Principles of Online Decision-Making (CS-303)

Problem Set 6 - Solutions

Problem 1

We will implement a taxi driver agent on Gymnasium Taxi environment. You first need to install the Gymnasium library if you haven't done so:

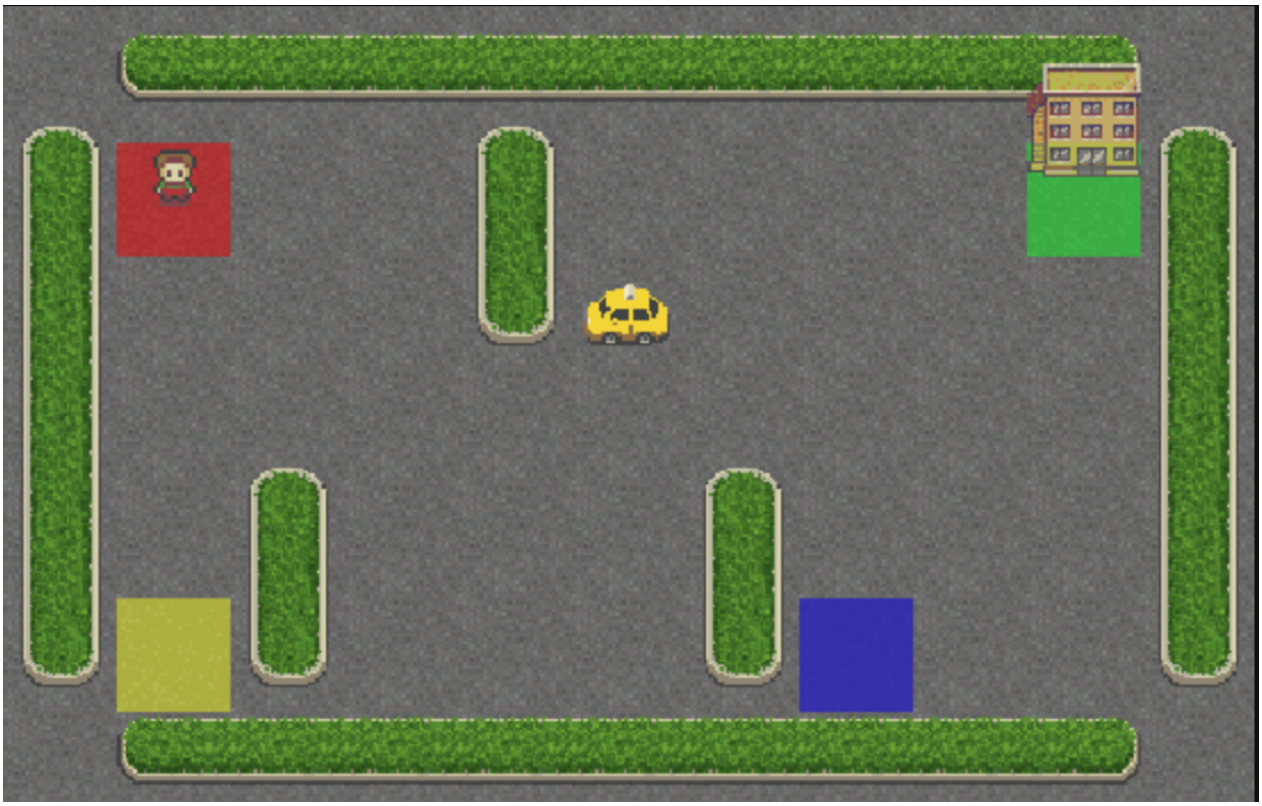
```
pip install "gymnasium[all]"
```

And tqdm library for progress bar:

```
conda install tqdm
```

Installing gymnasium can cause some issues depending on your environment. If you encounter the error with box2d-py, this webpage might help: `https://github.com/openai/spinningup/issues/32`.

You are given the following map of the environment: The taxi starts at a random location (one of



the empty squares), and there are four designated locations assigned by four color (R, G, Y, B). At the start of each episode, a passenger appears at one of these locations and has a destination of another location (also one of these four). The taxi driver agent must pick up the passenger and drop them off at the destination. The agent can take the following six actions:

- Move south
- Move north

- Move east
- Move west
- Pick up the passenger
- Drop off the passenger

The agent receives a reward of -1 for each time step, $+20$ for a successful drop-off, and -10 for an illegal pick-up or drop-off. The state is given by the taxi's row and column, the passenger's location, and the destination location: (taxi_row, taxi_col, passenger_loc, destination_loc) (More precisely, this tuple is converted to an integer by 4 (5 (5 taxi_row + taxi_col) + passenger_loc) + destination_loc). The origin of the taxi's row and column is at the top-left corner, and both row and column indices start from 0. The passenger location passenger_loc and destination location destination_loc are encoded as follows:

- 0: Red
- 1: Green
- 2: Yellow
- 3: Blue
- 4: In the taxi (only for passenger location).

