1 Introduction

In this lab, you will be working with our Java trading platform to implement agent strategies for the **Lemonade Stand Game**,[1] a generalization of a famous game of electoral politics studied by Hotelling in 1929.[2] You will build Q-learning agents to play this game, which are very basic reinforcement learning (RL) agents.

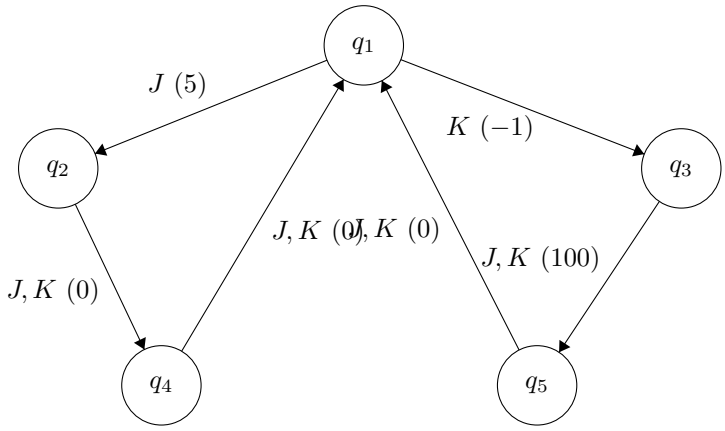
2 Q-Learning

In this section, we will introduce **Q-Learning**, one of the most popular reinforcement learning algorithms.

Reinforcement learning is a machine learning paradigm in which an agent learns the best actions to play at a state through repeated experience. RL algorithms operate in environments called **Markov Decision Processes** (MDPs), which are characterized by:

1. S: the set of states
2. A: the set of actions
3. $T(s, a, s')$: a transition function describing the probability of going from state s to s' via action a
4. $R(s, a, s')$ the reward for taking action a at state s and ending up at state s'
5. an initial state, or a probability distribution over initial states

Example: Here is an example of a Markov Decision Process:



1. $S = \{q_1, q_2, q_3, q_4, q_5\}$, with initial state q_1
2. $A = \{J, K\}$
3. T: All transitions are deterministic, and are represented in the diagram below. (E.g., if you start at state q_1 and take action J, you will end up at state q_2.)
4. R: The rewards corresponding to each transition are depicted in parentheses, next to the relevant action. (e.g., $R(q_1, J, q_2) = 5$)

[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.

A **policy** in an MDP is a function $\pi : S \to A$ from states to actions. An **optimal policy** is one that selects an optimal action at all states. See if you can figure out the optimal policy in our sample MDP.

At state q_1, action J earns you an immediate reward of 5, while action K earns you an immediate reward of -1. However, after taking your first action and transitioning to the next state (either q_2 or q_3), you will earn 100 if you had chosen K, compared to 0, if you had chosen J. Therefore, the optimal choice at q_1 is K.

The goal of Q-learning is to learn an optimal policy in an MDP from simulated experience, without prior knowledge of either the transitions or the rewards. In its most basic form, it learns a Q-table of dimension $|S| \times |A|$ that stores the long-term expected return of choosing action a in state s, for all $s \in S$ and $a \in A$.

The Q-table can be initialized arbitrarily.[3] Then each time the agent moves from state s to s' via action a and earns reward r, the Q-table is updated according to the following rule:

$$Q(s,a) = \alpha(r + \gamma \max_{a'} Q(s',a')) + (1-\alpha)Q(s,a) \ ,$$

where $\alpha \in [0,1]$ is a **learning rate**, which controls how fast new information is incorporated into Q-values; and $\gamma \in [0,1)$ is **discount factor**, which describes how much the agent cares about its present vs. its future rewards—a value close to 1 means it values its present rewards nearly as much as its future rewards.

Question: Starting from an initial Q-table of all zeroes, apply the Q-learning update rule to our sample MDP a few times to see if/how it learns the optimal policy. Assume an initial state of q_1, a learning rate of 0.5 and a discount factor of 0.9.

A Q-learning agent can learn online or offline. To learn **offline** means to learn while simulating play, without actually earning any rewards. To learn **online** means to learn while playing out in the wild, possibly foregoing rewards because the learning process is ongoing.

If an agent is learning online, it might prefer to play **on-policy**, meaning according to its latest and greatest policy, so that it does not forego too many rewards. In contrast, when playing offline, an agent can play **off-policy**, perhaps even uniformly at random, thereby exploring the environment much faster than otherwise. The trade-off between playing according the latest and the greatest vs. favoring exploring as-yet-unexplored parts of the environment is called the **exploration–exploitation** trade-off.

