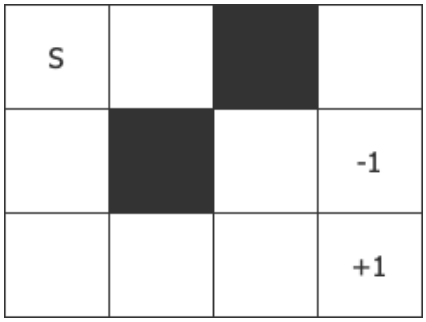
CS 486/686 Assignment 3
Winter 2023

1 Markov Decision Process & Reinforcement Learning (60 marks)

You will explore a set of grid worlds using (asynchronous) value iteration algorithm and passive adaptive dynamic programming (ADP).

1.1 Grid World environment

A simple 3x4 grid world



As a testing environment, you are provided an implementation of the 3x4 grid world often used as an example in lectures. The start state is at `(0,0)` and the goal states are at `(2,3)` and `(1,3)`. A wall barrier within the grid world is situated at `(0,2)` and `(1,1)`.

The are four available actions that an agent can take: up, right, down, and left. The actions are implemented following a clockwise direction such that `Action.UP = 0`, `Action.RIGHT = 1`, `Action.DOWN = 2`, and `Action.LEFT = 3`.

The agent moves in the intended direction with probability 0.8 (`prob_direct`), to its left with probability 0.1, and to its right with probability 0.1 (`prob_lateral`). The agent will stay in the same grid if it moves in a given direction and hits a wall. For any goal states s_g, for all states s', and all actions a, $P(s'|s_g, a) = 0$.

The immediate reward for entering any state that is not a goal state is -0.04. When the agent enters the goal state at `(2,3)` it will receive a `+1` reward; however, if it enters `(1, 3)`, it will receive a `-1` reward. For this assignment, rewards will be implemented as a 2D array and should be created following the example below.

```
rewards = np.array([
    [-0.04, -0.04, -0.04, -0.04],
    [-0.04, -0.04, -0.04, -1.00],
```

```
        [-0.04, -0.04, -0.04, 1.000]
    ])
```

We have provided an example of how to define a grid world environment in `main.py`.

Function(s) specific to the Grid World environment that you need to implement are explained below.

- `fill_T(self)`: initializes and populates the transition probabilities for all possible (s, a, s') combinations.

Unseen grid world environment

Your value iteration algorithm and passive ADP implementations will be tested on a set of grid world environments for marking. These would not be made available to you; however, you will be provided 2 grid worlds of varying dimensions and setup, in addition to the 3x4 grid world to test some edge cases. You are encouraged to create your own grid worlds for personal testing use.

The unseen grid worlds will have different dimensions and at least one goal state. The agent can take the same four available action at each state. If it moves in a direction and hits a wall, the agent will stay in the same grid. The transition probabilities (`prob_direct` and `prob_lateral` from `GridWorld.py`) will vary in the unseen grid worlds.

Note: It's important to keep `np.random.seed(486)` in `main.py` to ensure results are reproducible for functions that depend on psuedo-random `numpy` functions namely `make_move`, `simulate`, and `passive_ADP`. `np.random.seed(486)` must be run first before executing any of these functions.

1.2 Value Iteration Algorithm

In value iteration, the agent finds the optimal state values $V(s)$ (i.e., the expected utility of the optimal policies starting from state s) by iteratively updating its estimates after which, it derives an optimal policy based said estimates.

Let's first define some variables you will need in your implementation:

- γ: discount factor $\in (0, 1]$; default = `0.99`
- *R(s)*: the immediate reward for entering state s.

- *V(s)*: state values with initial values set to `0`.
- $P(s'|s,a)$: probability of transitioning to state *s'* if the agent executes action a at state s.

Recall the algorithm to solving for $V(s)$ iteratively from Lecture 14:

1. Start with arbitrary initial values for $V_0(s)$. For this assignment, $V(s)$ are initialized to 0.
2. At the i^{th} iteration, compute $V_{i+1}(s)$ as follows:

$$V_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s')$$

   Note that the right-hand side uses values from the previous iteration.
3. Terminate when $\max_s |V_i(s) - V_{i+1}(s)|$ is small enough. For this assignment, the default `tolerance` is 0.001.

Functions specific to this algorithm that you need to implement are listed below. Please refer to the docstrings in `Agent.py` for more details.

- `value_iteration(self, gamma, tolerance, max_iter)`: performs value iteration on the loaded environment following the steps above then returns the state values and number of iterations.
- `find_policy(self, V)`: finds the best action to take at each state based on the state values obtained (i.e., computing $\pi(s) = \arg\max_a \sum_{s'} P(s'|s,a)V(s')$).

1.3 Passive Adaptive Dynamic Programming (ADP)

In Passive ADP, the goal of the agent is to learn the expected value of following a fixed policy π i.e., $V^\pi(s)$, without knowledge of the transition probabilities or the reward function.

Recall the Passive ADP algorithm from Lecture 15:

1. Repeat steps 2 to 5.
2. Follow policy π and generate an experience $\langle s, a, s', r' \rangle$.
3. Update the reward function: $R(s') \leftarrow r'$.

4. Update the transition probability.

$$N(s,a) = N(s,a) + 1$$
$$N(s,a,s') = N(s,a,s') + 1$$
$$P(s'|s,a) = N(s,a,s')/N(s,a)$$

5. Derive $V^{\pi}(s)$ by using the Bellman equations.

$$V(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi(s))V(s')$$

Functions specific to this algorithm that you need to implement are listed below. Please refer to the docstrings in `Agent.py` for more details.

- `make_move(self, state, action)`: performs a single step in the environment under stochasticity based on the probability distribution and returns a tuple containing coordinates of s' and immediate reward of entering s'.
- `passive_adp(self, policy, gamma, adp_iters)`: performs passive ADP on a given policy using simulations from the environment and returns the state value array obtained by following given policy and a list containing the state values from every iteration.

Note: You are not allowed to use the true transition probabilities from the environment. Your implementation must learn an estimate of the transition probabilities through experiences from `GridWorld.simulate`.