EXERCISE 9
Policy and value iteration

Tue Herlau
tuhe@dtu.dk

4 April, 2025

Objective: Today we will consider methods for solving a finite MDP assuming the underlying probability structure is known. The methods we will consider will closely mirror DP as introduced in lecture 2, however since we are considered a discounted, infinite-horizon setting there will be important differences. Even though the setting is typically not realistic, the solution methods are worth taking note of as the methods we will encounter later can be seen as approximate solutions. (29 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex09.html

Contents

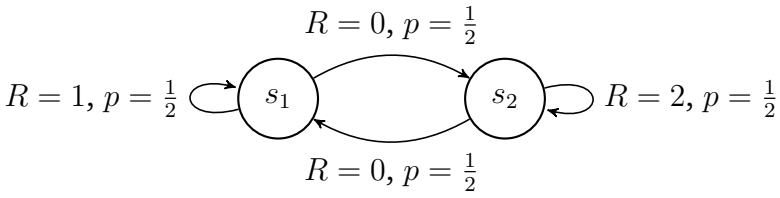


Figure 1: A simple MRP

1 Conceptual problems

- Solve **problem 3.14** from [SB18] (Hint: When you use [SB18, figure 3.2] as suggested only the right-hand pane is important)
- Solve **problem 4.1** from [SB18] (Hint: Remember to use [SB18, figure 4.1])

2 Exam problem: Bellmans equations

Consider a Markov Reward Process with two states s_1 and s_2 and a discount factor of $0 < \gamma < 1$ shown in fig. 1 [1]. With equal probability $\frac{1}{2}$, the system will either stay in the current state or transition to the other state. The MRP never terminates.

- When the system transition s_1 to s_2 (or s_2 to s_1) it will receive a reward of 0,
- If it transition from s_1 to s_1 it will receive a reward of 1,
- If it transition from s_2 to s_2 it will receive a reward of 2.

Since there are two states, the value function v_π takes two values $v_1, v_2 \in \mathbb{R}$ defined as: $v_1 = v_\pi(s_1)$ and $v_2 = v_\pi(s_2)$.

(a.) Assume for a moment that $v_2 = \frac{5}{2}$ and $\gamma = \frac{2}{3}$. What is v_1? (i.e. the value function in s_1, $v_\pi(s_1)$)

(b.) Ignore the previous question and consider the general form of the problem. As the name suggest, when Bellmans expectation equations are applied to this problem we obtain two equations. Write them as a linear system of the form $\boldsymbol{b} = A\boldsymbol{v}$ where $\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$. State what A and $\boldsymbol{b}$ are as functions of γ.

Bonus question: Determine v_1 by solving the equation in problem **b**.

[1] Recall a Markov Reward Process is a Markov decision process without actions, see [SB18, Example 6.2].

3 Preliminaries: The Markov Decision Process

The methods we consider this week will assume we have access to a model of the environment (the MDP) and they will then use this model for planning. A MDP consist of four elements:

- An initial state[2]
- A function to determine if a state is terminal (and therefore that the environment needs to stop)
- A function to compute the transition probability $p(s', r|s, a)$
- A function to compute actions in a state $\mathcal{A}(s)$

In python, these functions are defined by implementing the following class:

```python
# mdp.py
class MDP:
    def __init__(self, initial_state=None, verbose=False):
        self.verbose=verbose
        self.initial_state = initial_state  # Starting state s_0 of the MDP.
def is_terminal(self, state) -> bool:
    return False # Return true if the given state is terminal.

def Psr(self, state, action) -> dict:
    raise NotImplementedError("Return state distribution as a dictionary (see class documentation)")

def A(self, state) -> list:
    raise NotImplementedError("Return set/list of actions in given state A(s) = {a1, a2, ...}")
```

