CS 4180/5180: Reinforcement Learning and Sequential Decision Making (Spring 2024) Lawson L.S. Wong
Northeastern University Due January 19, 2024

Exercise 0: An Invitation to Reinforcement Learning

Please remember the following policies:

- Exercises are due at **11:59 PM** Boston time (ET).
- **Submissions should be made electronically on Canvas.**
  Please ensure that your solutions for both the written and programming parts are present.
  You can upload multiple files in a single submission, or you can zip them into a single file.
  You can make as many submissions as you wish, but only the latest one will be considered, and late days will be computed based on the latest submission.
- Each exercise may be handed in up to two days late (24-hour period), penalized by 5% per day.
  Submissions later than this will not be accepted. There is no limit on the total number of late days used over the course of the semester.
- Written solutions may be handwritten or typeset. For the former, please ensure handwriting is legible. If you write your answers on paper and submit images of them, that is fine, but please put and order them correctly in a single .pdf file. One way to do this is putting them in a Word document and saving as a PDF file.
- You are welcome to discuss these problems with other students in the class, but you must understand and write up the solution **and code** yourself, *and* indicate who you discussed with (if any). If you are collaborating with a large language model (LLM), acknowledge this and include your entire interaction with this system. We strongly encourage you to formulate your own answers first before consulting other students / LLMs.
- Contact the teaching staff if there are *extenuating* circumstances.

The purpose of this assignment is to get used to writing your own reinforcement learning environments and agents/policies, perform computational experiments with them, and analyze the results you get. This skill will be frequently used and reinforced throughout the course. ~~As such, no starter code is provided.~~ Some starter code is provided, which you may build on, get inspiration from, or completely ignore (implementing from scratch is recommended if you are sufficiently comfortable with Python and the RL material).

Implement your solution in Python 3. Include brief comments about the various functions/components, and explain how to run your code. Some questions will require written answers as well and ask you to show plots. Instead of inlining your written answers into the code and attaching images of plots, we recommend writing a separate PDF document that contains both written answers and plots for questions that require them.

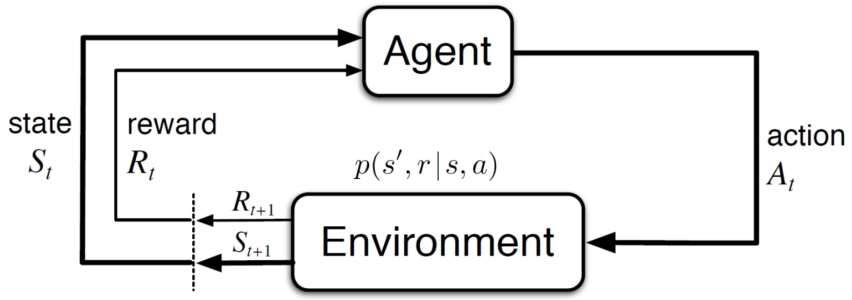
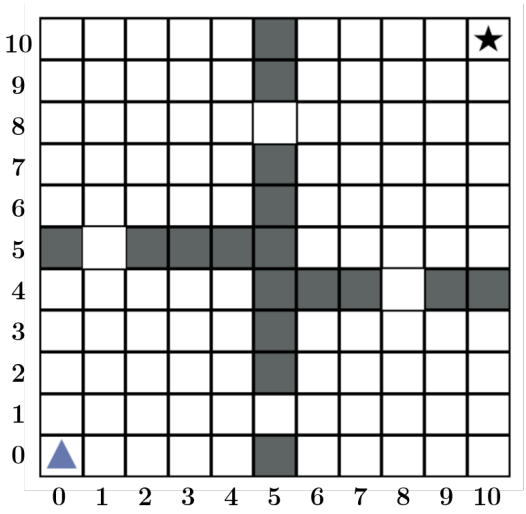


Figure 3.1: The agent–environment interaction in a Markov decision process.

The above figure illustrates the interface between a reinforcement learning agent and its environment. In this assignment, you will implement all of the above for the Four Rooms domain shown in lecture. You will first implement a simulator (environment), then design and implement several simple agents to interact with the simulator.



