EXERCISE 1
The finite-horizon decision problem

Tue Herlau
tuhe@dtu.dk

7 February, 2025

Objective: In this exercise, we will familiarize ourselves with the finite-horizon decision problem, with an emphasis on how it is practically implemented using an agent and an environment. These will be the two recurring building-blocks in the rest of the exercises. (43 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex01.html

Contents

1 Bobs financially challenged friend

☞ Bob has $x_0 = 20$ kroner. He can either:

- Action $u = 0$: Put them in the bank at a 10% interest, thereby ending up with 22 kroner.
- Action $u = 1$: Lend them to a friend.
  - With probability $\frac{1}{4}$ he looses everything
  - With probability $\frac{3}{4}$ his friend gives him 12 kroner (aka one beer) as a thank you, and thus he will have $20 + 12 = 32$ kroner total.

We can consider this as a decision problem starting in the state x_0 and where Bob takes a single action.

(a.) Which action u is the better when Bob starts out with $x_0 = 20$ kroner?

(b.) Determine the policy $\mu_0(x_0)$ which maximize Bobs earnings

(c.) (Bonus ethics questions: Who is the worse friend)

2 Implementing the Bob-friend environment (`bobs_friend.py`)

Part A: The Bob-friend-environment Implement the Bob-friend decision problem described in section 1 above as an environment. To get you started, I have written the init-method:

```python
# bobs_friend.py
class BobFriendEnvironment(gymnasium.Env):
    def __init__(self, x0=20):
        self.x0 = x0
        self.action_space = Discrete(2)      # Possible actions {0, 1}
```

As you can see, the init-method stores the initial amount of money Bob has (i.e., the first state x_0), and the reward the environment should return under the two actions is the **change** to Bobs monetary amount. I.e., when $u = 0$ and $x_0 = 20$ the reward should be $r = 22 - x_0 = 2$.

Problem 1 *Bobs friend*

Implement `reset` and `step`-functions in the environment class. Note that the environment should work for any initial amount `x0`.

When you are done, you can run the tests to check your work. Since you have not solved all problems, you can expect some tests to work and some to fail. Those that will work should be labeled according to the problem, `BobsFriendTest`, and the sub-question, meaning those starting with `test_a_` should work. See 02465material.pages.compute.dtu.dk/02465pu for more information.

Part B: Bobs choice In this problem, we will implement two agents: One that always take action $u_0 = 0$, and one that always take action $u_0 = 1$. To implement the agents, you only need to change the policy function (i.e., `self.pi` in the code).

Problem 2 *Bobs policy*

- Complete the two agents `AlwaysAction_u0` and `AlwaysAction_u1`. Their policies should be very simple.
- The rest of the code simulate them for a large number of episodes and compute the average reward. Explain why you get these numbers using your calculations in section 1
- Run the rest of the tests associated with Bob. They should all work if your implementation is correct.

Once implemented, read through the code to see how we train them for a large number of episodes and compute the average reward. Does this conform to

3 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 [Her25, section 4.4.2]).

Problem 3 *Inventory environment*

Implement the missing code for the inventory environment and the `RandomAgent` (code for the environment can be found in [Her25, section 4.4.2] and the following sections). The random agent should return one of the possible actions selected at random. 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 -4
[RandomAgent class] Average cost of random policy J_pi_random(0)= 4.226
[Agent class] Average cost of random policy J_pi_random(0)= 4.28
```

If you are stuck on how to select a random action (i.e., a random number) you can perhaps use the `np.random.choice`-function, or `action_space.sample()` (see the online documentation).

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

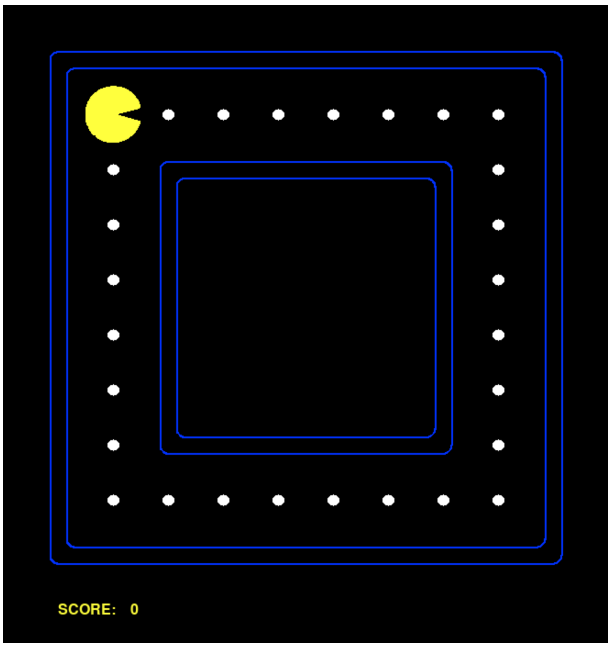

Figure 1: A simple pacman maze

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 reward of the rollout.

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) = 4.292
```

If you are stuck, note that the missing code can be found in the lecture notes.

4 Pacman and a simple agent (`pacman_hardcoded.py`)

In this problem, we will program an agent (pacman) that eats all the dots in the small maze shown in fig. 1. The agent you write should implement a single method, the policy `pi(x,k,info)`, which should return the action to take in state `x` at time step `k`. Remember that `info` can be ignored.

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 on how to work with breakpoints in the videos online.

Problem 5 *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.

Info:

- You should not write a general pacman-solver. Just make it work for this problem.
- You can verify the solution to this (and many of the other exercises) by running the test script found in the `irlc/tests` directory.

5 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.

The chance of winning the match can be written as:

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

(see [Her25, section X.1]) where 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. Although it is difficult to specify $p(x)$ because the matches can have different games,

we can quite easily *simulate* from p and thereby approximate, using the Monte-Carlo estimate:

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

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 6 *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:  0.7928
Agent: Average tournament length 34.1206
```

References

[Her25] Tue Herlau. Sequential decision making. (Freely available online), 2025.