15-888 Computational Game Solving (Fall 2025) Released on: Oct. 12, 2025

Homework 2

Student: boxiangf **Due on: Oct. 31, 2025**

Instructions Submit your homework on Gradescope. Submit a single `pdf` file containing **both** your written solutions **and** your code attached to the end (for example, using a `verbatim` environment). You can (but are not required to) typeset your written solutions in the `.tex` file provided in the homework `.zip`.

1 (Coarse) Correlated equilibria and dominated actions (20 points)

An action $a_i \in \mathcal{A}_i$ *strictly dominates* another action $a'_i \in \mathcal{A}_i$ if $u_i(a_i, a_{-i}) > u_i(a'_i, a_{-i})$ for every possible opponent action profile a_{-i}. In this case, we call a'_i *strictly dominated*. If a_i strictly dominates every other action, then we call it *strictly dominant*.

Problem 1.1 (5 points). Prove that if an action $\hat{a}_i \in \mathcal{A}_i$ is *strictly dominant*, then Player i plays $\hat{a}_i$ with probability 1 in every coarse correlated equilibrium (CCE).

Solution. Let $\hat{a}_i \in \mathcal{A}_i$ be strictly dominant for Player i. So $u_i(\hat{a}_i, a_{-i}) > u_i(a_i, a_{-i})$ for all a_{-i} and for all $a_i \neq \hat{a}_i$. We proceed by contradiction. Suppose there exists a CCE $\boldsymbol{\mu} \in \Delta(\mathcal{A}_1 \times \cdots \times \mathcal{A}_n)$ such that Player i assigns positive probability to some strategy profile with $a_i \neq \hat{a}_i$. By definition of a CCE, we have

$$\mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(a_1, \cdots, a_n) \geq \mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(a'_i, a_{-i})$$

for any deviation $a'_i \in \mathcal{A}_i$. Let $a'_i = \hat{a}_i$. The CCE definition implies

$$\mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(a_1, \cdots, a_n) \geq \mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(\hat{a}_i, a_{-i}). \tag{1}$$

But the action $\hat{a}_i \in \mathcal{A}_i$ is strictly dominant, so $u_i(\hat{a}_i, a_{-i}) > u_i(a_i, a_{-i})$ whenever $a_i \neq \hat{a}_i$. Since we assumed Player i assigns positive probability to a strategy profile with $a_i \neq \hat{a}_i$, taking expectations over $\boldsymbol{\mu}$ gives the strict inequality

$$\mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(a_1, \cdots, a_n) < \mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(\hat{a}_i, a_{-i})$$

which contradicts Equation 1. Therefore, Player i cannot assign positive probability to a strategy profile with $a_i \neq \hat{a}_i$. Hence Player i plays $\hat{a}_i$ with probability 1 in every CCE. □

Problem 1.2 (5 points). Prove that if an action $\hat{a}_i \in \mathcal{A}_i$ is *strictly dominated*, then Player i plays $\hat{a}_i$ with probability 0 in every correlated equilibrium (CE).

Solution. Let $\hat{a}_i \in \mathcal{A}_i$ be strictly dominated by some $a_i^* \in \mathcal{A}_i$. So $u_i(a_i^*, a_{-i}) > u_i(\hat{a}_i, a_{-i})$ for all a_{-i}. We proceed by contraction. Suppose there exists a CE $\boldsymbol{\mu} \in \Delta(\mathcal{A}_1 \times \cdots \times \mathcal{A}_n)$ such that Player i assigns positive probability to some strategy profile including $\hat{a}_i$. By definition of a CE, we have

$$\mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(a_1, \cdots, a_n) \geq \mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(\phi_i(a_i), a_{-i}) \tag{2}$$

for all $\phi_i : \mathcal{A}_i \to \mathcal{A}_i$. Define the deviation function as

$$\phi_i(\hat{a}_i) = a_i^*, \quad \phi_i(a_i) = a_i \text{ for } a_i \neq \hat{a}_i.$$

We have

$$u_i(\phi_i(a_i), a_{-i}) = \begin{cases} u_i(a_i, a_{-i}), & a_i \neq \hat{a}_i, \\ u_i(a_i^\star, a_{-i}) > u_i(\hat{a}_i, a_{-i}), & a_i = \hat{a}_i. \end{cases}$$

Since we assumed Player i assigns positive probability to some strategy profile including $\hat{a}_i$, taking expectations over $\boldsymbol{\mu}$ gives the strict inequality

$$\mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(a_1, \cdots, a_n) < \mathbb{E}_{(a_1,\cdots,a_n)\sim\mu} u_i(\phi_i(a_i), a_{-i})$$

which contradicts Equation 2. Therefore, Player i cannot assign positive probability to a strategy profile including $\hat{a}_i$. Hence Player i plays $\hat{a}_i$ with probability 0 in every CE. □

Problem 1.3 (10 points). Show that there is a normal-form game and a CCE in that game in which Player 1 plays a strictly dominated action with strictly positive probability.

★ Hint: Consider the normal-form game

$$\begin{bmatrix} (2,2) & (0,0) \\ (1,1) & (1,1) \\ (0,0) & (0,0) \end{bmatrix}.$$

Show that there is a CCE in this game with the desired property.

Solution. Define a normal-form game as follows:

	C_1	C_2
R_1	$(2,2)$	$(0,0)$
R_2	$(1,1)$	$(1,1)$
R_3	$(0,0)$	$(0,0)$

.

Note that R_3 is strictly dominated by R_2 for Player 1 since whatever action Player 2 plays, R_2 obtains an utility of 1, whereas R_3 obtains an utility of 0. Define the correlated distribution μ as

$$\mu(R_1, C_1) = 0.5, \quad \mu(R_3, C_2) = 0.5.$$

We show that μ is a CCE. That is, we show

$$\mathbb{E}_{(a_1,a_2)\sim\mu} u_i(a_1, a_2) \geq \mathbb{E}_{(a_1,a_2)\sim\mu} u_i(a_i', a_{-i}) \tag{3}$$

for all deviations $a_i' \in \mathcal{A}_i$ and $i \in \{1, 2\}$. For $i = 1$, the utility under μ gives

$$\mathbb{E}_{(a_1,a_2)\sim\mu} u_1(a_1, a_2) = 0.5 \cdot 2 + 0.5 \cdot 0 = 1.$$

Deviating to R_1 gives

$$\mathbb{E}_{(a_1,a_2)\sim\mu} u_1(R_1, a_2) = 0.5 \cdot 2 + 0.5 \cdot 0 = 1.$$

Deviating to R_2 gives

$$\mathbb{E}_{(a_1,a_2)\sim\mu} u_1(R_2, a_2) = 0.5 \cdot 1 + 0.5 \cdot 1 = 1.$$

Deviating to R_3 gives

$$\mathbb{E}_{(a_1,a_2)\sim\mu} u_1(R_3, a_2) = 0.5 \cdot 0 + 0.5 \cdot 0 = 0.$$

So Player 1 has no incentive to deviate to any other deterministic action strategy as Equation 3 holds true. For $i = 2$, the utility under μ gives

$$\mathbb{E}_{(a_1,a_2)\sim\mu} u_2(a_1, a_2) = 0.5 \cdot 2 + 0.5 \cdot 0 = 1.$$

Deviating to C_1 gives

$$\mathbb{E}_{(a_1,a_2)\sim\mu} u_2(a_1, C_1) = 0.5 \cdot 2 + 0.5 \cdot 0 = 1.$$

Deviating to C_2 gives

$$\mathbb{E}_{(a_1,a_2)\sim\mu} u_2(a_1, C_2) = 0.5 \cdot 0 + 0.5 \cdot 0 = 0.$$

So Player 2 has no incentive to deviate to any other deterministic action strategy as Equation 3 holds true. So the correlated distribution μ is a CCE where Player 1 plays the strictly dominated action R_3 with strictly positive probability. Also note that this is not the only CCE with this property. In fact, any μ satisfying

$$\mu(R_1, C_1) = p, \quad \mu(R_3, C_2) = 1 - p \quad \text{for } p \in \mathbb{R}_{[0.5,1)}$$

satisfies the same properties. □

2 Safe Abstraction (30 points)

Consider a two-player zero-sum extensive-form game Γ that proceeds as follows.

Step 1:

- Each player $i \in \{1, 2\}$ privately rolls a fair n-sided die, and observes its outcome $\theta_i \in \{1, \ldots, n\}$.
- Both players simultaneously pick actions a_1, a_2 from some action set $\mathcal{A}$ of size m.

Step 2:

- Each player $i \in \{1, 2\}$ privately rolls another fair n-sided die, and observes its outcome $\theta'_i \in \{1, \ldots, n\}$.
- Both players again simultaneously pick actions $a_1, a_2 \in \mathcal{A}$.

The utility of P1[1] depends only on the sums $\theta_1 + \theta'_1$ and $\theta_2 + \theta'_2$ and the actions a_1, a_2, a'_1, a'_2. That is, each player only cares about the *sum* of their two dice, not on their individual values.

Note that we can identify a decision point (information set) of a player i by specifying either θ_i (for Step 1), or a tuple $(\theta_i, \boldsymbol{a}, \theta'_i)$ (for Step 2), where $\boldsymbol{a} = (a_1, a_2)$.

2.1 Warm-up

Problem 2.1 (3 points). In terms of m and n, how many *terminal nodes* does this game have? How many *information sets* does each player have? How about *sequences*? (No proofs required; just state the answers.)

[1]This is a zero-sum game: P2's utility is the negative of P1's.

Solution. **Terminal nodes**: n^4m^4
Information sets: $n + n^2m^2$
Sequences: $nm + n^2m^3$ (using definition in Lecture 5) or $nm + n^2m^3 + 1$ (using definition where the null set is included to indicate a sequence of *no actions taken*) □

2.2 Symmetric strategies and CFR

Let $j_i = (\theta_i, a_i, a_{-i}, \theta_i')$ and $\tilde{j}_i = (\tilde{\theta}_i, a_i, a_{-i}, \tilde{\theta}_i')$ be two infosets of the same player i in Step 2. We say that j_i and $\tilde{j}_i$ are *equivalent* if $\theta_i + \theta_i' = \tilde{\theta}_i + \tilde{\theta}_i'$ (note that this forces the set of actions to be the same in both infosets).

