EXERCISE 1
The finite-horizon decision problem

Tue Herlau
tuhe@dtu.dk

4 February, 2022

Objective: In this exercise, we will familiarize ourselves with the finite-horizon decision problem (the basic problem) as well as the agent-class and openais environment class, both of which we will use in the remaining exercises. (40 lines of code)
Material: Obtain exercise material from our gitlab repository at
`https://gitlab.gbar.dtu.dk/02465material/02465students`

Contents

1 Pacman and your first agent (`pacman_hardcoded.py`) 1

2 Inventory control environment (`inventory_environment.py`) 2

3 Frozen lake and openai Gym (`frozen_lake.py` 4
3.0.1 A slightly smarter policy 4
3.0.2 The states 5
3.0.3 Obtaining trajectories 5

4 The chess tournament (`chess.py`)★ 6

1 Pacman and your first agent (`pacman_hardcoded.py`)

In this problem, we will program an agent that helps pacman eat all the dots in the small maze shown in fig. 1. The agent you write should implement a single method, `pi(x,k)`, which should return an action.

The code contains hints in the comments, including how you can figure out what the available actions are. I recommend inserting a breakpoint in the code and trying to work out the actions using the command line. You can see more information in the videos on how to this on DTU learn.

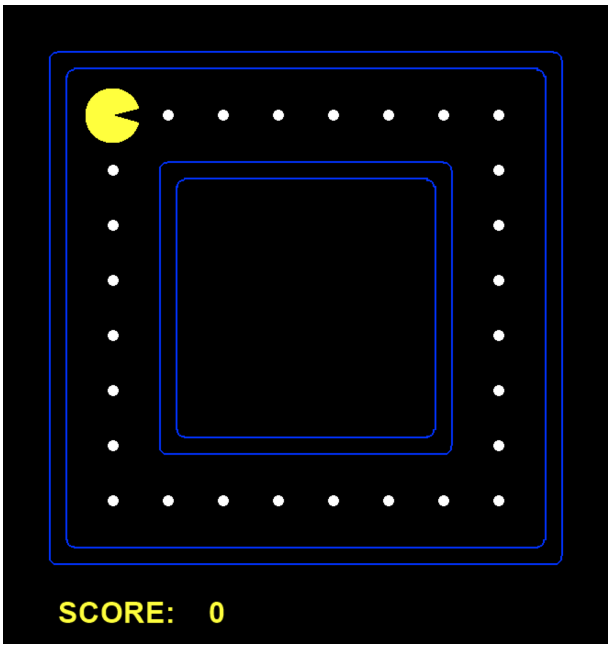


Figure 1: A simple pacman maze

Problem 1 *Deterministic pacman*

Implement the missing policy function using the hints in the code. Verify pacman eat all the food pellets by looking at the visualization. I recommend looking at the example carefully before moving on – this course will contain a lot of code similar to this.

2 Inventory control environment (`inventory_environment.py`)

This example will illustrate the various parts of the world-loop in greater details. Note most of the code you have to write is already given in the slides/lecture notes (See [Her21, section 4.4.2]).

Problem 2 *Look at the Agent class*

Open the file `ex01/agent.py`. The file contains the `Agent` class and the code for the `train` function. Read the Agent class and discuss the policy/training function. The input to the training function corresponds to x_k, u_k and x_{k+1}. Do you think we could omit x_{k+1} from the input-arguments and just let the agent experience x_{k+1} at the next step and then perform the training update?

After you are done, scroll through the training function `train` and see if you can locate where it implements the world loop.

Info: The training function is messy because it supports a number of special cases. Don't worry about understanding the input parameters at this point; we will see examples throughout the course.

The next example will illustrate how Agents can interact with the environment class.

Problem 3 *Inventory environment*

Implement the missing code for the inventory environment and ensure it works (code for the environment can be found in [Her21, section 4.4.2] and the following sections), then review the output and compare against the description in the notes.

