Discovering Multiagent Learning Algorithms with Large Language Models

Zun Li
Google DeepMind
New York City, USA
lizun@google.com

John Schultz
Google DeepMind
New York City, USA
jhtschultz@google.com

Daniel Hennes
Google DeepMind
Zurich, Switzerland
hennes@google.com

Marc Lanctot
Google DeepMind
Montreal, Canada
lanctot@google.com

ABSTRACT

Much of the advancement of Multi-Agent Reinforcement Learning (MARL) in imperfect-information games has historically depended on manual iterative refinement of baselines. While foundational families like Counterfactual Regret Minimization (CFR) and Policy Space Response Oracles (PSRO) rest on solid theoretical ground, the design of their most effective variants often relies on human intuition to navigate a vast algorithmic design space. In this work, we propose the use of AlphaEvolve, an evolutionary coding agent powered by large language models, to automatically discover new multiagent learning algorithms. We demonstrate the generality of this framework by evolving novel variants for two distinct paradigms of game-theoretic learning. First, in the domain of iterative regret minimization, we evolve the logic governing regret accumulation and policy derivation, discovering a new algorithm, Volatility-Adaptive Discounted (VAD-)CFR. VAD-CFR employs novel, non-intuitive mechanisms—including volatility-sensitive discounting, consistency-enforced optimism, and a hard warm-start policy accumulation schedule—to outperform state-of-the-art baselines like Discounted Predictive CFR+. Second, in the regime of population based training algorithms, we evolve training-time and evaluation-time meta strategy solvers for PSRO, discovering a new variant, Smoothed Hybrid Optimistic Regret (SHOR-)PSRO. SHOR-PSRO introduces a hybrid meta-solver that linearly blends Optimistic Regret Matching with a smoothed, temperature-controlled distribution over best pure strategies. By dynamically annealing this blending factor and diversity bonuses during training, the algorithm automates the transition from population diversity to rigorous equilibrium finding, yielding superior empirical convergence compared to standard static meta-solvers.

KEYWORDS

Multi-Agent Reinforcement Learning, Game Theory, Large Language Models, Meta-Learning

ACM Reference Format:
Zun Li, John Schultz, Daniel Hennes, and Marc Lanctot. 2026. Discovering Multiagent Learning Algorithms with Large Language Models. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 27 pages.

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. These advances have been driven by a diverse array of methods, including game-theoretic regret minimization [4] and population-based league training [23]. The practical performance of these algorithms relies heavily on specific structural choices—such as how regret is discounted over time or how a specific equilibrium solution concept is derived. Historically, the refinement of these choices has been a largely manual endeavor. Researchers must rely 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 [17], 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 the search to transcend simple parameter tuning and discover entirely new mechanisms for equilibrium finding.

We validate the generality of this framework by applying it to the two dominant paradigms of imperfect-information game solving:

1. Evolving Counterfactual Regret Minimization (CFR) [32]: The CFR family of algorithms relies on recursive definitions for accumulating regret and deriving an average policy. State-of-the-art variants like PCFR+ [8] and DCFR [3] were developed by manually identifying specific weighting schemes or regret targets to accelerate convergence. We apply AlphaEvolve to the symbolic operations governing regret accumulation, policy aggregation, and current policy derivation. The search yields Volatility-Adaptive Discounted (VAD-)CFR, a novel variant that dynamically adjusts its discounting parameters based on the volatility of regret updates and utilizes a

regret-magnitude weighted warm-start to construct the average strategy. This approach outperforms existing baselines across a diverse set of game benchmarks.

2. Evolving Policy-Space Response Oracles [12] (PSRO): PSRO generalizes the Double Oracle algorithm [13], expanding a population of policies by iteratively computing best responses and solving for a meta-strategy. Standard meta-solvers (e.g., Projected Replicator Dynamics or Uniform) enforce static trade-offs between exploration (expanding the game graph) and exploitation (refining the equilibrium). These static heuristics often fail to adapt to the changing topology of the empirical game during training. We apply AlphaEvolve to discover dynamic training-time and evaluation-time meta-solvers that optimize this schedule. The resulting variant, Smoothed Hybrid Optimistic Regret (SHOR-)PSRO, utilizes a hybrid update rule that mixes regret-based stability with aggressive greedy exploitation. By annealing the weight of these components over the training horizon, SHOR-PSRO empirically shows both algorithmic robustness and iteration efficiency.

The contributions of this paper are summarized as follows:
(1) We showcase using LLM-driven evolution to design multi-agent learning algorithms, moving beyond parameter tuning to symbolic code evolution.
(2) We identify VAD-CFR, an evolved algorithm that demonstrates how automated search can uncover update rules that are effective yet non-intuitive to human designers. In Appendix 7.2, we also present AOD-CFR, a variant discovered in an early trial that employs more conventional mechanisms while maintaining competitive performance.
(3) We demonstrate the versatility of the framework by evolving PSRO meta-solvers that improve convergence speed and stability automating the transition from exploration to exploitation, yielding a new variant, SHOR-PSRO.

By automating the discovery of these mechanisms, we anticipate that future game-theoretic solvers will derive from a blend of human ingenuity and AI-driven insights.

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 $\Gamma = \langle \mathcal{N}, \mathcal{H}, \mathcal{Z}, \mathcal{A}, u, \mathcal{I}, \sigma \rangle$, where $\mathcal{N} = \{1, \ldots, N\}$ denotes the set of players [20]. $\mathcal{H}$ 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 neither player can increase their utility by deviating unilaterally:

$$u_i(\sigma_i^*, \sigma_{-i}^*) \geq u_i(\sigma_i', \sigma_{-i}^*) \quad \forall \sigma_i', \forall i \in \mathcal{N} \tag{1}$$

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):

$$\text{Exploitability}(\sigma) = \frac{1}{N} \sum_{i \in \mathcal{N}} \left(\max_{\sigma_i'} u_i(\sigma_i', \sigma_{-i}) - u_i(\sigma) \right) \tag{2}$$

A strategy is an ϵ-Nash Equilibrium if the exploitability is less than or equal to ϵ. 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** [32]. 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 σ. We define $\pi_{-i}^\sigma(h)$ as the contribution of all players *except* i (including chance) to reaching h. The **counterfactual value** of reaching information set I and playing action a is the expected utility given that player i played to reach I:

$$v_i(\sigma, I, a) = \sum_{h \in I} \pi_{-i}^\sigma(h) \sum_{z \in \mathcal{Z}, h \sqsubset z} \pi^\sigma(z \mid h, a) u_i(z) \tag{3}$$

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:

$$r^t(I, a) = v_i(\sigma^t, I, a) - \sum_{a' \in \mathcal{A}(I)} \sigma^t(I, a') v_i(\sigma^t, I, a') \tag{4}$$

Standard CFR accumulates these values linearly over iterations T:

$$R^T(I, a) = \sum_{t=1}^{T} r^t(I, a) \tag{5}$$

The current policy σ^{t+1} is derived from accumulated regret, typically using **Regret Matching (RM)**, which assigns probabilities proportional to positive regret:

$$\sigma^{t+1}(I, a) = \frac{\max(R^t(I, a), 0)}{\sum_{a'} \max(R^t(I, a'), 0)} \tag{6}$$

The strategy σ^t at any single iteration may not converge to NE. Instead, CFR outputs the **average strategy** $\bar{\sigma}^T$, computed by weighting the iteration strategy σ^t by the player's contribution to the reach probability $\pi_i^{\sigma^t}(I)$.

CFR Variants: Several variants modify these update rules. For example, **CFR+** [22] replaces the regret accumulation R^T with floor bounding $\max(R^{T-1} + r^t, 0)$ 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 [1, 12] acts as a meta-solver that generalizes the Double Oracle algorithm [13]. It operates on a higher level of abstraction called the **Meta-Game** (or Empirical Game [25]). 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.

The PSRO Loop: 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:

$$\sigma_i^{k+1} \in \arg\max_{\sigma_i} \mathbb{E}_{\sigma_{-i} \sim \phi_{-i}} [u_i(\sigma_i, \sigma_{-i})] \tag{7}$$

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, we employ a distinct **evaluation-time meta-strategy solver** to compute a distribution over the current policy population; this distribution serves as the basis for the exploitability metric. It is critical [24] to distinguish this from the training process: the training-time MSS drives the *generation* of new policies (often prioritizing exploration), while the evaluation-time MSS ensures a robust *evaluation* of the population's quality. We regard both the training-time and evaluation-time MSS as integral components of PSRO when employed as a single game-solver. Standard PSRO relies on fixed static meta-solvers for both training-time and eval-time (e.g., always using Nash or always using Uniform). However, the optimal meta-solver may change during training—for instance, encouraging exploration (Uniform) early on and robustness (Nash) later. We aim to discover a dynamic meta-solver schedule.