Call a behavioral strategy $\boldsymbol{b}_i : \mathcal{J}_i \to \Delta(\mathcal{A})$, where $\mathcal{J}_i$ is player i's information partition, *symmetric* if equivalent infosets have identical action distributions, that is, $\boldsymbol{b}_i(\cdot|j_i) = \boldsymbol{b}_i(\cdot|j_i')$ whenever j_i, j_i' are equivalent.

Suppose both players run CFR with simultaneous (as opposed to alternating) updates. Let $\boldsymbol{b}_i^{(t)}$ be the behavioral strategy played by player i at time t. In CFR, the counterfactual utility given to the regret minimizer at infoset $j_i \in \mathcal{J}_i$ at time t is given by

$$u_i^{(t)}(a_i|j_i) = \sum_{\substack{h \in j_i \\ z \succeq ha}} \boldsymbol{b}_i^{(t)}(z|(h, a_i)) \cdot \boldsymbol{b}_{-i}^{(t)}(z) \cdot c(z) \cdot u_i(z), \tag{4}$$

where

- (h, a_i) is the child of node h reached by playing action a_i,
- the sum is over terminal nodes z that are descendants of nodes (h, a_i) for $h \in j_i$,
- $\boldsymbol{b}_i(z|(h, a_i))$ is the probability that player i plays all actions on the $(h, a_i) \to z$ path,
- $\boldsymbol{b}_{-i}(z)$ (resp. $c(z)$) is the probability that the opponent (resp. chance) plays all actions on the path from the root node to z, and
- $u_i(z)$ is the utility of terminal node z for player i.

Problem 2.2 (8 points). Let $\boldsymbol{b}_i^{(t)} : \mathcal{J}_i \to \Delta(\mathcal{A})$ be the behavioral strategy played by player i at time t in CFR. Show that if $\boldsymbol{b}_{-i}^{(1)}, \ldots, \boldsymbol{b}_{-i}^{(t-1)}$ are all symmetric, then $\boldsymbol{b}_i^{(1)}, \ldots, \boldsymbol{b}_i^{(t)}$ are also symmetric.

★ Hint: Given two equivalent infosets $(\theta_i, \boldsymbol{a}, \theta_i')$, $(\tilde{\theta}_i, \boldsymbol{a}, \tilde{\theta}_i')$, show that these two infosets will observe the same counterfactual utility $u_i^{(t)}(\cdot|j)$ at every timestep t.

Lemma. Let $\boldsymbol{b}_i^{(t)} : \mathcal{J}_i \to \Delta(\mathcal{A})$ be the behavioral strategy played by player i at time t in CFR. Let $\boldsymbol{b}_{-i}^{(1)}, \ldots, \boldsymbol{b}_{-i}^{(t-1)}$ be all symmetric. If j_i and $\tilde{j}_i$ are equivalent information sets, then $u_i^{(s)}(a_i|j_i) = u_i^{(s)}(a_i|\tilde{j}_i)$ for all timestep $s < t$. □

Proof. At Step 1, the equivalence class of information sets is undefined. Therefore, we only need to consider Step 2. Let j_i and $\tilde{j}_i$ be equivalent information sets in Step 2. Let timestep $s < t$ be arbitrary but fixed. At Step 2, after choosing a_i the game terminates, so $(h, a_i) = \perp$. So

$$\boldsymbol{b}_i^{(s)}(z|(h, a_i)) = \boldsymbol{b}_i^{(s)}(z|\perp) = 1$$

as we reach leaf z with certainty. Therefore, Equation 4 reduces to

$$u_i^{(s)}(a_i|j_i) = \sum_{\substack{h \in j_i \\ z \succeq ha}} \boldsymbol{b}_{-i}^{(s)}(z) \cdot c(z) \cdot u_i(z)$$

and

$$u_i^{(s)}(a_i|\tilde{j}_i) = \sum_{\substack{h \in \tilde{j}_i \\ \tilde{z} \sqsupseteq ha}} \boldsymbol{b}_{-i}^{(s)}(\tilde{z}) \cdot c(\tilde{z}) \cdot u_i(\tilde{z})$$

at Step 2 information sets j_i and $\tilde{j}_i$ respectively. We now show $u_i^{(s)}(a_i|j_i) = u_i^{st)}(a_i|\tilde{j}_i)$ by comparing each product inside the summation in turn.

For the environment chance term, we note that the die is fair. So

$$c(z) = \frac{1}{n^4}$$

for all leaf nodes z. So $c(z) = c(\tilde{z})$.

For the utility term, recall that j_i and $\tilde{j}_i$ are equivalent, so $\theta_i + \theta_i' = \tilde{\theta}_i + \tilde{\theta}_i'$ by definition. Since the joint action profile to reach z and $\tilde{z}$ is the same & the utility of Player 1 is dependent on the sums $\theta_i + \theta_i'$ and $\tilde{\theta}_i + \tilde{\theta}_i'$, these two being equal means that $u_i(z) = u_i(\tilde{z})$.

For the opponent's behavioral strategy term, recall that $\boldsymbol{b}_{-i}^{(s)}$ is symmetric. Since j_i and $\tilde{j}_i$ are equivalent & the joint action profile is the same for both z and $\tilde{z}$, we have

$$\boldsymbol{b}_{-i}^{(s)}(z) = \boldsymbol{b}_{-i}^{(s)}(\tilde{z})$$

as $\boldsymbol{b}_i(\cdot|j_i) = \boldsymbol{b}_i(\cdot|j_i')$ whenever j_i, j_i' are equivalent. Since all three terms are equal, this means that

$$u_i^{(s)}(a_i|j_i) = u_i^{(s)}(a_i|\tilde{j}_i)$$

for all timestep $s < t$. □

Solution. Let $\boldsymbol{b}_i^{(t)} : \mathcal{J}_i \to \Delta(\mathcal{A})$ be the behavioral strategy played by player i at time t in CFR. Let $\boldsymbol{b}_{-i}^{(1)}, \ldots, \boldsymbol{b}_{-i}^{(t-1)}$ be all symmetric. We prove $\boldsymbol{b}_i^{(1)}, \ldots, \boldsymbol{b}_i^{(t)}$ are also symmetric. That is, if j_i and $\tilde{j}_i$ are equivalent, then $\boldsymbol{b}_i^{(s)}(\cdot|j_i) = \boldsymbol{b}_i^{(s)}(\cdot|\tilde{j}_i)$ for all $s \leq t$. Let j_i and $\tilde{j}_i$ be equivalent information sets. We proceed by induction on s.

Base: For $s = 1$, every information set j_i and every action a_i is initialized with zero regret. So the regret minimizer outputs identical $\boldsymbol{b}_i^{(1)}$ for every information set j_i in Step 2. So $\boldsymbol{b}_i^{(1)}(\cdot|j_i) = \boldsymbol{b}_i^{(1)}(\cdot|\tilde{j}_i)$ trivially.

Induction Hypothesis: Assume $\boldsymbol{b}_i^{(s)}(\cdot|j_i) = \boldsymbol{b}_i^{(s)}(\cdot|\tilde{j}_i)$ for all $s < t$.

Induction Step: For $s = t$, recall from our Lemma that $u_i^{(s)}(a_i|j_i) = u_i^{(s)}(a_i|\tilde{j}_i)$ for all timestep $s < t$. So the history of counterfactual utilities $u_i^{(s)}$ and action distributions $\boldsymbol{b}_i^{(s)}$ is the same for j_i and $\tilde{j}_i$. Since regret minimizers are a deterministic function of these two histories, for identical histories, the minimizer outputs the same future action distribution. So $\boldsymbol{b}_i^{(t)}(\cdot|j_i) = \boldsymbol{b}_i^{(t)}(\cdot|\tilde{j}_i)$.

By induction, we have $\boldsymbol{b}_i^{(1)}, \ldots, \boldsymbol{b}_i^{(t)}$ are also symmetric. □

It follows by induction that both players will play symmetric strategies at all timesteps. Use this fact for the remainder of the problem.

Problem 2.3 (8 points). Show that, in this game, CFR can be implemented in $O(n^2m^4)$ time per iteration. Briefly discuss: how does this time complexity compare to that of the naive implementation of CFR, whose runtime would be linear in the number of nodes in the tree?

★ Hint: Because of the symmetry condition that you have just proven, many infosets are "copies" of each other, and you don't need to do the work of storing multiple copies of the same infoset. The naive computation will not work—you have to be a bit clever about saving reusable work across infosets.

Solution. Assume that each leaf visit is evaluated in constant time $O(1)$. By the symmetry condition, whenever information sets j_i and $\tilde{j}_i$ are equivalent, the action distributions are equal. So CFR only needs to do one evaluation for each equivalence class of information sets defined by the equivalence relation $\theta_i + \theta'_i = \tilde{\theta}_i + \tilde{\theta}'_i$. The number of equivalence classes at Step 1 is $n = O(n)$, and the number of equivalence classes at Step 2 is $2n - 1 = O(n)$. So the total number of information set equivalence classes for Player i is

$$O(n) + O(n) = O(n).$$

Since both players run CFR simultaneously, the total number of distinct equivalence classes for the entire game is

$$O(n) \cdot O(n) = O(n^2)$$

corresponding to $O(n)$ from Player 1 and $O(n)$ from Player 2. The size of the action profile is

$$m^2 \cdot m^2 = O(m^4)$$

corresponding to the m^2 possible actions at Step 1 multiplied by the m^2 possible actions at Step 2 (from both players). Therefore, the total CFR cost per iteration is the enumeration of all distinct equivalence class and action profiles

$$O(n^2) \cdot O(m^4) = O(n^2m^4).$$

The naive implementation of CFR costs $O(n^4m^4)$ per iteration as it has to enumerate all possible information sets and action profiles. By exploiting the symmetry condition, the time complexity is cut down by $O(n^2)$, a comparable speedup if the number of die sides n is large. This also means the runtime is sub-linear in the number of nodes in the tree. □

2.3 Imperfect-recall abstraction

Now consider the imperfect-recall abstraction of Γ in which equivalent information sets have been merged into larger infosets. That is, infosets in Step 2 are now identified by tuples $(\theta_i^*, \boldsymbol{a})$ where $\theta_i^* = \theta_i + \theta'_i$. Call this game $\hat{\Gamma}$. We define CFR in imperfect-recall games using the formula (4).

