number,
      cfr_regrets,
      base_alpha,
      base_beta,
      volatility_sensitivity,
      max_expected_instantaneous_regret,
      ewma_decay_factor,
      current_ewma_magnitude,
  ):
    """Calculates adaptive discounting parameters for a given iteration.
    This static method centralizes the logic for EWMA volatility, adaptive
    alpha/beta, and discount factors to ensure consistency across components.
    """
    t_plus_one = float(iteration_number + 1)

    instantaneous_regret_magnitude = max(
        (abs(r) for r in cfr_regrets.values()), default=0.0
    )

    if iteration_number == 0:
        projected_ewma = instantaneous_regret_magnitude
    else:
        projected_ewma = (
            ewma_decay_factor * instantaneous_regret_magnitude +
            (1.0 - ewma_decay_factor) * current_ewma_magnitude
        )

    if max_expected_instantaneous_regret > 0:
        normalized_volatility = min(1.0, projected_ewma / max_expected_instantaneous_regret)
    else:
```

```
            normalized_volatility = 0.0

        effective_alpha = max(0.1, base_alpha - volatility_sensitivity * normalized_volatility)
        effective_beta = base_beta - volatility_sensitivity * normalized_volatility
        effective_beta = min(effective_alpha, effective_beta)

        discount_factor_positive = (t_plus_one**effective_alpha) / (t_plus_one**effective_alpha
        ↪ + 1.0)
        discount_factor_negative = (t_plus_one**effective_beta) / (t_plus_one**effective_beta +
        ↪ 1.0)

        return projected_ewma, normalized_volatility, discount_factor_positive,
        ↪ discount_factor_negative

    def __init__(self, base_alpha=1.5, base_beta=-0.1, volatility_sensitivity=0.5,
                 max_expected_instantaneous_regret=2.0, instantaneous_regret_boost_factor=1.1,
                 ewma_decay_factor=0.1, negative_regret_cap=-20.0):
        """Initializes the regret accumulator with adaptive discounting parameters.

        Args:
          base_alpha: The baseline exponent for discounting positive cumulative regrets.
          base_beta: The baseline exponent for discounting negative cumulative regrets.
          volatility_sensitivity: Controls how strongly the instantaneous regret
            magnitude influences the adaptive alpha/beta. A higher value means the
            exponents will be more reduced by high volatility.
          max_expected_instantaneous_regret: An estimate of the maximum possible
            instantaneous regret magnitude, used for normalizing the volatility.
          instantaneous_regret_boost_factor: Boost factor for positive instantaneous regrets.
            A factor > 1.0 makes the algorithm more reactive to current good actions.
          ewma_decay_factor: Decay factor for the EWMA of instantaneous regret magnitude.
          negative_regret_cap: The minimum value for cumulative regret, to prevent
            regret lock-in and improve adaptability.
        """
        self._base_alpha = base_alpha
        self._base_beta = base_beta
        self._volatility_sensitivity = volatility_sensitivity
        self._max_expected_instantaneous_regret = max_expected_instantaneous_regret
        self._instantaneous_regret_boost_factor = instantaneous_regret_boost_factor
        self._ewma_decay_factor = ewma_decay_factor
        self._negative_regret_cap = negative_regret_cap
        self._ewma_instantaneous_regret_magnitude = 0.0

    def update_accumulate_regret(
        self, info_state_node, iteration_number, cfr_regrets
    ):
        """Updates cumulative regret for each action at an information set.
        Cumulative regrets are now signed.

        Args:
          info_state_node: a data structure corresponding to an information set with
            cumulative_regret and cumulative_policy stored.
          iteration_number: the current CFR iteration (0-indexed).
          cfr_regrets: the instantaneous counterfactual regrets of the current policy at the
            current information set. cfr_regrets haven't been added to info_state_node.
        ↪ cumulative_regret.
        Returns:
          updated cumulative regret for each action at the current information set (signed).
        """
        # Centralize adaptive parameter calculation to ensure consistency and reduce redundancy.
        (
            self._ewma_instantaneous_regret_magnitude,
            _,  # normalized_volatility is not used here
            discount_factor_positive,
            discount_factor_negative,
        ) = RegretAccumulator._calculate_adaptive_params(
            iteration_number=iteration_number,
            cfr_regrets=cfr_regrets,
            base_alpha=self._base_alpha,
            base_beta=self._base_beta,
            volatility_sensitivity=self._volatility_sensitivity,
            max_expected_instantaneous_regret=self._max_expected_instantaneous_regret,
            ewma_decay_factor=self._ewma_decay_factor,
            current_ewma_magnitude=self._ewma_instantaneous_regret_magnitude,
        )

        updated_cumulative_regret = {}
        for action in cfr_regrets:
          old_regret = info_state_node.cumulative_regret[action]

          instantaneous_regret_component = cfr_regrets[action]
          if instantaneous_regret_component > 0:
```

```
            instantaneous_regret_component *= self._instantaneous_regret_boost_factor

        # Apply different discount factors based on the sign of the old regret.
        if old_regret >= 0:
            discounted_old_regret = discount_factor_positive * old_regret
        else:
            discounted_old_regret = discount_factor_negative * old_regret

        new_regret = discounted_old_regret + instantaneous_regret_component

        # Cap the negative regret to prevent lock-in and improve adaptability.
        new_regret = max(self._negative_regret_cap, new_regret)

        # Crucially, regrets are NOT clipped to zero here. They can be negative.
        updated_cumulative_regret[action] = new_regret

      return updated_cumulative_regret


class PolicyFromRegretAccumulator:
  """A class that obtains a current policy from a consistent optimistic projection of
     ↪ regrets.
  It aligns the policy generation with the adaptive, asymmetric, and boosted logic from
     ↪ RegretAccumulator.
  """

  def __init__(self,
               initial_optimism_factor=1.0,
               optimism_decay_factor=100.0,
               positive_policy_exponent=1.5,
               base_alpha=1.5,
               base_beta=-0.1,
               volatility_sensitivity=0.5,
               max_expected_instantaneous_regret=2.0,
               instantaneous_regret_boost_factor=1.1,
               ewma_decay_factor=0.1):
    """Initializes the PolicyFromRegretAccumulator with parameters consistent with
       ↪ RegretAccumulator.

    Args:
      initial_optimism_factor: The initial weighting for the instantaneous regret component
       ↪ in the projection.
      optimism_decay_factor: Controls how quickly the optimism weight decays.
      positive_policy_exponent: Exponent for non-linear scaling of positive regrets.
      base_alpha: The baseline exponent for discounting positive regrets.
      base_beta: The baseline exponent for discounting negative regrets.
      volatility_sensitivity: Controls influence of volatility on both discounting
        and optimism dampening.
      max_expected_instantaneous_regret: Used for normalizing volatility.
      instantaneous_regret_boost_factor: Boost factor for positive instantaneous regrets.
      ewma_decay_factor: Decay factor for the EWMA of volatility.
    """
    self._initial_optimism_factor = initial_optimism_factor
    self._optimism_decay_factor = optimism_decay_factor
    self._positive_policy_exponent = positive_policy_exponent
    self._base_alpha = base_alpha
    self._base_beta = base_beta
    self._volatility_sensitivity = volatility_sensitivity
    self._max_expected_instantaneous_regret = max_expected_instantaneous_regret
    self._instantaneous_regret_boost_factor = instantaneous_regret_boost_factor
    self._ewma_decay_factor = ewma_decay_factor
    self._ewma_instantaneous_regret_magnitude = 0.0

  def get_updated_current_policy(self, info_state_node, iteration_number, cfr_regrets,
     ↪ previous_policy):
    """Obtains the current policy using a projection that is consistent with the
       ↪ RegretAccumulator's update rule.

    This method creates a tighter feedback loop by basing the current policy
    on a projection of what the regrets will be *after* the current
    iteration's update. This projection uses the same adaptive, asymmetric
    discounting and boosting logic as the main regret accumulation step.
    """
    # Centralize adaptive parameter calculation, ensuring consistency with RegretAccumulator
       ↪ .
    (
        self._ewma_instantaneous_regret_magnitude,
        normalized_volatility,
        discount_factor_positive,
        discount_factor_negative,
    ) = RegretAccumulator._calculate_adaptive_params(
```

```
            iteration_number=iteration_number,
            cfr_regrets=cfr_regrets,
            base_alpha=self._base_alpha,
            base_beta=self._base_beta,
            volatility_sensitivity=self._volatility_sensitivity,
            max_expected_instantaneous_regret=self._max_expected_instantaneous_regret,
            ewma_decay_factor=self._ewma_decay_factor,
            current_ewma_magnitude=self._ewma_instantaneous_regret_magnitude,
        )

        base_optimism = self._initial_optimism_factor / (1.0 + float(iteration_number) / self.
        ↪ _optimism_decay_factor)
        # Dampen optimism during volatile periods to increase stability.
        optimism_dampening_factor = max(0.0, 1.0 - self._volatility_sensitivity *
        ↪ normalized_volatility)
        optimism_strength = base_optimism * optimism_dampening_factor

        action_to_projected_regret = {}
        for action in info_state_node.legal_actions:
          old_cumulative_regret = info_state_node.cumulative_regret.get(action, 0.0)
          instantaneous_regret = cfr_regrets.get(action, 0.0)

          instantaneous_regret_component = instantaneous_regret
          if instantaneous_regret_component > 0:
              instantaneous_regret_component *= self._instantaneous_regret_boost_factor

          if old_cumulative_regret >= 0:
              discounted_old_regret = discount_factor_positive * old_cumulative_regret
          else:
              discounted_old_regret = discount_factor_negative * old_cumulative_regret

          projected_regret = discounted_old_regret + optimism_strength *
        ↪ instantaneous_regret_component
          action_to_projected_regret[action] = projected_regret

        positive_scaled_projected_regrets = {
            action: (max(0.0, regret) ** self._positive_policy_exponent)
            for action, regret in action_to_projected_regret.items()
        }

        sum_positive_scaled_projected_regrets = sum(positive_scaled_projected_regrets.values())

        info_state_policy = {}
        if sum_positive_scaled_projected_regrets > 0:
          for action, scaled_regret in positive_scaled_projected_regrets.items():
            info_state_policy[action] = scaled_regret / sum_positive_scaled_projected_regrets
        else:
          num_legal_actions = len(info_state_node.legal_actions)
          for action in info_state_node.legal_actions:
            info_state_policy[action] = 1.0 / num_legal_actions

        return info_state_policy


    class PolicyAccumulator:
      """A class that updates cumulative policy using regret-informed weighted
      averaging with a warmup period."""

      def __init__(self, base_gamma=2.0, gamma_max=4.0,
                   gamma_volatility_sensitivity=1.5, warmup_iterations=500,
                   stability_exponent=1.5, max_expected_instantaneous_regret=2.0,
                   regret_magnitude_weighting_exponent=0.5): # New parameter
        """Initializes the PolicyAccumulator with adaptive gamma parameters and regret-magnitude
        ↪  weighting.

