1 Introduction

In this lab, you will be implementing agent strategies for the game **Battle of the Sexes**. You will be playing both a complete-information and an incomplete-information version of the game. The agent strategies will take the form of **finite state machines**.

2 Setup

The stencil code for Lab 2 is available here. It includes six Python files:

- `bos_competition_agent.py`*
- `bos_finite_state_agent1.py`*
- `bos_finite_state_agent2.py`*
- `bos_punitive.py`
- `bos_reluctant.py`
- `bosii_competition_agent.py`*

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

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/lab02-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`

3 Battle of the Sexes

Battle of the Sexes is another classic game theory problem, just like the Prisoners' Dilemma. In this problem, Alice and Bob made plans to go to either a concert or a lecture together, but they both forgot which one they agreed on, and cannot communicate with each other beforehand. Instead, they each must choose one to go to, and hope the other one also shows up. They are both unhappy (i.e., zero payoffs) if they go to different events, and are both happy if they go to the same event. However, Alice prefers the concert to the lecture, and Bob prefers the lecture to the concert.

The payoff matrix of the game as is follows (Alice is the row player):

	C	**L**
C	7, 3	0, 0
L	0, 0	3, 7

An interesting feature of the game is the presence of two cooperative outcomes, each one favoring one of the players. The players both receive a positive payoff if they choose the same event, but how can they figure out whose preferred event they should go to?

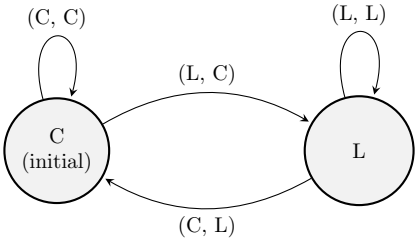
Question: How would you play this game against an opponent whose strategy is unknown to you?

4 Finite State Machines

Finite state machines (FSMs) can be used to model strategies in repeated normal-form games. Specifically, they maintain a state, which captures selected aspects of the history of the game, based on which the agent chooses its next action. Formally, in the context of a repeated game, a FSM consists of the following:

- A set of states with a select initial state.
- A function that assigns an action to every state.
- Rules that govern the transition from one state to the next based on the outcome (i.e., action profile) of one round of the game.

An example of one such strategy for Alice in the Battle of the Sexes is represented by this machine. Here, Alice begins by going to the concert, and continues to do so as long as Bob also goes. However, if Bob attends the lecture, Alice's next move will be to go to the lecture, again and again, for as long as Bob also goes. This strategy could be described as a **"follower"** strategy, as Alice always plays Bob's last move.



Question: Against which types of players would this be a strong strategy? Against which types of players would this be a weak strategy?

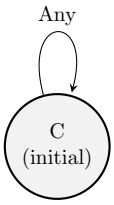
4.1 Finite State Machine Strategies

Below are some additional examples of FSMs (i.e., strategies) for Battle of the Sexes. Think about the strengths and weaknesses of each one, and which elements you may want to incorporate into your own strategy. Your strategy can be as simple or as complicated as you want. Any idea is worth trying!

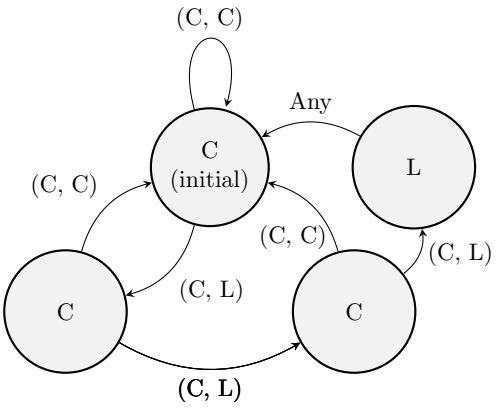
N.B. All of the strategies below are for the row player. Keep in mind that you may not be the row player in the simulation, but since the game is symmetric, you can just substitute the moves with their opposites.

The states in these diagrams are labelled with Alice's action, and the transitions, with Bob's (and Alice's).

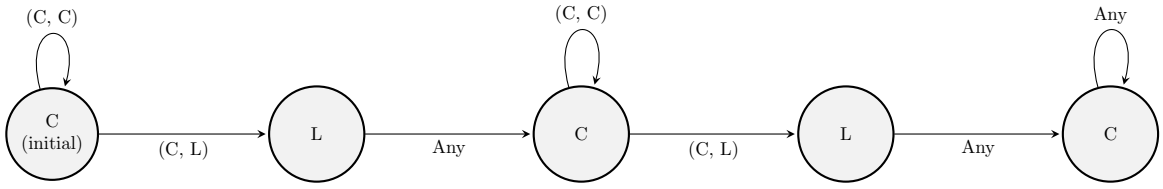
- **"Uncompromising":** Alice disregards Bob's wishes and unrelentingly and unrepentantly always goes to the concert. He can join her if he wants.