CFR on imperfect-recall zero-sum games is *not* generally guaranteed to converge to Nash equilibrium, even in averages, but you will show that *in this particular class of games* CFR works.

Problem 2.4 (8 points). Show that, when the local regret minimizer is regret matching, running CFR on $\hat{\Gamma}$ produces the same iterates as running CFR on Γ.

★ Hint: Read the next problem before solving this one.

Lemma. Let the local regret minimizer be Regret Matching ($\mathcal{RM}(\cdot)$). Then $\mathcal{RM}(kR) = \mathcal{RM}(R)$ for $k > 0$ (i.e. scale invariant). □

Proof. By definition of $\mathcal{RM}(\cdot)$ (Lecture 4):

$$\begin{aligned}\mathcal{RM}(kR) &= \frac{\max(kR, 0)}{\|\max(kR, 0)\|_1} \\ &= \frac{k \cdot \max(R, 0)}{k \cdot \|\max(R, 0)\|_1} \\ &= \frac{\max(R, 0)}{\|\max(R, 0)\|_1} \\ &= \mathcal{RM}(R)\end{aligned}$$

as the scalar k commutes with $\max(\cdot)$ and L1 norm. □

Solution. Let the local regret minimizer be Regret Matching ($\mathcal{RM}(\cdot)$). Let Γ be the full game with information sets $j_i \in \mathcal{J}_i$ for player $i \in \{1, 2\}$. Let $\hat{\Gamma}$ be the abstracted game with merged information sets $\hat{j}_i \in \hat{\mathcal{J}}_i$ for player $i \in \{1, 2\}$. That is, each $\hat{j}_i$ is an equivalence class $C(\hat{j}_i)$ of equivalent information sets defined by the equivalence relation $\theta_i + \theta_i' = \hat{\theta}_i + \hat{\theta}_i'$. We show that $CFR(\Gamma)$ and $CFR(\hat{\Gamma})$ produce the same iterates. That is, for all $s \leq t$,

$$\boldsymbol{b}_i^{(s)}(\cdot|j_i) = \boldsymbol{b}_i^{(s)}(\cdot|\hat{j}_i)$$

for all $j_i \in C(\hat{j}_i)$ and $\hat{j}_i \in \hat{\mathcal{J}}_i$. We proceed by induction on s.

Base: For $s = 1$, every information set j_i, abstracted information set $\hat{j}_i$ and their (identical) action profiles is initialized with zero regret. So the regret minimizer outputs identical $\boldsymbol{b}_i^{(1)}$ for every information set or abstracted information set. So $\boldsymbol{b}_i^{(1)}(\cdot|j_i) = \boldsymbol{b}_i^{(1)}(\cdot|\hat{j}_i)$ trivially.

Induction Hypothesis: Assume $\boldsymbol{b}_i^{(s)}(\cdot|j_i) = \boldsymbol{b}_i^{(s)}(\cdot|\hat{j}_i)$ for all $s < t$.

Induction Step: For $s = t$. Let $j_i \in C(\hat{j}_i)$ and $\hat{j}_i \in \hat{\mathcal{J}}_i$ be arbitrary but fixed. Define the aggregated regret vector over an equivalence class $C(\hat{j}_i)$ as

$$\hat{R}_i^{(t)}(a_i \mid \hat{j}_i) \coloneqq \sum_{j_i \in C(\hat{j}_i)} R_i^{(t)}(a_i \mid j_i).$$

Recall from Problem 2.2 that under the induction hypothesis, the same counterfactual utility $u_i^{(s)}(\cdot|j)$ and strategy $\boldsymbol{b}_i^{(s)}(\cdot|j)$ for $s < t$ is observed for all information sets in the equivalence class. This means that for the action a_i, the regret must also be the same across all $j_i \in C(\hat{j}_i)$ as it is a function of the same history of strategies and counterfactual utilities. So for $s = t - 1$,

$$\hat{R}_i^{(t-1)}(a_i \mid \hat{j}_i) = |C(\hat{j}_i)| \cdot R_i^{(t-1)}(a_i \mid j_i)$$

for any representative $j_i \in C(\hat{j}_i)$. Using $\mathcal{RM}$, we have

$$\begin{aligned}\boldsymbol{b}_i^{(t)}(\cdot|\hat{j}_i) &= \mathcal{RM}(\hat{R}_i^{(t-1)}(\cdot \mid \hat{j}_i)) \\ &= \mathcal{RM}(|C(\hat{j}_i)| \cdot R_i^{(t-1)}(\cdot \mid j_i)) \\ &= \mathcal{RM}(R_i^{(t-1)}(\cdot \mid j_i)) \\ &= \boldsymbol{b}_i^{(t)}(\cdot|j_i)\end{aligned}$$

where we used the scale invariance of $\mathcal{RM}$ proven in the Lemma above.

By induction, we have for all timestep t, $\boldsymbol{b}_i^{(t)}(\cdot|\hat{j}_i) = \boldsymbol{b}_i^{(t)}(\cdot|j_i)$ for all $j_i \in C(\hat{j}_i)$ and $\hat{j}_i \in \hat{\mathcal{J}}_i$. So the iterates from $CFR(\Gamma)$ and $CFR(\hat{\Gamma})$ are the same. □

Problem 2.5 (3 points). Your proof in the previous problem should have used a property specific to (variants of) regret matching. What is that property? Suppose that we were to instead run CFR in Γ with FTRL at each infoset and regularization parameter η. What would we need to do to η in $\hat{\Gamma}$ to recover the same iterates? (No proof required—just state the change)

Solution. We used (and proved in the Lemma) the property that Regret Matching is scale invariant. That is, $\mathcal{RM}(kR) = \mathcal{RM}(R)$. Since the aggregated regret vector over an equivalence class is scaled by $|C(\hat{j}_i)|$ compared to the initial regret vector over j_i, for FTRL we need to decrease the learning rate η by $|C(\hat{j}_i)|$ to account for the increased magnitude of the aggregated regret vector. That is,

$$\eta_{\hat{\Gamma}} = \frac{\eta_\Gamma}{|C(\hat{j}_i)|}$$

is required to recover the same iterates. □

3 Counterfactual Regret Minimization (50 points)

In this problem, you will implement the CFR regret minimizer for sequence-form decision problems.

You will run your CFR implementation on three games: rock-paper-superscissors (a simple variant of rock-paper-scissors, where beating paper with scissors gives a payoff of 2 instead of 1) and two well-known poker variants: Kuhn poker [Kuhn, 1950] and Leduc poker [Southey et al., 2005]. A description of each game is given in the zip of this homework, according to the format described in Section 3.1. The zip of the homework also contains a stub Python file to help you set up your implementation.

3.1 Format of the game files

Each game is encoded as a json file with the following structure.

- At the root, we have a dictionary with three keys: `decision_problem_pl1`, `decision_problem_pl2`, and `utility_pl1`. The first two keys contain a description of the tree-form sequential decision problems faced by the two players, while the third is a description of the bilinear utility function for Player 1 as a function of the sequence-form strategies of each player. Since both games are zero-sum, the utility for Player 2 is the opposite of the utility of Player 1.
- The tree of decision points and observation points for each decision problem is stored as a list of nodes. Each node has the following fields
  `id` is a string that represents the identifier of the node. The identifier is unique among the nodes for the same player.
  `type` is a string with value either `decision` (for decision points) or `observation` (for observation points).
  `actions` (only for decision points). This is a set of strings, representing the actions available at the decision node.
  `signals` (only for observation points). This is a set of strings, representing the signals that can be observed at the observation node.

`parent_edge` identifies the parent edge of the node. If the node is the root of the tree, then it is `null`. Else, it is a pair `(parent_node_id, action_or_signal)`, where the first member is the `id` of the parent node, and `action_or_signal` is the action or signal that connects the node to its parent.

`parent_sequence` (only for decision points). Identifies the parent sequence p_j of the decision point, defined as the last sequence (that is, decision point-action pair) encountered on the path from the root of the decision process to j.

> **Remark 1.** The list of nodes of the tree-form sequential decision process is given in top-down traversal order. The bottom-up traversal order can be obtained by reading the list of nodes backwards.

- The bilinear utility function for Player 1 is given through the payoff matrix $\boldsymbol{A}$ such that the (expected) utility of Player 1 can be written as

$$u_1(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^\top \mathbf{A} \boldsymbol{y},$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are sequence-form strategies for Players 1 and 2 respectively. We represent $\mathbf{A}$ in the file as a list of all non-zero matrix entries, storing for each the row index, column index, and value. Specifically, each entry is an object with the fields

`sequence_pl1` is a pair `(decision_pt_id_pl1, action_pl1)` which represents the sequence of Player 1 (row of the entry in the matrix).

`sequence_pl2` is a pair `(decision_pt_id_pl2, action_pl2)` which represents the sequence of Player 2 (column of the entry in the matrix).

`value` is the non-zero float value of the matrix entry.