        Args:
          base_gamma: The baseline exponent for polynomial weighting of policies.
          gamma_max: The maximum value the adaptive gamma can reach.
          gamma_volatility_sensitivity: Controls how strongly volatility influences gamma.
          warmup_iterations: Number of initial iterations to skip for policy averaging.
          stability_exponent: Exponent for the stability factor based on regret magnitude.
          max_expected_instantaneous_regret: Normalization factor for instantaneous regret
        ↪ magnitude.
          regret_magnitude_weighting_exponent: Exponent for up-weighting policies based on
            the absolute magnitude of instantaneous regrets. Higher values give more
            emphasis to policies from iterations with large regrets.
        """
        self._base_gamma = base_gamma
        self._gamma_max = gamma_max
        self._gamma_volatility_sensitivity = gamma_volatility_sensitivity
        self._warmup_iterations = warmup_iterations
```

```
        self._stability_exponent = stability_exponent
        self._max_expected_instantaneous_regret = max_expected_instantaneous_regret
        self._regret_magnitude_weighting_exponent = regret_magnitude_weighting_exponent # Stored

    def update_accumulate_policy(
        self,
        info_state_node,
        iteration_number,
        info_state_policy,
        cfr_regrets,
        reach_prob,
        counterfactual_reach_prob,
    ):
        """Updates cumulative policy using delayed, regret-informed, and regret-magnitude
        ↪ weighted averaging.
        """
        if iteration_number < self._warmup_iterations:
            return info_state_node.cumulative_policy

        # Calculate instantaneous regret magnitude (L-infinity norm) for this iteration
        instantaneous_regret_magnitude = max(
            (abs(r) for r in cfr_regrets.values()), default=0.0
        )

        # Normalize volatility using the shared parameter
        if self._max_expected_instantaneous_regret > 0:
            normalized_volatility = min(1.0, instantaneous_regret_magnitude / self.
        ↪ _max_expected_instantaneous_regret)
        else:
            normalized_volatility = 0.0

        # Adapt gamma based on volatility: higher volatility -> higher gamma
        effective_gamma = self._base_gamma + self._gamma_volatility_sensitivity *
        ↪ normalized_volatility
        effective_gamma = min(self._gamma_max, effective_gamma)

        # Standard polynomial weighting gives more weight to later iterations, now with adaptive
        ↪  gamma.
        temporal_weight = (float(iteration_number) + 1.0) ** effective_gamma

        # Calculate a stability factor from the L-infinity norm of instantaneous regrets.
        # Higher regret magnitude -> lower stability factor, using L-infinity norm for
        ↪ consistency.
        regret_stability_factor = 1.0 / (1.0 + instantaneous_regret_magnitude**self.
        ↪ _stability_exponent)

        # NEW: Regret Magnitude Weighting Factor
        # Policies from iterations with higher regret magnitude contribute more to the average.
        # This factor boosts the weight, with higher values of the exponent giving more emphasis
        ↪ .
        # Ensure it's never zero to avoid division by zero or completely nullifying weight.
        regret_magnitude_factor = (
            1.0 + (instantaneous_regret_magnitude / self._max_expected_instantaneous_regret)
        ) ** self._regret_magnitude_weighting_exponent
        regret_magnitude_factor = max(0.1, regret_magnitude_factor) # Ensure minimum value to
        ↪ avoid zero weight if normalization results in very small number

        # The final weight combines the temporal, stability, and regret-magnitude-based
        ↪ components.
        weight = temporal_weight * regret_stability_factor * regret_magnitude_factor

        return {
            action: (
                info_state_node.cumulative_policy[action]
                + weight * reach_prob * info_state_policy[action]
            )
            for action in info_state_policy
        }
```

Listing 4: SHOR-PSRO

```
import numpy as np


def _smoothed_best_pure_strategy(payoff_vec, temperature=1.0):
    """Computes a smoothed distribution biased towards the best pure strategy.
    The softmax function ensures that strategies with higher payoffs are
    given higher probability, with 'temperature' controlling the sharpness
    of the distribution. A lower temperature makes the distribution more
```

```
    concentrated on the best strategy, while a higher temperature
    leads to a more uniform distribution.
    """
    # Subtract max payoff for numerical stability (standard softmax trick)
    stable_payoffs = payoff_vec - np.max(payoff_vec)
    exp_payoffs = np.exp(stable_payoffs / temperature)

    sum_exp_payoffs = np.sum(exp_payoffs)
    if sum_exp_payoffs > 1e-12:  # Avoid division by zero
      return exp_payoffs / sum_exp_payoffs
    else:
      # Fallback to uniform distribution if all exponentiated payoffs are
      # effectively zero (e.g., due to very low temperature and negative payoffs,
      # or all payoffs being identical after stabilization).
      return np.ones_like(payoff_vec) / len(payoff_vec)

  def _hybrid_orm_solver(meta_games, iterations,
                         blending_factor=0.0,
                         temperature=0.1,
                         momentum_beta=0.0,
                         gain_normalization=True,
                         diversity_bonus_coeff=0.0,
                         return_average_strategy=True): # New: Flag to return average or last-
      ↪ iterate strategy
    """Computes meta-strategies using Optimistic Regret Matching+ enhanced with
    optimistic updates, gain normalization, and a diversity bonus, then blended
    with a smoothed best pure strategy.

    This solver combines the stability and convergence properties of Optimistic
    Regret Matching+ (ORM+) with an explicit pull towards highly rewarding
    pure strategies, smoothed by a temperature-controlled softmax. This hybrid
    approach aims to leverage ORM+'s ability to find mixed equilibria while
    also quickly identifying and exploring strong pure-strategy modes in the
    meta-game, thereby potentially accelerating the discovery of low-exploitable
    policies in PSRO. The blending factor controls the trade-off between
    these two dynamics.

    Args:
      meta_games: A list of n-dimensional numpy arrays, one per player.
      iterations: Number of internal solver iterations.
      blending_factor: Weight (0 to 1) for blending ORM+ output with the
        smoothed best pure strategy. A factor of 0 means pure ORM+; 1 means
        pure smoothed best pure strategy.
      temperature: Temperature for softmax smoothing when calculating the
        smoothed best pure strategy. Lower values make the smoothing sharper.
      momentum_beta: Momentum parameter for optimistic updates to payoff gains.
      gain_normalization: If True, normalizes payoff gains to make learning rate
        more robust across games.
      diversity_bonus_coeff: Coefficient for diversity bonus, encouraging
        exploration of less-chosen policies.
      return_average_strategy: If True, returns time-averaged strategies.
        If False, returns last-iterate strategies.

    Returns:
      A list of mixed-strategies, one for each player, as numpy arrays.
    """
    num_players = len(meta_games)
    num_strats = [m.shape[i] for i, m in enumerate(meta_games)]

    if any(n_s == 0 for n_s in num_strats):
        return [np.array([]).tolist() for _ in range(num_players)]

    strategies = [np.ones(s, dtype=float) / s for s in num_strats]
    cum_regrets = [np.zeros(s, dtype=float) for s in num_strats]
    avg_strategies = [np.zeros(s, dtype=float) for s in num_strats]
    prev_centered_payoff_gains = [np.zeros(s, dtype=float) for s in num_strats]

    for t in range(iterations):
      current_centered_payoff_gains = [np.zeros(s, dtype=float) for s in num_strats]
      orm_strategies_this_iter = [np.zeros(s, dtype=float) for s in num_strats]

      for p in range(num_players):
        payoff_vec = meta_games[p]
        for other_p in reversed(range(num_players)):
          if other_p != p:
            payoff_vec = np.tensordot(payoff_vec, strategies[other_p], axes=([other_p], [0]))

        centered_payoff_gains = payoff_vec - np.mean(payoff_vec)
        current_centered_payoff_gains[p] = centered_payoff_gains

        optimistic_payoff_gains = (1 + momentum_beta) * centered_payoff_gains - \
```

```
                                momentum_beta * prev_centered_payoff_gains[p]

        diversity_bonus = diversity_bonus_coeff * (1.0 - strategies[p])

        gains_for_regret_update = optimistic_payoff_gains + diversity_bonus

        if gain_normalization:
          max_abs_gain = np.max(np.abs(gains_for_regret_update))
          if max_abs_gain > 1e-8:
            gains_for_regret_update /= max_abs_gain

        cum_regrets[p] += gains_for_regret_update
        cum_regrets[p] = np.maximum(0, cum_regrets[p])

        sum_pos_regret = cum_regrets[p].sum()
        if sum_pos_regret > 1e-12:
          orm_strategies_this_iter[p] = cum_regrets[p] / sum_pos_regret
        else:
          orm_strategies_this_iter[p] = np.ones(num_strats[p]) / num_strats[p]

        smoothed_best_pure = _smoothed_best_pure_strategy(payoff_vec, temperature)

        strategies[p] = (1 - blending_factor) * orm_strategies_this_iter[p] + \
                        blending_factor * smoothed_best_pure

        prev_centered_payoff_gains[p] = current_centered_payoff_gains[p]

        if return_average_strategy: # Accumulate blended strategy only if average is requested
          avg_strategies[p] += strategies[p]

    if return_average_strategy:
      final_strategies = []
      for p in range(num_players):
        sum_avg_strat = np.sum(avg_strategies[p])
        if sum_avg_strat > 0:
          final_strategies.append(avg_strategies[p] / sum_avg_strat)
        else:
          final_strategies.append(np.ones(num_strats[p]) / num_strats[p])
      return final_strategies
    else:
      # If not returning average, return the last-iterate strategies
      return strategies


class TrainMetaStrategySolver:
  """A hybrid meta-solver for training that blends ORM+ with smoothed best pure strategies.

  This solver aims to accelerate convergence to low-exploitable strategies by
  dynamically balancing regret-minimization with a pull towards high-performing
  (but smoothed) pure strategies. Optimistic updates, gain normalization, and
  a diversity bonus are incorporated for improved learning dynamics. The
  blending factor, temperature, and diversity bonus are annealed over the
  outer PSRO iterations.
  """

  def __init__(self,
               base_solver_iterations=1000, # Base number of internal iterations
               iterations_per_policy_scale=20, # How much iterations scale per added policy
               max_solver_iterations=5000, # Max internal solver iterations
               initial_blending_factor=0.3, final_blending_factor=0.05,
               initial_temperature=0.5, final_temperature=0.01,
               momentum_beta=0.5,
               gain_normalization=True,
               initial_diversity_bonus_coeff=0.05,
               final_diversity_bonus_coeff=0.001,
               max_psro_iterations_for_annealing=75):
    """Initializes hybrid ORM solver parameters for training.