3 METHOD: AUTOMATING ALGORITHM DISCOVERY VIA ALPHAEVOLVE

We propose a framework to automate the design of multi-agent learning algorithms by shifting the design process from manual heuristics to automated discovery. We utilize **AlphaEvolve** [17], an evolutionary coding agent that leverages Large Language Models (LLMs) to evolve executable code. We apply this framework to two distinct paradigms: Regret Minimization (CFR) and Population based Training (PSRO).

3.1 The AlphaEvolve Framework

AlphaEvolve combines the creative code-generation capabilities of LLMs with the rigorous selection pressure of evolutionary algorithms. Unlike traditional genetic programming which relies on random syntactic mutations, AlphaEvolve uses an LLM to propose semantically meaningful modifications to a code file. The process operates as follows:

- **Population Initialization:** We initialize a population $\mathcal{P}$ containing the standard implementations of the baseline algorithms (e.g., standard CFR code or Uniform PSRO code).
- **LLM-Driven Mutation:** In each generation, a parent algorithm $A \in \mathcal{P}$ is selected based on fitness. Notice that AlphaEvolve supports *multi-objective* optimization: if multiple fitness metrics are defined, one of them (say f) will be randomly selected, and A is then sampled in favor of high value of f. The source code of A is fed into an LLM (e.g., Gemini [7]) with a prompt instructing it to *"Modify the following code to improve fitness (reduce exploitability)."* The LLM generates a new candidate variant A' by applying several lines of code changes to A.
- **Automated Evaluation:** The candidate A' is executed on a set of proxy games (e.g., Kuhn Poker). An automated evaluator computes its fitness scores (final negative exploitability).
- **Evolutionary Selection:** Valid candidates are added to the population $\mathcal{P}$. This loop repeats, allowing the system to discover complex, non-intuitive optimizations.

3.2 Evolving Regret Minimization Code

To discover new variants of CFR, we expose the core update functions of the Counterfactual Regret Minimization algorithm to the AlphaEvolve agent. The search space is defined by the Python functions responsible for accumulating regret and updating the average policy. Specifically, we design the components with the primitives in Listing 1. This search space is expressive enough to encompass the entire family of known CFR variants as special cases. For instance, standard CFR uses eq (5) for `RegretAccumulator` and eq (6) for `PolicyFromRegretAccumulator`. These components are Python classes which allow for maintaining state across iterations, in contrast to pure functions.

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.
    """
    ...

```

```python

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.
    """
    ...
```

3.3 Evolving Meta-Strategy Solvers Code

For PSRO, we evolve `TrainMetaStrategySolver` for generating the mixed strategies used during the oracle training phase, and `EvalMetaStrategySolver` for computing a strategy profile to report metrics such as exploitability. The code skeleton we designed is shown in Listing 2. This interface 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.

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.

```python
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."""
    ...
```

3.4 Optimization Objective

We manually select a set $\mathcal{G}$ of training games for AlphaEvolve to compute $|\mathcal{G}| + 1$ fitness scores. These are the negative exploitability $-\text{Exploitability}(A(g)_K)$ of the final strategy profile after K iterations for each $g \in \mathcal{G}$, as well as their average

$$-\frac{1}{|\mathcal{G}|}\sum_{g\in\mathcal{G}}\text{Exploitability}(A(g)_K).$$

Here $A(g)_K$ 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* 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*. 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 11 games are provided in Appendix 7.3. We utilized the OpenSpiel [11] framework for all experiments. Our implementation of AlphaEvolve is backboned by Gemini 2.5 pro [7].

4.2 Experimental Evaluation: VAD-CFR[1]

We evaluate the efficacy of the evolved algorithm, VAD-CFR, against established game-theoretic baselines. We benchmarked VAD-CFR against a suite of state-of-the-art regret minimization algorithms: standard CFR [32], CFR+ [22], Linear CFR (LCFR), Discounted CFR (DCFR) [3], Predictive CFR+ (PCFR+) [8], Discounted Predictive CFR+ (DPCFR+) [28], and a Hyperparameter Schedule-powered

[1]Appendix 7.2 details an alternative variant (AOD-CFR) discovered on a different set of training games that also achieves robust results. Notably, this variant relies on more conventional mechanisms than VAD-CFR.

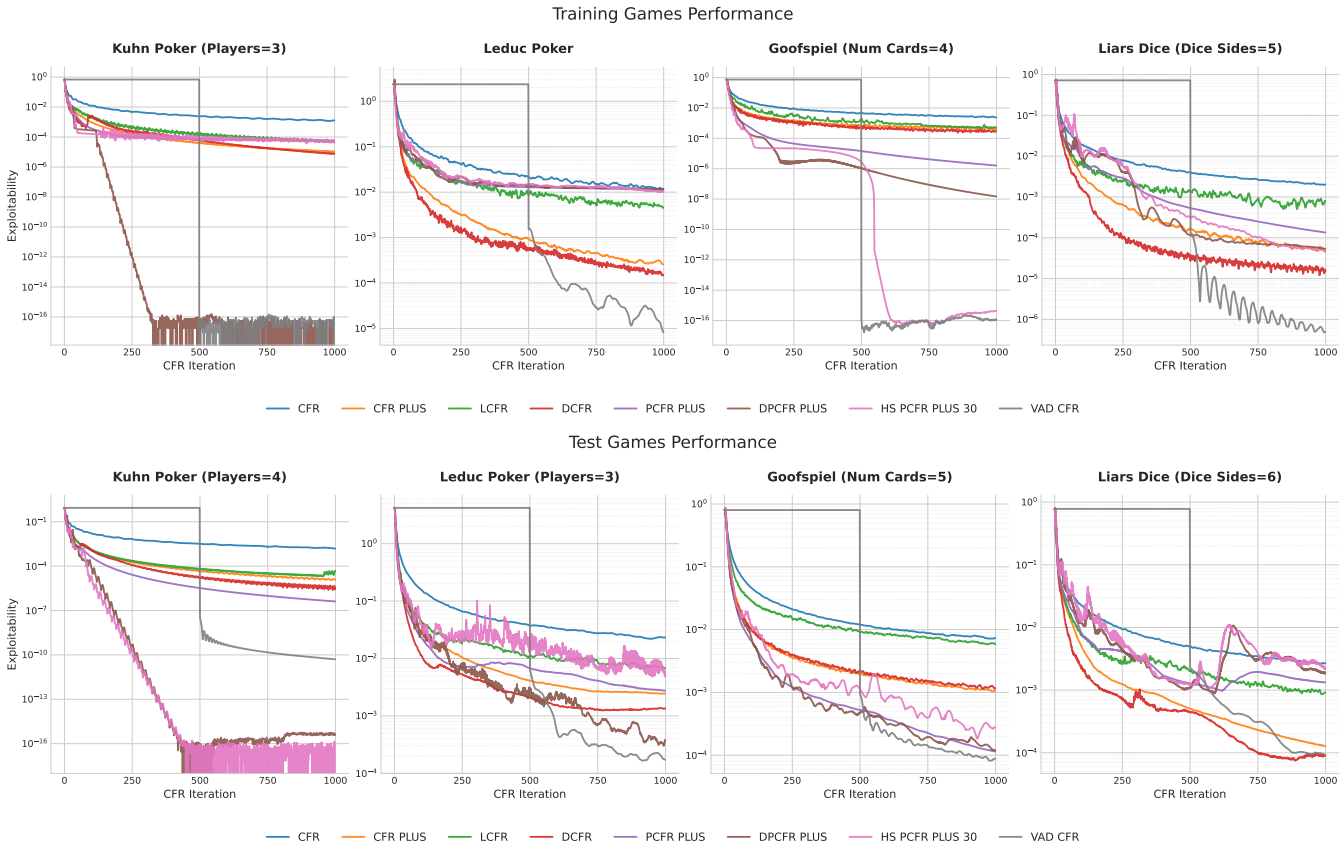

Figure 1: CFR variants performances.

PCFR variant HS-PCFR+(30) [31]. Performance was quantified using exploitability (measured on a logarithmic scale) over a fixed horizon of 1000 iterations, e.g., $K = 1000$. For all domains the exploitability scores are computed exactly. We use CFR+ as the seed program. The prompt we use is shown in Listing 8 in the Appendix.

4.2.1 Discovered Algorithmic Architecture. The source code of the discovered VAD-CFR algorithm is provided in Listing 5 in Appendix 7.1. A high-level abstraction is shown in Listing 3. The evolutionary search discarded the standard linear averaging and static discounting of the CFR family in favor of three distinct, non-intuitive mechanisms:

• **Volatility-Adaptive Discounting (vs. Static Discounting):** Standard DCFR applies fixed discount factors (α, β) to cumulative regrets. In contrast, VAD-CFR makes these parameters reactive. It tracks the volatility of the learning process via an Exponential Weighted Moving Average (EWMA) of the instantaneous regret magnitude. When volatility is high (indicating the strategy is in flux), the algorithm dynamically increases discounting to "forget" the unstable history faster; when volatility drops, it retains more history for fine-tuning.

• **Asymmetric Instantaneous Boosting:** While DCFR applies asymmetry to accumulated history, VAD-CFR applies a novel asymmetry to the instantaneous update itself. Positive instantaneous regrets (actions that are currently good) are boosted by a factor of 1.1. This asymmetry enables the immediate exploitation of beneficial deviations, eliminating the lag associated with accumulation.

• **Hard Warm-Start & Regret-Magnitude Weighting:** Perhaps the most interesting discovery is the policy accumulation schedule. Standard CFR variants begin averaging policies immediately at $t = 1$. In contrast, VAD-CFR enforces a hard warm-start[2], postponing the start of policy averaging until iteration 500. The underlying regret accumulation process remains fully active during this phase, ensuring the agent continues to learn and update its strategy. Furthermore, once accumulation begins, policies are weighted not just linearly (t), but by the magnitude of the instantaneous regret. This acts as a sophisticated filter, ensuring the final equilibrium approximation is constructed only from high-information iterations, preventing early-stage noise from polluting the solution quality. Warm starting mechanisms have been studied in a previous work [2].

Listing 3: High-level abstraction of VAD-CFR.