Example: Rock-paper-superscissors In the case of rock-paper-superscissors the decision problem faced by each of the players has only one decision points with three actions: playing rock, paper, or superscissors. So, each tree-form sequential decision process only has a single node, which is a decision node. The payoff matrix of the game[2] is

$$\begin{array}{c} \\ \mathrm{r} \\ \mathrm{p} \\ \mathrm{s} \end{array} \begin{array}{c} \begin{array}{ccc} \mathrm{r} & \mathrm{p} & \mathrm{s} \end{array} \\ \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -2 \\ -1 & 2 & 0 \end{pmatrix} \end{array}.$$

So, the game file in this case has content:

```
{
  "decision_problem_pl1": [
    {"id": "d1_pl1", "type": "decision", "actions": ["r", "p", "s"],
     "parent_edge": null, "parent_sequence": null}
  ],
  "decision_problem_pl2": [
    {"id": "d1_pl2", "type": "decision", "actions": ["r", "p", "s"],
     "parent_edge": null, "parent_sequence": null}
  ],
  "utility_pl1": [
    {"sequence_pl1": ["d1_pl1", "r"], "sequence_pl2": ["d1_pl2", "p"], "value": -1},
    {"sequence_pl1": ["d1_pl1", "r"], "sequence_pl2": ["d1_pl2", "s"], "value": 1},
    {"sequence_pl1": ["d1_pl1", "p"], "sequence_pl2": ["d1_pl2", "r"], "value": 1},
    {"sequence_pl1": ["d1_pl1", "p"], "sequence_pl2": ["d1_pl2", "s"], "value": -2},
```

[2] A Nash equilibrium of the game is reached when all players play rock with probability 1/2, paper with probability 1/4 and superscissors with probability 1/4. Correspondingly, the game value is 0.

```
    {"sequence_pl1": ["d1_pl1", "s"], "sequence_pl2": ["d1_pl2", "r"], "value": -1},
    {"sequence_pl1": ["d1_pl1", "s"], "sequence_pl2": ["d1_pl2", "p"], "value": 2}
  ]
}
```

3.2 Learning to best respond

Let $\mathcal{X}$ and $\mathcal{Y}$ be the sequence-form strategy polytopes corresponding to the tree-form sequential decision problems faced by Players 1 and 2 respectively. A good smoke test when implementing regret minimization algorithms is to verify that they learn to best respond. In particular, you will verify that your implementation of CFR applied to the decision problem of Player 1 learns a best response against Player 2 when Player 2 plays the *uniform* strategy, that is, the strategy that at each decision points picks any of the available actions with equal probability.

Let $\boldsymbol{y}_{\text{uni}} \in \mathcal{Y}$ be the sequence-form representation of the strategy for Player 2 that at each decision point selects each of the available actions with equal probability. When Player 2 plays according to that strategy, the utility vector for Player 1 is given by $\boldsymbol{u} \coloneqq \mathbf{A}\boldsymbol{y}_{\text{uni}}$, where $\mathbf{A}$ is the payoff matrix of the game.

For each of the three games, take your CFR implementation for the decision problem of Player 1, and let it output strategies $\boldsymbol{x}^{(t)} \in \mathcal{X}$ while giving as feedback at each time t the same utility vector $\boldsymbol{u}$. As $T \to \infty$, the average strategy

$$\bar{\boldsymbol{x}}^{(T)} \coloneqq \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{x}^{(t)} \in \mathcal{X} \tag{5}$$

will converge to a best response to the uniform strategy $\boldsymbol{y}_{\text{uni}}$, that is,

$$\lim_{T\to\infty} \langle \bar{\boldsymbol{x}}^{(T)}, \mathbf{A}\boldsymbol{y}_{\text{uni}} \rangle = \max_{\hat{\boldsymbol{x}}\in\mathcal{X}} \hat{\boldsymbol{x}}^{\top}\mathbf{A}\boldsymbol{y}_{\text{uni}}.$$

If the above doesn't happen empirically, something is wrong with your implementation.

Problem 3.1 (15 points). In each of the three games, apply your CFR implementation to the tree-form sequential decision problem of Player 1, using as local regret minimizer at each decision point the regret matching algorithm (Lecture 4). At each time t, give as feedback to the algorithm the same utility vector $\boldsymbol{u} = \mathbf{A}\boldsymbol{y}_{\text{uni}}$, where $\boldsymbol{y}_{\text{uni}} \in \mathcal{Y}$ is the uniform strategy for Player 2. Run the algorithm for 1000 iterations. After each iteration $T = 1, \dots, 1000$, compute the value $v^{(T)} \coloneqq \langle \bar{\boldsymbol{x}}^{(T)}, \mathbf{A}\boldsymbol{y}_{\text{uni}} \rangle$ where $\bar{\boldsymbol{x}}^{(T)} \in \mathcal{X}$ is the average strategy output so far by CFR, as defined in (5).

Plot v^T as a function of T. Empirically, what is the limit you observe v^T is converging to?

★ Hint: represent vectors on $\mathbb{R}^{|\Sigma|}$ (including the sequence-form strategies output by CFR and utility vectors given to CFR) in memory as dictionaries from sequences (tuples `(decision_point_id, action)`) to floats.
★ Hint: in rock-paper-superscissor, v^T should approach the value 1/3. In Kuhn poker, the value 1/2. In Leduc poker, the value 2.0875.

Solution. For Rock-Paper-Superscissors, after running for $T = 1000$ iterations, the expected utility is 0.333000 and is approaching the limit $\frac{1}{3}$.

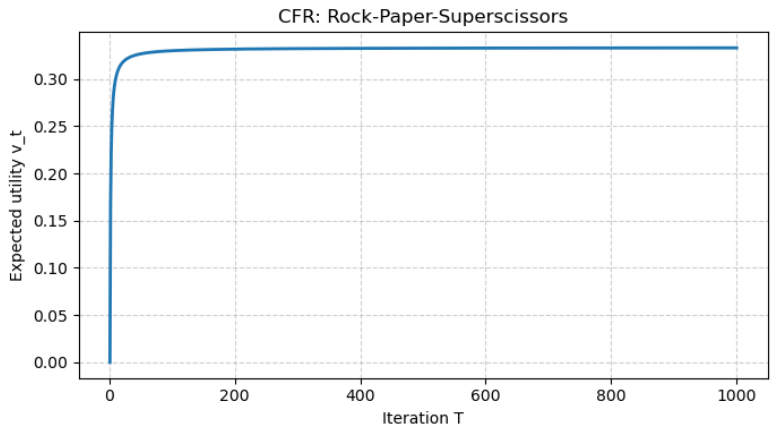


Figure 1: Plot for Rock-Paper-Superscissors

For Kuhn poker, after running for $T = 1000$ iterations, the expected utility is 0.499625 and is approaching the limit $\frac{1}{2}$.

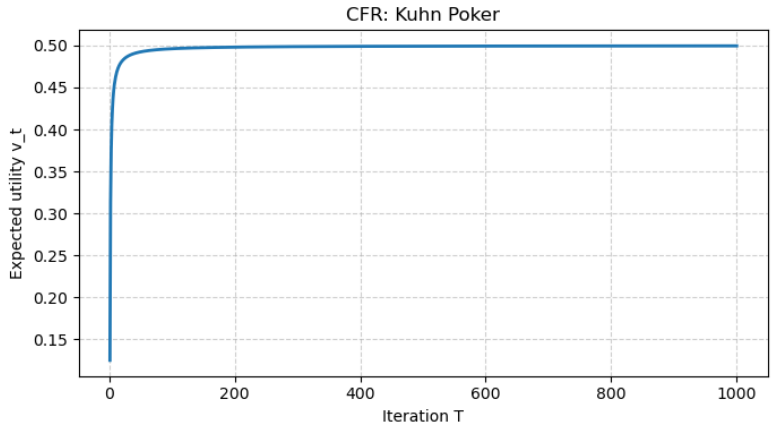


Figure 2: Plot for Kuhn Poker

For Leduc poker, after running for $T = 1000$ iterations, the expected utility is 2.085045 and is approaching the limit 2.0875 (empirically obtained by running for $T = 50000$ iterations).

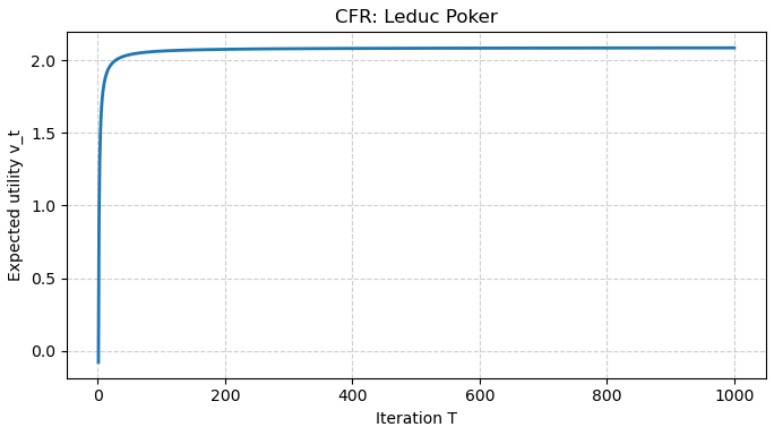


Figure 3: Plot for Leduc Poker

□

3.3 Learning a Nash equilibrium

Now that you are confident that your implementation of CFR is correct, you will use CFR to converge to Nash equilibrium using the self-play idea described in Lecture 3 and recalled next.

The idea behind using regret minimization to converge to Nash equilibrium in a two-player zero-sum game is to use *self play*. We instantiate two regret minimization algorithms, $\mathfrak{R}_{\mathcal{X}}$ and $\mathfrak{R}_{\mathcal{Y}}$, for the domains of the maximization and minimization problem, respectively. At each time t the two regret minimizers output strategies $\boldsymbol{x}^{(t)}$ and $\boldsymbol{y}^{(t)}$, respectively. Then they receive as feedback the vectors $\boldsymbol{u}_{\mathcal{X}}^{(t)}, \boldsymbol{u}_{\mathcal{Y}}^{(t)}$ defined as

$$\boldsymbol{u}_{\mathcal{X}}^{(t)} := \mathbf{A}\boldsymbol{y}^{(t)}, \qquad \boldsymbol{u}_{\mathcal{Y}}^{(t)} := -\mathbf{A}^{\top}\boldsymbol{x}^{(t)}, \tag{6}$$

where $\mathbf{A}$ is Player 1's payoff matrix.

