1 Introduction

In this lab, you will be implement agent strategies for the **Lemonade Stand Game**,[1] a generalization of a famous game of electoral politics studied by Hotelling.[2]

2 Setup

You can find and clone the stencil code for Lab 5 here. The stencil code includes:

- `my_lemonade_agent.py`
- `my_rl_lemonade_agent.py`
- `my_agent.py`*
- `cslogin.txt`*

The star annotations indicate which files you will edit during this lab. The others are purely support code and/or already-implemented opponent bots for the simulations.

The only file you are required to develop to complete this lab is `my_agent.py`, but you may find it helpful to experiment with different strategies in `my_lemonade_agent.py` or `my_rl_lemonade_agent.py` before developing your submission.

Please be sure to read the `README.md`. Here is an abridged version of the setup guide described in there:

1. Clone the repository with the command
   `git clone https://github.com/brown-agt/lab05-stencil.git`
2. Create a python virtual environment and activate it. Be sure to use `python 3.10` or higher.
3. Run `pip install --upgrade agt_server`

IMPORTANT: The `README.md` will also contain helpful methods for implementing your agent.

3 The Lemonade Stand Game (Again!)

The Lemonade Stand Game is a three-player game, in which the players are lemonade stand proprietors trying to maximize their revenue (i.e., sell as much lemonade as possible).

The three players have been issued permits to set up their lemonade stands at one of twelve possible evenly-spaced spots on a circular beach around a lake—imagine the possible lemonade-stand locations as the hours on a clockface.

Every morning, 12 beachgoers spread themselves out evenly on the beach, one at each hour. Over the course of the day, they all buy two cups of lemonade, one from the closest stand to their left, and another from the closest stand to their right.

If there is one lemonade stand at a beachgoer's location, she buys both cups from that one stand. If there are two lemonade stands at a beachgoer's location, she buys one from each. If all three lemonade stands are at her location, she buys from only two of them, selecting those two at random.

A player's payoff equals the total number of cups of lemonade it sells.

[1] Martin Zinkevich, Michaels Bowling and Wunder. Solving Unsolvable Games, *ACM SIGecom Exchanges*, 10(1):35–38, 2011.
[2] Harold Hotelling. Stability in Competition. *Economic Journal*, 39:41–57, 1929.

As you can see, your payoff in this game depends heavily on what your opponents do. You might even reason that without any idea of how your opponents will play, it is impossible to design a meaningful strategy![3]

Question: Think about some possible strategies for this game. If you know how your opponents tend to play, how would you respond to their strategies? How would you try to learn how your opponents are playing?

4 Your Task

Your goal in this lab is to build an agent for for the Lemonade Stand game using any strategy you like. Your agent will then compete in multiple 100-round competitions, each one against two other agents, chosen uniformly at random among your classmates' submissions.

This lab is completely open ended. We are not suggesting that you implement any particular type of strategy, or any other. We are providing stencil code that supports RL strategies (with states, actions, transitions, etc.), along with (more generic) stencil code that supports non-RL strategies.

You only need to complete the stencil code in the `my_agent`, and not the stencil in `my_lemonade_agent` or `my_rl_lemonade_agent`, but you may find it helpful to complete those as well, so that you can experiment with your agent against RL and non-RL strategies.

4.1 Implementing an RL Agent

The `my_rl_lemonade_agent.py` file contains the stencil code for implementing an RL-based agent.

- `my_rl_lemonade_agent.py` is where you will implement your particular RL algorithm, whether it is Q-learning, deep Q-learning, or something else. In order to implement an RL algorithm, you should fill in the `get_action` and `update` methods. `get_action` should return an `int` between 0 and 11 (inclusive), indicating the spot along the beach where your agent plans to set up shop. `update` is where you update your model's weights, should you choose to train a model.
- `my_rl_lemonade_agent.py` is populated with a few instance variables such as a Q-table, but depending on your strategy, you are free to add, remove, or change some of them. Make sure you initialize your Q-values appropriately. **Hint:** Be optimistic!
- You must choose a state space and a state-space representation. Since your opponents are likely to be learning agents, the environment is non-stationary. As a result, it is necessary—in principle, at least—to encode the entire history of observation-action pairs as the state. As this is intractable in general, the two practical options are: 1. hand design a tuple of features using domain knowledge of the game history (e.g., include a Tit-for-Tat flag), or 2. build a deep neural network that learns an embedding, i.e., a high-dimensional vector representation of the history.
- In addition, you can write an optional `train` function so that your agent can train itself in self play for 100,000 steps before the competition is run. Doing so will automatically produce a weights file as a `npy`. If you choose *not* to write a `train` function, please save your Q-table or a set of neural network model weights as a `npy`.

4.2 Implementing a Non-RL Agent

To implement a generic (i.e., non-RL) agent, you should fill out `my_lemonade_agent.py`.

[3] Garry Kasparov, chess grandmaster and perhaps the greatest chess player of all time, claims that this was part of (or perhaps even entirely) the reason he ultimately lost his match to IBM's Deep Blue. Just like pitchers know the batters they face, Kasparov always studied the past plays of his other opponents. But he was not permitted access to Deep Blue's algorithmic workings, so when the machine did something unusual, Kasparov was caught off guard.

There are two methods to fill in: `get_action` and `update`. As in prior labs, these are called once per round, with `get_action` getting your agent's action for that round, and `update` specifying your between-round updates. Just as in the RL version, `get_action` should return an `int` between 0 and 11 (inclusive), indicating the spot along the beach where your agent plans to set up shop.