    Args:
      base_solver_iterations: Base number of internal solver iterations for
      ↪ _hybrid_orm_solver.
      iterations_per_policy_scale: Amount to increase internal solver iterations per added
      ↪ policy.
      max_solver_iterations: Maximum internal solver iterations.
      initial_blending_factor: Initial weight for the smoothed best pure strategy component.
      final_blending_factor: Final weight for the smoothed best pure strategy component.
      initial_temperature: Initial temperature for softmax smoothing.
      final_temperature: Final temperature for softmax smoothing.
      momentum_beta: Momentum for optimistic updates.
      gain_normalization: Normalizes gains for scale-invariance.
      initial_diversity_bonus_coeff: Max initial diversity bonus coefficient.
```

```
        final_diversity_bonus_coeff: Min initial diversity bonus coefficient across PSRO
        ↪ iterations.
        max_psro_iterations_for_annealing: PSRO iterations over which outer annealing occurs.
      """
      self._base_solver_iterations = base_solver_iterations
      self._iterations_per_policy_scale = iterations_per_policy_scale
      self._max_solver_iterations = max_solver_iterations
      self._initial_blending_factor = initial_blending_factor
      self._final_blending_factor = final_blending_factor
      self._initial_temperature = initial_temperature
      self._final_temperature = final_temperature
      self._momentum_beta = momentum_beta
      self._gain_normalization = gain_normalization
      self._initial_diversity_bonus_coeff = initial_diversity_bonus_coeff
      self._final_diversity_bonus_coeff = final_diversity_bonus_coeff
      self._max_psro_iterations_for_annealing = max_psro_iterations_for_annealing
      self._current_psro_iteration = 0

    def get_meta_strategy(self, game, policy_sets, meta_games):
      """Returns blended meta strategies for training.

      Args:
        game: The pyspiel game object.
        policy_sets: A list of lists of policies, one list per player.
          policy_sets[p][i] is player p's i-th policy. len(policy_sets[p]) ==
          meta_games[0].shape[p].
        meta_games: A list of n-dimensional numpy arrays, one per player. Each
          array has shape (num_strats_p0, num_strats_p1, ..., num_strats_pn-1) and
          meta_games[p][i0, i1, ..., in-1] is the payoff of player p when player k
          chooses strategy ik.

      Returns:
        A list of blended mixed-strategies.
      """
      del game, policy_sets # Unused

      self._current_psro_iteration += 1
      current_psro_iter = self._current_psro_iteration

      # Adaptive solver iterations: scale with current population size
      num_current_policies_p0 = len(meta_games[0]) # Assuming symmetric populations
      solver_iterations = int(self._base_solver_iterations +
                              self._iterations_per_policy_scale * (num_current_policies_p0 -
      ↪ 1))
      solver_iterations = np.clip(solver_iterations, self._base_solver_iterations, self.
      ↪ _max_solver_iterations)

      annealing_progress = min(1.0, current_psro_iter / self.
      ↪ _max_psro_iterations_for_annealing)

      blending_factor = (self._initial_blending_factor * (1.0 - annealing_progress) +
                         self._final_blending_factor * annealing_progress)

      temperature = (self._initial_temperature * (1.0 - annealing_progress) +
                     self._final_temperature * annealing_progress)

      diversity_bonus_coeff = (self._initial_diversity_bonus_coeff * (1.0 - annealing_progress
      ↪ ) +
                               self._final_diversity_bonus_coeff * annealing_progress)

      blending_factor = np.clip(blending_factor, self._final_blending_factor, self.
      ↪ _initial_blending_factor)
      temperature = np.clip(temperature, self._final_temperature, self._initial_temperature)
      diversity_bonus_coeff = np.clip(diversity_bonus_coeff, self._final_diversity_bonus_coeff
      ↪ , self._initial_diversity_bonus_coeff)

      strategies = _hybrid_orm_solver(
          meta_games,
          iterations=solver_iterations, # Use adaptive iterations
          blending_factor=blending_factor,
          temperature=temperature,
          momentum_beta=self._momentum_beta,
          gain_normalization=self._gain_normalization,
          diversity_bonus_coeff=diversity_bonus_coeff,
          return_average_strategy=True # Training always uses averaged strategies for
      ↪ stability
      )
      return [s.tolist() for s in strategies]


class EvalMetaStrategySolver:
```

```
    """Returns meta strategies for evaluation in PSRO.

    This solver uses a hybrid approach, blending Optimistic Regret Matching+
    with a smoothed best pure strategy, tailored for robust and accurate
    exploitability measurement. The parameters are set to emphasize
    exploitation for evaluation purposes, including optimistic updates and
    gain normalization for stability, while keeping diversity bonus minimal.
    Crucially, it returns the *last-iterate* strategy for a reactive estimate
    of exploitability.
    """

    def __init__(self,
                 base_solver_iterations=8000, # Base number of internal iterations
                 iterations_per_policy_scale=50, # How much iterations scale per added policy
                 max_solver_iterations=15000, # Max internal solver iterations
                 blending_factor=0.01,
                 temperature=0.001,
                 momentum_beta=0.2,
                 gain_normalization=True,
                 diversity_bonus_coeff=0.0):
      """Initializes hybrid ORM solver parameters for evaluation meta-strategies.

      Args:
        base_solver_iterations: Base number of internal solver iterations for
        ↪ _hybrid_orm_solver.
        iterations_per_policy_scale: Amount to increase internal solver iterations per added
        ↪ policy.
        max_solver_iterations: Maximum internal solver iterations.
        blending_factor: Weight (0 to 1) for the smoothed best pure strategy component.
        temperature: Temperature for softmax smoothing.
        momentum_beta: Momentum for optimistic updates.
        gain_normalization: Normalizes gains for scale-invariance.
        diversity_bonus_coeff: Diversity bonus, kept very low for evaluation.
      """
      self._base_solver_iterations = base_solver_iterations
      self._iterations_per_policy_scale = iterations_per_policy_scale
      self._max_solver_iterations = max_solver_iterations
      self._blending_factor = blending_factor
      self._temperature = temperature
      self._momentum_beta = momentum_beta
      self._gain_normalization = gain_normalization
      self._diversity_bonus_coeff = diversity_bonus_coeff

    def get_meta_strategy(self, game, policy_sets, meta_games):
      """Returns blended meta strategies for evaluation in policy-space response oracles.

      Args:
        game: The pyspiel game object.
        policy_sets: A list of lists of policies, one list per player.
          policy_sets[p][i] is player p's i-th policy. len(policy_sets[p]) ==
          meta_games[0].shape[p].
        meta_games: A list of n-dimensional numpy arrays, one per player. Each
          array has shape (num_strats_p0, num_strats_p1, ..., num_strats_pn-1) and
          meta_games[p][i0, i1, ..., in-1] is the payoff of player p when player k
          chooses strategy ik.

      Returns:
        A list of mixed-strategies, one for each player. Each mixed strategy is
        a list of non-negative weights (not necessarily normalized). It is used
        for evaluation of the current PSRO policies. E.g., computing
        exploitability.
      """
      del game, policy_sets # Unused

      num_current_policies_p0 = len(meta_games[0]) # Assuming symmetric populations
      solver_iterations = int(self._base_solver_iterations +
                              self._iterations_per_policy_scale * (num_current_policies_p0 -
      ↪ 1))
      solver_iterations = np.clip(solver_iterations, self._base_solver_iterations, self.
      ↪ _max_solver_iterations)

      strategies = _hybrid_orm_solver(
          meta_games,
          iterations=solver_iterations, # Use adaptive iterations
          blending_factor=self._blending_factor,
          temperature=self._temperature,
          momentum_beta=self._momentum_beta,
          gain_normalization=self._gain_normalization,
          diversity_bonus_coeff=self._diversity_bonus_coeff,
          return_average_strategy=False # Eval explicitly requests last-iterate strategy
      )
```

```
        return [s.tolist() for s in strategies]
```

7.2.1 Detailed Specification of VAD-CFR Ablation Components

To understand the contribution of each mechanism discovered by AlphaEvolve within the Volatility-Adaptive Discounted (VAD-) CFR, we conducted a systematic ablation study. Each ablation removes or reverts a specific evolved component to its standard Counterfactual Regret Minimization (CFR) or Discounted CFR (DCFR) equivalent. The following variants correspond to the empirical results presented in Table 1.

Regret Accumulation Variants

• **VAD-CFR (no_asym_discount):** VAD-CFR utilizes highly asymmetric discounting factors for positive and negative regrets. This ablation forces a symmetric baseline discount factor by setting `base_beta = 1.5` (matching `base_alpha`), testing the hypothesis that an aggressive "forget-gate" ($\beta = -0.1$) is necessary for recovery in non-stationary multi-agent environments.
• **VAD-CFR (no_boost):** This variant removes the "Exploration Boost" mechanism by reverting `instantaneous_regret_boost_factor` to 1.0. In the full algorithm, the 1.1 boost accelerates the accumulation of regrets for under-explored, high-potential actions.

Policy Accumulation and Strategy Distillation

• **VAD-CFR (no_prediction):** VAD-CFR uses a non-linear projection when deriving the current iteration policy from accumulated regrets. This ablation reverts the `PolicyFromRegretAccumulator` to a standard linear Regret Matching mapping by setting `positive_policy_exponent = 1.0` and removing the anticipatory `initial_optimism_factor = 0.0`.
• **VAD-CFR (no_warmstart):** One of the most significant discovered features is a hard warm-start delay. This ablation bypasses the `warmup_iterations = 500` condition, forcing the algorithm to begin policy averaging from $t = 1$. This tests whether a "burn-in" period is critical for preventing early, high-variance policies from polluting the final average strategy.

Environmental Adaptation

• **VAD-CFR (no_volatility):** The core dynamic of VAD-CFR is its ability to adjust discounting parameters based on the perceived volatility of the game state, tracked via an Exponential Weighted Moving Average (EWMA) of instantaneous regrets. This ablation disables the `calculate_adaptive_params` logic, replacing the adaptive volatility scaling with static hyperparameters. This directly tests the value of the meta-adaptation dynamics discovered by the LLM versus a fixed discount schedule.

7.2.2 Code of WOP-CFR

See Listing 5.

7.2.3 Detailed Specification of SHOR-PSRO Ablation Components

To understand the contribution of each mechanism discovered by AlphaEvolve within the Smoothed Hybrid Optimistic Regret (SHOR-) PSRO framework, we conducted a comprehensive ablation study. The raw SHOR-PSRO algorithm relies on a highly parameterized, asymmetric design. Each ablation isolates a specific heuristic by either neutralizing its parameters or reverting the logic to standard baselines. The following variants correspond to the empirical results presented in Section 4.4, with direct references to the algorithmic components defined in Listing 4.

Train/Eval Asymmetry Ablations
The raw LLM discovery evolved two distinct, highly specialized solvers for the training-time Oracle generation and the evaluation-time exploitability measurement.

• **SHOR-PSRO (train_as_eval):** This ablation forces the evaluation phase to utilize the `SHORTrainMetaStrategySolver`. It tests whether the exploratory, high-temperature heuris-

Listing 5: WOP-CFR

