1 Introduction

In this lab, you will be using our Java Trading Platform to implement 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 this week will take the form of **finite state machines**.

2 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 who gets to go to their preferred event?

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

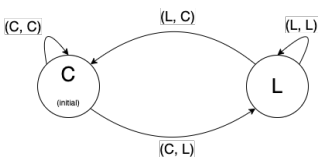
3 Finite State Machines

Finite state machines (FSMs) can be used to model of strategies in repeated normal-form games. Specifically, they maintain a state, which captures some aspect of the history of the game, based on which the agent chooses its next action.

Formally, in the context of a repeated game, a finite state machine consists of the following:

- A set of states, and 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, 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?

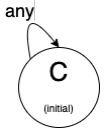
3.1 Additional Strategies with Finite State Machines

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!

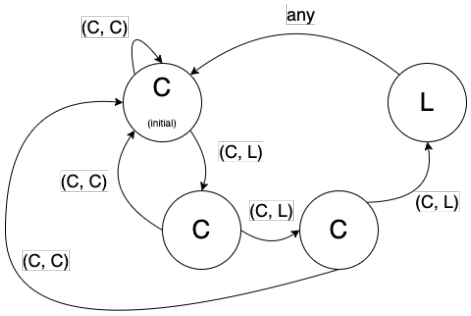
Note: All of the strategies below are for the row player. Keep in mind that you may not be the row player in the simulation, but since the game is symmetric, you can just substitute the moves with their opposites.

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

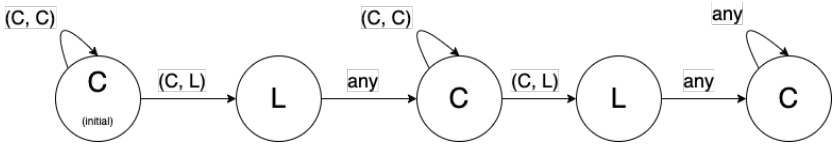
- **"Uncompromising":** Alice disregards Bob's wishes and unrelentingly and unrepentantly always goes to the concert. He can join her if he wants.



- **"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 will compromise 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 finite state machines rather than the strategies from Lab 1? What are some drawbacks?

3.2 Simulation: Implementing Finite State Machines

For the first simulation, you will be implementing a strategy as a finite state machine. 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

3.2.1 Installation

If you have not completed Lab 1, please complete its installation instructions (Section 4.3.1). If you have done Lab 1, there is no need to repeat this step.

Once you have installed the Game Simulator, download the agent files from the course website and place them in a package in your existing project. Then edit line 1 in each agent to reflect the correct package name (just as you did in Lab 1). After correcting the package name, there should be no errors in your code that prevent it from compiling. If you have errors related to importing the GameSimulator code, try **Project** → **Clean**. If this doesn't work, try re-importing the external JAR, or ask a TA for help.

3.2.2 Implementation

You will be writing the same method in two different agent files.

In `BoSFiniteStateAgent1.java`, you will implement a strategy to beat the **"reluctant to compromise"** strategy from Section 3.1. In `BoSFiniteStateAgent2.java`, you will implement a strategy to beat the **"punitive"** strategy from Section 3.1. Take a look at `BoSReluctantAgent.java` and `BoSPunitiveAgent.java` to see how the opponent code works.

To implement your own strategy, fill in the following two methods:

- `nextMove` should return either `STUBBORN` or `COMPROMISE` based on the current state of the game, as stored in the instance variable `this.currentState`. Note that `this.currentState` is initialized to 0, which represents your initial state.
- `transitionState(myMove, opponentMove)` should, based on your current state, and the actions taken by each player in the previous round of the game (which are derived from the game report received after each round), updates `this.currentState` 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 finite state machine and it beats the sample agents. There are many possible ways to beat these simple strategies – for fun, try and see just how much payoff you can get!

3.2.3 Running the Simulation

To run the simulation, press the **Run** button in your agent files. Note that your agents in this lab should return their moves within 1 second. If ever either agent fails to do so, that round of the game is skipped.

3.3 Competition: Finite State Machines

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, with which your agent will compete for, as usual, 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.

3.3.1 Implementing your Agent

Just like in the simulation, you will be writing the `nextMove` and `transitionState` methods. For the competition, you will do this in `BoSCompetitionAgent.java`. 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 might respond.

Just as in Lab 1, please be sure to give your agent a name, and to change the `HOST` variable to the correct hostname, as instructed by a TA.

3.3.2 Entering the Competition

To enter the competition, await TA instructions, and then press the **Run** button.

4 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.

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

	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 Finite State Machines are all still viable strategies.

In complete-information games, the Game Simulator's Game Reports after each round of play contain all players' actions, and their rewards. For incomplete-information games, the Game Reports also contain Bob's mood each round. That way, Alice can design a strategy based not only on how Bob acted in the past, but how he acted when he was in either a good or a bad mood.

