EXERCISE 9
Policy and value iteration

Tue Herlau
tuhe@dtu.dk

1 April, 2022

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. (24 lines of code)
Material: Obtain exercise material from our gitlab repository at
`https://gitlab.gbar.dtu.dk/02465material/02465students`

Contents

1 The Markov Decision Process

In this exercise, we will consider reinforcement-learning methods in the idealized case where a model of the environment is known. The algorithms considered will be very reminiscent of the dynamical programming algorithm (the DP algorithm is nearly value iteration), however, there will be changes in the notation and structure due to the infinite-horizon aspect of the reinforcement-learning problem as well as the renaming of the variables as in [SB18].

This will bring us to the final model considered in the course, the Markov Decision Process (MDP). The MDP defines the set of available states, actions, the transition function $P(s', r|s, a)$, and a check to see whether a state is terminal:

```python
# mdp.py
class MDP:
    """
    The basic MDP class. It defines three main components:

    1) In a state s, the available actions as
        > mdp.A(s)
    2) In a state s, taking action a, the availble transitions
        > mdp.Psr(s, a)
        This function returns a dictionary of the form:
        > { (sp1, r1): p1, (sp2, r2): p2, ...}
        such that
            P(S' = sp, R = r | S=s, A=a) = mdp.Psr[ (sp,r) ]

    3) A terminal test which is true if state is terminal
        > mdp.is_terminal(state)

    4) An initial state distribution of the form of a dictionary:
        {s: p(S_0=s), ...}

    In addition to this, it also defines a list of all states and all non-terminal states:
        >>> mdp.states
        >>> mdp.nonterminal_states

    The set of states is computed on-the-fly by finding all states that can be reached from the initial state distribution.
    In other words, when you first call mdp.states, it may take some time for the call to finish.
    The advantage of this approach is you could implement and MDP with an infinite amount of states and as long as you never
    call mdp.states you wont run out of memory.
    """
    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):
        return False # Return true if the given state is terminal.

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

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

To implement your own MDP, all you need to do is to specify an initial state, a terminal condition (true when a state is terminal), a set of actions available in each state, and finally the transition probability $P(s', r'|s, a)$. Note the later function returns a dictionary where the keys are (s', r') and the value is the corresponding probability. Note the same file contains helpful functionality to transform a MDP into an environment and a (discrete) gym environment into a MDP; you don't need to read this functionality at this point, just be aware than once you have specified a MDP, there is an easy way to turn it into an environment.

2 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 the various states under the policy, which in turn allows us to plan by selecting actions which lead to states with high value, which will be a theme in the remainder of this course.

In this exercise, we will look at how we can evaluate the value function in the idealized setting where the MDP is known using dynamical programming. The MDP will be an instance of the MDP class given earlier and implement a small $4x4$ gridworld example from [SB18, Example 4.1]. To familiarize ourselves with the MDP class the implementation is given below:

```python
# small_gridworld.py
UP,RIGHT, DOWN, LEFT = 0, 1, 2, 3
class SmallGridworldMDP(MDP):
    def __init__(self, rows=4, cols=4):
        self.rows = rows # Number of rows, columns.
        self.cols = cols
        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))
        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.

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).

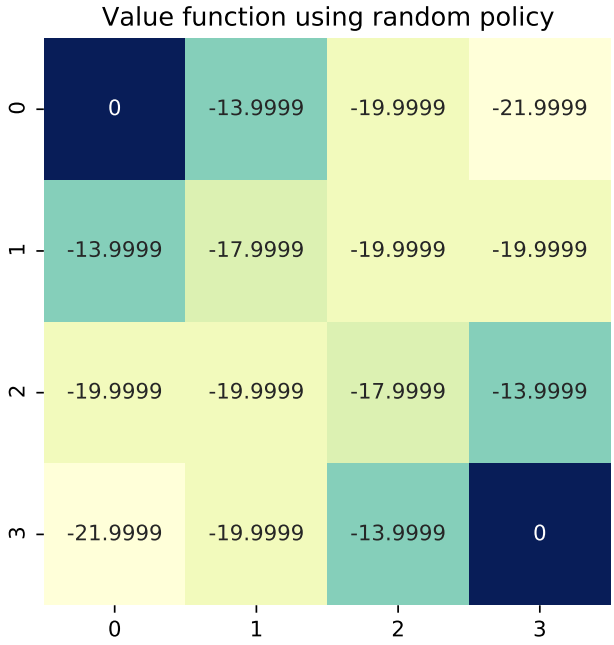
Problem 1 *Policy Evaluation*

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 described in the pseudo code.