```
class WOPRegretAccumulator:
  def update_accumulate_regret(self, node, iter_num, cfr_regrets):
    t = float(iter_num + 1)
    d_pos, d_neg = (t**1.5)/(t**1.5 + 1.0), (t**-0.1)/(t**-0.1 + 1.0)
    return {a: (d_pos if node.cumulative_regret[a] >= 0 else d_neg) * node.cumulative_regret
     ↪ [a] + cfr_regrets[a] for a in cfr_regrets}

class WOPPolicyFromRegretAccumulator:
  def get_updated_current_policy(self, node, iter_num, cfr_regrets, previous_policy):
    t = float(iter_num + 1)
    d_pos, d_neg = (t**1.5)/(t**1.5 + 1.0), (t**-0.1)/(t**-0.1 + 1.0)
    opt_s = 1.0 / (1.0 + float(iter_num) / 100.0)
    proj = {}
    for a in node.legal_actions:
      d = d_pos if node.cumulative_regret.get(a, 0.0) >= 0 else d_neg
      r = d * node.cumulative_regret.get(a, 0.0) + opt_s * cfr_regrets.get(a, 0.0)
      proj[a] = max(0.0, r) ** 1.5
    total = sum(proj.values())
    return {a: v/total if total > 0 else 1.0/len(proj) for a, v in proj.items()}

class WOPPolicyAccumulator:
  def update_accumulate_policy(self, node, iter_num, policy, cfr_regrets, reach, cf_reach):
    if iter_num < 500: return node.cumulative_policy # Hard Warm-Start
    weight = (float(iter_num) + 1.0) ** 2.0 # Static Polynomial Weight
    return {a: node.cumulative_policy[a] + weight * reach * policy[a] for a in policy}
```

tics used to generate new policies are fundamentally detrimental when used to measure exact exploitability.

• **SHOR-PSRO (eval_as_train):** This ablation forces the training phase to utilize the `SHOREvalMetaStrategySolver`, stripping away the dynamic annealing and diversity bonuses during population expansion to test if the training solver's complexity is genuinely necessary for exploration.

Blending and Annealing Ablations

SHOR-PSRO's core innovation is the active interpolation between an Optimistic Regret Matching+ (ORM+) loop and a smoothed best pure strategy, guided by automated schedules.

• **SHOR-PSRO (no_blending_pure):** Removes the ORM+ engine entirely by strictly setting `initial_blending_factor=1.0` and `final_blending_factor=1.0`. The solver relies exclusively on the temperature-smoothed pure strategy.

• **SHOR-PSRO (no_blending_orm):** Removes the pure strategy bias by strictly setting `initial_blending_factor=0.0` and `final_blending_factor=0.0`. The solver relies exclusively on the ORM+ update loop.

• **SHOR-PSRO (no_annealing):** Disables the dynamic scaling of hyperparameters across PSRO epochs. It freezes the `blending_factor` at 0.05, `temperature` at 0.01, and `diversity_bonus_coeff` at 0.001, preventing the solver from transitioning from exploration to exploitation as the population grows.

• **SHOR-PSRO (fixed_iterations):** Disables the `iterations_per_policy_scale` mechanism, testing whether scaling the internal solver iterations dynamically with the size of the empirical game tensor is a critical mechanism or an over-parameterized heuristic.

ORM Engine Heuristic Ablations

The underlying ORM+ loop discovered by the LLM contains several specialized micro-mechanics intended to accelerate convergence.

• **SHOR-PSRO (no_diversity):** Removes the uniform exploration incentive by strictly setting `diversity_bonus_coeff=0.0`.

• **SHOR-PSRO (no_momentum):** Disables the optimistic update of the payoff gains by setting `momentum_beta=0.0`, reverting the update rule closer to standard Regret Matching.

• **SHOR-PSRO (no_gain_norm):** Sets `gain_normalization=False`, disabling the adaptive scaling of the optimistic gains matrix, which tests the algorithm's vulnerability to extreme payoff magnitudes.

Table 2: Exhaustive Algorithmic Audit of SHOR-PSRO Components. Results are reported as IQM Log-Improvement Scores (s_g) relative to the raw discovery. Test IQM is decomposed into within-family scaling (11 games) and out-of-family transfer (3 games).

Variant	Total IQM	Train IQM	Test (All)	Test (Within)	Test (Out-of-Family)
Distilled Solver					
PM-PSRO	**+0.032**	**-0.038**	**+0.059**	**+0.062**	**+0.514**
Train/Eval Asymmetry Ablations					
train_as_eval	+0.048	+0.224	+0.021	+0.024	-0.028
eval_as_train	-0.557	-0.584	-0.541	-0.493	-15.808
Blending & Annealing Ablations					
no_blending_pure (Removes ORM entirely)	**-1.525**	**-2.292**	**-1.268**	**-1.391**	**-0.855**
no_blending_orm (Removes pure blend)	+0.012	-0.069	+0.020	+0.023	**+0.351**
no_annealing	+0.013	-0.104	+0.016	+0.024	-0.053
fixed_iterations	-0.038	-0.070	-0.031	-0.009	-16.041
ORM Engine Heuristic Ablations					
no_diversity	+0.007	-0.015	+0.012	+0.019	-0.020
no_momentum	-0.017	-0.018	-0.016	+0.006	-0.152
no_gain_norm	-0.012	+0.012	-0.017	-0.013	-0.078

7.2.4 CODE OF PM-PSRO

See Listing 6.

7.3 EVALUATED GAME SUITE

We evaluate all algorithms on a suite of 18 game instances from the OpenSpiel framework (Lanctot et al., 2019), spanning five game families. The suite is designed to cover a wide range of structural properties: sequential vs. simultaneous moves, 2–4 players, and game trees ranging from 2 to 347,810 information states with branching factors from 1 to 66. All games in our suite involve imperfect information. Table 3 provides a summary; detailed descriptions follow.

Table 3: Summary of the 18 evaluated game instances. All games involve imperfect information. "Seq" = sequential, "Sim" = simultaneous (converted to sequential via `turn_based_simultaneous_game`). Information state counts and branching factors are exact, computed by full game tree traversal.

Game Instance	Family	Players	Type	Info States	Avg. Actions	Action Range
Kuhn Poker	Poker	2	Seq	12	2.0	2
Kuhn Poker (3p)	Poker	3	Seq	48	2.0	2
Kuhn Poker (4p)	Poker	4	Seq	160	2.0	2
Leduc Poker	Poker	2	Seq	936	2.3	2–3
Leduc Poker (3p)	Poker	3	Seq	25,800	2.3	2–3
Universal Poker	Poker	2	Seq	20,160	2.4	2–4
Goofspiel-3	Goofspiel	2	Sim	114	2.1	2–3
Goofspiel-4	Goofspiel	2	Sim	6,056	2.1	2–4
Goofspiel-5	Goofspiel	2	Sim	347,810	2.1	2–5
Goofspiel-3 (limited info)	Goofspiel	2	Sim	90	2.1	2–3
Goofspiel-4 (limited info)	Goofspiel	2	Sim	3,608	2.1	2–4
Goofspiel-5 (limited info)	Goofspiel	2	Sim	236,450	2.1	2–5
Liar's Dice (4-sided)	Liar's Dice	2	Seq	1,024	2.0	1–8
Liar's Dice (5-sided)	Liar's Dice	2	Seq	5,120	2.0	1–10
Liar's Dice (6-sided)	Liar's Dice	2	Seq	24,576	2.0	1–12
Colonel Blotto	Blotto	2	Sim	2	66.0	66
Battleship (2×2)	Battleship	2	Seq	10,198	4.0	4
Battleship (3×2)	Battleship	2	Seq	160,491	6.0	6–7

Listing 6: PM-PSRO.

```python
import numpy as np

class PMMetaStrategySolver:
    """Projection Matching (PM) PSRO Solver.
    A minimal meta-solver utilizing geometric tangent
    projection, and a dynamic iteration budget.
    """
    def __init__(self, base_iterations=1000, iter_scale=20):
        self.base_iterations = base_iterations
        self.iter_scale = iter_scale

    def get_meta_strategy(self, game, policy_sets, meta_games):
        num_players = len(meta_games)
        num_strats = [m.shape[i] for i, m in enumerate(meta_games)]
        num_current_policies = num_strats[0]
        iterations = self.base_iterations + self.iter_scale * (num_current_policies - 1)
        strategies = [np.ones(s) / s for s in num_strats]
        duals = [np.zeros(s) for s in num_strats]
        avg_strategies = [np.zeros(s) for s in num_strats]
        for _ in range(iterations):
            for p in range(num_players):
                payoff_vec = meta_games[p]
                for other_p in reversed(range(num_players)):
                    if other_p != p:
                        payoff_vec = np.tensordot(
                            payoff_vec, strategies[other_p], axes=([other_p], [0])
                        )

                projected_grad = payoff_vec - np.mean(payoff_vec)
                duals[p] += projected_grad
                positive_duals = np.maximum(duals[p], 0)
                sum_duals = np.sum(positive_duals)
                if sum_duals > 0:
                    strategies[p] = positive_duals / sum_duals
                else:
                    strategies[p] = np.ones(num_strats[p]) / num_strats[p]
                avg_strategies[p] += strategies[p]