Info: The first part of the problem should work and produce the output:

```
Accumulated reward of first episode -2
[RandomAgent class] Average cost of random policy J_pi_random(0)= 4.168
[Agent class] Average cost of random policy J_pi_random(0)= 4.271
```

Note that once the environment is defined, the last example requires no extra functionality to run.

Although this example illustrates the *proper* way to interact with an environment using the `train` function, I think it is worth to re-produce the result without using api calls. We will therefore implement our own training function:

Problem 4 *Do-it-yourself training function*

Implement the missing functionality for the simplified training function. The function computes a single rollout of the agent's policy and return the total cost of the rollout. If you are stuck, note that the missing code can be found in the lecture notes.

Info: The estimate based on your simplified training function should match that of the real training function: The first part of the problem should work and produce the output:

```
[simplified train] Average cost of random policy J_pi_random(0) = 4.308
```

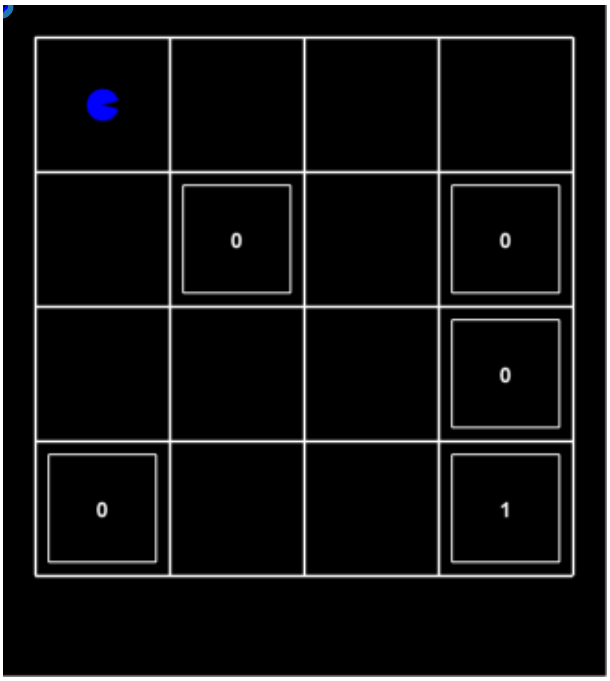


Figure 2: frozen lake layout. 0 means holes in the ice (terminate with reward 0). The goal gives a reward of 1. Whats special about the environment is that the ice is slippery: If the agent e.g. takes the action up, it will move either up, left and right (but not down) with equal probability. This makes it hard to avoid the holes in the ice!

3 Frozen lake and openai Gym (`frozen_lake.py`

To familiarize ourselves with the environment class, we will consider the frozen-lake environment from openai Gym (see fig. 2). This is a stochastic gridworld environment, but follows the same convention as [Her21, section 4.4.2]. For specific details about what the environment does, see the links:

- `https://github.com/openai/gym/blob/master/gym/envs/toy_text/frozen_lake.py`

3.0.1 A slightly smarter policy

Recall the Agent class defaults to a random policy, i.e. one which randomly goes left, right, up and down. We can improve upon it slightly be implementing an agent with a different move pattern:

Problem 5 *A smarter policy*

Implement an agent which moves down if k is even and otherwise right and evaluate it.

Info: The output should be:

```
Average reward of random policy 0.0205
Average reward of down_right policy 0.039
```

3.0.2 The states

The states are stored linearly as integers in the frozen lake environment. Write a function which invert the linear index of states to the row/column format used in the constructor of the frozen lake environment.

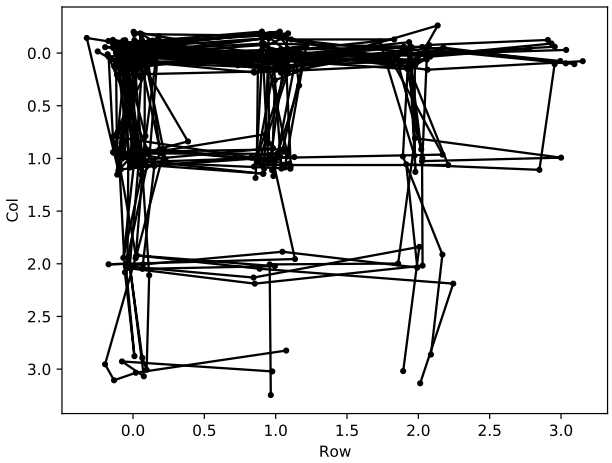
Problem 6 *State indexing*

Complete the function body of `def to_s`.