```
class RegretAccumulator:
    """Volatility-Adaptive Discounting & Asymmetric Boosting."""
    def update_accumulate_regret(self, info_state_node,
          iteration_number, cfr_regrets):
        # 1. Volatility & Adaptive Discount Calculation
```

[2]The LLM generated this threshold without knowing the fixed 1000-iteration evaluation horizon in its prompt context (Listing 8).

```
        inst_mag = max((abs(r) for r in cfr_regrets.values()),
            default=0.0)
        self.ewma = 0.1 * inst_mag + 0.9 * self.ewma
        volatility = min(1.0, self.ewma / 2.0)
        alpha = max(0.1, 1.5 - 0.5 * volatility)
        beta = -0.1 - 0.5 * volatility
        beta = min(alpha, beta)
        t_plus_one = float(iteration_number) + 1.0
        disc_pos = (t_plus_one ** alpha) / (t_plus_one ** alpha +
            1.0)
        disc_neg = (t_plus_one ** beta) / (t_plus_one ** beta +
            1.0)
        updated_regrets = {}
        for a, r in cfr_regrets.items():
            # 2. Asymmetric Instantaneous Boosting
            r_boosted = r * 1.1 if r > 0 else r
            # 3. Sign-Dependent History Discounting
            prev_R = info_state_node.cumulative_regret.get(a, 0.0)
            discount = disc_pos if prev_R >= 0 else disc_neg
            # 4. Update with Negative Cap
            new_R = (prev_R * discount) + r_boosted
            updated_regrets[a] = max(-20.0, new_R)
        return updated_regrets

class PolicyFromRegretAccumulator:
    """Derives policy from a 'Future Projection' of regrets."""
    def get_updated_current_policy(self, info_state_node,
        iteration_number, cfr_regrets, previous_policy):
        # 1. Consistency: Re-calculate exact adaptive params
        inst_mag = max((abs(r) for r in cfr_regrets.values()),
            default=0.0)
        volatility = min(1.0, inst_mag / 2.0)
        # 2. Decaying Optimism Schedule
        base_optimism = 1.0 / (1.0 + iteration_number / 100.0)
        optimism = base_optimism * max(0.0, 1.0 - 0.5 * volatility
            )
        policy = {}
        sum_pos_regret = 0.0
        for a in info_state_node.legal_actions:
            # 3. Projected Regret Calculation
            r_boosted = cfr_regrets.get(a, 0.0)
            if r_boosted > 0: r_boosted *= 1.1
            prev_R = info_state_node.cumulative_regret.get(a, 0.0)
            discount = get_adaptive_discount_pos(prev_R) if prev_R
                >= 0 else get_adaptive_discount_neg(prev_R)
            proj_R = (prev_R * discount) + r_boosted + optimism
            # 4. Non-Linear Probability Scaling
            if proj_R > 0:
                scaled_r = proj_R ** 1.5
                policy[a] = scaled_r
                sum_pos_regret += scaled_r
            else:
                policy[a] = 0.0
        return normalize(policy) if sum_pos_regret > 0 else
            uniform(policy)