1. **1 point.** **Code:** Implement a `simulate` function that acts as the Four Rooms environment.
   It should take in the current state s and desired action a, and return the next state s' as well as the reward r.
   The specification of the domain is:

   - The state consists of integer (x, y) coordinates, where $x \in [0, 10]$ and $y \in [0, 10]$.
     The shaded cells are walls and the agent should never be in those cells.
   - The feasible actions are `left`, `down`, `right`, `up`. All four actions may be taken at any valid state.
   - To determine the effect of taking an action in a given state:
     - The agent typically moves in the specified direction
       (i.e., taking `right` in $(2, 3)$ typically moves the agent to $(3, 3)$).
     - However, there is occasional noise in the domain (e.g., the agent slips).
       With probability 0.8, the agent moves in the correct direction. With probability 0.1 each, it moves in one of the two "perpendicular" directions.
       For example, taking `right` in $(2, 3)$ will take it to $(3, 3)$ with probability 0.8, to $(2, 4)$ with probability 0.1 (the effect of the `up` action), and to $(2, 2)$ with probability 0.1 (the effect of the `down` action).
     - If the result of the noisy action causes it to collide into a wall, or take the agent out of bounds, then the state does not change.
   - The agent starts in $(0, 0)$ (triangle in figure), and the goal state (for now) is $(10, 10)$ (star in figure).
   - Taking any action from the goal state $(10, 10)$ teleports the agent to the start state $(0, 0)$ (i.e., reset).
   - If the resulting state is the goal state, the reward returned is $+1$; otherwise, it is 0.
   - Check your understanding: Moving $a =$ `up` from $s = (0, 10)$ will result in $s' = (0, 10)$ with probability 0.9 and $s' = (1, 10)$ with probability 0.1, and will return a reward of $r = 0$. In other words, calling `simulate((0,10), 'up')` should return `(0,10), 0` 90% of the time, and `(1,10), 0` 10% of the time.

2. **1 point.** **Code:** Implement a `manual` policy, and an `agent` that interacts with the simulator and the policy.

   - The *policy* is a state-action mapping $\pi : S \to A$, i.e., a function that produces an action given a state as input. Here, implement a policy that queries you to input an appropriate action for the given state.
   - The *agent*, as shown in the diagram on the previous page, is the entity that provides actions to the environment (simulator), and receives next states and rewards from the environment. Here, the agent should be in charge of getting the current state s, querying the policy for an appropriate action $a = \pi(s)$, calling the simulator with (s, a) to get (s', r), and repeating.

   Run the agent with you providing the action on each step. Check that the simulator working as expected, and that you able to get to the goal reliably.

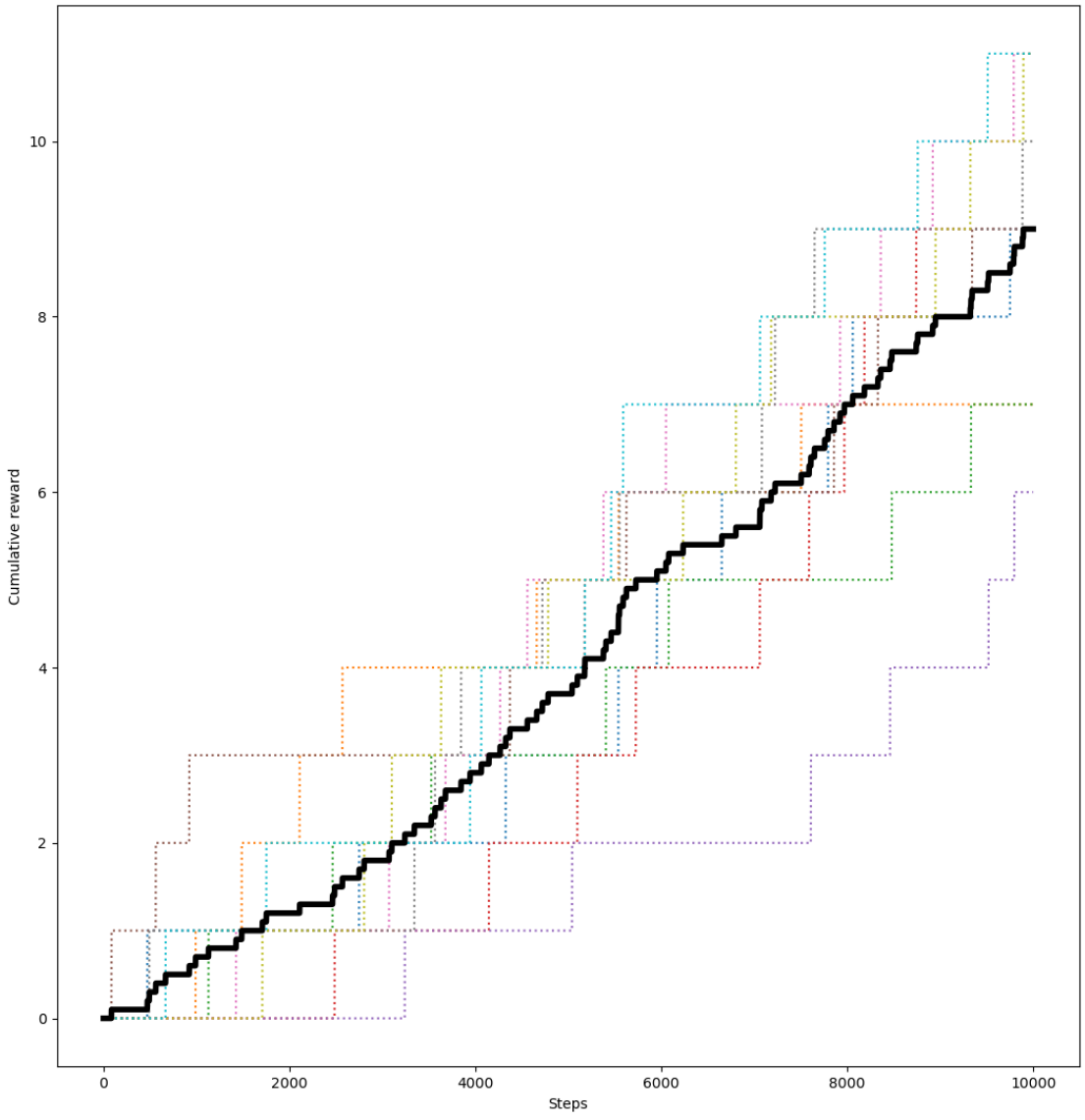
   You will probably get bored of interacting with the agent rather quickly! Let's try to automate this.