To play well, given its current knowledge—called **exploitation**—the agent chooses an action with the highest expected long-term rewards, as recorded in the current Q table. To continue learning—called **exploration**—the agent occasionally chooses an action at random; specifically, with **exploration rate** $\epsilon \in [0,1]$.

There is a vast quantity of resources about MDPs available for your consumption on the web. For example, we refer you to this video: `https://www.youtube.com/watch?v=qhRNvCVVJaA`. We encourage you to watch and learn more—after lab!

3 Implementing Q-Learning for Chicken

Although definitively *not* a Markov decision process, you will nonetheless be designing Q-learning agents for repeated games. As a test game, we will revisit the game of **Chicken**, which you also played in Lab 1.

The game of Chicken is shown below. However, instead of naming the actions, we have enumerated them.

Action	**0**	**1**
0	0, 0	-1, 1
1	1, -1	-5, -5

[3] Initializations closer to the expected long-term rewards lead to faster convergence.

You will write an agent that returns its next action, 0 or 1, based on the Q-learning algorithm. In order to do so, you will need to morph the game into an MDP. The key step in doing this is to decide on a state-space representation. For example, your agent can choose as the state its last action, its opponents' last action, both agents' last actions, its own past two actions and its opponents' past four actions, etc. Once you settle on a state-space representation, you will write a method that updates the Q-table using this representation, based on its experience playing the game. As the choice of state-space representation is key to learning, this method will determine how effectively your agent learns.

3.1 Installing the Stencil Code

If you have yet 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 Labs 1 and 2). 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 Implementing the Q-Learning Algorithm

To implement the Q-learning algorithm, you will need to navigate to `AbsQLearner.java`.

This class contains some very important instance variables:

- `this.Q` is your Q-table. It has dimensions `[this.numPossibleStates][this.numPossibleActions]`.
- `this.currentState` is the current state of the game.
- `this.numPossibleStates` is the size of the state space.
- `this.numPossibleActions` is the size of the action space.
- `this.learningRate` is the learning rate hyperparameter (α).
- `this.discountFactor` is the discount factor hyperparameter (γ).
- `this.explorationRate` is the exploration rate hyperparameter (ϵ).

You will need to implement the following two methods:

- `nextMove()`. This returns your agent's next action based on the game's current state, as follows:
  - With a probability of `this.explorationRate`, chooses a random action in the range $[0, $`numPossibleActions`$)$. See `Math.random()` and `Random.nextInt(a)`.
  - Otherwise, choose the best possible action based on `this.currentState`: i.e.,

$$\underset{\texttt{i} \in [0,\ \texttt{this.numPossibleActions})}{\arg\max} \texttt{this.Q[this.currentState][i]}$$

- `updateQ(myAction, prevState, nextState, reward)`. This updates the Q-table after each iteration of the game, based on a transition from `prevState` to `nextState`, by way of action `myAction`, producing reward `reward`. Use the Q-learning equation to update `this.Q[prevState][myAction]`.

Be sure to also check out the helper functions `max` and `argmax`.

Finally, you will notice an abstract method called `determineState()`. This is the main method you will implement for your agents. Its purpose is to allow you to translate your agent's experience playing the game (i.e., the information in the Game Report) into an integer representing the game's current state, as you have defined states in your agent's MDP.

3.3 Sample Q-Learning Representations of Chicken

You will be training two separate sample Q-learning Chicken agents which we provide. This exercise will serve to verify the correctness of your Q-learning implementation, and to illustrate the importance of using a robust state-space representation. After running our sample agents, you will design your own agent for Chicken, meaning your own state-space representation, and you will then train your agent to play the game.

You will be training our sample agents against a *very* simple agent located in `BasicChickenQOpponent.java`. Feel free to read through the code for this basic agent—it simply plays actions $[0, 0, 1]$ in sequence, repeatedly. Can Q-learning pick up on this pattern and play a perfect response? As it turns out, the answer depends on the state-space representation!

3.3.1 `LastMoveChickenQLearner.java`: An Insufficient State Space

Navigate to `LastMoveChickenQLearner.java`. Here, you will see the most basic form of a Chicken-playing Q-learning agent. This agent has a state space of 2 possible states. Each state corresponds to the opponent's last move. If the opponent played 0, the state will be 0, and if the opponent played 1, the state will be 1. Note the value of the `NUM_POSSIBLE_STATES` constant and the implementation of `determineState`.