class PolicyAccumulator:
    """Warm-Start & Magnitude Weighting."""
    def update_accumulate_policy(self, info_state_node,
        iteration_number, info_state_policy, cfr_regrets,
        reach_prob, counterfactual_reach_prob):
        # 1. Hard Warm-Start
        if iteration_number < 500:
            return info_state_node.cumulative_policy
        # 2. Adaptive Gamma (Polynomial Averaging)
        inst_mag = max((abs(r) for r in cfr_regrets.values()),
            default=0.0)
        volatility = min(1.0, inst_mag / 2.0)
        gamma = min(4.0, 2.0 + 1.5 * volatility)
        # 3. Multi-Factor Weighting
        w_time = (float(iteration_number) + 1.0) ** gamma
        w_mag = (1.0 + (inst_mag / 2.0)) ** 0.5
        w_stable = 1.0 / (1.0 + inst_mag ** 1.5)
        final_weight = w_time * w_mag * w_stable
        # 4. Update Cumulative Policy
        updated_policy = {}
        for a, prob in info_state_policy.items():
            prev_P = info_state_node.cumulative_policy.get(a, 0.0)
            updated_policy[a] = prev_P + (final_weight *
                reach_prob * prob)
        return updated_policy
```

4.2.2 Empirical Analysis: Training Domain. As shown in Figure 1, VAD-CFR demonstrates superior convergence across the training 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.

4.2.3 Generalization Capabilities: Test Domain. Results illustrated in Figure 1 highlights 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 game topologies.

Overall, VAD-CFR demonstrates efficient convergence rate and generalization across a broad variety of domains: as shown in Figure 3 in the Appendix, **VAD-CFR matches or surpasses previous state-of-the-art performances in 10 of the 11 games**, with 4-player Kuhn Poker as the only exception.

4.3 Experimental Evaluation: SHOR-PSRO

Next, we evaluated the performance of the evolved 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 uniformly randomizes 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 [15], Projected Replicator Dynamics (PRD) [12], and Regret Matching (RM). We use Uniform as the initial program for both solver classes. The prompt we use is shown in Listing 9 in the Appendix.

4.3.1 Discovered Algorithmic Architecture. The source code of the discovered SHOR-PSRO algorithm is provided in Listing 6 in Appendix 7.1. A high-level abstraction is in Listing 4. The evolutionary search yielded a "Hybrid" meta-solver that constructs a meta-strategy σ by linearly blending two distinct components: Optimistic Regret Matching (ORM) and a Smoothed Best Pure Strategy.

- **Hybrid Blending Mechanism:** At every internal solver iteration, the meta-strategy is computed as:

$$\sigma_{hybrid} = (1 - \lambda) \cdot \sigma_{ORM} + \lambda \cdot \sigma_{Softmax} \tag{8}$$

where σ_{ORM} provides the stability of regret minimization, and $\sigma_{Softmax}$ is a Boltzmann distribution over pure strategies that aggressively biases the solver toward high-reward modes. The blending factor λ controls the trade-off between these dynamics.

- **Dynamic Annealing Schedule:** Unlike standard PSRO which uses fixed solvers, SHOR-PSRO employs a dynamic schedule for the training-time solver. Over the course of the PSRO iterations:

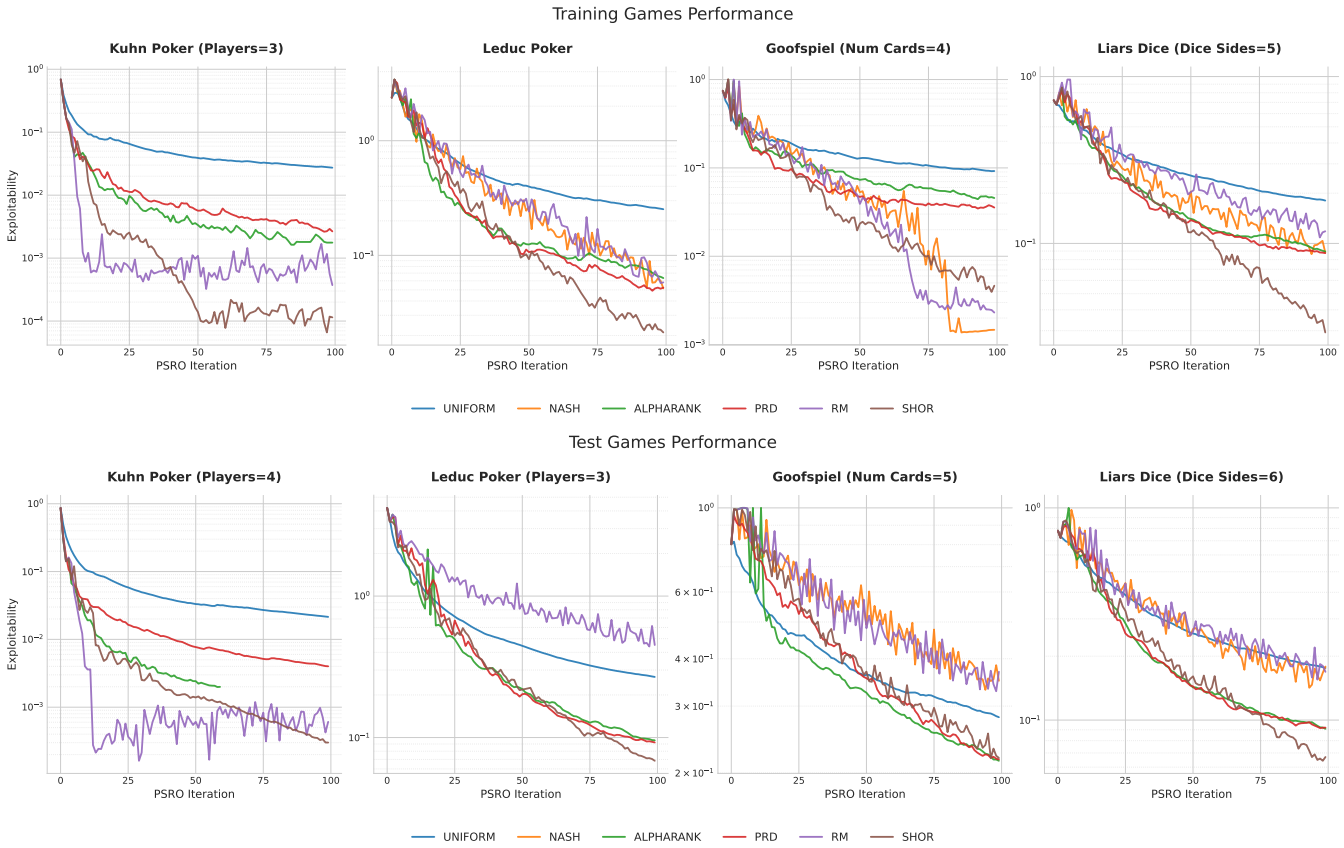

Figure 2: PSRO variants performances.

– The blending factor λ anneals from 0.3 $\longrightarrow$ 0.05, gradually shifting focus from greedy exploitation to robust equilibrium finding.
– An explicit **diversity bonus** is applied to the payoff gains, which also decays from 0.05 $\longrightarrow$ 0.001, ensuring early expansion of the game graph followed by late-stage refinement.

• **Training vs. Evaluation Asymmetry:** The search discovered distinct configurations for training and evaluation. The **Training Solver** utilizes the dynamic annealing schedule described above and returns the *average* strategy over internal iterations to ensure stability. In contrast, the **Evaluation Solver** employs a fixed, low blending factor ($\lambda = 0.01$) and returns the *last-iterate* strategy. This decoupling allows the algorithm to explore safely during training while providing reactive, low-noise exploitability estimates during evaluation.

Listing 4: High-level abstraction of SHOR-PSRO.

```python
class TrainMetaStrategySolver:
    def get_meta_strategy(self, game, policy_sets, meta_games):
        """Hybrid Training Solver with Full Parameter Annealing"""
        # 1. Annealing Schedules (Exploration -> Exploitation)
        prog = min(1.0, self.current_psro_iter / 75.0)
        blend = 0.30 - (0.25 * prog)  # Blend: 0.3 -> 0.05
        div = 0.05 - (0.049 * prog)   # Diversity: 0.05 -> 0.001
        temp = 0.50 - (0.49 * prog)   # Temp: 0.5 -> 0.01
        # 2. Adaptive Solver Iterations
        pop_size = len(meta_games[0])
        n_iters = int(1000 + 20 * (pop_size - 1))
        # 3. Hybrid Internal Loop (Training returns Average)
        return _hybrid_solver(meta_games, n_iters, blend, div,
            temp, momentum=0.5, return_avg=True)


class EvalMetaStrategySolver:
    def get_meta_strategy(self, game, policy_sets, meta_games):
        """Evaluation Solver (Fixed Parameters, Last-Iterate)"""
        # 1. Fixed Parameters for Pure Exploitation
        blend, div, temp = 0.01, 0.0, 0.001
        # 2. Aggressive Iteration Scale for Convergence
        pop_size = len(meta_games[0])
        n_iters = int(8000 + 50 * (pop_size - 1))
        # 3. Hybrid Loop returns Last-Iterate for eval reactivity
        return _hybrid_solver(meta_games, n_iters, blend, div,
            temp, momentum=0.2, return_avg=False)

def _hybrid_solver(meta_games, n_iters, blend, div, temp, momentum
    , return_avg):
    sigma = uniform_strategy(meta_games)
    sigma_avg = zeros_like(sigma)
    for i in range(n_iters):
        # A. Optimistic Regret Matching (ORM)
        gains = compute_payoffs(meta_games, sigma) + div * (1 -
            sigma)
        gains_norm = normalize_scale(apply_momentum(gains, beta=
            momentum))
        sigma_orm = regret_matching(gains_norm)
        # B. Smoothed Best Pure Strategy (Softmax)
        exp_vals = compute_expected_values(meta_games, sigma)
        sigma_pure = softmax(exp_vals, temperature=temp)
        # C. Hybrid Blending
        sigma = ((1 - blend) * sigma_orm) + (blend * sigma_pure)
```