For instance, this code implements the $4x4$ gridworld example from [SB18, Example 4.1], which we will use in some of the other exercises:

```python
# small_gridworld.py
UP,RIGHT, DOWN, LEFT = 0, 1, 2, 3
class SmallGridworldMDP(MDP):
    def __init__(self, rows=4, cols=4):
        self.rows, self.cols = rows, cols # Number of rows, columns.
        super().__init__(initial_state=(rows//2, cols//2) ) # Initial state is in the middle of the
        ↪  board.

    def A(self, state):
        return [UP, DOWN, RIGHT, LEFT] # All four directions available.

    def Psr(self, state, action):
        row, col = state # state is in the format  state = (row, col)
        if action == UP:    row -= 1
        if action == DOWN:  row += 1
        if action == LEFT:  col += 1
        if action == RIGHT: col -= 1

        col = min(self.cols-1, max(col, 0)) # Check boundary conditions.
```

[2]Or more generally a distribution over the initial states. This is not relevant for Todays exercises.

```
        row = min(self.rows-1, max(row, 0))
        reward = -1  # Always get a reward of -1
        next_state = (row, col)
        # Note that P(next_state, reward | state, action) = 1 because environment is deterministic
        return {(next_state, reward): 1}

    def is_terminal(self, state):
        row, col = state
        return (row == 0 and col == 0) or (row == self.rows-1 and col == self.cols-1)
```

The states are defined as tuples of integers (`state = (row, col)`), and the transition probabilities are 1 since the environment is deterministic.

4 Relationships between Q, V and the MDP (`mdp_warmup.py`)

The purpose of this exercise is to familiarize you with the MDP class (see discussion above and the online documentation) and how to compute expectations using python loops.

Expected reward

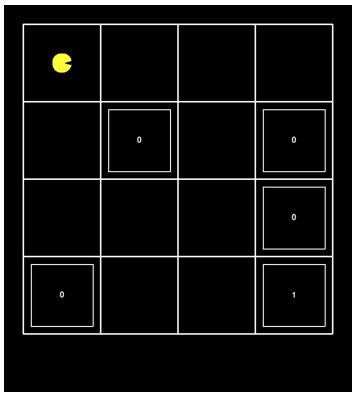
Problem 1 *Expected reward*

Implement a function which uses the MDP class to compute the expected reward in a state s given that we take a specific action a. In other words, it should compute:

$$\mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{s',r} p(s', r|s, a)r$$

Info: To solve this problem you must work with the function `mdp.Psr` of the MDP class. Here is an example of how to use it to print out all values of $p(s', r|s, a)$ in the starting state of the frozen-lake environment:

```
>>> from irlc.gridworld.gridworld_environments import FrozenLake
>>> mdp = FrozenLake().mdp # Get the MDP of the frozen lake environment
>>> s = mdp.initial_state
>>> s     # Initial state
(0, 3)
>>> a = 1 # Go east.
>>> for (s_, r), p in mdp.Psr(s, a).items():
...     print(f"Prob. of moving to state {s_=} and getting reward {r=} is p(s_, r | s, a)", p)
...
Prob. of moving to state s_=(1, 3) and getting reward r=0 is p(s_, r | s, a) 0.3333333333333337
Prob. of moving to state s_=(0, 3) and getting reward r=0 is p(s_, r | s, a) 0.3333333333333333
Prob. of moving to state s_=(0, 2) and getting reward r=0 is p(s_, r | s, a) 0.3333333333333333
```



When done, the code:

```
    # mdp_warmup.py
    V = {}
    for s in mdp.nonterminal_states:
        V[s] = s[0] + 2*s[1]
    print("Value function is", V)
    # Compute the corresponding Q(s,a)-values in state s0:
    q_ = value_function2q_function(mdp, s=s0, gamma=0.9, v=V)
    print(f"Q-values in {s0=} is", q_)
```

Should produce this output:

```
Value function is {(0, 1): 2, (1, 2): 5, (2, 1): 4, (0, 0): 0, (3, 1): 5, (1, 1): 3, (0, 3): 6, (2,
↪  0): 2, (3, 0): 3, (2, 3): 8, (0, 2): 4, (3, 3): 9, (2, 2): 6, (1, 0): 1, (3, 2): 7, (1, 3): 7}
Q-values in s0=(0, 3) is {0: 5.9, 1: 5.300000000000001, 2: 5.3, 3: 5.000000000000001}
```

Value function to Q-function Next, find the slide "*Fundamental properties of the value function*" from today. The two operations shown in the slide are used several times throughout [SB18], and we will therefore implement them here.