- **"Reluctant to Compromise":** Alice always attends the concert, except when Bob went to the lecture three times in a row. After attending the lecture once, Alice goes straight back to the concert.



- **"Punitive":** Alice goes to the concert as long as Bob does. Once Bob breaks this compromise, Alice compromises by going to the lecture once. However, if Bob breaks the compromise 3 times, Alice retaliates by going to the concert forever after.



Question: What are some advantages of using FSMs over the strategies from Lab 1? What are some disadvantages?

4.2 Simulations

In this lab, you will be implementing strategies as FSMs. Your goal will be to beat several sample agents, which are also FSMs. The games will last 100 rounds.

You will play as Bob. However, to generalize the moves to both players, instead of `CONCERT` and `LECTURE`, your available moves will be `STUBBORN` (for Bob this means going to the lecture) and `COMPROMISE` (for Bob this means going to the concert). In other words, the game you are now playing looks like this:

	S (Lecture)	**C** (Concert)
S (Concert)	0, 0	7, 3
C (Lecture)	3, 7	0, 0

Take a look at `bos_reluctant.py` and `bos_punitive.py`, which, respectively, implement the **"reluctant to compromise"** and **"punitive"** strategies from Section 4.1.

To **counter** the "reluctant to compromise" strategy, implement an effective response in `bos_finite_state_agent1.py`. To **counter** the "punitive" strategy, implement an effective response in `bos_finite_state_agent2.py`.

Hint: Consider an objective such as one of these two when designing your strategies:

- **Maximize your agent's absolute utility**: i.e., aim for the highest possible payoff.
- **Maximize your agent's utility relative to your opponent's**, even if that ultimately means making both agents worse off.

Task: To implement your strategies, fill in the following two methods in these files:

- `get_action()`: Return either `self.STUBBORN` or `self.COMPROMISE`, depending on the current state of the game, as stored in the instance variable `self.curr_state`. Note that `self.curr_state` is initialized to 0, which represents your initial state.
- `update()`: Return the next state, based on your current state and the actions taken by each player in the previous round of the game, and update `self.curr_state` to reflect the new state of the game.

You are free to make your strategy as simple or complicated as you want, so long as it is a FSM that beats the sample agents. There are many possible ways to beat these simple strategies. For fun, you might try to see just how high of payoffs you can achieve!

4.3 Competition

Now that you have designed a FSM that has defeated the sample agents, whose strategies were known to you, you will design a FSM to be paired up against a classmate's agent for another 100 rounds.

You will not know whether you are Alice or Bob, but this should not matter, since the game as we defined it, in terms of `STUBBORN` and `COMPROMISE`, is symmetric.

Task: For the competition, just as for the simulations, you job is to write the `get_action()` and `update()` methods. You should do this in `bos_competition_agent.py`. Keep in mind that this task is notably harder than the last, since you do not know the opponent's strategy. Be creative. Try to think about how your classmates' agents may play, and from there, think about ways in which you might respond.

Task: Test your agent before submitting by running your agent file, `bos_competition_agent.py`. Doing so will launch a 1000-round local competition in which your agent competes against itself. You should run this test to make sure that your agent will not crash in the class competition.

5 Battle of the Sexes: Incomplete Information

Next, we'll explore a variation of Battle of the Sexes in which Alice has **incomplete information**. In particular, Alice does not know whether Bob is in a good mood or a bad mood. If Bob is in a good mood, he would like to see Alice, and the original payoffs are maintained. But if Bob is in a bad mood, he receives higher payoffs from avoiding Alice rather than from meeting her. Let's assume Bob is in a good mood with probability $2/3$; this outcome is represented by the payoff matrix on the left. Bob is then in a bad mood with probability $1/3$; this outcome is represented by the payoff matrix on the right.