As this is a very open-ended task, you may implement any strategy here, including algorithms from past labs like Exponential Weights, Fictitious Play, or a Finite State Machine.

4.3 Testing your Competition Agent Locally

To test your agent before submitting, simply run your `my_agent.py` file. Doing so will launch a 1000-round local competition in which your agent competes against two copies of itself. You should run this test to make sure that your agent will not crash in the class competition.

If you chose to implement an RL agent, this process will also include a 100,000-round training phase, before launching the competition.

5 Tips for RL-Based Agents

The Lemonade Stand game is more complex than the repeated games played Labs 1, 2, and 3. In this section, we spell out some of the main considerations in scaling up an RL-based agent to this more complicted domain.

5.1 A Larger State and Action Space

Recall that Q-learning associates each action, taken at each state, with an expected reward. Your agent learns this expected reward during the training stage by repeatedly choosing actions at a sampling of states. If, however, a state-action pair is never experienced by the agent in the training stage, your agent will learn nothing about the expected reward of that pair.

In a game with a small number of states and actions, like those in Lab 3, an RL agent can probably visit each state-action pair often enough to learn a good estimate of its expected reward. But in a larger game like the Lemonade Stand game, this becomes increasingly difficult. At a high level, there are two ways to tackle this difficulty, while still representing Q-values in a table.[4] The first is to use a coarse state representation, so that there are relatively few state-action pairs to visit. The second is to try to figure out which state-action pairs are the most relevant, and to focus on learning about them.

Reducing the number of states, and hence state-action pairs: Imagine that you wanted your state to depend on both of your opponents' last 2 moves. In the Lemonade Stand game, this would mean a state space consisting of $12^4 = 20736$ states, resulting in a Q-table of $20736 \cdot 12 = 248832$ state-action pairs. Even a few million rounds of training would result in a highly sparse Q-table, and thus an underspecified policy.

Perhaps the easiest way to create a coarse state representation in the Lemonade Stand game is to divide the board up into sections—as halves, thirds, quarters, or sixths. Intuitively, if you could predict which third/quarter/sixth of the board your opponents would visit, you could play a near-best-response. So the idea is: instead of recording your opponents' *exact* locations, you just record the *section* of the board on which they play. Using thirds as an example, and again recording the opponents' moves during the past two rounds, would now require only $3^4 = 81$ states, and thus under 1000 state-action pairs—a vast improvement!

[4] An alternative might be to represent Q-tables using a neural network, which you are welcome to do if you are versed in such topics, but is by no means necessary to complete this lab.

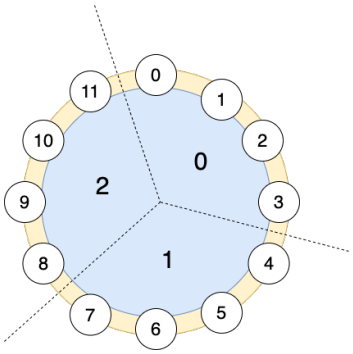

Figure 1: Dividing an opponent's action by 4 yields the "third" of the board they play. This trick works analogously for any factor of 12!

Inferring your opponents' strategies: Another idea is to enumerate a few possible opponent strategies, and then to try to relate your actual opponents' behaviors to these possibilities. For example, consider the two possible strategies "consistent" and "erratic". You could observe your opponents' previous actions, and then determine, based on the distances they travel between one round and the next, whether you consider them "consistent" or "erratic." You could then incorporate that information into your state space, thereby recording information that spans multiple past rounds in very few states.

These and similar ideas can be refined and combined in search of the perfect balance of expressiveness and conciseness. For example, you could break the board into thirds, devise two possible opponent strategies, and then your states could comprise both a strategy and both opponents' past two moves in terms of board sections. In this way, you would have created a representation that incorporates quite a lot of coarse information, stored in roughly the same number of states as both opponents' precise last moves.

Note that you can also always *de*crease the number of rounds in the training phase. But please do not increase it, so that your agent concludes its training phase fast enough to enter the competition. (To test for this failure mode, follow the testing instructions in Section 4.3.)

5.2 Q-table Initialization

Another key difference between the Lemonade Stand game and the games your RL agents played in Lab 3 is the larger range of payoffs. In Chicken, for example, the payoff range was from -5 to 1, and the equilibrium payoffs were centered around 0. In the Lemonade Stand game, in contrast, your agent can achieve a payoff of up to 12, and the symmetric equilibrium payoffs are 8.

The way a Q-table is initialized is important, especially for agents that train on-policy. Suppose the Q-value at all state-action pairs were initialized to -1. Then, since payoffs are non-negative in the Lemonade Stand game, the very first action to be played at a state, regardless of the quality of the move, would always look wonderful compared to all the other actions, whose values are negative. That action would then be favored, so that learning to move away from that action towards a better one would be slow.

A more promising approach is to initialize the Q-value at all state-action pairs to 12. Then, all the actions that have been played at a state would look worse than those have not yet been played, leading the agent to explore all actions at a state at least once before being able to formulate a preferred strategy.

6 Submitting your Lab

Submit the folder containing `my_agent.py` via Gradescope, along with any helper files necessary to run your agent, e.g., a data file containing a Q-table. (The support code is already configured to save a Q-table for you automatically.)

Please be sure to include both you and your partner's CS logins as separate lines in `cs_login.txt`.
And please make sure to name your agent.

N.B. Please be sure that `my_agent.py` contains a valid agent under `agent_submission`; otherwise, your agent will not run in the competition.