In the first problem we will focus on the update defined for any function V:

$$Q(s,a) = \mathbb{E}[R_{t+1} + \gamma V(S_{t+1})|s,a] = \sum_{s',r} p(s',r|s,a)(r + \gamma V(s'))$$

We will assume that s is known, and the function should therefore return a dictionary `Q` of the form `Q = {a0 : q0, a1: q1, ...}` so that `Q[a]` contains $Q(s,a)$.

Problem 2 *Value to action-value function*

Implement a function that transform the V function to the Q-function as described above.

Info: When done, the following code:

```
# mdp_warmup.py
V = {}
for s in mdp.nonterminal_states:
    V[s] = s[0] + 2*s[1]
print("Value function is", V)
# Compute the corresponding Q(s,a)-values in state s0:
q_ = value_function2q_function(mdp, s=s0, gamma=0.9, v=V)
print(f"Q-values in {s0=} is", q_)
```

Should produce this output:

```
Value function is {(0, 1): 2, (1, 2): 5, (2, 1): 4, (0, 0): 0, (3, 1): 5, (1, 1): 3, (0, 3): 6, (2,
↪  0): 2, (3, 0): 3, (2, 3): 8, (0, 2): 4, (3, 3): 9, (2, 2): 6, (1, 0): 1, (3, 2): 7, (1, 3): 7}
Q-values in s0=(0, 3) is {0: 5.9, 1: 5.300000000000001, 2: 5.3, 3: 5.000000000000001}
```

Q-function to value-function In our last example, we will convert a Q-function, represented as a dictionary so that `Q[s,a]` is the Q-value, to a value function. We assume that the state s is known. To do this we need a policy which we also represent as a dictionary. In other words, `pi[a]` is the probability $\pi(a|s)$.

Problem 3 *Action-value to value*

Implement a function that transform the Q function to the V-function defined as:

$$V(s) = \mathbb{E}_\pi[Q(s,a)] = \sum_{a \in \mathcal{A}(s)} \pi(a|s) Q(s,a).$$

Info: When done, the following code:

```
    # mdp_warmup.py
    Q = {}
    for s in mdp.nonterminal_states:
        for a in mdp.A(s):
            Q[s,a] = s[0] + 2*s[1] - 10*a # The particular values are not important in this example
    # Create a policy. In this case pi(a=3) = 0.4.
    pi = {0: 0.2,
          1: 0.2,
          2: 0.2,
          3: 0.4}
    print(f"Value-function in {s0=} is", q_function2value_function(pi, Q, s=s0))
```

Should produce this output:

```
Value-function in s0=(0, 3) is -12.000000000000002
```

5 Iterative Policy Evaluation (`policy_evaluation.py`)

Given a policy π, the value function v_π is a fundamental quantity in reinforcement learning as it provides an evaluation of how the policy perform in each state.

The policy `pi` will be implemented so that `pi[s][a]` is the probability that the policy selects a in state s (see the code for how the policy is initialized).