        final_strategies = []
        for avg, s in zip(avg_strategies, num_strats):
            sum_avg = np.sum(avg)
            if sum_avg > 0:
                final_strategies.append((avg / sum_avg).tolist())
            else:
                final_strategies.append((np.ones(s) / s).tolist())
        return final_strategies
```

7.3.1 POKER VARIANTS

Kuhn Poker (Kuhn, 1950). A simplified poker game played with a three-card deck (Jack, Queen, King). Each player is dealt one card and places a single ante. Players then engage in a single round of betting with check/bet/call/fold actions. Despite its simplicity, Kuhn Poker has a non-trivial mixed Nash equilibrium and serves as a canonical benchmark for imperfect-information game-solving algorithms. We evaluate on the standard 2-player version (12 information states) and extended variants with 3 players (48 information states, 4-card deck) and 4 players (160 information states, 5-card deck).

Leduc Poker (Southey et al., 2005). A two-round poker game using a deck of six cards (two suits $\times$ three ranks). Each player is dealt a single private card, followed by a round of betting; then a single community card is dealt, followed by a second betting round. Hands are ranked by pair (private card matches community card) and then by card rank. Leduc Poker is a standard benchmark for imperfect-information algorithms as it is small enough for exact Nash equilibrium computation yet complex enough to test key algorithmic properties. We evaluate on the 2-player version (936 information states) and a 3-player variant (25,800 information states).

Universal Poker. OpenSpiel's parameterized poker game that can represent arbitrary limit and no-limit hold'em variants through the ACPC (Annual Computer Poker Competition) protocol. In its default configuration, it implements a small limit hold'em game with a 4-card deck, a single betting round, and fixed bet sizes. With 20,160 information states, it provides a more complex poker benchmark than Leduc while remaining computationally tractable for tabular methods.

7.3.2 GOOFSPIEL (THE GAME OF PURE STRATEGY)

Goofspiel (Ross, 1971). A simultaneous-move card game in which two players compete to win "prize" cards using their "bid" cards. Each player holds a hand of cards numbered 1 through n, and a sequence of prize cards (also 1 through n) is revealed one at a time. In each round, both players simultaneously play a bid card; the higher bid wins the current prize (ties result in the prize being discarded). The player who accumulates the highest total prize value wins. Since players choose actions simultaneously, the game inherently involves imperfect information—each player must commit to a bid without knowing the opponent's choice. We convert the simultaneous-move game to sequential form via the `turn_based_simultaneous_game` wrapper for compatibility with CFR-based algorithms.

We evaluate on six variants organized along two dimensions:

- **Scale:** $n \in \{3, 4, 5\}$ cards, yielding progressively larger game trees (114 to 347,810 information states).
- **Information structure:** In the standard version, both players observe each other's bids after each round. In the *limited information* version (`imp_info=True`), players only observe whether they won or lost each round, but not the opponent's specific bid. Interestingly, the limited-information variant has *fewer* information states (e.g., 236,450 vs. 347,810 for n=5) because more game histories are observationally equivalent from each player's perspective.

7.3.3 LIAR'S DICE

Liar's Dice (Lisý et al., 2015). A bluffing game in which each player secretly rolls a single die, then players alternate making increasingly higher claims about the total count of a specific face value across *all* dice. A player may either raise the current claim or challenge ("call liar") the previous player's claim. If challenged, all dice are revealed: the challenger wins if the claim was false, and the claimant wins if it was true. We evaluate on variants with $d \in \{4, 5, 6\}$ die faces, yielding 1,024, 5,120, and 24,576 information states respectively.

7.3.4 COLONEL BLOTTO

Colonel Blotto (Borel, 1953). A classic resource-allocation game in which two players simultaneously distribute a fixed budget of units across multiple battlefields. Each battlefield is won by the player who allocates more units to it (ties broken randomly). The player who wins the most battlefields

wins the game. In the default OpenSpiel configuration, each player has 10 units to distribute across 3 battlefields (66 possible allocations per player). As a single simultaneous decision, the game has only 2 information states (one per player) but a rich 66×66 payoff matrix with complex mixed Nash equilibria. The game is converted to sequential form via `turn_based_simultaneous_game`.

7.3.5 BATTLESHIP

Battleship (Farina et al., 2019). A parametric two-player game inspired by the classic board game, formalized for game-theoretic analysis. Each player secretly places ships on a grid, then players alternate taking shots to locate and sink the opponent's ships. The game has imperfect information (players do not observe the opponent's ship placement) and is parameterized by board dimensions, ship configurations, and scoring rules. We evaluate on two configurations:

- **Battleship 2×2:** A 2×2 board with one ship of size 2 and value 2.0, with 3 shots allowed (10,198 information states).
- **Battleship 3×2:** A 3×2 board with the same ship and shot configuration (160,491 information states).

7.3.6 GAME SELECTION RATIONALE

The suite is designed to stress-test algorithm performance across several axes:

1. **Number of players:** Two-player (most games) vs. multi-player (Kuhn 3p/4p, Leduc 3p).
2. **Game tree scale:** From 2 information states (Blotto) to 347,810 (Goofspiel-5), testing scalability.
3. **Strategic depth:** From pure bluffing (poker, Liar's Dice) to resource allocation (Blotto) to bidding under uncertainty (Goofspiel).
4. **Move structure:** Sequential (poker, Liar's Dice, Battleship) vs. simultaneous converted to sequential (Goofspiel, Blotto).
5. **Observability:** Varying degrees of information—from fully observable bids (Goofspiel) to hidden bids with outcome feedback only (Goofspiel limited info) to hidden card placement (Battleship).

7.4 RESULTS ON 18 GAMES

Figures 3 and 4 show full per-game exploitability curves for all 18 games in our evaluation suite for CFR and PSRO variants respectively.

7.5 ADDITIONAL PSRO EVALUATION ON RANDOMLY GENERATED NORMAL-FORM GAMES

To complement the OpenSpiel benchmark with an evaluation that is structurally independent of any specific extensive-form game, we evaluate all PSRO meta-solvers on randomly generated N-player normal-form games (NFGs). This setting tests the meta-solver itself in isolation, removing any influence of the underlying extensive-form structure that AlphaEvolve might have implicitly overfit to during training. Sampling 1000 NFGs per configuration also provides a large-sample statistical complement to the deterministic per-game evaluation in the main body.

Game generation. For each configuration (N, A), we sample 1000 random N-player NFGs with A actions per player. Each game is constructed by drawing independent payoff tensors $u_i \in \mathbb{R}^{A^N}$ for each player from either $\mathcal{N}(0, 1)$ (Gaussian) or $\mathcal{U}[-1, 1]$ (Uniform), and then centering across players to enforce constant-sum structure:

$$u_i \leftarrow u_i - \tfrac{1}{N} \sum\nolimits_{j=1}^{N} u_j.$$

After centering, $\sum_i u_i(a) = 0$ for every joint action profile a, generalizing two-player zero-sum to the N-player setting. Distinct random seeds are used across the 1000 sampled games and the two distributions.

Why $N \geq 3$. We restrict the NFG evaluation to $N \geq 3$ because two-player zero-sum NFGs admit a polynomial-time exact solution via linear programming. In that regime the LP-Nash baseline trivially

dominates all other meta-solvers and the comparison contains no signal about meta-solver design choices.

Experimental setup. Each PSRO run starts from a single random pure strategy per player, expands the population by one exact best response per iteration, and terminates after $K = A$ outer iterations (full population growth). We benchmark Uniform, AlphaRank, PRD, RM, SHOR-PSRO, and PM-PSRO under identical conditions; the LP-Nash baseline is omitted as it does not extend to $N \geq 3$. Following Agarwal et al. (2021), we report the per-iteration interquartile mean (IQM) of exploitability across the 1000 sampled games, with 95% confidence intervals computed by 50,000 bootstrap resamples at the run level (i.e., resampling games rather than individual iterations, which preserves within-trajectory correlation).

Results. Figures 5 and 6 show per-iteration IQM exploitability under Gaussian and Uniform payoffs respectively, across six (N, A) configurations spanning 3-, 4-, and 5-player games with A ranging from 20 to 100 actions. The qualitative pattern is identical across all configurations and both distributions: SHOR-PSRO and PM-PSRO consistently reach lower exploitability than every established baseline, with the gap widening as iteration count grows. Established baselines (Uniform, AlphaRank, PRD, and RM) plateau early — typically before iteration $A/2$ — while SHOR-PSRO and PM-PSRO continue to descend. PM-PSRO begins more slowly during the first ~ 5 iterations, since its tangent-projected utility signal requires several iterations to accumulate, but matches or surpasses SHOR-PSRO by mid-run in the larger configurations (3P/50A, 3P/100A, 4P/50A). The 95% bootstrap CIs are tighter than line width across most of each curve, indicating that the observed gaps are statistically significant rather than artifacts of game-level variance. Insensitivity to the payoff distribution (Gaussian vs. Uniform results are qualitatively identical) suggests that the discovered solvers' advantage is not specific to any particular generative model.

7.6 PROMPTS

Listing 7: Prompt for Evolving CFR

````
Act as an expert in game theory, multiagent learning, online learning and optimization. Your
    ↪ task is to iteratively improve a new variant of counterfactual regret minimization.
    ↪ The primary goal is to speed up convergence to low-exploitable strategies.

Always adhere to best practices in Python coding.


A key data structure that is used is infostate_node:

```python
@attr.s
class _InfoStateNode(object):
  """An object wrapping values associated to an information state."""
  # The list of the legal actions.
  legal_actions = attr.ib()
  index_in_tabular_policy = attr.ib()
  # Map from information states string representations and actions to the
  # counterfactual regrets, accumulated over the policy iterations
  cumulative_regret = attr.ib(factory=lambda: collections.defaultdict(float))
  # Same as above for the cumulative of the policy probabilities computed
  # during the policy iterations
  cumulative_policy = attr.ib(factory=lambda: collections.defaultdict(float))
```

You are allowed to modify three key components of CFR: (1) how are the regret values
    ↪ accumulated at each infoset (RegretAccumulator) (2) how to obtain a current policy
    ↪ at the current iteration from the current cumulative_regret (
    ↪ PolicyFromRegretAccumulator) and (3) how to accumulate policies across iterations to
    ↪  compute an average policy (PolicyAccumulator).

# Prior programs
Previously we found that the following programs performed well on the task at hand:

{previous_programs}

# Current program
Here is the current program we are trying to improve (you will need to propose a
    ↪ modification to it below):
````

```
{code}

# *SEARCH/REPLACE block* Rules:

Every *SEARCH/REPLACE block* must use this format:
1. The opening fence: ```python
2. The start of search block: <<<<<<< SEARCH
3. A contiguous chunk of up to 4 lines to search for in the existing source code
4. The dividing line: =======
5. The lines to replace into the source code
6. The end of the replace block: >>>>>>> REPLACE
7. The closing fence: ```

Every *SEARCH* section must *EXACTLY MATCH* the existing file content, character for
    ↪ character, including all comments, docstrings, etc.