We summarize the process pictorially in Figure 4.

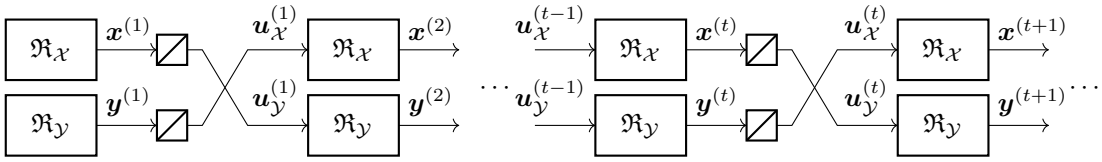


Figure 4: The flow of strategies and utilities in regret minimization for games. The symbol ⊠ denotes computation/construction of the utility vector.

As we saw in class, the pair of average strategies produced by the regret minimizers up to any time T converges to a Nash equilibrium, where convergence is measured via the *saddle point gap*

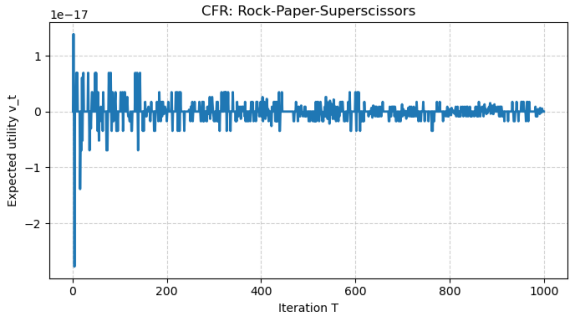
$$\max_{\hat{\boldsymbol{x}} \in \mathcal{X}} (\hat{\boldsymbol{x}}^{\top}\mathbf{A}\boldsymbol{y}) - \min_{\hat{\boldsymbol{y}} \in \mathcal{Y}} (\boldsymbol{x}^{\top}\mathbf{A}\hat{\boldsymbol{y}}).$$

A point $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{X} \times \mathcal{Y}$ has zero saddle point gap if and only if it is a Nash equilibrium of the game.

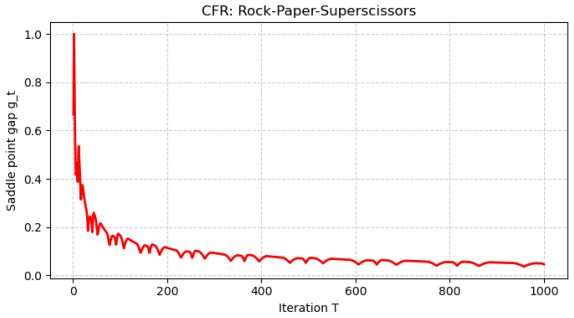
Problem 3.2 (15 points). Let the CFR implementation (using regret matching as the local regret minimizer at each decision point) for Player 1's and Player 2's tree-form sequential decision problems play against each other in self play, as described above.

Plot the saddle point gap and the expected utility (for Player 1) of the average strategies as a function of the number of iterations $T = 1, \ldots, 1000$.

★ Hint: represent vectors on $\mathbb{R}^{|\Sigma|}$ (including the sequence-form strategies output by CFR and utility vectors given to CFR) in memory as dictionaries from sequences (tuples `(decision_point_id, action)`) to floats.
★ Hint: to compute the saddle-point gap, feel free to use the function `gap(game, strategy_pl1, strategy_pl2)` provided in the Python stub file.
★ Hint: the saddle point gap should be going to zero. The expected utility of the average strategies in rock-paper-superscissor should approach the value 0. In Kuhn poker it should approach -0.055. In Leduc poker it should approach -0.085.

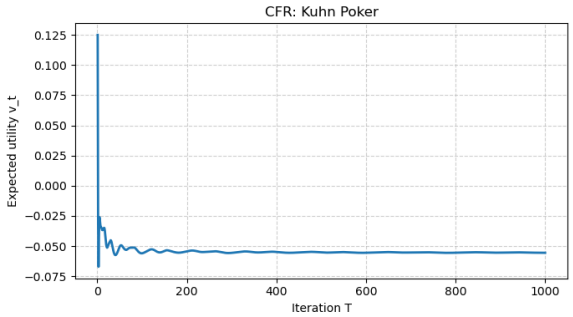


(a) Utility (Note y-axis is scaled by 1e-17)

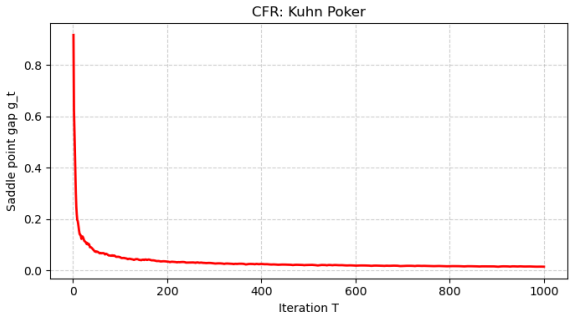


(b) Saddle Point Gap

Figure 5: Plots for Rock-Paper-Superscissors

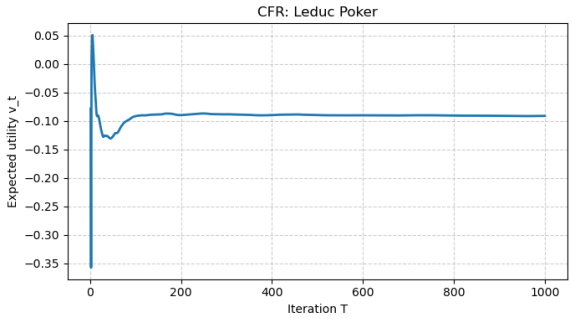


(a) Utility

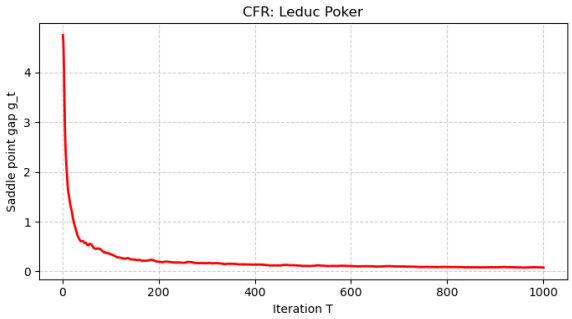


(b) Saddle Point Gap

Figure 6: Plots Kuhn Poker



(a) Utility



(b) Saddle Point Gap

Figure 7: Plots Leduc Poker

Solution. □

3.4 CFR+, Alternating Iterates, and Linear Averaging

To achieve better performance in practice when learning Nash equilibria in two-player zero-sum games, we consider the following modifications to the setup of the previous subsection.

- Instead of regret matching, CFR is set up to use the regret matching plus algorithm (see Lecture 4) at each decision point.
- Instead of using the classical self-play scheme described in Figure 4, players *alternate* the iterates and feedback: the utility vector $\boldsymbol{u}_{\mathcal{X}}^{(t)}$ is as defined in (6), whereas

$$\tilde{\boldsymbol{u}}_{\mathcal{Y}}^{(t)} \coloneqq -\mathbf{A}^\top \boldsymbol{x}^{(t+1)}.$$

- Finally, the output is weighted average of the strategies,

$$\bar{\bar{\boldsymbol{x}}}^{(T)} \coloneqq \frac{1}{\sum_{t=1}^{T} t^\gamma} \sum_{t=1}^{T} t^\gamma \boldsymbol{x}^{(t)}, \qquad \bar{\bar{\boldsymbol{y}}}^{(T)} \coloneqq \frac{1}{\sum_{t=1}^{T} t^\gamma} \sum_{t=1}^{T} t^\gamma \boldsymbol{y}^{(t)}$$

  is considered instead of the regular averages when computing the saddle point gap.

Collectively, the modified setup we just described is referred to as "running CFR+".

Problem 3.3 (20 points). Modify your implementation of CFR so that it also supports each of the following variants.

(a) (10 points) CFR+, with alternating iterates and $\gamma = 1$,
(b) (4 points) DCFR, with alternating iterates and parameters $\alpha = 1.5, \beta = 0, \gamma = 2$,
(c) (3 points) PCFR+, with alternating iterates and $\gamma = 2$, and
(d) (3 points) PCFR+, with alternating iterates and $\gamma = \infty$ (*i.e.,* do not average; keep only the last iterate).

For each variant, run it for 1000 iterations, plotting the expected utility for Player 1 and the saddle point gap of the averages $\bar{\bar{\boldsymbol{x}}}^{(T)}, \bar{\bar{\boldsymbol{y}}}^{(T)}$ after each iteration T. Add these lines to your plots from Problem 3.2.