There are 500 discrete states in total. Check more detail on the webpage.

In this homework, you will implement a taxi driver agent using policy iteration and value iteration algorithms assuming that the environment dynamics (transition probabilities and rewards) are known. In the next homework, you will implement an algorithms that learns from experience without knowing the environment dynamics.

(a) To implement these algorithms, first we need to specify the MDP corresponding to this environment (*i.e.,* $p(s', r|s, a)$). Since there are 500 possible states and there are six actions from each state, it is very tedious to specify $p(s', r|s, a)$ by hand. Leveraging the fact that the transition happens deterministic given the pair of the current state and action (s, a), we only need to specify the next state and reward for each (s, a) pair. To do this, we can categorize the (s, a) pair into several sets, and express transition and reward for each set. Specifically, let S be the set of all possible (s, a) pairs, and consider the following sets of (s, a) pair ($S_x \subsetneq S$):

- S_p: (s, a) pair that successfully picks up the passenger
- S_s: (s, a) pair that successfully drops off the passenger to the destination
- S_d: (s, a) pair that drops off the passenger to a colored tile but not to the destination
- S_w: (s, a) pair that makes the taxi to hit the walls.

Assume that the state s is given by the list [(taxi_row, taxi_col), passenger_loc, destination_loc], and action a is given by the list [(taxi_move_row, taxi_move_col), j], where $j \in \{-1, 0, 1\}$ and $j = 1$ is pick up, and $j = -1$ is drop off actions. Furthermore, let $f((x, y))$ be the function that returns the color of the tile on the coordinate (x, y): 0 (Red), 1 (Green), 2 (Yellow), and 3 (Blue). If the tile of the input coordinate is not colored, f returns -1. You can assume that the S_w is given.

First, give the explicit definition of each set except S_w. (*i.e.,* express the condition that a given (s, a) pair belongs to each set using s and a. *e.g,* $S_p = \{(s, a) \in S : f(s[0]) = s[1], a[1] = 1\}$.)

Then, express the transition and reward for all $(s, a) \in S$ pair using these sets.

Hint: you might want to consider other sets than S_p, S_s, S_d, S_w.

- $S_p = \{(s, a) \in S : f(s[0]) = s[1], a[1] = 1\}$

- $S_s = \{(s,a) \in S : s[1] = 4, f(s[0]) = s[2], a[1] = -1\}$
- $S_d = \{(s,a) \in S : s[1] = 4, f(s[0]) \in \{0,1,2,3\} \backslash \{s[2]\}, a[1] = -1\}$

We also consider the following sets/events:

- S_{ip}: (s,a) pair that tries illegal pick up (*i.e.,* try to pick up when there's no passenger on the tile.)
- S_{id}: (s,a) pair that illegal drop off(*i.e.,* try to drop off when there is no passenger or drop off to not-colored tile.)
- S_m: (s,a) pair that moves the taxi without hitting any wall

These sets can be defined as follows:

- $S_{ip} = \{(s,a) \in S \backslash S_p : a[1] = 1\} = \{(s,a) \in S : f(s[0]) \neq s[1], a[1] = 1\}$
- $S_{id} = \{(s,a) \in S \backslash S_d : a[1] = -1\} = \{(s,a) \in S : s[1] = 4, f(s[0]) \notin \{0,1,2,3\}, a[1] = -1\}$
- $S_m = \{(s,a) \in S \backslash S_w :, a[0] \in \{(1,0),(0,1),(-1,0),(0,-1),\}\}$

Now, the transition and reward for each set is expressed as follows:

$$(s', r) = \begin{cases} ([s[0], 4, s[2]], -1) & \text{if } (s,a) \in S_p \\ (s, -10) & \text{if } (s,a) \in S_{ip} \\ ([s[0], f(s[0]), s[2]], 10) & \text{if } (s,a) \in S_s \\ ([s[0], f(s[0]), s[2]], -1) & \text{if } (s,a) \in S_d \\ (s, -10) & \text{if } (s,a) \in S_{id} \\ ([s[0] + a[0], s[1], s[2]], -1) & \text{if } (s,a) \in S_m \\ (s, -1) & \text{if } (s,a) \in S_w \end{cases}$$

(b) The function transition_function in taxi_util.py computes the next state and the reward given the current state and the action. Fill out the blanks marked as #PODMexercis in the function transition_function.

(c) Fill out the blanks marked as #PODMexercis in the function value_iteration in value_iteration.py, and complete the function.

(d) Fill out the function policy_iteration in policy_iteration.py, and complete the function.

(e) Run the compute_policy.py to compute the optimal policy using your implementations of value iteration and policy iteration. Compare the convergence time. Which one converges faster, and reason why. You can also visualize the policies by running taxi_run.py with the computed policy.