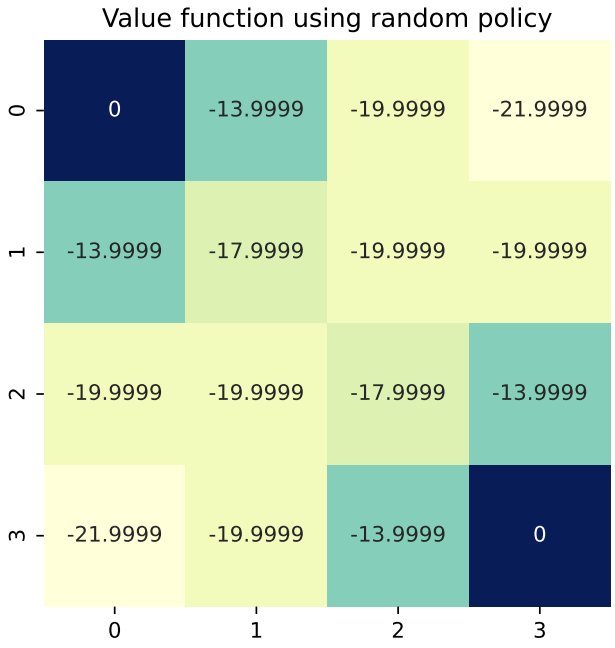
Problem 4 *Policy Evaluation*

The script contains a policy which just selects actions from a uniform distribution. Evaluate this policy, i.e. find v_π, from an environment where the full description of the environment's dynamics is available. To do that, implement the policy evaluation algorithm described in [SB18, Section 4.1]. We will use the asynchronous version which is described in the pseudo code.

Info: When you implement policy evaluation, note that the central update is equivalent to the function `value_function2q_function` you implemented in the previous problem. (see [SB18, Section 4.1] and the comments in the code). Recall that what the function does is evaluate the Q-values for a given state using value-function values:

$$Q(s,a) = \mathbb{E}[r + v_\pi(s')|s,a]$$

Once done, you can run the code and obtain a plot of the value function for a random policy as follows:



6 Policy Iteration (`policy_iteration.py`)

The policy improvement theorem tells us that if we know the action-value function q_π, we find a policy which is at least as good as π by being greedy with respect to q_π:

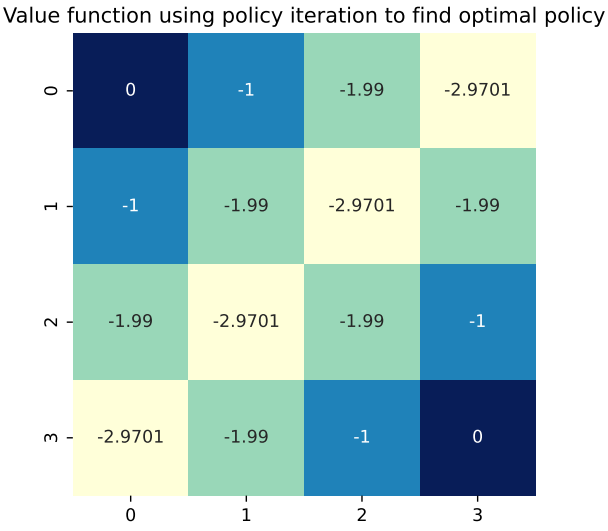
$$\pi'(s) = \arg\max_a q_\pi(s,a) = \arg\max_a \sum_{s',r} p\left(s', r|s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right]$$

Since we just saw how to compute the value function, we can combine these two ideas into one to get an algorithm which produces an optimal policy. This technique is known as *policy iteration*.

Problem 5 *Policy Iteration*

Implement policy iteration ([SB18, Section 4.3]) to find the optimal policy for the small gridworld example. For policy evaluation, re-use the function from section 5.

Info: Your code should produce the optimal policy and value function. For the value function, I obtain the following result:



Problem 6 *Convergence behavior*

The policy iteration algorithm on [SB18, p. 80] has a subtle bug in that it that the termination condition in step 3 may never be triggered even if $\gamma < 1$ and step 2 converges. Why? There are two different modifications that can fix the issue. Discuss what they are and which you would recommend.