Info: The code contains a check for the state $s = 5$, which should output the same result.

3.0.3 Obtaining trajectories

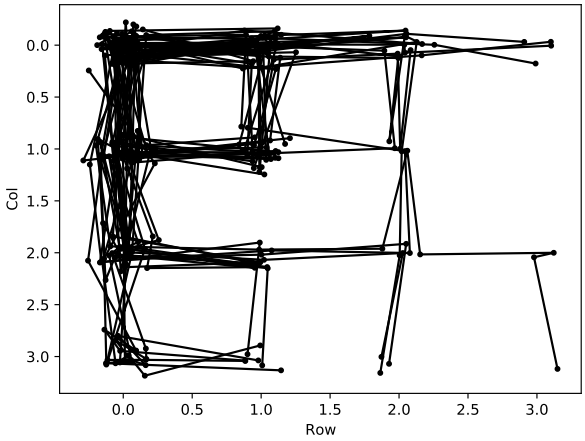
Once done, the next part of the code showcase how we can obtain information from the individual trajectories/rollouts using an api call. The trajectories from the Random agent is given below:



Problem 7 *Displaying trajectories*

Generate and plot 50 trajectories from the down-right agent as (x, y) lines similar to the figure above.

Info: The previous figure should provide some pretty big hints. At any rate, the code should produce something akin to the following result:



There is not a lot of difference compared to the random policy, which is to be expected considering they have nearly identical value functions.

4 The chess tournament (`chess.py`)★

This problem is inspired by the following youtube video: `https://www.youtube.com/watch?v=5UQU1oBpAic`. The problem goes as follows:

- You are playing a chess match against a skilled opponent. For a given game, there is a 75% chance the game will end in a draw
- Of the games that do not end in a draw, there is a $2/3$ chance you will win, and a $1/3$ chance you will lose.
- The first player to win 2 games in a row is declared the winner of the chess match. What is the probability you will win the match?

The video describes how the problem can be solved using basic probability theory, however we will do something much more fun and solve the problem by converting it to an Environment and then use the course software.

To review, we will use the general Monte-Carlo formulate:

$$\mu = \mathbb{E}[h(x)] = \sum_x p(x)h(x)$$

(see [Her21, section X.1]). But in our case, x corresponds to a chess match (i.e., a list of outcomes of the various games), for instance

$$x = (0, -1, 1, 0, 0, 1, 1)$$

corresponds to a draw, a loss, a win, two losses and two wins.

In this case, h should be a function which takes a chess match and determines who won. Note it can be a bit difficult to specify $p(x)$ because the matches can have different

games, however we can quite easily *simulate* from p and thereby approximate, using the MC principle:

$$\hat{\mu} \approx \frac{1}{T} \sum_{i=1}^{T} h(x_i)$$

where x_i is a random chess match generated one game at a time as described in the problem.

To implement this, we will consider the chess-match as an environment, where the state is a list of the form of x above, and in each step of the environment the outcome of a random game is generated (according to the description) and added to the end of x. The environment then checks if the match is over and return a reward of 1 if the player won and otherwise zero. If we use this procedure the average reward, computed over many episodes, will correspond to the average win rate.

Problem 8 *Chess match*

Implement the chessmatch environment according to the above description. What should the `reset` function do?. Note that the action will not be used.

As a bonus, we will also compute the average game length.

Info: The output should be approximately:

```
Agent: Estimated chance I won the tournament:  0.7944
Agent: Average tournament length 33.3922
```

References

[Her21] Tue Herlau. Sequential decision making. (See **02465_Notes.pdf**), 2021.