```
        if return_avg: sigma_avg += sigma
    return normalize(sigma_avg) if return_avg else sigma
```

4.3.2 Empirical Analysis: Training Domain. As shown in Figure 2, SHOR-PSRO demonstrates superior convergence speed and stability across the training set. In simpler domains like Kuhn Poker, SHOR-PSRO achieves exploitability levels ($< 10^{-3}$) significantly faster than PRD or RM. This acceleration is likely attributable to the "Hybrid Blending" mechanism, which allows the solver to leverage the stability of regret minimization while aggressively exploiting high-reward modes via the smoothed softmax component.

4.3.3 Generalization Capabilities: Test Domain. To assess the robustness of the evolved mechanism, we evaluated SHOR-PSRO on a Test Set comprising larger and more complex game variants. The results, illustrated in Figure 2, highlight the algorithm's generalization capabilities. 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 demonstrates a clear advantage. While static solvers struggle with the expanded branching factor, the hybrid solver's ability to bias towards "smoothed best pure strategies" allows it to navigate the expanded game graph efficiently. The results suggest that the evolved evaluation-time asymmetry—using a fixed, low-noise configuration for measurement—ensures that the exploitability metric accurately reflects the population's growing strength without being obscured by exploration noise.

Overall, SHOR-PSRO proves to be a highly robust meta-solver, capable of generalizing to unseen game topologies without requiring manual re-tuning of its annealing schedules. As shown in Figure 4 in the Appendix, **SHOR-PSRO matches or surpasses state-of-the-art performances in 8 of the 11 games**.

5 RELATED WORK

Our work sits at the intersection of game-theoretic multi-agent learning, meta-learning, and automated algorithmic discovery. Here, we review the manual heuristic design that characterizes the current state-of-the-art and contrast it with our automated approach.

Significant research effort has been dedicated to improving the convergence speed of CFR through specific weighting schemes and regret targets. Notable variants include CFR+ [22], which introduced floor bounding and linear averaging; Discounted CFR (DCFR) [3], which applies fixed discount factors to historical regrets; and Predictive CFR+ (PCFR+) [8], which utilizes predictive Blackwell approachability. However, these improvements were largely derived through human intuition and trial-and-error to navigate the vast combinatorial space of potential update rules. In contrast to strategy-based warm starting [2], which relies on discrete, heuristic-driven resets, VAD-CFR employs a static warmup schedule with continuous volatility-based discounting to regulate regret retention.

For larger games where traversing the full tree is infeasible, Policy Space Response Oracles (PSRO) [1] generalizes the Double Oracle algorithm [13] by iteratively expanding a population of policies via exact or reinforcement learning best responses. While PSRO rests on solid theoretical ground [30], its practical application faces challenges in terms of convergence speed and population quality. Recent surveys highlight the diversity of PSRO variants [1], but like CFR, the design of meta-solver schedules has traditionally been static or manually tuned rather than dynamically evolved.

The research of automating machine learning algorithm design has been evolving for both neural-based approaches and symbolic-based approaches. Meta-learning approaches such as meta-RL [18, 29] and meta-learning optimizers [14] have parameterized update rules using neural networks to optimize learning dynamics. For symbolic ML discovery, a foundational work in this domain is AutoML-Zero [19], which demonstrated that complete machine learning algorithms could be evolved from scratch using basic mathematical operations. A following work also applies program search for discovering optimizers [5]. The authors of [6] also studied discovering novel symbolic reinforcement learning algorithms.

Within the specific domain of multi-agent learning, there has been prior work on automating algorithm design [9, 26, 27]. More directly related, one can meta-learn the regret minimization algorithm directly via a CFR-like no-regret framework from examples [21]. However, these approaches often faced limitations: they either operated within a relatively constrained search space or relied on neural parameterizations that hindered interpretability. Our work builds directly on AlphaEvolve [17], utilizing Large Language Models (LLMs) to perform semantic mutation on interpretable code, bridging the gap between expressive neural meta-learning and symbolic discovery. This approach has already been shown great success in the field of math [10] and combinatorial algorithms [16].

6 CONCLUSION

In this work, we proposed a methodology for using LLM-driven evolution to automate the design of multi-agent learning algorithms, effectively shifting the paradigm from manual heuristic tuning to symbolic code evolution. By leveraging AlphaEvolve, we demonstrated that algorithm design can be treated as a search problem within a combinatorial space of symbolic operations, allowing for the discovery of complex, non-intuitive optimization mechanisms.

First, we identified **VAD-CFR**, which breaks from the tradition of static discount schedules used in DCFR. By introducing volatility-adaptive parameters, instantaneous regret boosting, and a warm-start for policy averaging, VAD-CFR demonstrates that filtering out early-stage noise is as critical as optimizing late-stage convergence. These mechanisms significantly outperform state-of-the-art baselines in domains where exact exploitability is computable.

Second, we addressed the efficiency challenges of Policy Space Response Oracles (PSRO) by evolving **SHOR-PSRO**. This algorithm utilizes a hybrid meta-solver that blends Optimistic Regret Matching with a smoothed, temperature-controlled best response. By automating the annealing of this blending factor, SHOR-PSRO effectively manages the transition from population diversity to equilibrium refinement, yielding solvers that are significantly more effective for equilibrium finding in large-scale scenarios.

Our results suggest that the automated discovery of algorithmic asymmetries—specifically those that manage regret scaling and dynamic mixing schedules—can yield solvers that are elusive to human intuition but highly effective in practice. Future work will explore the application of this evolutionary framework to fully deep reinforcement learning agents and the discovery of cooperative mechanisms in general-sum games.

REFERENCES

[1] Ariyan Bighashdel, Yongzhao Wang, Stephen McAleer, Rahul Savani, and Frans A Oliehoek. 2024. Policy Space Response Oracles: A Survey. In *Thirty-Third International Joint Conference on Artificial Intelligence (IJCAI-24) Survey Track*.

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

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

[4] Noam Brown and Tuomas Sandholm. 2019. Superhuman AI for multiplayer poker. *Science* 365, 6456 (2019), 885–890. https://doi.org/10.1126/science.aay2400

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

[6] John D Co-Reyes, Yingjie Miao, Daiyi Peng, Esteban Real, Quoc V Le, Sergey Levine, Honglak Lee, and Aleksandra Faust. [n.d.]. Evolving Reinforcement Learning Algorithms. In *Ninth International Conference on Learning Representations*.

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

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

[9] Xidong Feng, Oliver Slumbers, Ziyu Wan, Bo Liu, Stephen Marcus McAleer, Ying Wen, Jun Wang, and Yaodong Yang. [n.d.]. Neural Auto-Curricula in Two-Player Zero-Sum Games. In *Thirty-Fifth International Conference on Neural Information Processing Systems*.

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

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

[12] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérétat, David Silver, and Thore Graepel. 2017. A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning. In *Thirty-First International Conference on Neural Information Processing Systems*. 4190–4203.

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

[14] Luke Metz, Niru Maheswaranathan, Jeremy Nixon, C Daniel Freeman, and Jascha Sohl-Dickstein. 2019. Understanding and Correcting Pathologies in the Training of Learned Optimizers. In *Thirty-Sixth International Conference on Machine Learning*. 4556–4565.

[15] Paul Muller, Shayegan Omidshafiei, Mark Rowland, Karl Tuyls, Julien Perolat, Siqi Liu, Daniel Hennes, Luke Marris, Marc Lanctot, Edward Hughes, et al. [n.d.]. A Generalized Training Approach for Multiagent Learning. In *Ninth International Conference on Learning Representations*.

[16] Ansh Nagda, Prabhakar Raghavan, and Abhradeep Thakurta. 2025. Reinforced Generation of Combinatorial Structures: Hardness of Approximation. *arXiv preprint arXiv:2509.18057* (2025).

[17] 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. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131* (2025).

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

[19] Esteban Real, Chen Liang, David So, and Quoc V Le. 2020. AutoML-Zero: Evolving Machine Learning Algorithms From Scratch. In *Thirty-Seventh International Conference on Machine Learning*. 8007–8019.

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

[21] David Sychrovsk'y, Michal Šustr, Elnaz Davoodi, Michael Bowling, Marc Lanctot, and Martin Schmid. [n.d.]. Learning to Not Regret. In *Thirty-Eighth AAAI Conference on Artificial Intelligence*.

[22] Oskari Tammelin. 2014. Solving large imperfect information games using CFR+. *arXiv preprint arXiv:1407.5042* (2014).

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

[24] Yongzhao Wang, Qiurui Ma, and Michael P Wellman. 2022. Evaluating Strategy Exploration in Empirical Game-Theoretic Analysis. In *Twenty-First International Conference on Autonomous Agents and Multiagent Systems*. 1346–1354.

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

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

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

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

[29] Zhongwen Xu, Hado van Hasselt, and David Silver. 2018. Meta-Gradient Reinforcement Learning. In *Thirty-Second International Conference on Neural Information Processing Systems*, Vol. 31.

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

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

[32] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. 2007. Regret Minimization in Games with Incomplete Information. In *Twenty-First International Conference on Neural Information Processing Systems*. 1729–1736.

7 APPENDIX

7.1 Source code of discovered algorithms

Listing 5: VAD-CFR

```python
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 using ADCFRP.
    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 = {}
```

```python
    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 6: 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 Asymmetric Optimistic Discounted CFR

In an early trial, we discover another variant of CFR by training on 2-player Kuhn Poker, 2-player Leduc Poker, 4-Card Goofspiel and 4-side Liar Dice. It also shows good empirical performance and employs relatively more conventional mechanisms than VAD-CFR. We show its empirical performance together with other CFR variants in Figure 3. The source code is in Listing 7. Unlike manual heuristics which often rely on symmetric or fixed update rules, AOD-CFR utilizes a set of dynamic, asymmetric mechanisms discovered via evolutionary search.