Run this agent. Doing so will launch a training phase against the aforementioned `BasicChickenAgent`, which will last 200 rounds, printing updates every 5 rounds. At the end of this training phase, you can see how much payoff was earned. If you correctly implemented Q-learning, the agent should have achieved a payoff of about 0 (a slightly negative value is possible, due to randomness, but it should earn no less than -20 or -30).

Although the agent plays imperfectly, it does learn to avoid the `(CONTINUE, CONTINUE)` outcome—a payoff of -5.0 should be *extremely* rare. Thus, it avoids a large negative payoff. So it is doing *something* right, but it wasn't quite able to learn the entirety of the other agent's strategy. How can we improve its performance?

3.3.2 `LookBackChickenQLearner.java`: A Sufficient State Space

Navigate to `LookBackChickenQLearner.java`. You will notice a very similar setup in this agent, except that it uses four states, rather than two. Look at `determineState`. Each state represents a combination of the opponent's past *two* moves. Is this enough to predict the agent's next move, and generate a best response?

Run this agent. It will enter the same training phase the other agent did. But this time, it should learn to play perfectly, achieving a high positive payoff (a few dozen over 200 rounds). A small difference in the state space led to a large difference in performance between our two sample agents.

Question: Why was the second state-space representation sufficient to produce a best response to the opponent's strategy, whereas the first one was not?

3.3.3 Implementing your Own State Space Representation

Navigate to `MyChickenQLearner.java`.

Your next task is to devise your own representation of the game of Chicken as an MDP, in order to maximize your payoff against a *mystery* agent. Your opponent's strategy this time will be more intricate than the strategies faced by the sample agents, so you will have to design a more robust state-space representation, in order to account for its possible strategies. Feel free to incorporate its last move, your own last move, and any further historical instances thereof.

Important: After you decide on a state-space representation, be sure to edit the `NUM_POSSIBLE_STATES` constant to reflect the number of states in your representation. The `determineState` method should then return a value in `[0, NUM_POSSIBLE_STATES)`.

Run your agent. It will play Chicken against the mystery agent for 10^5 rounds—the additional rounds should give your agent a chance to learn to play well against a more sophisticated strategy.

How did you do? *If you don't do well, do not despair!* It can be very difficult to achieve positive payoff against an unknown opponent in Chicken, so achieving even a small negative payoff is itself impressive. You should aim for at least about -0.05, on average across all 10^5 rounds.

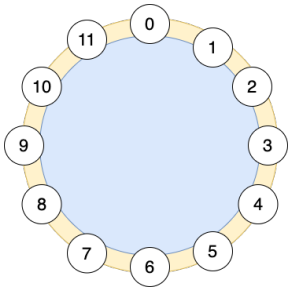
Try out a few different ideas, but feel free to move on when you feel your agent is performing as well as it can, given the time constraints imposed by the lab.

4 The Lemonade Stand Game

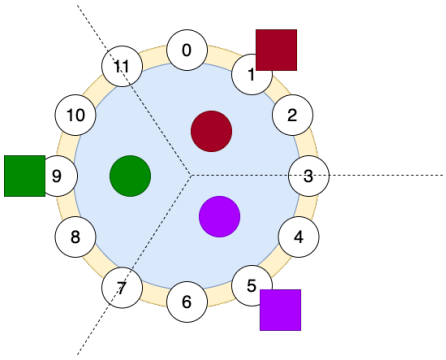
The Lemonade Stand Game is a three-player repeated game, in which the players are lemonade stand proprietors trying to maximize their revenue (i.e., sell as much lemonade as possible).

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, 24 beach go-ers spread themselves out evenly on the beach, two between 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. A player's payoff equals the total number of cups of lemonade it sells. (When two or more players choose the same location, the proceeds are split evenly among all players at that location.)

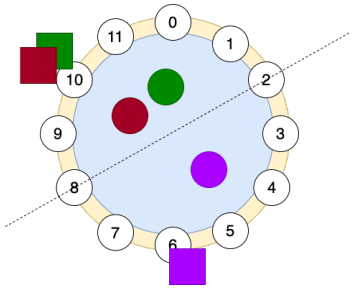
The game, along with several examples, is illustrated below.