*SEARCH/REPLACE* blocks will replace *all* matching occurrences.
Include enough lines to make the SEARCH blocks uniquely match the lines to change.

Keep *SEARCH/REPLACE* blocks concise.
Break large *SEARCH/REPLACE* blocks into a series of smaller blocks that each change a small
    ↪  portion of the file.
Include just the changing lines, and a few surrounding lines if needed for uniqueness.
Do not include long runs of unchanging lines in *SEARCH/REPLACE* blocks.

To move code within a file, use 2 *SEARCH/REPLACE* blocks: 1 to delete it from its current
    ↪ location, 1 to insert it in the new location.

Make sure that the changes you propose are consistent with each other. For example, if you
    ↪ refer to a new config variable somewhere, you should also propose a change to add
    ↪ that variable.

Example:
```python
<<<<<<< SEARCH
    return total_loss
=======
    # Add sparsity-promoting regularization to the loss.
    total_loss += self.hypers.l1_reg_weight * l1_reg

    return total_loss
{replace}
```
and
```python
<<<<<<< SEARCH
  return hyper.zipit([
=======
  return hyper.zipit([
      hyper.uniform('l1_reg_weight', hyper.interval(0.0, 0.01)),
{replace}
```

{lazy_prompt}
ONLY EVER RETURN CODE IN A *SEARCH/REPLACE BLOCK*!

Task
{task_instruction} {focus_sentence} {trigger_chain_of_thought}
Describe each change with a *SEARCH/REPLACE block*.
```

Listing 8: Prompt Config for Evolving CFR

```
{
  "template": {
    "prompt_sampler_type": "TemplatePromptSampler",
    "probability": 1.0,
    "template_path": "meta_cfr_prompt.md",
    "format_args": {
      "lazy_prompt": [
        ["\nYou are diligent and tireless!\nYou NEVER leave comments describing code without
    ↪  implementing it!\nYou always COMPLETELY IMPLEMENT the needed code!", 0.5],
        ["", 0.5]
      ],
      "task_instruction": [
        ["Propose modifications to current program that combine the strengths of all the
    ↪ programs above and achieved high scores on the task.", 0.25],
        ["Suggest an unconventional modification to improve our implementation.", 0.25],
        ["Suggest a new idea to improve the code.", 0.25],
```

```
        ["Suggest a new idea to improve the code that is inspired by your expert knowledge
    ↪ of game theory, multiagent reinforcement learning, economics, stochastic processes,
    ↪ machine learning, and optimization.", 0.25]
      ],
      "focus_sentence": [
        ["Focus on simplifying the code, instead of adding new functionality.", 0.2],
        ["", 0.8]
      ],
      "trigger_chain_of_thought": [
        ["\n\nStart with providing a comprehensive explanation for the proposed changes
    ↪ including\n* The specific issue or limitation it addresses.\n* The underlying
    ↪ rationale and expected impact.\n\nYou need to specify this *before providing code*.\
    ↪ n\n", 0.5],
        ["\n\n", 0.5]
      ]
    }
  }
}
```

Listing 9: Prompt for Evolving PSRO

```
Act as an expert in game theory, multiagent learning, online learning and optimization. Your
    ↪  task is to iteratively improve a variant of Policy-Space Response Oracles (PSRO).
    ↪ The primary goal is to speed up convergence to low-exploitable strategies.

# PSRO Overview

PSRO iteratively builds a population of policies for each player and computes meta-
    ↪ strategies (distributions over policies) to guide training and evaluation.

**Each iteration:**
1. **Empirical game**: Simulate payoffs between all policy combinations to form a game
    ↪ tensor.
2. **Train-time meta-strategy**: Compute a distribution over current policies for each
    ↪ player. This determines what opponents the best-response oracle trains against.
3. **Best response**: Add a new policy for each player that best responds to opponents'
    ↪ train-time meta-strategies.
4. **Eval-time meta-strategy**: Compute a (possibly different) distribution for evaluation,
    ↪ e.g., to measure exploitability.

**Your task**: Improve both the **train-time** and **eval-time** meta-strategy solvers.
    ↪ These serve different purposes: train-time guides population growth, eval-time
    ↪ assesses solution quality.

# Available Utilities

**Best Response**
- `BestResponsePolicy(game, player_id, policy)`: Returns a best-response policy for `
    ↪ player_id` against `policy`.
- Use `.value(game.new_initial_state())` to get the BR value.

**Policy Aggregation**
- `PolicyAggregator(game).aggregate(pids, policy_sets, weights)`: Creates a mixed policy
    ↪ from weighted pure policies.
  - `pids`: `list(range(game.num_players()))`
  - `weights`: `[weights_p0, weights_p1, ...]`, each matching `len(policy_sets[p])`

**Policy Evaluation**
- `expected_game_score.policy_value(state, joint_policy)`: Returns expected payoffs (list,
    ↪ one per player).
  - `joint_policy`: `[policy_p0, policy_p1, ...]`

Always adhere to best practices in Python coding.

# Prior programs
Previously we found that the following programs performed well on the task at hand:

{previous_programs}

# Current program
Here is the current program we are trying to improve (you will need to propose a
    ↪ modification to it below):

{code}

# *SEARCH/REPLACE block* Rules:

Every *SEARCH/REPLACE block* must use this format:
1. The opening fence: ```python
```

```
2. The start of search block: <<<<<<< SEARCH
3. A contiguous chunk of up to 4 lines to search for in the existing source code
4. The dividing line: =======
5. The lines to replace into the source code
6. The end of the replace block: >>>>>>> REPLACE
7. The closing fence: ```

Every *SEARCH* section must *EXACTLY MATCH* the existing file content, character for
    ↪ character, including all comments, docstrings, etc.

*SEARCH/REPLACE* blocks will replace *all* matching occurrences.
Include enough lines to make the SEARCH blocks uniquely match the lines to change.

Keep *SEARCH/REPLACE* blocks concise.
Break large *SEARCH/REPLACE* blocks into a series of smaller blocks that each change a small
    ↪  portion of the file.
Include just the changing lines, and a few surrounding lines if needed for uniqueness.
Do not include long runs of unchanging lines in *SEARCH/REPLACE* blocks.

To move code within a file, use 2 *SEARCH/REPLACE* blocks: 1 to delete it from its current
    ↪ location, 1 to insert it in the new location.

Make sure that the changes you propose are consistent with each other. For example, if you
    ↪ refer to a new config variable somewhere, you should also propose a change to add
    ↪ that variable.

Example:
```python
<<<<<<< SEARCH
    return total_loss
=======
    # Add sparsity-promoting regularization to the loss.
    total_loss += self.hypers.l1_reg_weight * l1_reg

    return total_loss
{replace}
```
and
```python
<<<<<<< SEARCH
  return hyper.zipit([
=======
  return hyper.zipit([
      hyper.uniform('l1_reg_weight', hyper.interval(0.0, 0.01)),
{replace}
```

{lazy_prompt}
ONLY EVER RETURN CODE IN A *SEARCH/REPLACE BLOCK*!

# Task
{task_instruction} {focus_sentence} {difficulty_hint} {exploration_nudge} {
    ↪ trigger_chain_of_thought}
Describe each change with a *SEARCH/REPLACE block*.
```

Listing 10: Prompt Config for Evolving PSRO

```
{
  "template": {
    "prompt_sampler_type": "TemplatePromptSampler",
    "probability": 1.0,
    "template_path": "meta_psro_prompt.md",
    "format_args": {
      "lazy_prompt": [
        ["\nYou are diligent and tireless!\nYou NEVER leave comments describing code without
    ↪  implementing it!\nYou always COMPLETELY IMPLEMENT the needed code!", 0.5],
        ["", 0.5]
      ],
      "task_instruction": [
        ["Propose modifications to current program that combine the strengths of all the
    ↪ programs above and achieved high scores on the task.", 0.25],
        ["Suggest an unconventional modification to improve our implementation.", 0.25],
        ["Suggest a new idea to improve the code.", 0.25],
        ["Suggest a new idea to improve the code that is inspired by your expert knowledge
    ↪ of game theory, multiagent reinforcement learning, economics, stochastic processes,
    ↪ machine learning, and optimization.", 0.25]
      ],
      "focus_sentence": [
        ["Focus on simplifying the code, instead of adding new functionality.", 0.2],
```