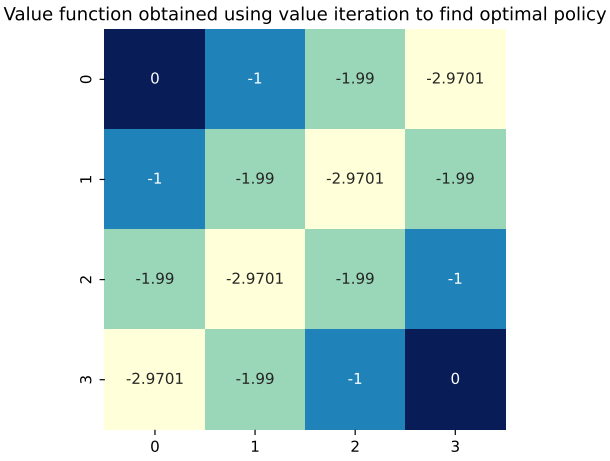
7 Value Iteration (`value_iteration.py`)

Value iteration merge the policy improvement step with the policy evaluation step above to create a single algorithm which usually converge faster; the result is very reminiscent of the DP algorithm from Lecture 2 except for a change in notation. Implement value iteration and use it once more to solve the small gridworld example.

Problem 7 *Value iteration*

Implement the value iteration algorithm from [SB18, Section 4.4] and use it to evaluate the small gridworld example.

Info: It should be no surprise the result of the value function should agree with the previous exercise. Specifically you should obtain:



7.1 A value-iteration agent (`value_iteration_agent.py`)

In this problem, you will implement an `Agent` that takes action based on the policy computed by value iteration. This agent will be useful to benchmark against since it by definition takes actions according to the optimal policy. It will also offer some nice visualization options similar to what was used during todays lecture.

The agent you implement will compute (and store) the policy computed the method `value_iteration` you implemented in section 7, and then in the policy method you should implement a policy π (`agent.pi`) which:

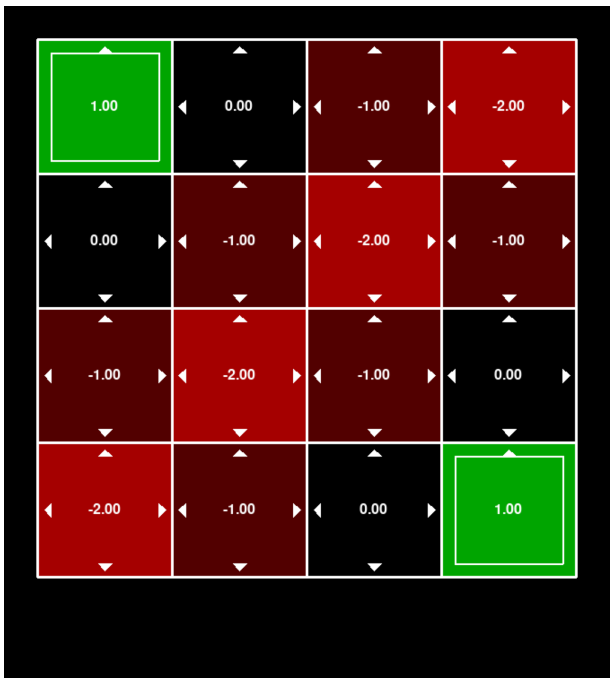
- With probability $1 - \varepsilon$ takes the optimal action
- With probability ε takes a random action

When $\varepsilon > 0$ the agent will not behave optimally – this is a bit illogical right now, but next week we will compare against methods which contains such a parameter and therefore it is very convenient to implement for a more fair comparison.

Problem 8 *Value iteration agent*

Complete the code for the value-iteration agent and read the code to understand what the example does. Note that the value-iteration should take random actions with probability `epsilon`. This may seem strange, but it will offer a more fair comparison to e.g. Q-learning as we will see in the coming weeks.