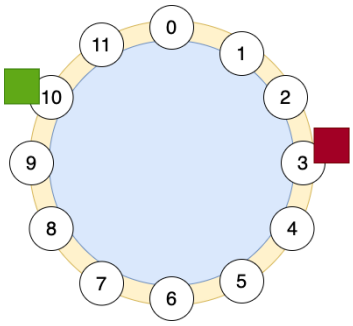
The setup for the Lemonade Stand Game. Players submit actions numbered between 0 and 11 to indicate their chosen location.



A possible outcome in the Lemonade Stand Game. The players are evenly spaced along the lake, so all earn payoffs of 8.



Another possible outcome in the Lemonade Stand Game. The Green and Red players both choose 10, while the Purple player chooses 6. Together, the Red and Green players earn 12, which they split evenly, while the Purple player earns 12.



Question: Imagine you are the third player in the Lemonade Stand Game. Your opponents' moves are pictured above. What is your best response? Is there a unique answer? What is your best possible payoff?

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![4]

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]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.

5 Q-Learning in the Lemonade Stand Game

Navigate to `LastMoveLemonadeQLearner.java`. This agent is analogous to `LastMoveChickenQLearner`—it only takes into account each opponent's previous move. Since you have two opponents in this game, each with 12 possible moves, the size of this state space is 144. If you were to incorporate your own move as well, this size would increase to $12^3 = 1728$. Thus, even simple state-space representations in the Lemonade Stand Game are *far* bigger than their Chicken counterparts. Hence, Q-learning in this game, is destined to be much slower than in Chicken, perhaps too slow for adequate learning in a lab.

Run this agent. It plays a 10^5-round simulation against two pretty simple agents, but it doesn't manage to learn very much. This is because the state space is large, and at the same time, it contains relatively little information about the opponents' strategies (only one data point each). Since the Lemonade Stand Game is more complex than Chicken, strategies can likewise be far more complex, in which case storing only an opponent's last move does not even begin to approach the amount of information necessary to learn an opponent's strategy.

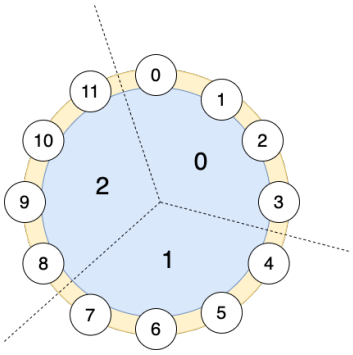
How can we improve upon the amount of information we represent while avoiding a state-space explosion? We will explore possible solutions to this conundrum in the remainder of this lab.

5.1 Competition: Implementing your Q-Learning Lemonade Agent

Navigate to `MyLemonadeQLearner.java`. Your task in the Lemonade Stand Game is exactly as it was in Chicken—to edit the `NUM_POSSIBLE_STATES` constant, and fill in `determineState`. However, things are much more complicated now, as in order for your agent to learn effectively, you will need to balance the need for a state space that stores enough information to be sufficiently expressive with the need for a state space that is small enough to be learned effectively.

Here are some suggestions. Feel free to use these and/or to invent your own:

- **Dividing up the board:** It may not be feasible to represent the opponents' exact positions, of which there are 12 possibilities, for more than a couple of past rounds. Even just two rounds would require $12^4 = 20736$ states! But what if instead of recording the opponents' *exact* locations, you only record the *section* of the board in which they played. You could divide the board into sections in many ways—as halves, thirds, quarters, or sixths. Using thirds as an example, and again recording the opponents' moves during the past two rounds, now only requires $3^4 = 81$ states: a vast improvement! Modeling only your opponents' general locations may be sufficient to learn to play a nearly-best response.



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 move 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' last moves.

Try out a few different state-space representations.

5.1.1 Pre-training your Agent

Run `MyLemonadeQLearner.java` to train against a mystery agent for 10^5 rounds.

You will learn your payoff at the end of the training, which will tell you how your agent did against the mystery agent. Additionally, your Q-table will be saved to a file, to be used in the competition.

All of your classmates will do the same.

5.1.2 Competing with your Pre-trained Agent

Navigate to `CompetitionLemonadeQAgent.java`. To enter the competition, please await the TA instructions. You will be instructed to edit the `HOST` and `NAME` variables, just like in previous labs.

Finally, run your pre-trained agent. Three copies of your agent will be launched to play in three matches, each one a repeated Lemonade Stand Game that lasts 100 rounds against two other Q-learning agents developed by your classmates. Each agent will determine the state of the game based on its own state-space representation, and will look up which move to make in its Q-table, given the current state.

Let the most robust state-space representation win!