COS 445 - Homework 3

Due online Monday, April 3rd at 11:59 pm


- You must write up your solutions by yourself, without any collaborators or external references.

- Unless otherwise stated, you may collaborate with other students and consult external references, but may not take written/typed/recorded notes from these interactions.

- You must list all collaborators and external references consulted.

- Some problems will be marked as no collaboration problems. This is to make sure you have experience solving a problem start-to-finish by yourself in preparation for the midterms/final.

- Please upload each problem as a separate file via CS Dropbox.

Instructions for Problems 1-3

In each of the next three problems, we will describe a game, for which your task is to implement a strategy in Java. Specifically, for each game, your task is the following:

(a) Submit a Java source file that precisely follows the API specifications. We suggest starting by using the template code provided. Your source file must follow the naming convention ProblemName_netid.java, where netid is the NetID of the submitter. **List your teammates and their NetIDs in your submitted writeup.**

(b) Describe your strategy in English. (No pseudocode necessary, since we will have your code!)

(c) Describe, as rigorously as possible, why your strategy is a good idea. How does it behave against {Nash equilibrium players, a copy of itself, other simple strategies you can think of, irrational players}? What’s the worst that could happen? What’s the best that could happen? What do you expect other players to do? If you feel that your exact strategy is too complex to analyze, you are allowed to analyze a simpler version of your submitted algorithm.

Logistics and grading

- You may work in teams of up to 3 students. One team member should submit source files and the writeups (but please make sure all team members are clearly listed!). Each task will be worth 15 points total, broken down as described below.
• A submission that plays the game correctly according to the description will be worth 5 points per task.

• For part (c), the analysis, we will provide a few specific leading questions. Answering these questions clearly and concisely will get you another 5 points per task.

• The quality of the remainder of your analysis and your overall strategy will be worth the remaining 5 points per task. This portion is open-ended in nature, and will be graded on clarity, rigor, originality, and use of game-theoretic concepts. Please limit yourself absolutely to no more than two pages of analysis per problem; a few thoughtful paragraphs can suffice for a perfect score. Most analyses should be less than a page.
  
  – A well-thought-out strategy that is justified clearly in the analysis can receive full credit, regardless of its performance in the simulations.
  
  – A strategy that performs exceptionally well in the simulations can receive full credit, even if the analysis and justification is unremarkable. Note that performance will be measured against a benchmark of total accumulated payoff, not relative ranking among your classmates - you should try to get as much payoff as possible. If doing so gives your classmates more payoff too, even better!

• It is possible for every team to receive 15 points per task with well-thought-out and well-explained strategies.

• Bonus points (and special prizes TBA) will be awarded to teams that perform especially well. In particular, as noted above, a strategy that performs well might make up for a not-quite-perfect writeup.

• Please ensure that your code can play all rounds within 5 seconds on a reasonable desktop computer, and consume less than 1GB of RAM. Intentional violations of this rule to disrupt the game will result in a score of zero (it is extremely unlikely that you will accidentally violate this - just keep an eye out if your computer crashes when you test your strategies).

• You are encouraged to think outside the box, but within the confines of this assignment. Non-game-theoretic solutions (for example, bribery, intimidation, blackmail, or submitting ransomware) will result in a score of zero.

• You are allowed to post notes about the assignment to Piazza, to try and coordinate with other teams, etc., but not to share code. Examples of acceptable posts might include: “Our team is playing strategy X, and it is a Nash if every team does this. Please do so as well!!” or “Our team is playing a Grim Trigger strategy, don’t cross us or we’ll ruin your payoff, even if it hurts us too!” All such posts must be non-anonymous to instructors (anonymous to classmates is OK), and should be recorded in your strategy justification (how did making this post help?). Please keep all “strategic” communication about the assignment on Piazza and visible to the instructors.