3. **1 point.** **Code:** Implement a `random` policy. A random policy outputs one of the four actions, uniformly at random. This is an example of a stochastic policy.

   How well does this policy perform? In reinforcement learning, the objective is to maximize the expected sum of rewards, i.e., a policy is "better" if, on average, it leads to greater cumulative reward. Since the environment and policy in this case are both stochastic, the obtained rewards (and the number of steps to obtain them) will be different every time we run them (unless we take care to use the same seed for pseudorandom number generation – a very good tool for debugging and controlling experiments). To get a good overall sense of performance, we should run our agents for many steps, over many trials.

   Plot: Run your random-policy agent for a total of 10 trials, 10^4 steps in each trial, and produce a cumulative reward plot similar to the one shown below. The dotted lines are the cumulative reward curves for each of 10 trials, and the thick solid black line is the average of those 10 dotted curves.
   We plotted this using `matplotlib.pyplot`.

   Written: How do you think this compares with your manual policy? (You do not have to run your manual policy for 10^4 steps!) What are some reasons for the difference in performance?



4. **1 point.** **Code:** Devise and implement at least two more policies, one of which should be generally worse than the random policy, and one better.
   Written: Describe the strategy each policy uses, and why that leads to generally worse/better performance.
   Plot: Show the cumulative reward curves of each, similar to Q3.

   You may use any algorithms you know about to do this, including reinforcement learning algorithms, but this should not be necessary.

5. **1 point.** So far, depending on what policies you implemented for Q4, you may or may not have used information from the reward signal (notice that the agent has access to this, given by the environment). Using the reward signal was unnecessary because you already knew that the goal state is $(10, 10)$. Assuming that your agent does not use the reward signal, the agent cannot be considered to be *learning to act*.

   Code: Devise and implement a learning agent. To facilitate this, first modify your code to select a *random* goal state before any trial, instead of using the fixed goal state $(10, 10)$. The goal state should remain fixed throughout all trials. Your agent should *learn* where to find high reward and use this knowledge to act better in the future. Make sure your agent does not "cheat" by seeing what the goal state is – only the simulator can access this.

   To assess that learning has enabled better acting, you can keep track of the average number of steps it takes to find the goal the first time in a trial, and the average number of steps it takes to get from start to goal after that. If your learning agent is effective, it should take significantly fewer steps to get from start to goal after the first time the goal has been discovered.

   Written: Describe the strategy of your learning agent and the policy it uses, and explain how it has learned to act better.
   Plot: Show the cumulative reward curve, similar to Q3.

   The following curves were generated for the randomly selected goal state $(7, 7)$. The policies you devise can behave differently (the random policy should give similar results).