```
          ["", 0.8]
        ],
        "trigger_chain_of_thought": [
          ["\n\nStart with providing a comprehensive explanation for the proposed changes
      ↪ including\n* The specific issue or limitation it addresses.\n* The underlying
      ↪ rationale and expected impact.\n\nYou need to specify this *before providing code*.\
      ↪ n\n", 0.5],
          ["\n\n", 0.5]
        ]
      }
    }
  }
```

7.7 Recurrence of Discovered Mechanisms Across Independent Runs

The algorithms reported in the main text are deterministic artifacts, and their performance claims are validated on held-out games rather than on the search process that produced them. Nevertheless, because AlphaEvolve relies on stochastic LLM sampling, it is natural to ask whether the components our ablation audit distills are stable properties of the search or accidents of one trajectory. This section reports an analysis of ten further independent AlphaEvolve trajectories drawn from our broader experimentation — five in the CFR design space and five in the PSRO design space — run under the same setup, prompts, training games, and fitness definition as the reported runs, and differing only in the random seed of the evolutionary process.

For each trajectory we took the best program by average fitness and classified it along two axes: which of the core mechanisms identified by our audit it contains, and what additional secondary machinery it wraps around them. The result is summarized in Tables 4 and 5.

CFR trajectories. All five runs independently arrive at the same three mechanisms that our ablation audit distills for WOP-CFR. Every run discovers an aggressive treatment of negative regret, ranging from a hard floor at zero through to negative discount exponents; two runs ($R4$, $R5$) independently push the exponent negative, reaching the same "mathematical extreme" as VAD-CFR's forget-gate. Every run discovers an optimistic or predictive policy projection, and two runs ($R1$, $R4$) rediscover the power exponent 1.5 exactly. Every run suppresses the contribution of early policies to the average, three of them ($R1$, $R4$, $R5$) via an explicit hard warm-start with a fixed cutoff; the code comments accompanying these warm-starts give essentially the rationale we state in Section 4.3, namely avoiding the pollution of the average strategy with noisy policies from early, unstable iterations. The secondary machinery, in contrast, is different in every single run.

PSRO trajectories. The five PSRO runs span five structurally different solver families (fictitious play, projected replicator dynamics, regret matching$^+$ ensembles, Hedge, and a belief-based PRD variant), and yet all five contain a no-regret inner loop equipped with an optimistic or predictive update. Two runs reproduce PM-PSRO's mean-centered utility signal verbatim, and the remaining three use a close variant of it (division by the standard deviation, or strategy-weighted centering). As with CFR, the secondary machinery differs in every run.

Interpretation. Taken together, the ten trajectories support the paper's central claim as an empirical regularity rather than a single observation: the evolutionary search reliably converges on a small, stable algorithmic core, and reliably wraps that core in run-specific machinery that does not recur. This is precisely the structure our train-test audit is designed to detect, and it predicts that future discovery runs in these design spaces will produce different-looking programs that nonetheless reduce to the same cores under audit.

7.8 Behavior Beyond the Evaluation Horizon

The main text evaluates all CFR variants at a fixed horizon of $K = 1000$ iterations and all PSRO variants at $K = 100$ PSRO iterations. Since the discovered algorithms were selected by fitness measured at exactly these horizons, a natural concern is whether they are tuned to the horizon rather than convergent — that is, whether they are principled solvers or strong fitted heuristics. This section reports their behavior well beyond the horizon at which they were selected.

Table 4: **Recurrence of core mechanisms across five independent CFR trajectories** ($R1$–$R5$), compared with the reported VAD-CFR run. The core mechanisms recur in all five runs; the secondary machinery is distinct in each.

Core mechanism	R1	R2	R3	R4	R5	VAD (paper)
Aggressive neg.-regret forgetting	β_{neg}=0	floor at 0	β_{neg}=0.5	$\beta_{neg}\to-2.0$	$\beta_{neg}\to-1.5$	β=−0.1
Optimistic/predictive projection	yes	yes	yes	yes	yes	yes
Power exponent	1.5	—	—	1.5	2.0	1.5
Early-policy suppression	hard T=100	soft	soft	hard T=200	hard T=20	hard T=500
Secondary machinery (all distinct)	dual EMA	var. clipping	curiosity	tanh sharpen	KL damping	volatility EWMA

Table 5: **Recurrence of core mechanisms across five independent PSRO trajectories** (A–E), which span five structurally different solver families, compared with the reported SHOR-PSRO run. A no-regret inner loop with an optimistic update appears in all five; the secondary machinery is distinct in each.

Core mechanism	A (FP)	B (PRD)	C (RM+ ens.)	D (Hedge)	E (belief-PRD)	SHOR (paper)
No-regret inner loop	RM+	PRD+rectif.	RM+	Hedge	belief-PRD+rectif.	ORM+
Optimistic/predictive update	yes	yes	yes	yes	yes (belief EWMA)	yes
Mean-centered utilities	verbatim	÷std variant	strategy-wtd.	implicit	strategy-wtd.	verbatim
Secondary machinery (all distinct)	velocity EMAs	stagnation amp.	risk-aversion ens.	restricted-game pruning	trend-triggered expl.	annealing+blending

WOP-CFR. We extended WOP-CFR to $K = 10{,}000$ iterations, ten times the reported horizon, on three of the hardest test games (Table 6). Exploitability continues to decrease by roughly another order of magnitude, with no plateau and no divergence. The only non-monotonicity is a minor fluctuation on 3-player Leduc Poker between $K = 5{,}000$ and $K = 10{,}000$ at the 2×10^{-5} level; the value at $K = 10{,}000$ remains $3.5\times$ below the value at the reported horizon, and fluctuations of this magnitude are consistent with the small oscillations that CFR-family average strategies exhibit throughout Figures 1 and 3.