Problem 1: Iterated Prisoner’s Dilemma (15 points)

In this problem, you will implement a strategy for the Prisoner's Dilemma, repeated over a time horizon of $T = 1000$ rounds. At each round, a player can cooperate ($C$) or defect ($D$), receiving rewards according to the following payoff matrix:
Your player will be able to access play history through a callback function, so that you can implement strategies that maintain state. Your performance will be measured by the sum of your payoffs, played head-to-head against each other submission in the class.

**API specification**

Your class should implement the Prisoner interface, which requires the following methods:

- `public boolean cooperate()`: called at each round, when you must make an action. Return `true` to cooperate or `false` to defect.
- `public void receive(boolean action)`: will be called at the end of each round with the partner’s action, so that you can record play history as you like.

A simple example is provided in `Prisoner_cxzhang.java`.

**Questions**

- What is the unique Nash equilibrium of this game (you don’t need to prove it’s unique)?
- How does your strategy behave against tit-for-tat?
- How does your strategy behave against itself?

**Problem 2: Centipede (15 points)**

In this problem, you will play an iterated version of the game Centipede, which we first describe. Centipede is a two-player game played over 100 alternating turns. Suppose Alice is the player to move first, and Bob is the second player. There is a growing pot of money, which is divided asymmetrically based on the actions of the players.

- At the first round $t = 1$, Alice can take or push the pot. If she takes, the game ends with Alice receiving $4$, and Bob receiving $1$. If she pushes, nobody receives any payoff, and control passes to Bob.
- At the second round $t = 2$, Bob can take or push. If he takes, the game ends with Bob receiving $5$, and Alice receiving $2$.
- In general, at odd rounds $t$, Alice can take the pot to end the game with payoffs $(t + 3, t)$. At even rounds, Bob can end the game with payoffs $(t, t + 3)$. Or, either player can choose to push to the next round.
- At round 100, if the players have pushed for all previous rounds. Bob must take the pot.

Your solutions will play an iterated version of Centipede in the following sense:

- Each “day”, you will be matched with two partners. You will play as Alice with one, and Bob with another.
At the beginning of each Centipede match, you will receive some information about your partner: \(a\) and \(b\), the average payoff received by your partner playing as Alice and Bob, respectively.

Your performance will be measured by the total payoff received after a large number of rounds. Everyone will be guaranteed to play with each other. To ensure fairness, we will run the entire process multiple times with random matchings assignments.

**API specification**

Your class should implement the `Centipede` interface, which requires the following method:

- `public boolean init(double a, double b):` called at the beginning of a game of Centipede. \(a\) and \(b\) are the average payoffs your partner received playing as Alice and Bob, respectively, or \(-1\) on the first day.
- `public boolean push(int t):` called at each round \(t\), when it is your player’s turn. Return `true` to push, and `false` to end the game.

A simple example is provided in `Centipede_karans.java`.

**Questions**

- What is the unique Nash equilibrium of this game (you don’t need to prove it’s unique)? Why might you *not* want to play according to a Nash, even though you are only playing one game with each partner?
- How does your strategy behave against one that chooses to push or take by flipping a fair independent coin?
- How does your strategy perform within a population of copies of itself?

**Problem 3: Ultimatum (15 points)**

In this problem, we play a reciprocal, iterated version of a classic asymmetric game. We begin by describing the basic game, which consists of just two turns.

- During Alice’s turn, she receives 100 coins, and her task is to propose a division of these coins. Alice must propose an *ultimatum*: a division of these coins \((x, 100 - x)\), where \(x \in \{0, 1, \ldots, 100\}\).
- During Bob’s turn, he can either accept or reject the proposal. If he accepts, Alice receives payoff \(x\), while Bob receives payoff \(100 - x\). If he rejects, both players receive nothing.

The iterated version will be similar to that for Centipede:

- Each “day”, you will be matched with two partners. You will play as Alice with one, and Bob with another.
- As either player, you will receive the average proposal value \(x\) made by your partner’s program while playing Alice.
• Your performance will be measured by the total payoff received after a large number of rounds. Everyone will be guaranteed to play with each other. To ensure fairness, we will run the entire process multiple times with random matchings assignments.