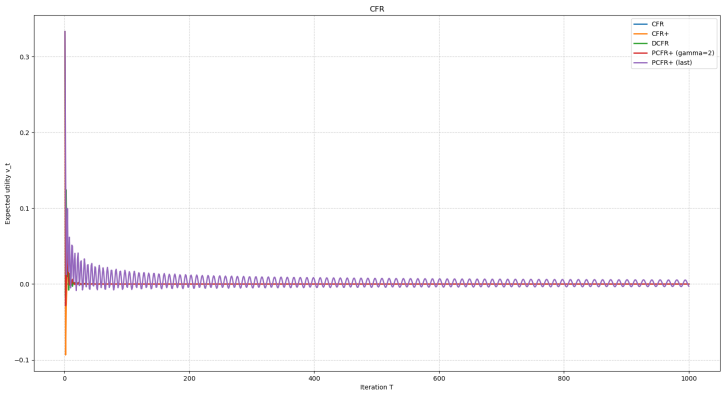
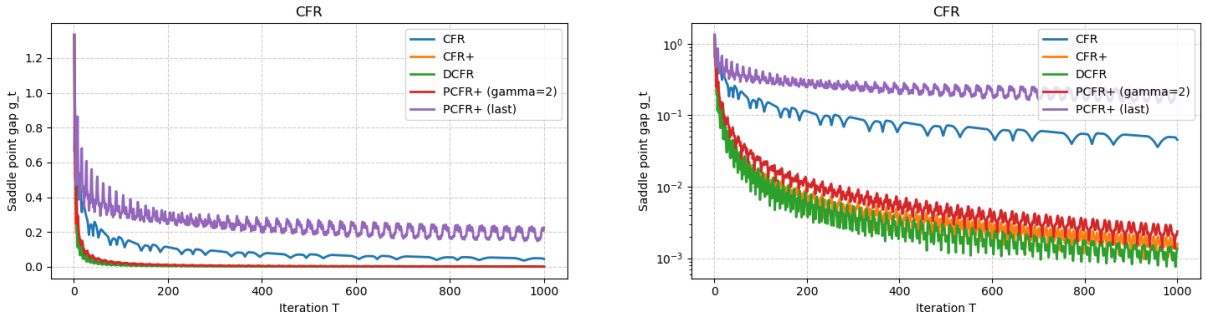


Figure 8: Utility Plot for Rock-Paper-Superscissors



(a) Normal Scale

(b) Log Scale

Figure 9: Saddle Point Gap for Rock-Paper-Superscissors

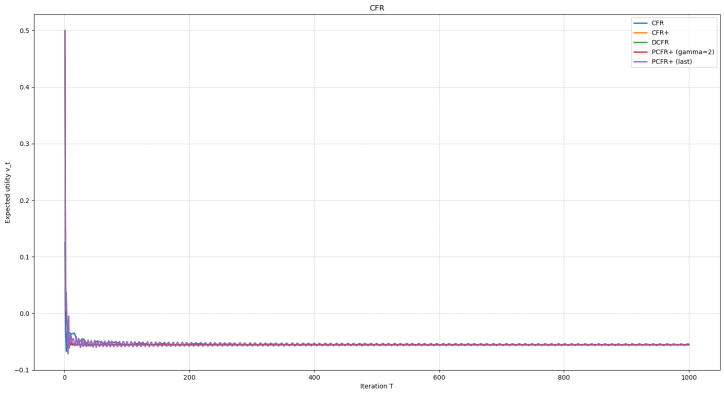
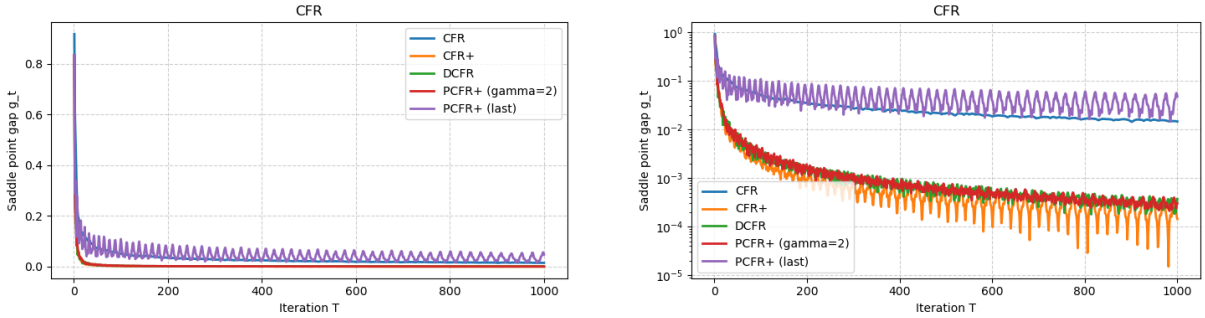


Figure 10: Utility Plot for Kuhn Poker



(a) Normal Scale

(b) Log Scale

Figure 11: Saddle Point Gap for Kuhn Poker

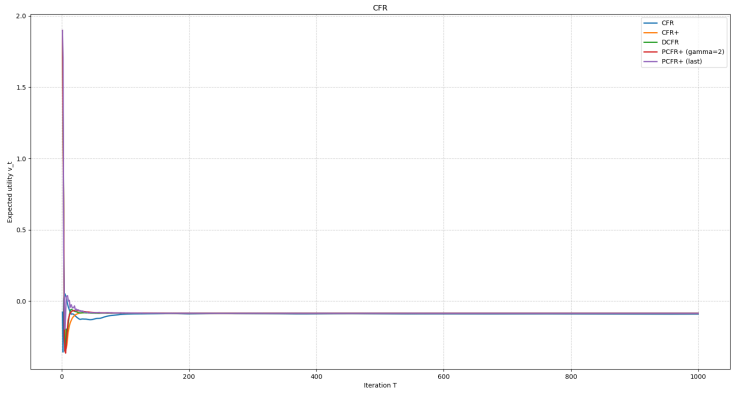
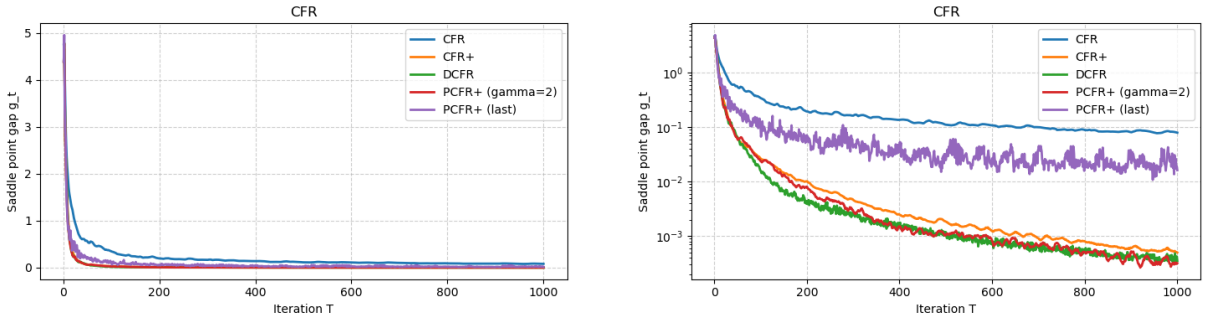