$P = 2/3$	**S** (Lecture)	**C** (Concert)
S (Concert)	0, 0	7, 3
C (Lecture)	3, 7	0, 0

$P = 1/3$	**S** (Lecture)	**C** (Concert)
S (Concert)	0, 7	7, 0
C (Lecture)	3, 0	0, 3

Note that Alice's payoffs are not affected by Bob's mood—she still prefers the concert, and wants to see Bob. So why should Bob's mood affect Alice's strategy? Because Bob's mood affects how Bob will play, and thus Alice's chance of reaching a cooperative state!

Although incomplete-information games are more complicated to analyze than complete-information games, Fictitious Play, Exponential Weights, and FSMs are all still viable strategies.

In complete-information games, the data collected after each round of play contains all players' actions and payoffs. In incomplete-information games, the post-round data also contains the players' private information, which in this game means Bob's mood during that round. That way, Alice can design a strategy based not only on how Bob acted in the past, but conditioned on whether he is in a good or bad mood, provided she also predicts his mood.

5.1 Fictitious Play

To use Fictitious Play in Incomplete Information Battle of the Sexes, Alice can keep track of *two* empirical probability distributions over Bob's past actions—one for each of his moods. Call these distributions $\hat{\pi}_G$ and $\hat{\pi}_B$. Alice should also keep track of how often Bob was in a good vs. a bad mood. Using this information, Alice can compute her expected of playing, for example $\mathbf{S}$, as follows:

$$\begin{aligned}\mathbb{E}_a[\mathbf{S}] = \Pr(\text{good})\ &[\hat{\pi}_G(\mathbf{S})\ u_a(\mathbf{S},\mathbf{S};\text{good}) + \hat{\pi}_G(\mathbf{C})\ u_a(\mathbf{S},\mathbf{C};\text{good})] \\ + \Pr(\text{bad})\ &[\hat{\pi}_B(\mathbf{S})\ u_a(\mathbf{S},\mathbf{S};\text{bad}) + \hat{\pi}_B(\mathbf{S})\ u_a(\mathbf{S},\mathbf{C};\text{bad})]\end{aligned}$$

Then, as usual, she can choose an action that maximizes her payoffs, given the game's history.

The situation is slightly simpler for Bob, since he knows his mood before he makes a move. But Bob still must maintain a probability distribution, say $\hat{\rho}_G$, over Alice's past actions, when Bob was in a good mood; and likewise, $\hat{\rho}_B$, when he was in a bad mood. With this information in hand, Bob can compute his expected payoff of playing, for example $\mathbf{S}$, when he is in a good mood as follows:

$$\mathbb{E}_b[\mathbf{S}] = \hat{\rho}_G(\mathbf{S})\ u_a(\mathbf{S},\mathbf{S};\text{good}) + \hat{\rho}_G(\mathbf{C})\ u_a(\mathbf{C},\mathbf{S};\text{good})$$

5.2 Exponential Weights

Similarly, to extend the Exponential Weights strategy to this incomplete-information game, the players should keep track of two average reward vectors, conditioned on Bob's mood.

From Alice's perspective, she can use these average reward vectors, in conjunction with the probability of each of Bob's moods, to calculate her *expected* average reward for each action. She can then use the classic Exponential Weights formula to construct a probability distribution over her possible actions.

For example, if Alice keeps track her average rewards per action when Bob is in a good mood and a bad mood in vectors $\hat{r}_G$ and $\hat{r}_B$, respectively, then her expected average reward for playing $\mathbf{S}$ is:

$$\mathbb{E}_a[\mathbf{S}] = \Pr(\text{good})\ \hat{r}_G(\mathbf{S}) + \Pr(\text{bad})\ \hat{r}_B(\mathbf{S})$$

So her probability distribution over her next action would be calculated as follows via Exponential Weights:

$$\Pr(\mathbf{S}) = \frac{e^{\mathbb{E}_a[\mathbf{S}]}}{e^{\mathbb{E}_a[\mathbf{S}]} + e^{\mathbb{E}_a[\mathbf{C}]}} \qquad \Pr(\mathbf{C}) = \frac{e^{\mathbb{E}_a[\mathbf{C}]}}{e^{\mathbb{E}_a[\mathbf{S}]} + e^{\mathbb{E}_a[\mathbf{C}]}}$$

From Bob's perspective, he ought to track the same data, but since he *knows* his mood, he can use the corresponding average reward vector as usual. Given $M \in \{\text{good}, \text{bad}\}$,

$$\Pr(\mathbf{S}) = \frac{e^{\hat{r}_M[\mathbf{S}]}}{e^{\hat{r}_M[\mathbf{S}]} + e^{\hat{r}_M[\mathbf{C}]}}, \quad \Pr(\mathbf{C}) = \frac{e^{\hat{r}_M[\mathbf{C}]}}{e^{\hat{r}_M[\mathbf{S}]} + e^{\hat{r}_M[\mathbf{C}]}}$$