- **Adaptive Regret Discounting:** The algorithm employs a linear schedule for discounting cumulative regrets. The positive regret discount factor, α, transitions from $1.0 \longrightarrow 2.5$ over 500 iterations, while the negative regret discount factor, β, transitions from $0.5 \longrightarrow 0.0$. This allows the algorithm to aggressively prune suboptimal actions early (via β) while increasingly emphasizing recent information for optimal actions (via α).
- **Asymmetric Instantaneous Scaling:** A distinct feature of AOD-CFR is its sign-dependent scaling of instantaneous regret. As revealed in the source code, the update rule scales instantaneous regret by 1.1 when both cumulative and instantaneous regrets are positive, but attenuates it by 0.9 when cumulative regret is positive and instantaneous regret is negative. This asymmetry likely functions as a momentum-preservation mechanism, reinforcing established beneficial actions while damping noise.
- **Trend-Based Policy Optimism:** In the policy derivation step, AOD-CFR incorporates a "trend-based optimism" term. It tracks an Exponential Moving Average (EMA) of cumulative regrets (decay ≈ 0.1) and adds a scaled deviation term to the accumulated regret before applying Regret Matching. Specifically, the code implements a scaling factor `optimism_trend_scale` applied to the difference between current cumulative regret and its EMA.
- **Polynomial Policy Averaging:** The average strategy is computed using an adaptive polynomial weighting scheme where the exponent γ scales linearly from $1.0 \longrightarrow 5.0$.