Figure 12: Utility Plot for Leduc Poker



(a) Normal Scale

(b) Log Scale

Figure 13: Saddle Point Gap for Leduc Poker

Solution. □

4 Code

```
#!/usr/bin/env python3

import os
import argparse
import json
import math
import matplotlib.pyplot as plt

################################################################################
# The next functions are already implemented for your convenience
#
# In all the functions in this stub file, ‘game‘ is the parsed input game json
# file, whereas ‘tfsdp‘ is either ‘game["decision_problem_pl1"]‘ or
# ‘game["decision_problem_pl2"]‘.
#
# See the homework handout for a description of each field.


def get_sequence_set(tfsdp):
    """Returns a set of all sequences in the given tree-form sequential decision
    process (TFSDP)"""

    sequences = set()
    for node in tfsdp:
        if node["type"] == "decision":
            for action in node["actions"]:
                sequences.add((node["id"], action))
    return sequences


def is_valid_RSigma_vector(tfsdp, obj):
    """Checks that the given object is a dictionary keyed on the set of sequences
    of the given tree-form sequential decision process (TFSDP)"""

    sequence_set = get_sequence_set(tfsdp)
    return isinstance(obj, dict) and obj.keys() == sequence_set


def assert_is_valid_sf_strategy(tfsdp, obj):
    """Checks whether the given object ‘obj‘ represents a valid sequence-form
```

```
    strategy vector for the given tree-form sequential decision process
    (TFSDP)"""

    if not is_valid_RSigma_vector(tfsdp, obj):
        print("The sequence-form strategy should be a dictionary with key set equal to the set of sequences in the game")
        os.exit(1)
    for node in tfsdp:
        if node["type"] == "decision":
            parent_reach = 1.0
            if node["parent_sequence"] is not None:
                parent_reach = obj[node["parent_sequence"]]
            if abs(sum([obj[(node["id"], action)] for action in node["actions"]]) - parent_reach) > 1e-3:
                print(
                    "At node ID %s the sum of the child sequences is not equal to the parent sequence", node["id"])


def best_response_value(tfsdp, utility):
    """Computes the value of max_{x in Q} x^T utility, where Q is the
    sequence-form polytope for the given tree-form sequential decision
    process (TFSDP)"""

    assert is_valid_RSigma_vector(tfsdp, utility)

    utility_ = utility.copy()
    utility_[None] = 0.0
    for node in tfsdp[::-1]:
        if node["type"] == "decision":
            max_ev = max([utility_[(node["id"], action)]
                          for action in node["actions"]])
            utility_[node["parent_sequence"]] += max_ev
    return utility_[None]


def compute_utility_vector_pl1(game, sf_strategy_pl2):
    """Returns A * y, where A is the payoff matrix of the game and y is
    the given strategy for Player 2"""

    assert_is_valid_sf_strategy(
        game["decision_problem_pl2"], sf_strategy_pl2)

    sequence_set = get_sequence_set(game["decision_problem_pl1"])
    utility = {sequence: 0.0 for sequence in sequence_set}
    for entry in game["utility_pl1"]:
        utility[entry["sequence_pl1"]] += entry["value"] * \
            sf_strategy_pl2[entry["sequence_pl2"]]

    assert is_valid_RSigma_vector(game["decision_problem_pl1"], utility)
    return utility


def compute_utility_vector_pl2(game, sf_strategy_pl1):
    """Returns -A^transpose * x, where A is the payoff matrix of the
    game and x is the given strategy for Player 1"""

    assert_is_valid_sf_strategy(
        game["decision_problem_pl1"], sf_strategy_pl1)

    sequence_set = get_sequence_set(game["decision_problem_pl2"])
    utility = {sequence: 0.0 for sequence in sequence_set}
    for entry in game["utility_pl1"]:
        utility[entry["sequence_pl2"]] -= entry["value"] * \
            sf_strategy_pl1[entry["sequence_pl1"]]

    assert is_valid_RSigma_vector(game["decision_problem_pl2"], utility)
    return utility
```

```
def gap(game, sf_strategy_pl1, sf_strategy_pl2):
    """Computes the saddle point gap of the given sequence-form strategies
    for the players"""

    assert_is_valid_sf_strategy(
        game["decision_problem_pl1"], sf_strategy_pl1)
    assert_is_valid_sf_strategy(
        game["decision_problem_pl2"], sf_strategy_pl2)

    utility_pl1 = compute_utility_vector_pl1(game, sf_strategy_pl2)
    utility_pl2 = compute_utility_vector_pl2(game, sf_strategy_pl1)

    return (best_response_value(game["decision_problem_pl1"], utility_pl1)
            + best_response_value(game["decision_problem_pl2"], utility_pl2))


###############################################################################
# Starting from here, you should fill in the implementation of the
# different functions


def expected_utility_pl1(game, sf_strategy_pl1, sf_strategy_pl2):
    """Returns the expected utility for Player 1 in the game, when the two
    players play according to the given strategies"""

    assert_is_valid_sf_strategy(
        game["decision_problem_pl1"], sf_strategy_pl1)
    assert_is_valid_sf_strategy(
        game["decision_problem_pl2"], sf_strategy_pl2)

    # FINISH
    utility_pl1 = compute_utility_vector_pl1(game, sf_strategy_pl2)
    utility = sum(sf_strategy_pl1[seq] * utility_pl1[seq] for seq in utility_pl1)
    return utility


def uniform_sf_strategy(tfsdp):
    """Returns the uniform sequence-form strategy for the given tree-form
    sequential decision process"""

    # FINISH
    seqs = get_sequence_set(tfsdp)
    x = {seq: 0.0 for seq in seqs}

    for node in tfsdp:
        if node["type"] != "decision":
            continue

        if node["parent_sequence"] is None:
            parent_reach = 1.0
        else:
            parent_reach = x[node["parent_sequence"]]

        num_actions = len(node["actions"])
        if num_actions > 0:
            for action in node["actions"]:
                x[(node["id"], action)] = parent_reach / num_actions

    assert_is_valid_sf_strategy(tfsdp, x)
    return x


class RegretMatching(object):
```

```
    def __init__(self, action_set):
        self.action_set = set(action_set)

        # FINISH
        self.regret_sum = {action: 0.0 for action in action_set}
        self.last_strategy = None

    def next_strategy(self):

        # FINISH
        positive_regrets = {action: max(0.0, self.regret_sum[action]) for action in self.action_set}
        total_positive_regret = sum(positive_regrets.values())

        if total_positive_regret <= 1e-8:
            strategy = {action: 1.0 / len(self.action_set) for action in self.action_set}
        else:
            strategy = {action: positive_regrets[action] / total_positive_regret for action in self.action_set}

        self.last_strategy = strategy
        return strategy

    def observe_utility(self, utility):
        assert isinstance(utility, dict) and utility.keys() == self.action_set

        # FINISH
        if self.last_strategy is None:
            self.last_strategy = {action: 1.0 / len(self.action_set) for action in self.action_set}

        inner_product = sum(self.last_strategy[action] * utility[action] for action in self.action_set)
        for action in self.action_set:
            self.regret_sum[action] += utility[action] - inner_product


class RegretMatchingPlus(object):
    def __init__(self, action_set):
        self.action_set = set(action_set)

        # FINISH
        self.regret_sum = {action: 0.0 for action in action_set}
        self.last_strategy = None

    def next_strategy(self):

        # FINISH
        regrets = {action: self.regret_sum[action] for action in self.action_set}
        total_regret = sum(regrets.values())

        if total_regret <= 1e-8:
            strategy = {action: 1.0 / len(self.action_set) for action in self.action_set}
        else:
            strategy = {action: regrets[action] / total_regret for action in self.action_set}

        self.last_strategy = strategy
        return strategy

    def observe_utility(self, utility):
        assert isinstance(utility, dict) and utility.keys() == self.action_set

        # FINISH
        if self.last_strategy is None:
            self.last_strategy = {action: 1.0 / len(self.action_set) for action in self.action_set}

        inner_product = sum(self.last_strategy[action] * utility[action] for action in self.action_set)
        for action in self.action_set:
            self.regret_sum[action] = max(0.0, self.regret_sum[action] + utility[action] - inner_product)
```

```
class Cfr(object):
    def __init__(self, tfsdp, rm_class=RegretMatching):
        self.tfsdp = tfsdp
        self.local_regret_minimizers = {}
        self.last_strategies = {}
        self.children = {}

        # For each decision point, we instantiate a local regret minimizer
        for node in tfsdp:
            if node["type"] == "decision":
                self.local_regret_minimizers[node["id"]] = rm_class(
                    node["actions"])

        for node in tfsdp:
            parent_edge = node.get("parent_edge", None)
            if parent_edge is not None:
                self.children[parent_edge] = node["id"]

    def next_strategy(self):

        # FINISH
        seqs = get_sequence_set(self.tfsdp)
        x = {seq: 0.0 for seq in seqs}

        for node in self.tfsdp:
            if node["type"] != "decision":
                continue

            strategy = self.local_regret_minimizers[node["id"]].next_strategy()
            self.last_strategies[node["id"]] = strategy

            parent_reach = 1.0
            if node["parent_sequence"] is not None:
                parent_reach = x[node["parent_sequence"]]

            for action in node["actions"]:
                x[(node["id"], action)] = parent_reach * strategy[action]

        assert_is_valid_sf_strategy(self.tfsdp, x)
        return x

    def observe_utility(self, utility):

        # FINISH
        V = {None: 0.0}
        for node in self.tfsdp[::-1]:

            if node["type"] == "decision":
                j = node["id"]
                b = self.last_strategies.get(j, None)

                if b is None:
                    b = {action: 1.0 / len(node["actions"]) for action in node["actions"]}

                V_j = 0.0
                for action in node["actions"]:
                    child_id = self.children.get((j, action), None)
                    V_child = 0.0 if child_id is None else V.get(child_id, 0.0)
                    V_j += b[action] * (utility[(j, action)] + V_child)
                V[j] = V_j

            else:
                k = node["id"]
```

```
                V_k = 0.0
                for signal in node["signals"]:
                    child_id = self.children.get((k, signal), None)
                    V_child = 0.0 if child_id is None else V.get(child_id, 0.0)
                    V_k += V_child
                V[k] = V_k

        for node in self.tfsdp:
            if node["type"] != "decision":
                continue

            j = node["id"]
            u = {}
            for action in node["actions"]:
                child_id = self.children.get((j, action), None)
                V_child = 0.0 if child_id is None else V.get(child_id, 0.0)
                u[action] = utility[(j, action)] + V_child
            self.local_regret_minimizers[j].observe_utility(u)


def dict_add_in_place(dst, src, weight=1.0):
    for key, value in src.items():
        dst[key] = dst.get(key, 0.0) + weight * value


def dict_scale(dict, scale):
    return {key: value * scale for key, value in dict.items()}


def apply_dcfr_discount(cfr, t, alpha, beta):
    if alpha == 0.0 and beta == 0.0:
        return

    pos_scale = (t ** alpha) / (t ** alpha + 1.0)
    neg_scale = (t ** beta) / (t ** beta + 1.0)

    for rm in cfr.local_regret_minimizers.values():
        regret_sum = rm.regret_sum

        for (action, regret) in regret_sum.items():
            if regret >= 0.0:
                regret_sum[action] = regret * pos_scale
            else:
                regret_sum[action] = regret * neg_scale


def plotter_v(v):
    T = range(1, len(v) + 1)
    plt.figure(figsize=(7, 4))
    plt.plot(T, v, linewidth=2)
    plt.xlabel("Iteration T")
    plt.ylabel("Expected utility v_t")
    plt.title("CFR: Leduc Poker")
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.tight_layout()
    plt.show()


def plotter_gap(gap):
    T = range(1, len(gap) + 1)
    plt.figure(figsize=(7, 4))
    plt.plot(T, gap, linewidth=2, color='red')
    plt.xlabel("Iteration T")
    plt.ylabel("Saddle point gap g_t")
    plt.title("CFR: Leduc Poker")
```

```
    # plt.yscale("log")
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.tight_layout()
    plt.show()


def plotter_v_vs_baseline(v, baseline_v):
    T1 = range(1, len(v) + 1)
    T2 = range(1, len(baseline_v) + 1)
    plt.figure(figsize=(7, 4))
    plt.plot(T1, v, linewidth=2, label='Variant')
    plt.plot(T2, baseline_v, linestyle='--', color='r', linewidth=2, label='Baseline')
    plt.xlabel("Iteration T")
    plt.ylabel("Expected utility v_t")
    plt.title("CFR")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.tight_layout()
    plt.show()


def plotter_gap_vs_baseline(gap, baseline_gap):
    T1 = range(1, len(gap) + 1)
    T2 = range(1, len(baseline_gap) + 1)
    plt.figure(figsize=(7, 4))
    plt.plot(T1, gap, linewidth=2, label='Variant')
    plt.plot(T2, baseline_gap, linestyle='--', color='r', linewidth=2, label='Baseline')
    plt.xlabel("Iteration T")
    plt.ylabel("Saddle point gap g_t")
    plt.title("CFR")
    # plt.yscale("log")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.tight_layout()
    plt.show()


def plotter_v_aggregate(v_list, labels):
    T = range(1, len(v_list[0]) + 1)
    plt.figure(figsize=(7, 4))

    for v, label in zip(v_list, labels):
        plt.plot(T, v, linewidth=2, label=label)

    plt.xlabel("Iteration T")
    plt.ylabel("Expected utility v_t")
    plt.title("CFR")
    # plt.yscale("log")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.tight_layout()
    plt.show()


def plotter_gap_aggregate(gap_list, labels):
    T = range(1, len(gap_list[0]) + 1)
    plt.figure(figsize=(7, 4))

    for gap, label in zip(gap_list, labels):
        plt.plot(T, gap, linewidth=2, label=label)

    plt.xlabel("Iteration T")
    plt.ylabel("Saddle point gap g_t")
    plt.title("CFR")
    # plt.yscale("log")
```

```
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.tight_layout()
    plt.show()


def plotter_gap_aggregate_log(gap_list, labels):
    T = range(1, len(gap_list[0]) + 1)
    plt.figure(figsize=(7, 4))

    for gap, label in zip(gap_list, labels):
        plt.plot(T, gap, linewidth=2, label=label)

    plt.xlabel("Iteration T")
    plt.ylabel("Saddle point gap g_t")
    plt.title("CFR")
    plt.yscale("log")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.tight_layout()
    plt.show()


def solve_problem_3_1(game, T=1000):

    # FINISH
    v_list = []
    y_uniform = uniform_sf_strategy(game["decision_problem_pl2"])
    u = compute_utility_vector_pl1(game, y_uniform)
    cfr_pl1 = Cfr(game["decision_problem_pl1"], rm_class=RegretMatching)

    sequences = get_sequence_set(game["decision_problem_pl1"])
    x_sum = {seq: 0.0 for seq in sequences}

    for t in range(1, T + 1):
        x_t = cfr_pl1.next_strategy()
        cfr_pl1.observe_utility(u)

        dict_add_in_place(x_sum, x_t, weight=1.0)

        x_bar = dict_scale(x_sum, 1.0 / t)
        v_t = expected_utility_pl1(game, x_bar, y_uniform)
        v_list.append(v_t)

        if t % 100 == 0 or t == 1 or t == T:
            print("Iteration %d. Expected utility = %.6f" % (t, v_t))

    return v_list


def solve_problem_3_2(game, T=1000):

    # FINISH
    v_list = []
    gap_list = []

    cfr_pl1 = Cfr(game["decision_problem_pl1"], rm_class=RegretMatching)
    cfr_pl2 = Cfr(game["decision_problem_pl2"], rm_class=RegretMatching)

    sequences_pl1 = get_sequence_set(game["decision_problem_pl1"])
    sequences_pl2 = get_sequence_set(game["decision_problem_pl2"])
    x_sum = {seq: 0.0 for seq in sequences_pl1}
    y_sum = {seq: 0.0 for seq in sequences_pl2}

    for t in range(1, T + 1):
```

```
        x_t = cfr_pl1.next_strategy()
        y_t = cfr_pl2.next_strategy()

        u_pl1 = compute_utility_vector_pl1(game, y_t)
        u_pl2 = compute_utility_vector_pl2(game, x_t)

        cfr_pl1.observe_utility(u_pl1)
        cfr_pl2.observe_utility(u_pl2)

        dict_add_in_place(x_sum, x_t, weight=1.0)
        dict_add_in_place(y_sum, y_t, weight=1.0)

        x_bar = dict_scale(x_sum, 1.0 / t)
        y_bar = dict_scale(y_sum, 1.0 / t)

        v_t = expected_utility_pl1(game, x_bar, y_bar)
        g_t = gap(game, x_bar, y_bar)

        v_list.append(v_t)
        gap_list.append(g_t)

        if t % 100 == 0 or t == 1 or t == T:
            print("Iteration %d. Expected utility = %.6f. Gap = %.6f" % (t, v_t, g_t))

    return v_list, gap_list


def solve_problem_3_3(game, T=1000, alpha=0.0, beta=0.0, gamma=0.0, use_rm_plus=True):

    # FINISH
    print("Parameters: alpha =", alpha, "beta =", beta, "gamma =", gamma, "use_rm_plus =", use_rm_plus)
    v_list = []
    gap_list = []

    rm_class = RegretMatchingPlus if use_rm_plus else RegretMatching
    cfr_pl1 = Cfr(game["decision_problem_pl1"], rm_class=rm_class)
    cfr_pl2 = Cfr(game["decision_problem_pl2"], rm_class=rm_class)

    keep_last = (gamma == float('inf'))
    if keep_last:
        print("Using last iterate only (gamma = inf)")

    else:
        sequences_pl1 = get_sequence_set(game["decision_problem_pl1"])
        sequences_pl2 = get_sequence_set(game["decision_problem_pl2"])
        x_wsum = {seq: 0.0 for seq in sequences_pl1}
        y_wsum = {seq: 0.0 for seq in sequences_pl2}
        wsum = 0.0

    for t in range(1, T + 1):

        y_t = cfr_pl2.next_strategy()
        u_pl1 = compute_utility_vector_pl1(game, y_t)
        apply_dcfr_discount(cfr_pl1, t, alpha, beta)
        cfr_pl1.observe_utility(u_pl1)

        x_tp1 = cfr_pl1.next_strategy()
        u_pl2 = compute_utility_vector_pl2(game, x_tp1)
        apply_dcfr_discount(cfr_pl2, t, alpha, beta)
        cfr_pl2.observe_utility(u_pl2)

        if keep_last:
            x_bar = x_tp1
            y_bar = y_t
```

```
        else:
            w_t = t ** gamma
            dict_add_in_place(x_wsum, x_tp1, weight=w_t)
            dict_add_in_place(y_wsum, y_t, weight=w_t)

            wsum += w_t
            x_bar = dict_scale(x_wsum, 1.0 / wsum)
            y_bar = dict_scale(y_wsum, 1.0 / wsum)

        v_t = expected_utility_pl1(game, x_bar, y_bar)
        g_t = gap(game, x_bar, y_bar)

        v_list.append(v_t)
        gap_list.append(g_t)

        if t % 100 == 0 or t == 1 or t == T:
            print("Iteration %d. Expected utility = %.6f. Gap = %.6f" % (t, v_t, g_t))

    return v_list, gap_list


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Problem 3 (CFR)')
    parser.add_argument("--game", help="Path to game file")
    parser.add_argument("--problem", choices=["3.1", "3.2", "3.3"])
    parser.add_argument("--T", type=int, default=1000)
    parser.add_argument("--alpha", type=float, default=0.0)
    parser.add_argument("--beta", type=float, default=0.0)
    parser.add_argument("--gamma", type=float, default=0.0)
    parser.add_argument("--use_rm_plus", action="store_true", default=False)

    args = parser.parse_args()
    print("Reading game path %s..." % args.game)

    game = json.load(open(args.game))

    # Convert all sequences from lists to tuples
    for tfsdp in [game["decision_problem_pl1"], game["decision_problem_pl2"]]:
        for node in tfsdp:
            if isinstance(node["parent_edge"], list):
                node["parent_edge"] = tuple(node["parent_edge"])
            if "parent_sequence" in node and isinstance(node["parent_sequence"], list):
                node["parent_sequence"] = tuple(node["parent_sequence"])
    for entry in game["utility_pl1"]:
        assert isinstance(entry["sequence_pl1"], list)
        assert isinstance(entry["sequence_pl2"], list)
        entry["sequence_pl1"] = tuple(entry["sequence_pl1"])
        entry["sequence_pl2"] = tuple(entry["sequence_pl2
    print("... done. Running code for Problem", args.problem)
    print("Alpha =", args.alpha, "Beta =", args.beta, "Gamma =", args.gamma, "T =", args.T, "Use RM Plus =", args.use_rm_plus)

    if args.problem == "3.1":
        solve_problem_3_1(game, T=args.T)
        v_list = solve_problem_3_1(game, T=args.T)
        plotter_v(v_list)

    elif args.problem == "3.2":
        v_list, gap_list = solve_problem_3_2(game, T=args.T)
        plotter_v(v_list)
        plotter_gap(gap_list)

    else:
        assert args.problem == "3.3"
        v_cfr, gap_cfr = solve_problem_3_2(game, T=args.T) # CFR
        v_cfrp, gap_cfrp = solve_problem_3_3(game, T=args.T, alpha=0.0, beta=0.0, gamma=1.0, use_rm_plus=True) # CFR+
```

```
v_dcfr, gap_dcfr = solve_problem_3_3(game, T=args.T, alpha=1.5, beta=0.0, gamma=2.0, use_rm_plus=False) # DCFR
v_pcfrp_2, gap_pcfrp_2 = solve_problem_3_3(game, T=args.T, alpha=0.0, beta=0.0,
                                           gamma=2.0, use_rm_plus=True) # PCFR+ gamma=2
v_pcfrp_inf, gap_pcfrp_inf = solve_problem_3_3(game, T=args.T, alpha=0.0, beta=0.0,
                                               gamma=float('inf'), use_rm_plus=True) # PCFR+ last iterate

plotter_v_aggregate([v_cfr, v_cfrp, v_dcfr, v_pcfrp_2, v_pcfrp_inf],
                    ['CFR', 'CFR+', 'DCFR', 'PCFR+ (gamma=2)', 'PCFR+ (last)'])
plotter_gap_aggregate([gap_cfr, gap_cfrp, gap_dcfr, gap_pcfrp_2, gap_pcfrp_inf],
                    ['CFR', 'CFR+', 'DCFR', 'PCFR+ (gamma=2)', 'PCFR+ (last)'])
plotter_gap_aggregate_log([gap_cfr, gap_cfrp, gap_dcfr, gap_pcfrp_2, gap_pcfrp_inf],
                    ['CFR', 'CFR+', 'DCFR', 'PCFR+ (gamma=2)', 'PCFR+ (last)'])
```

References

H. W. Kuhn. A simplified two-person poker. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games*, volume 1 of *Annals of Mathematics Studies, 24*, pages 97–103. Princeton University Press, Princeton, New Jersey, 1950.

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