4.1 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 payoff of playing, for example $\mathbf{C}$, as follows:

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

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 $\mathbf{C}$, as follows:

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

4.2 Exponential Weights

Similarly, to extend the Exponential Weights strategy to this incomplete-information game, the players should keep track of two average rewards 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 form 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 $\mathbf{C}$ is:

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

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

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

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 $M \in \{\text{good}, \text{bad}\}$,

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

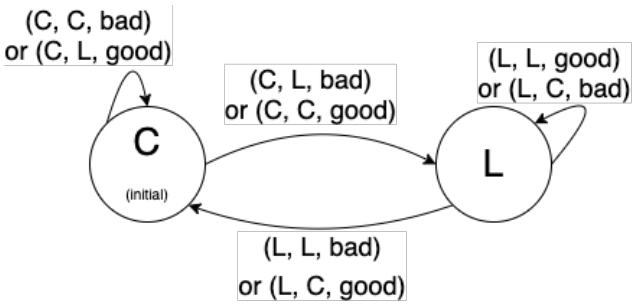
4.3 Finite State Machines

Finally, finite state machines 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.

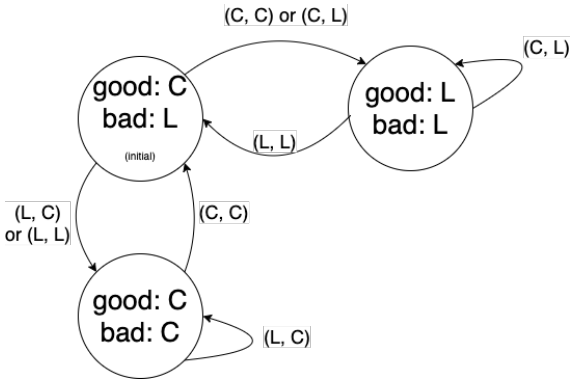
Here are some possible strategies for Alice, based on our previous ideas for the complete-information game:

- **"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're doing.



(a) A FSM for Alice.



(b) A FSM for Bob.

4.4 Competition: Incomplete-Information Battle of the Sexes

In this competition, you will be implementing an agent to compete against a classmate for 100 rounds in **Incomplete-Information Battle of the Sexes**. The payoff matrices are the ones depicted above. 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 100-round game, though.

To participate in the competition, you must write the `nextMove` method of `AbsBoSIIAgent.java`, which returns `STUBBORN` or `COMPROMISE`. Because the information is incomplete, we have provided you with the following helper methods (imported from `AbsBoSIIAgent`):

- `isRowPlayer()` returns `true` if you are the row player (and thus have incomplete information) and `false` if you are the column player.
- `getMood()` returns your current mood: either `GOOD_MOOD` or `BAD_MOOD`, provided you are the column player. The mood determines the payoff matrix. If you are the row player, this method returns `null`, because your mood does not vary and you do not know the opponent's mood.
- `getGameHistory()` returns an ordered `List<GameRound>` of the outcomes of all the rounds thus far in the game. To see the results of only the last round, simply access the last element of the List.

  `GameRound` objects provide the following methods:

  - `getMyMove()` returns what you played: either `STUBBORN` or `COMPROMISE`.
  - `getOpponentMove()` returns what your opponent played: either `STUBBORN` or `COMPROMISE`.
  - `getColumnPlayerMood()` returns the column player's mood when the move was made: `GOOD_MOOD` or `BAD_MOOD`.
  - `getMyReward()` gets your reward from this round of the game.
  - `getOpponentReward()` gets the opponent's reward from this round of the game.

- `rowPlayerRewardFrom(rowPlayerMove, columnPlayerMove)` returns the row player's hypothetical payoff from the specified action profile.
- `columnPlayerRewardFrom(rowPlayerMove, columnPlayerMove, columnPlayerMood)` is analogous to this, but returns the column player's hypothetical reward, and depends on their mood.
- `columnPlayerGoodMoodProbability()` returns the probability that the column player is in a good mood (this can be useful in probability/expected value calculations).

Because any given strategy can differ vastly between the row-player and column-player implementations, we recommend separating your two strategies. For example, you can use the following structure:

```
if (isRowPlayer()) {
    // ... (your row-player strategy)
} else {
    Integer myMood = getMood();
    // ... (your column-player strategy)
}
```

Just as in Lab 1, please be sure to give your agent a name, and to change the `HOST` variable to the correct hostname, as instructed by a TA.

4.4.1 Entering the Competition

To enter the competition, await TA instructions, and then press the **Run** button.

Because this game is unbalanced, we would like to give you a chance to play both roles and/or adjust your strategies. So time permitting, we will run multiple rounds of this competition.