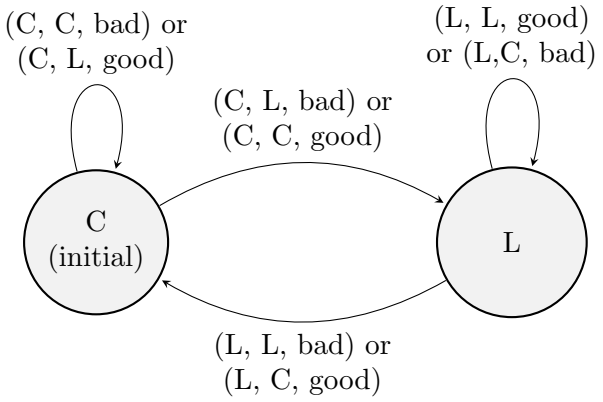
5.3 Finite State Machines

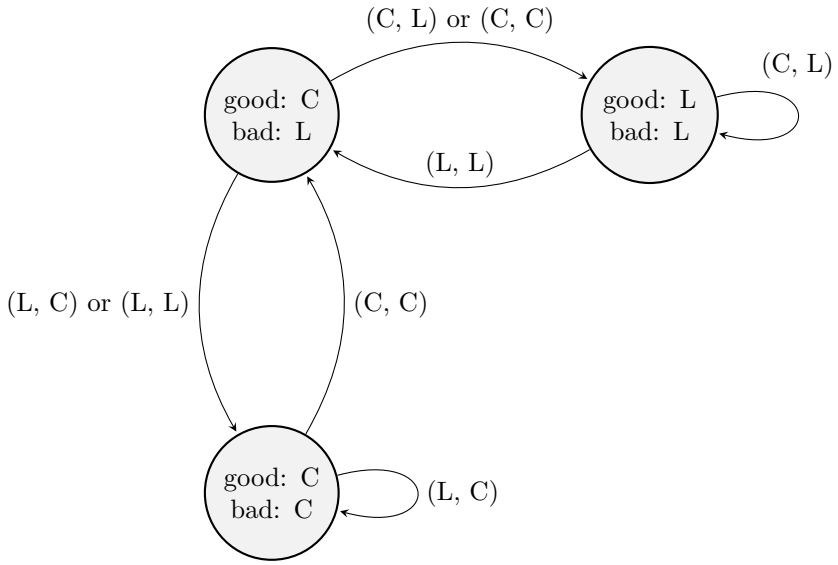
Finally, FSMs should also incorporate Bob's mood somehow. From Alice's perspective, she can use his past moods as part of the state space, just as she previously used his past actions. From Bob's perspective, he can incorporate his *current* mood into the state space.

Here are some possible strategies for Alice, based on our previous ideas for the complete-information game:

- **"Uncompromising":** Alice disregards Bob's wishes and unrelentingly and unrepentantly always goes to the concert, *irrespective of his past moods.* He can join her if he wants.
- **"Reluctant to Compromise":** Alice always attends the concert, except when Bob *was in a bad mood three times in a row,* and correspondingly went to the lecture all three times. After attending the lecture once, Alice goes straight back to the concert.
- **"Punitive":** Alice goes to the concert as long as Bob does. Once Bob breaks this compromise, Alice will compromise by going to the lecture once, *if Bob was in a bad mood.* However, if Bob breaks the compromise 3 times—going to the lecture three times *when he was in a good mood*—Alice retaliates by going to the concert forever after.

And here are two altogether different strategies. The first one is for Alice, and the second, for Bob. See if you can figure out what they are doing.





5.4 Competition: Incomplete-Information Battle of the Sexes

In this competition, you will be implementing an agent to compete against a classmate for 100 rounds (50 rounds as Alice and 50 rounds as Bob) in **Incomplete-Information Battle of the Sexes**. The payoff matrices are depicted at the beginning of Section 5. Note, however, you will not know until it is time to make your first move whether you are the row player or the column player—you do remain the same player throughout the entire 50-round games, though.