Listing 7: AOD-CFR

```
import math
import collections
import attr

class RegretAccumulator:
  """Updates cumulative regret with adaptive positive and negative discounting,
  and asymmetric instantaneous regret scaling."""

  def __init__(self,
               alpha_start: float = 1.0, alpha_max: float = 2.5, schedule_T_alpha: float = 500.0,
               beta_start: float = 0.5, beta_max: float = 0.0, schedule_T_beta: float = 500.0,
               pos_cum_pos_inst_scale: float = 1.1,  # R_cum > 0, r_inst > 0
               pos_cum_neg_inst_scale: float = 0.9,   # R_cum > 0, r_inst < 0
               neg_cum_pos_inst_scale: float = 0.7,   # R_cum < 0, r_inst > 0
               neg_cum_neg_inst_scale: float = 1.2):  # R_cum < 0, r_inst < 0
    """Initializes the regret accumulator.

    Args:
        alpha_start: Initial exponent for positive cumulative regret discounting.
        alpha_max: Final exponent for positive cumulative regret discounting.
```

```
        schedule_T_alpha: Iterations for alpha transition.
        beta_start: Initial exponent for negative cumulative regret discounting.
        beta_max: Final exponent for negative cumulative regret discounting.
        schedule_T_beta: Iterations for beta transition.
        pos_cum_pos_inst_scale: Scaling factor for instantaneous regret when
                                cumulative regret is positive and instantaneous is positive.
        pos_cum_neg_inst_scale: Scaling factor for instantaneous regret when
                                cumulative regret is positive and instantaneous is negative.
        neg_cum_pos_inst_scale: Scaling factor for instantaneous regret when
                                cumulative regret is negative and instantaneous is positive.
        neg_cum_neg_inst_scale: Scaling factor for instantaneous regret when
                                cumulative regret is negative and instantaneous is negative.
    """
    if not (alpha_start >= 0 and alpha_max >= 0 and schedule_T_alpha >= 0):
        raise ValueError("Alpha parameters and schedule must be non-negative.")
    if not (beta_start >= 0 and beta_max >= 0 and schedule_T_beta >= 0):
        raise ValueError("Beta parameters and schedule must be non-negative.")
    if not (pos_cum_pos_inst_scale >= 0 and pos_cum_neg_inst_scale >= 0 and
            neg_cum_pos_inst_scale >= 0 and neg_cum_neg_inst_scale >= 0):
        raise ValueError("Instantaneous regret scaling factors must be non-negative.")

    self._alpha_start = alpha_start
    self._alpha_max = alpha_max
    self._schedule_T_alpha = max(1.0, schedule_T_alpha)
    self._beta_start = beta_start
    self._beta_max = beta_max
    self._schedule_T_beta = max(1.0, schedule_T_beta)

    self._pos_cum_pos_inst_scale = pos_cum_pos_inst_scale
    self._pos_cum_neg_inst_scale = pos_cum_neg_inst_scale
    self._neg_cum_pos_inst_scale = neg_cum_pos_inst_scale
    self._neg_cum_neg_inst_scale = neg_cum_neg_inst_scale
    self._epsilon = 1e-9 # Small value for sign comparisons


def update_accumulate_regret(
    self, info_state_node, iteration_number, cfr_regrets
):
  """Updates cumulative regrets with adaptive discounting and asymmetric instantaneous regret scaling.

  Args:
    info_state_node: Data structure for the information set.
    iteration_number: The current CFR iteration (0-based).
    cfr_regrets: Instantaneous counterfactual regrets for the current iteration.
  Returns:
    Updated cumulative regret dictionary for each action.
  """
  t = float(iteration_number) + 1.0 # 1-indexed for weighting calculations
  iter_float = float(iteration_number) # 0-based for schedule progress

  # Calculate current_alpha for positive cumulative regret discounting
  t_norm_alpha = min(1.0, iter_float / self._schedule_T_alpha)
  current_alpha = self._alpha_start + (self._alpha_max - self._alpha_start) * t_norm_alpha

  # Calculate current_beta for negative cumulative regret discounting
  t_norm_beta = min(1.0, iter_float / self._schedule_T_beta)
  current_beta = self._beta_start + (self._beta_max - self._beta_start) * t_norm_beta

  updated_cumulative_regret = {}
  legal_actions = info_state_node.legal_actions

  for action in legal_actions:
      prev_cumulative_regret = info_state_node.cumulative_regret.get(action, 0.0)
      current_instantaneous_regret = cfr_regrets.get(action, 0.0)

      # Determine instantaneous regret scaling factor based on signs
      instantaneous_scaling_factor = 1.0 # Default

      if prev_cumulative_regret > self._epsilon: # Cumulative regret is positive
          if current_instantaneous_regret > self._epsilon:
              instantaneous_scaling_factor = self._pos_cum_pos_inst_scale
          elif current_instantaneous_regret < -self._epsilon:
              instantaneous_scaling_factor = self._pos_cum_neg_inst_scale
      elif prev_cumulative_regret < -self._epsilon: # Cumulative regret is negative
          if current_instantaneous_regret > self._epsilon:
              instantaneous_scaling_factor = self._neg_cum_pos_inst_scale
          elif current_instantaneous_regret < -self._epsilon:
              instantaneous_scaling_factor = self._neg_cum_neg_inst_scale
```

```
        # If one or both are near zero, instantaneous_scaling_factor remains 1.0

        scaled_instantaneous_regret = current_instantaneous_regret * instantaneous_scaling_factor

        # Calculate discount factor for the previous cumulative regret based on its sign
        discount_factor = 0.0
        if prev_cumulative_regret > 0:
            try:
                t_pow_alpha = math.pow(t, current_alpha)
                # If t^alpha overflows, the factor approaches 1.0
                discount_factor = t_pow_alpha / (t_pow_alpha + 1.0) if t_pow_alpha != float('inf') else 1.0
            except ValueError: # Fallback for potential issues with very large t and alpha
                discount_factor = 1.0 if t > 1.1 else 0.5 # t=1 is 0.5 (1/2), large t is 1.0 (inf/inf)
        else: # prev_cumulative_regret is non-positive
            try:
                t_pow_beta = math.pow(t, current_beta)
                # If t^beta overflows, the factor approaches 1.0
                discount_factor = t_pow_beta / (t_pow_beta + 1.0) if t_pow_beta != float('inf') else 1.0
            except ValueError: # Fallback for potential issues with very large t and beta
                discount_factor = 1.0 if t > 1.1 else 0.5 # t=1 is 0.5 (1/2), large t is 1.0 (inf/inf)

        discounted_prev_regret = prev_cumulative_regret * discount_factor

        # Add the scaled current instantaneous regret.
        new_cumulative_regret = discounted_prev_regret + scaled_instantaneous_regret

        updated_cumulative_regret[action] = new_cumulative_regret

    return updated_cumulative_regret


import math
import collections
import attr # Ensure attr is imported if not already globally


class PolicyFromRegretAccumulator:
  """Calculates the current policy using cumulative regrets plus an optimistic term
     based on the deviation of current cumulative regret from its EMA."""

  def __init__(self,
               use_squared_weights: bool = True,
               cumulative_regret_ema_alpha: float = 0.1, # EMA factor for cumulative regrets (lower means more smoothing)
               optimism_trend_scale: float = 0.5):      # Scaling factor for the R^t - EMA(R^{t-1}) term
      """Initializes the policy calculator with trend-based optimism.

      Args:
          use_squared_weights: If True (default), use max(0, R + scaled_optimism)^2 for policy weights (CFR+ style).
                               If False, use max(0, R + scaled_optimism) for policy weights (RM+ style).
          cumulative_regret_ema_alpha: Decay factor for the EMA of cumulative regrets (0 < alpha <= 1).
                                       Higher alpha gives more weight to recent cumulative regrets.
          optimism_trend_scale: Scaling factor applied to the difference between current cumulative regret
                                and its EMA (R^t - EMA(R^{t-1})) before adding it to R^t.
      """
      if not (0.0 < cumulative_regret_ema_alpha <= 1.0):
          raise ValueError("cumulative_regret_ema_alpha must be between 0 (exclusive) and 1 (inclusive).")
      if optimism_trend_scale < 0:
           raise ValueError("optimism_trend_scale must be non-negative.")

      self._use_squared_weights = use_squared_weights
      self._cumulative_regret_ema_alpha = cumulative_regret_ema_alpha
      self._optimism_trend_scale = optimism_trend_scale
      # Store EMA of *cumulative* regrets: {infoset_index: {action: ema_value}}
      self._ema_cumulative_regrets = collections.defaultdict(lambda: collections.defaultdict(float))
      # Epsilon for safe floating point comparisons for policy normalization.
      self._epsilon_norm = 1e-12


  def get_updated_current_policy(self, info_state_node, iteration_number, cfr_regrets, previous_policy):
    """Calculates the current policy using cumulative regrets plus trend-based optimism.

    Args:
      info_state_node: a data structure corresponding to an information set with
                       cumulative_regret and cumulative_policy stored.
                       cumulative_regret (R^t) has been updated by RegretAccumulator
                       for the current iteration.
      iteration_number: the current CFR iteration (0-based) (unused but kept for API).
      cfr_regrets: the instantaneous counterfactual regrets (r^t) for the current iteration (unused).
```

```
      previous_policy: the previous policy at the current information set (unused).
    Returns:
      updated current policy dictionary at the current information set.
    """
    cumulative_regrets_map = info_state_node.cumulative_regret  # R^t
    actions = info_state_node.legal_actions
    infoset_index = info_state_node.index_in_tabular_policy  # Unique key for EMA storage

    action_to_policy_weight = {}
    sum_policy_weights = 0.0
    action_to_new_ema = {} # Store new EMA values before overwriting state

    if not actions: # Handle cases with no legal actions
        return {}

    num_actions = len(actions) # Used for uniform policy fallback

    # Get the EMA map for this specific infoset
    infoset_ema_map = self._ema_cumulative_regrets[infoset_index]

    for action in actions:
        # Get the current cumulative regret R^t(a)
        current_cum_regret = cumulative_regrets_map.get(action, 0.0)
        # Get the previous EMA of cumulative regret EMA(R^{t-1})(a)
        prev_ema_cum_regret = infoset_ema_map.get(action, 0.0) # Defaults to 0.0 on first encounter

        # Calculate the deviation = R^t(a) - EMA(R^{t-1})(a)
        regret_trend = current_cum_regret - prev_ema_cum_regret

        # Calculate the scaled optimistic term
        scaled_optimism = self._optimism_trend_scale * regret_trend

        # Calculate the value used for Regret Matching: R^t(a) + scaled_optimism
        combined_value = current_cum_regret + scaled_optimism

        # Apply Regret Matching logic: take the positive part.
        positive_combined_value = max(0.0, combined_value)

        # Use linear or squared weighting based on the flag
        if self._use_squared_weights:
            policy_weight = positive_combined_value ** 2 # CFR+ style
        else:
            policy_weight = positive_combined_value      # RM+ style

        action_to_policy_weight[action] = policy_weight
        sum_policy_weights += policy_weight

        # Update the EMA of cumulative regret for the next iteration (t)
        # EMA^t = alpha * R^t + (1 - alpha) * EMA^{t-1}
        new_ema_cum_regret = self._cumulative_regret_ema_alpha * current_cum_regret + \
                             (1.0 - self._cumulative_regret_ema_alpha) * prev_ema_cum_regret
        action_to_new_ema[action] = new_ema_cum_regret


    # Update the stored EMA values *after* using the previous values for all actions
    for action in actions:
        infoset_ema_map[action] = action_to_new_ema[action]

    # Normalize to get the policy
    info_state_policy = {}
    if sum_policy_weights > self._epsilon_norm: # Use normalization epsilon
      # Normalize positive weights (or their squares) to get policy probabilities
      for action in actions:
        info_state_policy[action] = action_to_policy_weight[action] / sum_policy_weights
    else:
      # Default to uniform policy if sum of weights is zero or numerically close to zero
      uniform_prob = 1.0 / num_actions if num_actions > 0 else 0.0
      for action in actions:
        info_state_policy[action] = uniform_prob

    return info_state_policy


class PolicyAccumulator:
  """Updates cumulative policy using adaptive polynomial weighting (like DCFR/PDCFR+)."""

  def __init__(self,
               gamma_start: float = 1.0, gamma_max: float = 5.0, gamma_schedule_T: float = 500.0):
```

```
    """Initializes the policy accumulator with adaptive gamma.

    Args:
        gamma_start: Initial exponent for weighting the current policy (t^gamma).
        gamma_max: Max gamma value for later exploitation.
        gamma_schedule_T: Number of iterations over which gamma transitions.
    """
    if gamma_start < 0 or gamma_max < 0 or gamma_schedule_T < 0:
        raise ValueError("PolicyAccumulator gamma parameters must be non-negative.")
    self._gamma_start = gamma_start
    self._gamma_max = gamma_max
    self._gamma_schedule_T = max(1.0, gamma_schedule_T) # Avoid division by zero


def update_accumulate_policy(
    self,
    info_state_node,
    iteration_number,
    info_state_policy,
    cfr_regrets, # Not used
    reach_prob,
    counterfactual_reach_prob, # Not used
):
  """Updates the cumulative policy using adaptive polynomial weighting.

  Args:
    info_state_node: Data structure for the information set.
    iteration_number: The current CFR iteration (0-based).
    info_state_policy: The current policy profile for this iteration.
    reach_prob: Player's reach probability for this iteration.
  Returns:
    Updated cumulative policy dictionary for each action.
  """
  # Use 1-based iteration number t for calculations
  t = float(iteration_number) + 1.0
  # Use 0-based iteration for schedule progress calculation
  iter_float = float(iteration_number)

  # Calculate current_gamma based on linear schedule
  t_norm_gamma = min(1.0, iter_float / self._gamma_schedule_T)
  current_gamma = self._gamma_start + (self._gamma_max - self._gamma_start) * t_norm_gamma

  # Calculate t^current_gamma, which is used for both discounting and weighting.
  t_pow_gamma = 1.0
  if t > 0: # t is 1-indexed (float(iteration_number) + 1.0), so t=0 is not possible here.
      try:
          t_pow_gamma = math.pow(t, current_gamma)
      except ValueError: # Fallback for extremely large exponents to avoid overflow, treat as very large.
          t_pow_gamma = float('inf')

  # Calculate the discount factor for the previous cumulative policy sum.
  # This mirrors the discounting logic used for positive cumulative regrets in RegretAccumulator.
  # If t_pow_gamma is 'inf', discount factor approaches 1.0.
  policy_discount_factor = t_pow_gamma / (t_pow_gamma + 1.0) if t_pow_gamma != float('inf') else 1.0

  # Calculate the weight for the current policy's contribution (Adaptive CFR style).
  # This weights the current policy by t^gamma and the player's reach probability.
  current_policy_contribution_weight = t_pow_gamma * reach_prob

  updated_cumulative_policy = {}
  # Iterate over all legal actions and actions present in the cumulative policy
  # to ensure proper discounting and updating of all relevant entries.
  all_actions = set(info_state_node.legal_actions) | set(info_state_node.cumulative_policy.keys())

  for action in all_actions:
      previous_cumulative_policy = info_state_node.cumulative_policy.get(action, 0.0)
      current_policy_prob = info_state_policy.get(action, 0.0)

      # Discount the previously accumulated policy sum.
      discounted_previous_cumulative_policy = previous_cumulative_policy * policy_discount_factor

      # Add the current policy's weighted contribution.
      current_contribution = current_policy_contribution_weight * current_policy_prob

      updated_cumulative_policy[action] = (
          discounted_previous_cumulative_policy + current_contribution
      )