API specification

Your class should implement the Ultimatum interface, which requires the following methods:

• public int propose(double a): returns the proposal (an integer between 0 and 100 inclusive) that your program wishes to make. To inform your decision, $a$ will be the average proposal your partner made as Alice. On the first day, $a$ will be set to $-1$.

• public boolean accept(double a, int x): returns whether your program wants to accept a proposal of $x$ (such that you receive payoff $100 - x$). $a$ is the same as above.

A simple example is provided in Ultimatum_smattw.java.

Questions

• Why might Alice not just choose $(99, 1)$? Why might Bob not always accept?

• How does your strategy do within a population that always proposes a $(50, 50)$ split and accepts a $(100 - x, x)$ split with probability $x/100$?\footnote{Food for thought: by posing this question, did we change the strategies you’re considering? Did we change your beliefs about how your classmates might behave?}

• How does your strategy perform within a population of copies of itself?

Problem 4: A Tree Game (15 points, no collaboration)

Consider the following extended form game:

\begin{figure}[h]
\centering
\begin{tikzpicture}[level distance=1.5cm,
  level 1/.style={sibling distance=3cm},
  level 2/.style={sibling distance=1.5cm},
  level 3/.style={sibling distance=1cm},
  every node/.style={draw, circle, minimum size=0.5cm}]
  \node {1}
  child {node {2}
    child {node {1} child {node {a} edge from parent[draw=none]}
    child {node {b}
      child {node {4}
        edge from parent[draw=none]}
      child {node {1.5}
        edge from parent[draw=none]}
    }
    child {node {c}
      child {node {3}
        edge from parent[draw=none]}
      child {node {0}
        edge from parent[draw=none]}
    }
    child {node {d}
      child {node {0}
        edge from parent[draw=none]}
      child {node {1}
        edge from parent[draw=none]}
    }
    child {node {e}
      child {node {1}
        edge from parent[draw=none]}
      child {node {2}
        edge from parent[draw=none]}
    }
    child {node {f}
      child {node {3}
        edge from parent[draw=none]}
      child {node {0.5}
        edge from parent[draw=none]}
    }
    child {node {g}
      child {node {1}
        edge from parent[draw=none]}
      child {node {2}
        edge from parent[draw=none]}
    }
    child {node {h}
      child {node {2}
        edge from parent[draw=none]}
      child {node {1}
        edge from parent[draw=none]}
    }
    child {node {w}
      child {node {2}
        edge from parent[draw=none]}
      child {node {1}
        edge from parent[draw=none]}
    }
    child {node {z}
      child {node {1}
        edge from parent[draw=none]}
      child {node {2}
        edge from parent[draw=none]}
    }
    child {node {y}
      child {node {1}
        edge from parent[draw=none]}
      child {node {2}
        edge from parent[draw=none]}
    }
    child {node {x}
      child {node {1}
        edge from parent[draw=none]}
      child {node {2}
        edge from parent[draw=none]}
    }
  }
  child {node {A}
    child {node {1}}
  }
  child {node {B}
    child {node {1}}
  }
\end{tikzpicture}
\caption{An extended form game.}
\end{figure}
There are two players (named 1 and 2) and three rounds. First player 1 plays, then player 2, then player 1 again. The numbers on the leaves denote the payoffs to the first and second players, respectively (as labeled at the internal nodes of the tree). The labels on the edges denote the names of the actions they can play at that turn.

(a) Find a subgame-perfect Nash equilibrium for this game.

(b) Find a pure Nash equilibrium such that both players receive strictly higher payoff than in the subgame-perfect Nash equilibrium from part (a).

Recall that for both parts, you must list a pure strategy for every internal node (even if it is never actually “played”). For example, \{B, w, h\} is not a complete answer, but \{B, y, w, b, d, f, h\} is (i.e. you must still specify what actions “would have” been selected if those nodes were reached). In both parts, you should briefly argue why the strategies you pose form a (subgame-perfect) Nash.