To participate in the competition, you must write the `get_action()` method of `bosii_competition_agent.py`, which returns `S` or `C`. Because the information is incomplete, we have provided you with the following helper methods (imported from `bosii_competition_agent.py`):

- `is_row_player()` returns `true` if you are the row player (and thus have incomplete information) and `false` if you are the column player.
- `get_mood()` returns your current mood: either `self.GOOD_MOOD` or `self.BAD_MOOD`, provided you are the column player. The mood determines the payoff matrix. If you are the row player, this method returns `None`, because your mood does not vary and you do not know the opponent's mood.
- `get_action_history()` returns a list of the player's historical actions over all rounds played in the current matching so far
- `get_util_history()` returns a list of the player's historical payoffs over all rounds played in the current matching so far
- `get_opp_action_history()` returns a list of the opponent's historical actions over all rounds played in the current matching so far
- `get_opp_util_history()` returns a list of the opponent player's historical payoffs over all rounds played in the current matching so far
- `get_mood_history` returns a list of the column player's moods over all rounds played in the current matching so far, if you are the column player or `None`, if you are the row player.
- `get_last_action()` returns the player's actions in the last round if a round has been played, and `None` otherwise
- `get_last_util()` returns the player's payoff in the last round if a round has been played, and `None` otherwise
- `get_opp_last_action()` returns the opponent's action in the last round if a round has been played, and `None` otherwise
- `get_opp_last_util()` returns the opponent's payoff in the last round if a round has been played, and `None` otherwise
- `get_last_mood()` returns your last mood in the previous round if you are the column player and a round has been played, and `None` otherwise
- `row_player_calculate_util(row_move, col_move)` returns the row player's hypothetical utility given action profile `(row_move, col_move)`
- `col_player_calculate_util(row_move, col_move, mood)` is analogous, but returns the column player's hypothetical utility, and depending on her mood
- `col_player_good_mood_prob()` returns the probability that the column player is in a good mood, which is useful in expected utility calculations

Because only the column player has two moods, we recommend you separate your strategy into two, one for the row player, and the other for the column, as follows:

```
if self.is_row_player():
    #  (your row-player strategy)
else:
    my_mood = self.get_mood():
    #  (your column-player strategy)
}
```

Because this game is unbalanced, your agent will play both roles, roughly equally often.

Task: Test your agent before submitting by running your agent file, `bosii_competition_agent.py`. Doing so will launch a 100-round local competition in which your agent competes against itself. You should run this test to make sure that your agent will not crash in the class competition.

5.5 Testing your Competition Agent Locally

Before participating in the class competition, you can test your agent locally, in a mock competition against a TA bot. To do so, simply run your competition agent file, `bosii_competition_agent.py`, in your terminal of choice. We implemented this functionality so that you can make sure your agent does not crash in competition mode, thereby making it more likely your agent will run smoothly in the class competition.

The class competition is also launched via `bosii_competition_agent.py`, using the `join_server` boolean, along with the appropriate IP address and port. This latter information will be announced during lab.

A Simulation Details; TLDR

Recall from Lab 1 that at a high-level, the AGT_SERVER simulates a repeated game tournament as follows:

1. Your agent is paired against another agent (or set of agents, as the game rules require) in a round robin style. Each agent will face off against every other agent in a repeated game *twice*, once as player 1 and once as player 2.
2. Before each repeated game simulation begins, the AGT_SERVER calls each agent's `setup()` method.
3. Then, during each round of the simulation:
   (a) AGT_SERVER requests an action from each agent. Upon receiving this request, each agent calls its `get_action()` method, and then sends its action to AGT_SERVER.
   (b) AGT_SERVER executes these moves and calculates payoffs.
   (c) AGT_SERVER then broadcasts the results back to the agents in a *sanitized* json report summarizing the round's results. (The report is sanitized, as not all information is necessarily broadcast to all agents.) Upon receipt, your agent's history is automatically updated with the information it receives, which it can then retrieve using methods like `self.get_action_history()`, `self.get_util_history()`, etc..
   (d) Once the round has concluded, AGT_SERVER calls the agent's `update()` method, which gives it a chance to update its strategy.
4. After the simulation (i.e., all rounds of the repeated game) conclude, AGT_SERVER resets the game history and calls the agents `setup()` methods to allow them to reset for the next simulation.

The calls to the `get_action()` method are implemented differently for Fictitious Play and Exponential Weights agents. For the former, the AGT_SERVER calls the agent's `predict()` method, so that it can build its probability distribution, and then `optimize()` to solicit its next move. For the latter, the AGT_SERVER calls `calc_move_probs()`, and then samples from this distribution to arrive at the agent's next move.