Info: Once done, you should get 100% integration with the agent/environment interface and nicer visualizations such as:



Note that the visualization contains a reward of $+1$ at the corner locations. This may seem at odds with the definition of the environment in [SB18], but recall that our gridworld requires the agent to take one last action in the terminal (corner) states [a], and therefore it gets a terminal reward of +1 to cancel out the movement bonus of -1 for this last (extra) move.

[a] If you really want to get into the weeds, this is in turn a good idea to get better visualizations for the demos. Don't overthink it; visualizations always make things a bit messy

8 Gambler's problem (`gambler.py`)★

Let's re-do the Gamblers problem described in [SB18, Example 4.3]. Read the included description and notice in particular that the number of available actions (how much to bet) depends on the state (players total capital).

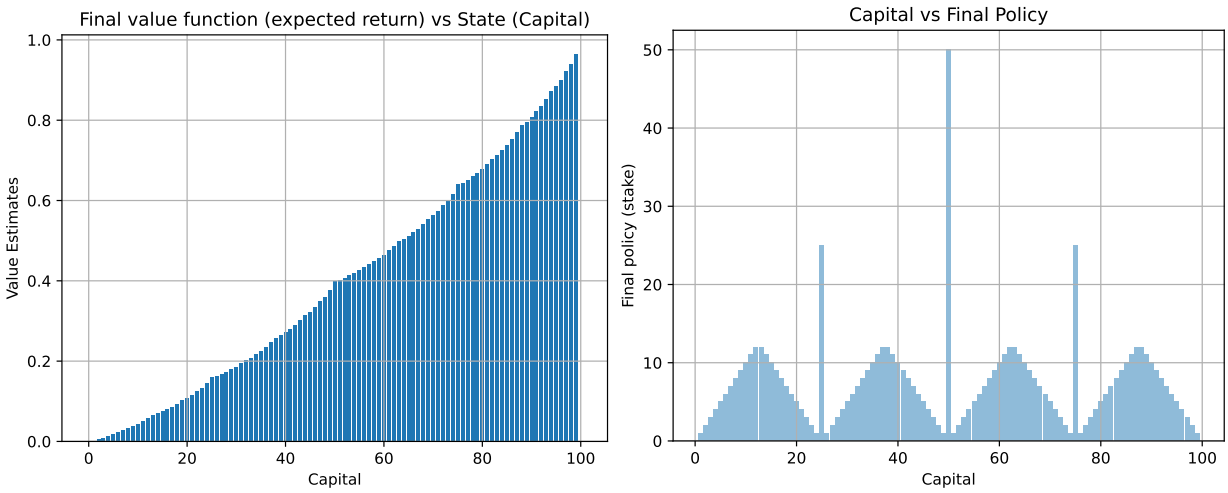
You should implement the Gambler problem as an MDP (see main portions of the code above), and then your code should take care of the rest. Note there are two terminal states : bankruptcy ($s = 0$) and winning $s = 100$. The states will therefore be $\mathcal{S} = (0, 1, \ldots, 100)$. Likewise, you cannot gamble more money than you have, or gamble so much that if you win you end up with more than $s = 100$ units of money (use this when you implement the action sets).

Problem 9 *Gambler's problem*

Implement the Gambler's problem environment and re-use your policy iteration code. You might need to make it a bit more robust if it uses `env.nA`, but these changes should be minimal. When done, check you get the same result as in the lecture notes.

Info: Reward should only be given when transitioning to the state $s = 100$; in all other cases it should be zero. When the gambler is in $s = 100$ or $s = 0$ he stays that way; apparently the world is harsh and he has good impulse control.

Once done, the value function should look as follows:



The policy might look as above and it might not – the problem is some actions can have same value (i.e. $q(s, a) = q(s, b)$) and the above figure is obtained by breaking ties by preferring the lowest gamble. I think, however, you can rely on the $s = 50, s = 25, s = 75$-spikes having the same height.

References

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (Freely available online).