Table 6: **WOP-CFR exploitability beyond the evaluation horizon.** The reported horizon is $K = 1000$.

Game	$\mathbf{K = 1000}$ **(paper)**	$\mathbf{K = 2000}$	$\mathbf{K = 5000}$	$\mathbf{K = 10000}$
Leduc Poker (3p)	6.0e-5	3.9e-5	1.3e-5	1.7e-5
Goofspiel-5	1.1e-4	4.6e-5	1.1e-5	4.3e-6
Liar's Dice (6-sided)	2.2e-4	9.4e-5	4.3e-5	1.9e-5

PM-PSRO. We extended PM-PSRO to $K = 300$ PSRO iterations, three times the reported horizon, on the same three games (Table 7). Exploitability decreases strictly monotonically throughout, roughly halving with each extension of the horizon.

Partial theoretical statements. We do not have a full convergence rate proof for either distilled solver, but two partial theoretical connections can be established. First, WOP-CFR's positive-regret discount $d_t^{(+)} = t^{1.5}/(t^{1.5} + 1)$ lies within the weighted-regret family analyzed for DCFR-style algorithms (Brown & Sandholm, 2019b); the nonstandard elements are the negative-exponent forget-gate applied to negative regrets and the 1.5-power projection used to derive the policy. We conjecture that the combination admits analysis as a predictive regret-matching variant (Farina et al., 2021) with time-varying weights, and leave a formal rate proof as an open problem. Second, PM-PSRO is a PSRO variant whose meta-solver only affects which best responses are added to the population; it therefore inherits the worst-case guarantee of the double-oracle family (Zhang & Sandholm, 2024), in that the full-game equilibrium is recovered once all strategy sets are fully expanded. Neither statement establishes a finite-time rate, and we present the extended-horizon results above as empirical evidence.

7.9 Compute Resources and LLM Cost

Our pipeline has two phases with very different cost profiles: In the *search phase*, the LLM acts as the proposal operator that generates candidate code; this is where all LLM usage occurs. In the *audit-*

Table 7: **PM-PSRO exploitability beyond the evaluation horizon.** The reported horizon is $K = 100$ PSRO iterations.

Game	$\mathbf{K = 100}$ **(paper)**	$\mathbf{K = 200}$	$\mathbf{K = 300}$
Leduc Poker (3p)	1.1e-1	4.0e-2	1.9e-2
Goofspiel-5	2.2e-1	9.4e-2	5.3e-2
Liar's Dice (6-sided)	6.4e-2	1.9e-2	7.4e-3

and-distillation phase, the discovered algorithm is a static program that is executed deterministically over exact game-tree evaluations: no code is generated or mutated, so this phase makes no LLM calls at all and its cost is purely the CPU cost of running the ablations. The audit methodology we contribute therefore adds negligible cost on top of any discovery run, though the discovery run it audits is of course LLM-driven.

Table 8 itemizes both reported runs, sliced up to the checkpoint at which the reported algorithm appeared. All candidate evaluation is CPU-only; no GPUs or TPUs were used at any point. Prompt counts include retries under rate limiting, and token counts are estimated from logged character volumes at approximately four characters per token. Monetary figures are computed at public list prices at the time of writing ($1.25 per million input tokens, $10 per million output tokens) without any prompt-caching discount, and are therefore upper estimates.

Table 8: **Search cost of the two reported AlphaEvolve runs** (Gemini 2.5 Pro, accessed via API), measured up to the checkpoint at which the reported algorithm appeared.

	CFR run (VAD-CFR)	**PSRO run (SHOR-PSRO)**
Candidate programs evaluated	8,401	12,342
Valid-execution rate	23.6%	10.3%
Prompts sent (incl. retries)	12,255	28,071
Successful generations	8,401	12,342
Input tokens (est.)	≈70M	≈140M
Output tokens (est.)	≈21M	≈54M
Cost at list prices (est.)	≈$290	≈$720
CPU-hours	≈2,260	≈16,100
Wall-clock time	≈2.4 days	≈4.1 days
Parallel evaluators	≈47	≈210

7.10 Extension to Deep Multi-Agent Architectures

While this work scopes evaluation to tabular games with exact oracles to cleanly isolate algorithmic mechanisms from neural approximation noise, both distilled update rules can be theoretically mapped to deep multi-agent reinforcement learning architectures:

Deep CFR Integration. In Deep CFR (Brown et al., 2019), separate neural networks approximate advantage values (instantaneous counterfactual regrets) $v^t(I, a)$ at each iteration t, and policy networks approximate the average strategy. WOP-CFR's update rule translates directly to sample weighting in the advantage replay buffer: positive advantage regression targets are weighted by $d_t^{(+)} = \frac{t^{1.5}}{t^{1.5}+1}$, while negative advantages are discounted by $d_t^{(-)} = \frac{t^{-0.1}}{t^{-0.1}+1}$. The hard warm-start translates to delaying training of the average policy network until iteration $T = 500$, training it with sample weights $w_t = t^2$ thereafter.

Neural PSRO Integration. In neural PSRO architectures (such as Pipeline PSRO (Mcaleer et al., 2020)), the meta-solver operates purely on the empirical payoff tensor constructed from evaluated matchups between population members, while the best-response oracle is implemented via deep RL (e.g., PPO or DQN). Because PM-PSRO's tangent projection $g_i^t = (I - \frac{1}{|\Pi_i|}\mathbf{1}\mathbf{1}^\top)u_i^t$ operates strictly at the meta-game level on the empirical payoff matrix, it acts as a drop-in replacement for PRD or

AlphaRank meta-solvers, completely independent of whether the underlying best-response oracle is tabular value iteration or an RL policy network.

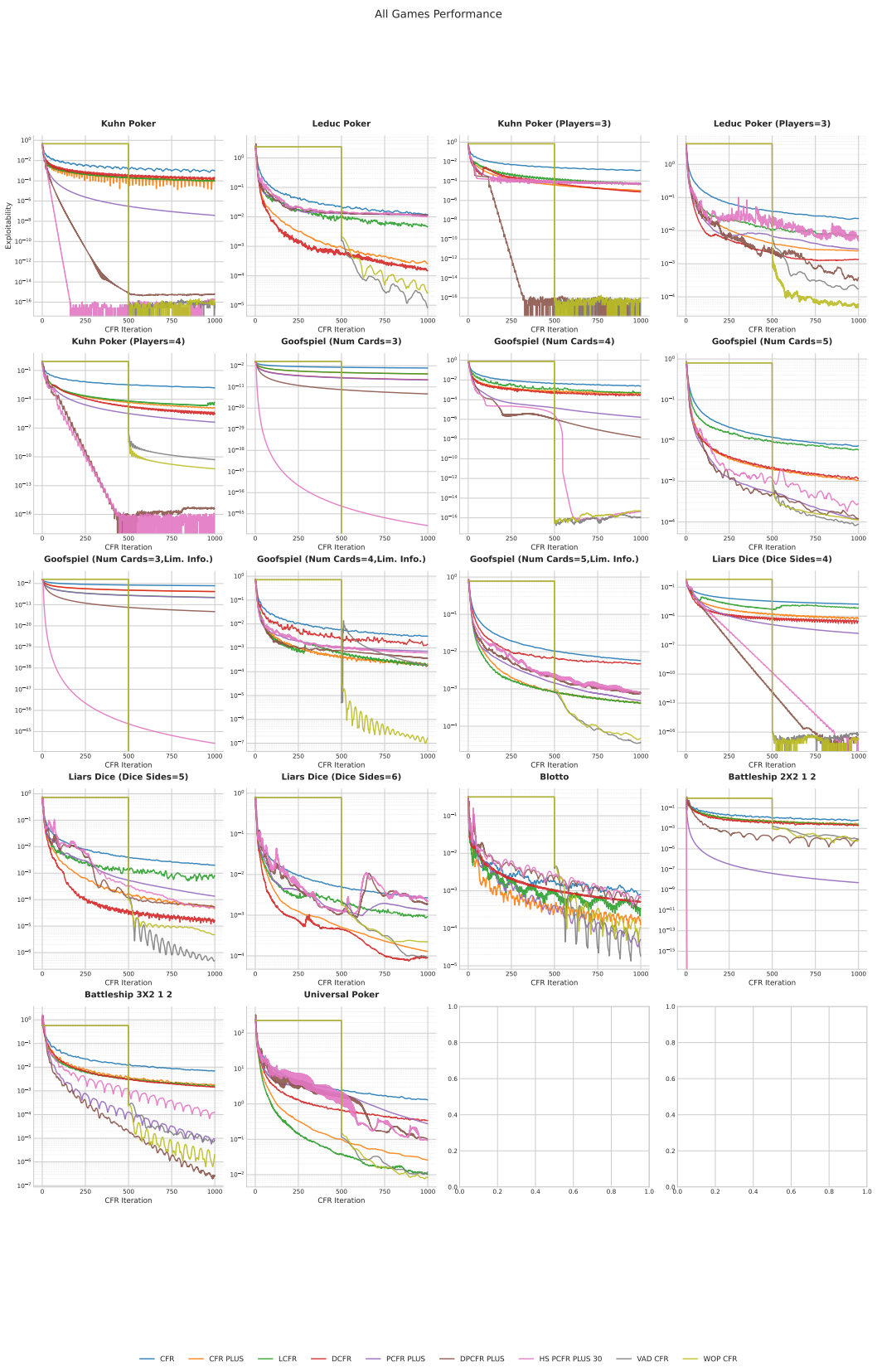

Figure 3: **CFR variants performances on All Games.**

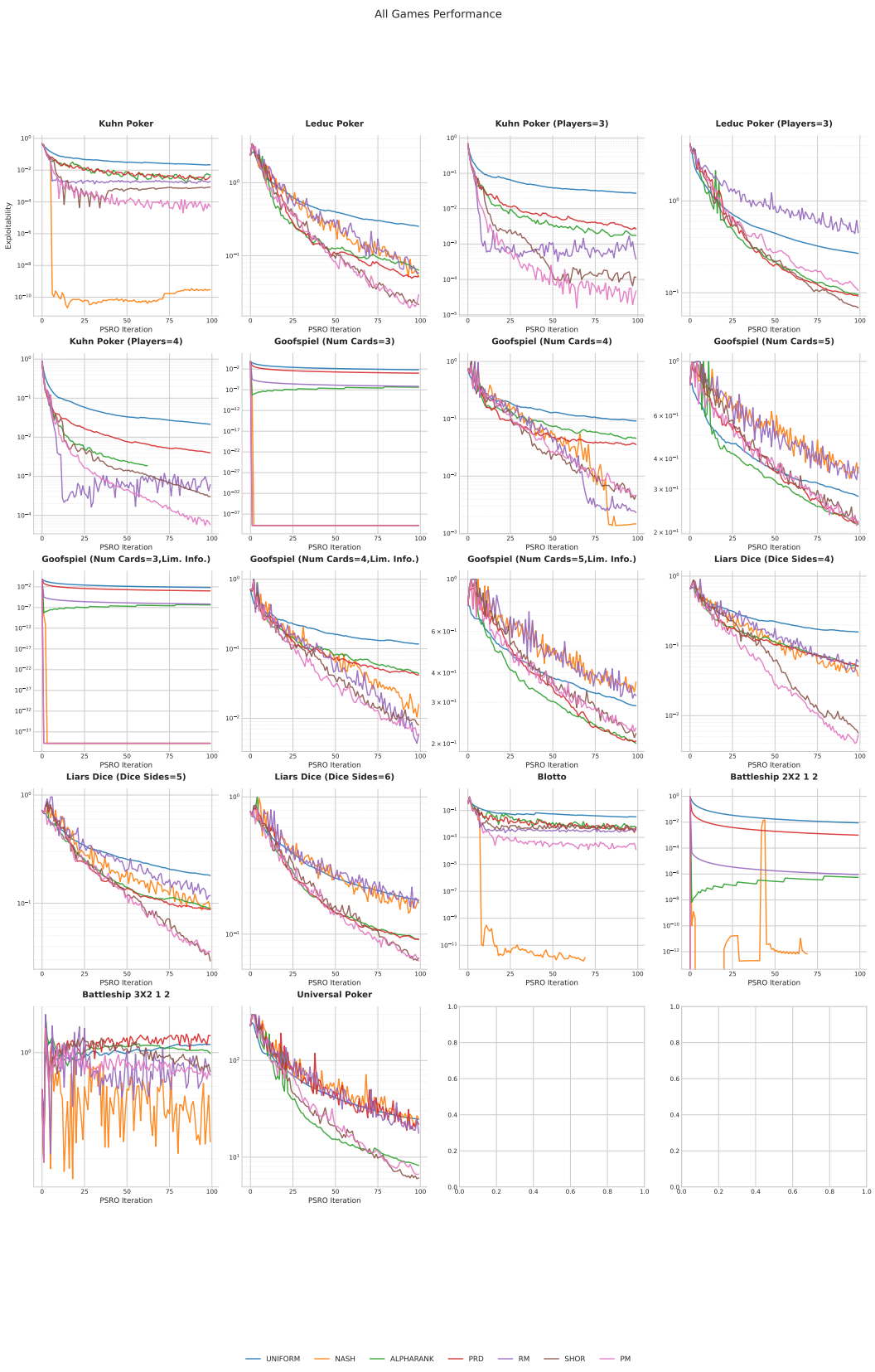

Figure 4: **PSRO variants performances on All Games.**

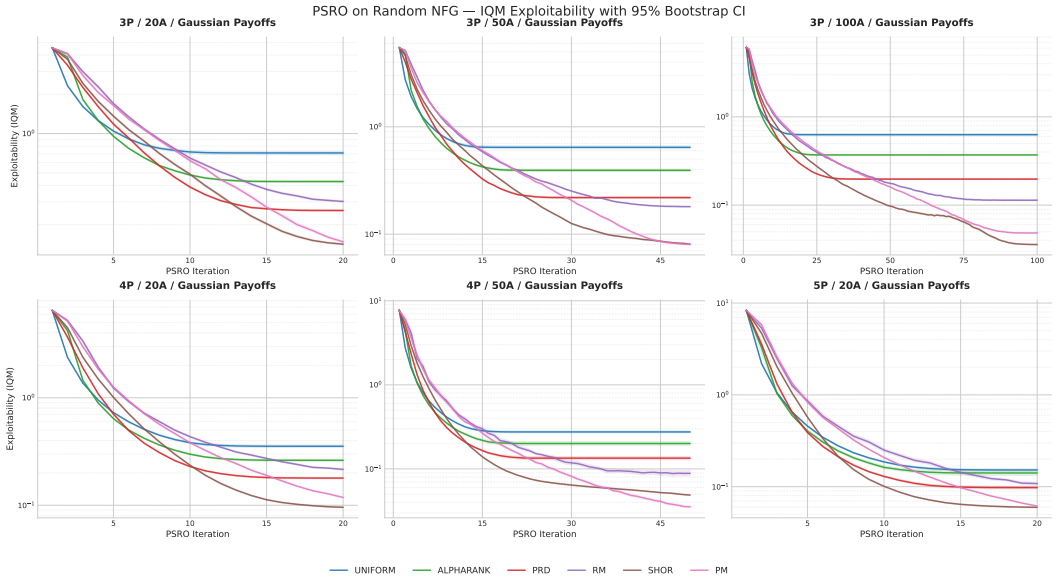

Figure 5: **PSRO variants performances on Gaussian payoff games.**

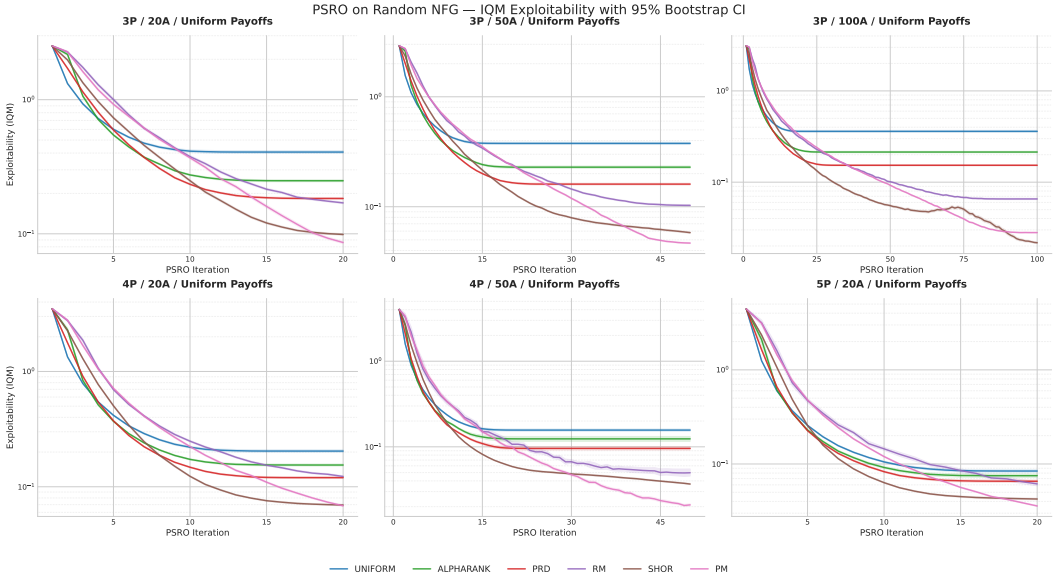

Figure 6: **PSRO variants performances on Uniform payoff games.**