```

```
    # Ensure all *legal* actions are present in the final dict, even if value is 0.0
    for action in info_state_node.legal_actions:
        if action not in updated_cumulative_policy:
            # This happens if an action was legal but never accumulated policy before
            # and isn't in current policy (prob=0). Its value should be 0
            # after discounting the previous 0 and adding 0 contribution.
            updated_cumulative_policy[action] = 0.0

    return updated_cumulative_policy
```

7.3 Results on 11 Games

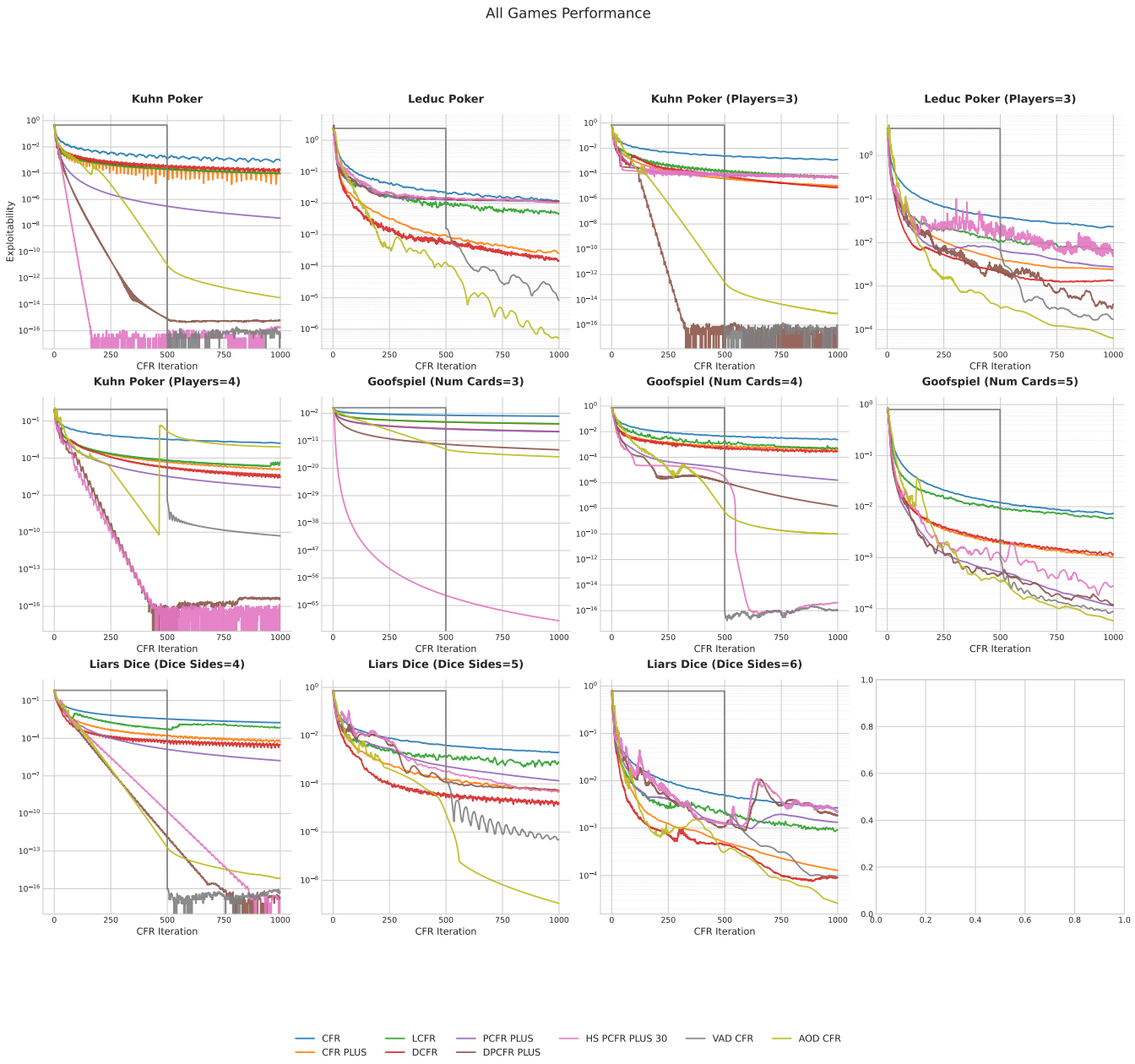

Figure 3: CFR variants performances on All Games.

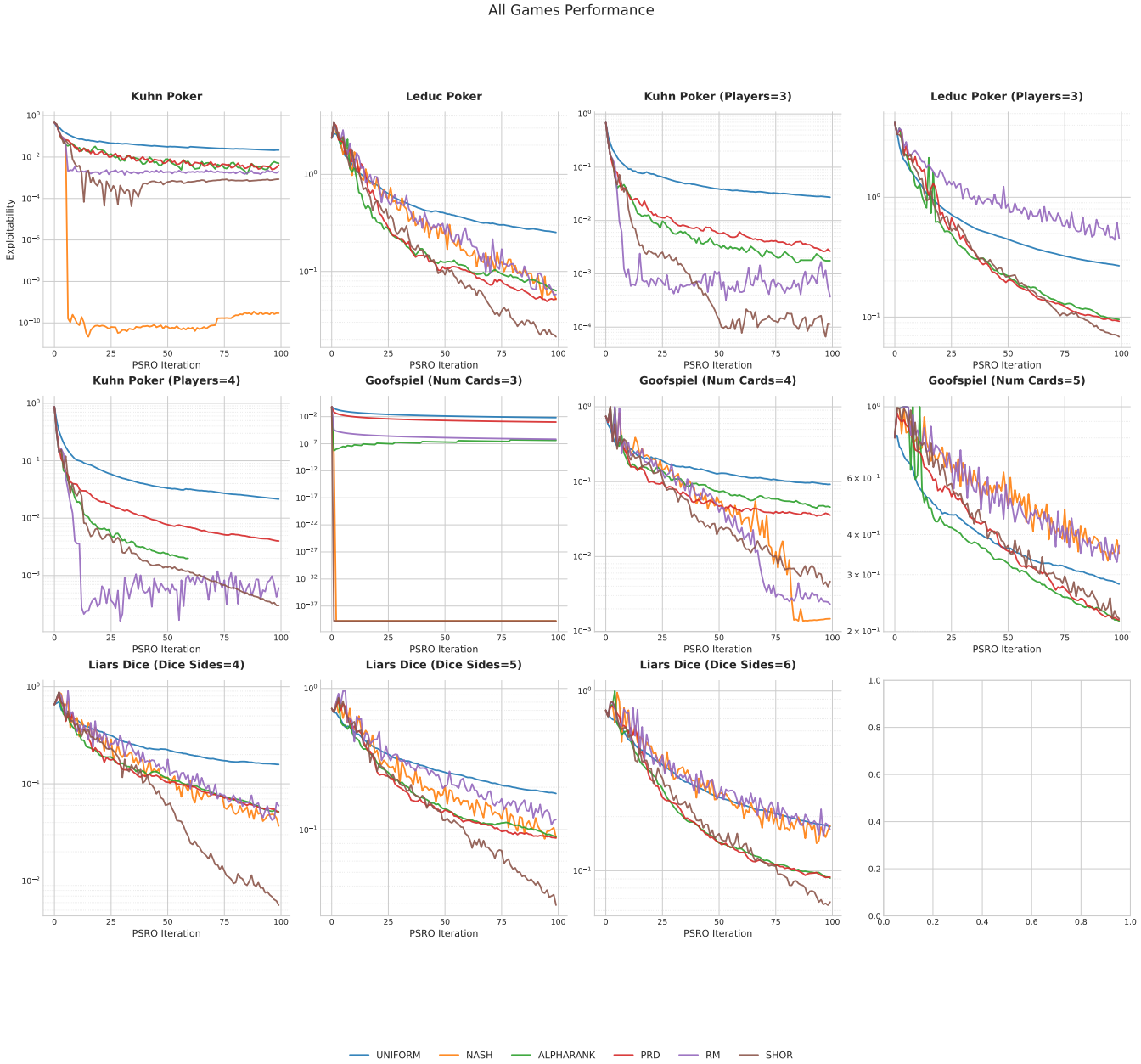

Figure 4: PSRO variants performances on All Games.

7.4 Prompts

Listing 8: 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 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*.