Info: When you implement policy evaluation, it is convenient to complete (and use) the little helper function which implement parts of the policy evaluation algorithm given in [SB18, Section 4.1] (see code comments). What the function does is evaluate the Q-values for a given state using value-function values:

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

This kind of function is used throughout the dynamical programming chapter. Once done, you can run the code and obtain a value function as follows:



3 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:

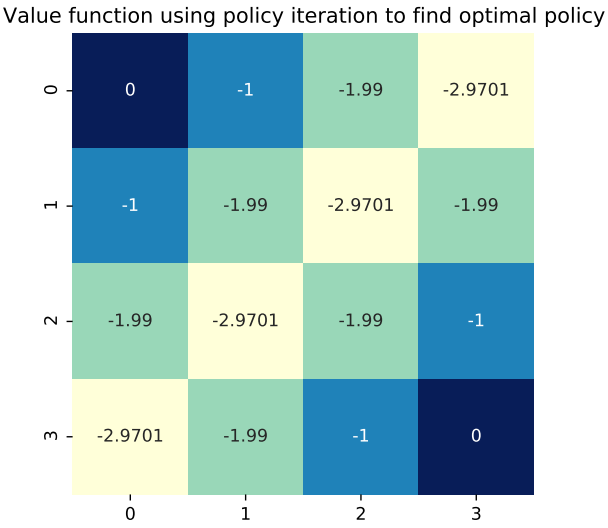
$$\pi'(s) = \arg\max_a q_\pi(s,a) = \arg\max_a \sum_{s',r} p\left(s', r|s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right]$$

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 2 *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 2.

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



Problem 3 *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.

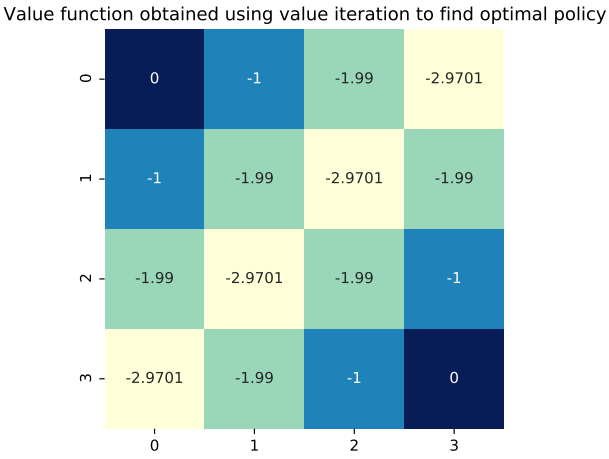
4 Value Iteration (`value_iteration.py`)

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 4 *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:



I have included two extra problems that are a bit more challenging and which highlights some practical issues with the DP algorithm.

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

If we build an Agent that takes actions based on the optimal policy computed using value iteration, we can automatically get a perfect agent to benchmark against. The gridworld environments we will mostly be using will (obviously) be build around an MDP, but also offer automatic visualization and an easy way to specify the grid-world map. As an example, the small gridworld we have experimented with can be defined as just:

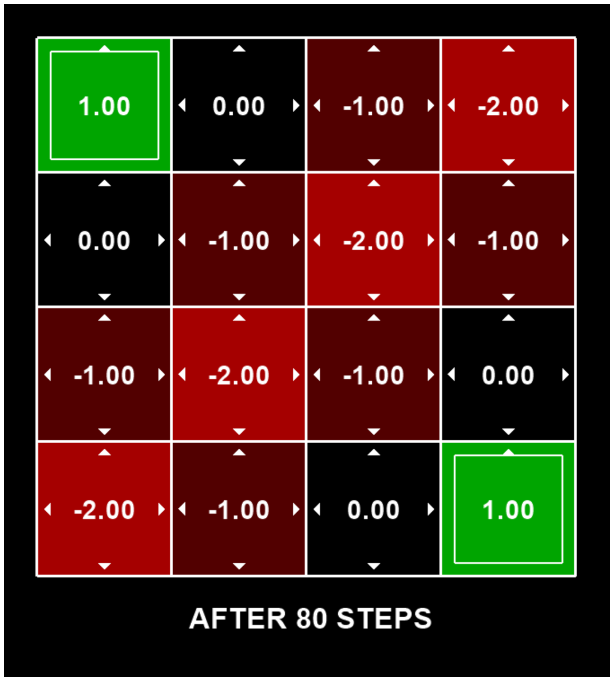
```
# gridworld_environments.py
sutton_corner_maze = [[  1, ' ', ' ', ' '],
                      [' ', ' ', ' ', ' '],
                      [' ', 'S', ' ', ' '],
                      [' ', ' ', ' ',   1]]
"""
Implement Suttons little corner-maze environment (see (SB18, Example 4.1)).
You can make an instance using:
> from irlc.gridworld.gridworld_environments import SuttonCornerGridEnvironment
> env = SuttonCornerGridEnvironment()
To get access the the mdp (as a MDP-class instance, for instance to see the states
↪  env.mdp.nonterminal_states) use
> env.mdp
You can make a visualization (allowing you to play) and train it as:
> from irlc import PlayWrapper, Agent, train
> agent = Agent()
> agent = PlayWrapper(agent, env) # allows you to play using the keyboard; omit to
↪  use agent as usual.
> train(env, agent, num_episodes=1)
"""
```

```
19 class SuttonCornerGridEnvironment(GridworldEnvironment):
20     def __init__(self, *args, living_reward=-1, **kwargs): # living_reward=-1 means
       ↪ the agent gets a reward of -1 per step.
21         super().__init__(sutton_corner_maze, *args, living_reward=living_reward,
           ↪ **kwargs)
```

Problem 5 *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

5 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).

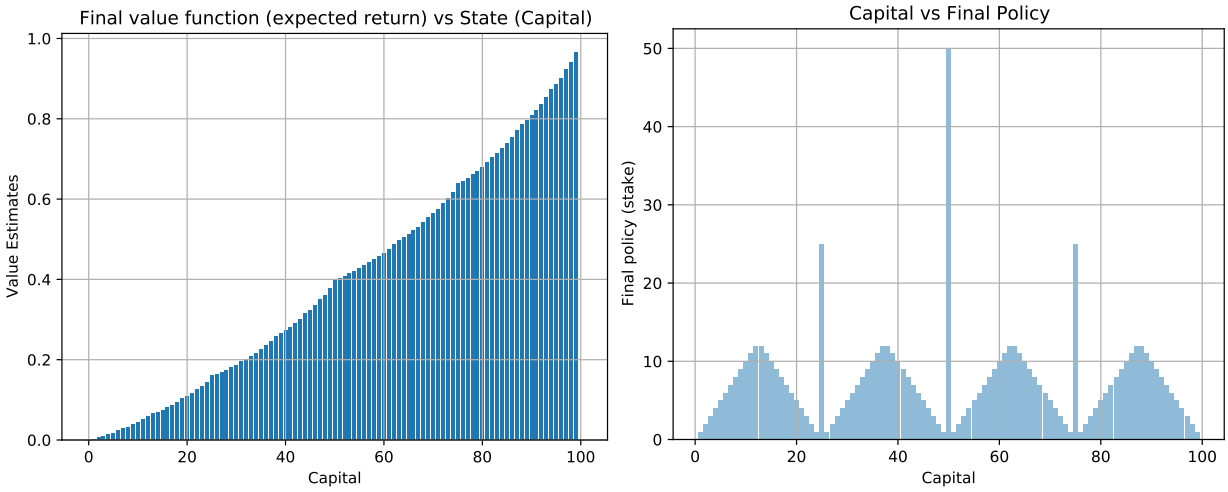
You should implement the Gambler problem as an MDP (see main portions of the code above), and then your code should take care of the rest. Note there are two terminal states : bankruptcy ($s = 0$) and winning $s = 100$. The states will therefore be $\mathcal{S} = (0, 1, \ldots, 100)$. Likewise, you cannot 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 6 *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.

6 Jack's car rental (`jacks_car_rental.py`) ★★

Finally, let's look at Jacks car rental problem [SB18, Example 4.2: Jack's Car Rental]. In the book, it is just listed as a student problem (exercise 4.7), but I think it is a little hard to build from the ground up so some help will follow: computationally fairly intensive and I have not found any obvious ways to speed it up, so I recommend starting with a much smaller problem to debug the implementation. A description of the problem can be found in [SB18, Example 4.2], but it took me a while to work out what I think is the right interpretation:

- Each state s is comprised of the cars at location 1 and 2: `s = (c1,c2)`. In other words, the `states` variables must be all possible combinations of c_1 and c_2, i.e. from 0 up to and including 20 cars.
- The actions are the amount of cars moved from c_2 to c_1. In other words an action of $a = 0$ keeps these the same, whereas $a = 1$ leads to $(c_1, c_2) \to (c_1 + 1, c_2 - 1)$ (this is before the renting/returning).
- According to the problem a is constrained by ± 5 AND that the car numbers cannot be negative: i.e. $c_1 + a$ and $c_2 - a$ cannot be negative.
- In addition to this cars are returned and rented out, which I thought was the really confusing part. Suppose we let $s_0 = (c_1^0, c_2^0)$ be the state of the dealership at $t = 0$, and let us suppose

$$d_1, d_2, \quad i_1, i_2 \tag{1}$$

  are the number of cars demanded and the number of cars returned in.
- I will make the assumption the above numbers are truncated at 9; this is a simplified way of posing the problem, but it makes things much easier. Code is included to compute the PDF.
  - During the evening of the first day the number of cars are s^0
  - The car-dealership move cars between the two locations by applying a
  - The above four numbers are generated from their respective Poisson distributions as described in the problem.
  - It now becomes morning on day $t = 1$
  - The number of *rented out* cars are applied to update s^0. Note $c_{1,2}$ cannot be negative, so excess rent-demand is losts and s^0 is updated. This gives us r_1
  - The reward is computed using the actual number rented out (no excess demand)
  - Cars are returned during the day, but they have to be cleaned in the afternoon; this is what happens next:
  - The number of returned cars are added to s^0; however if the number exceed the maximum car number (20), the excess cars are simply sent to the factory. This gives us finally the value s^1, as well as r_1

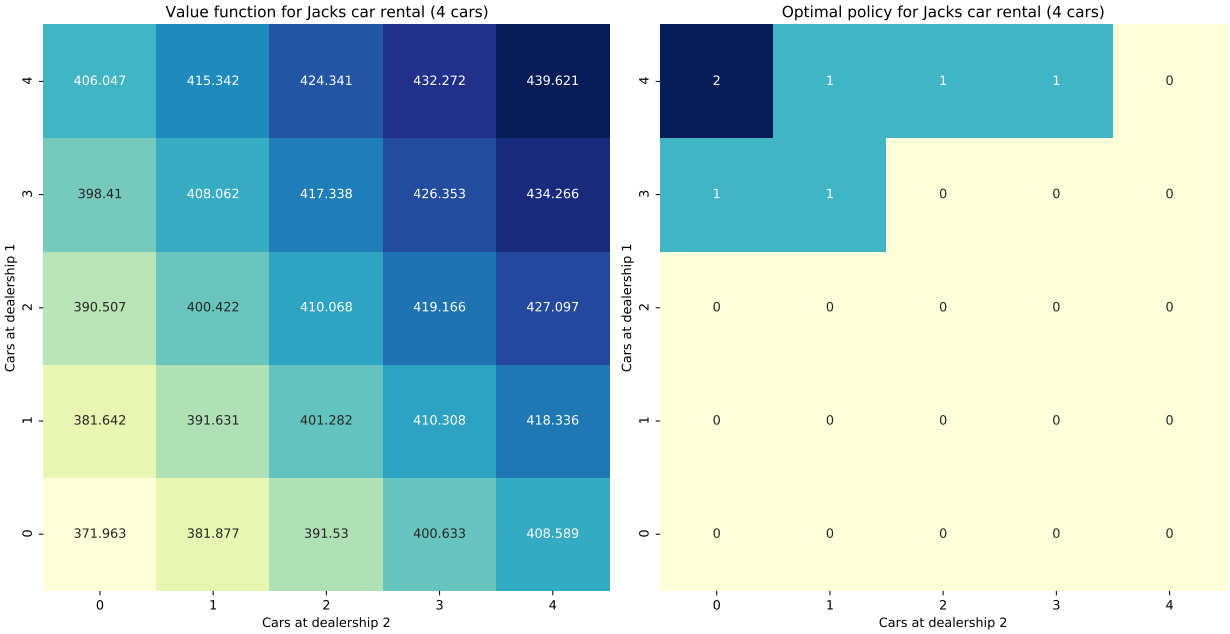
– The probability of this single event is the product of the probability of the four numbers in eq. (1); however the probability of an actual transition $(s_t, a_t) \to (s_{t+1}, r_{t+1})$ is the sum of all individual paths which lead to this transition.

I found it helpful to make a helper function to implement this update.

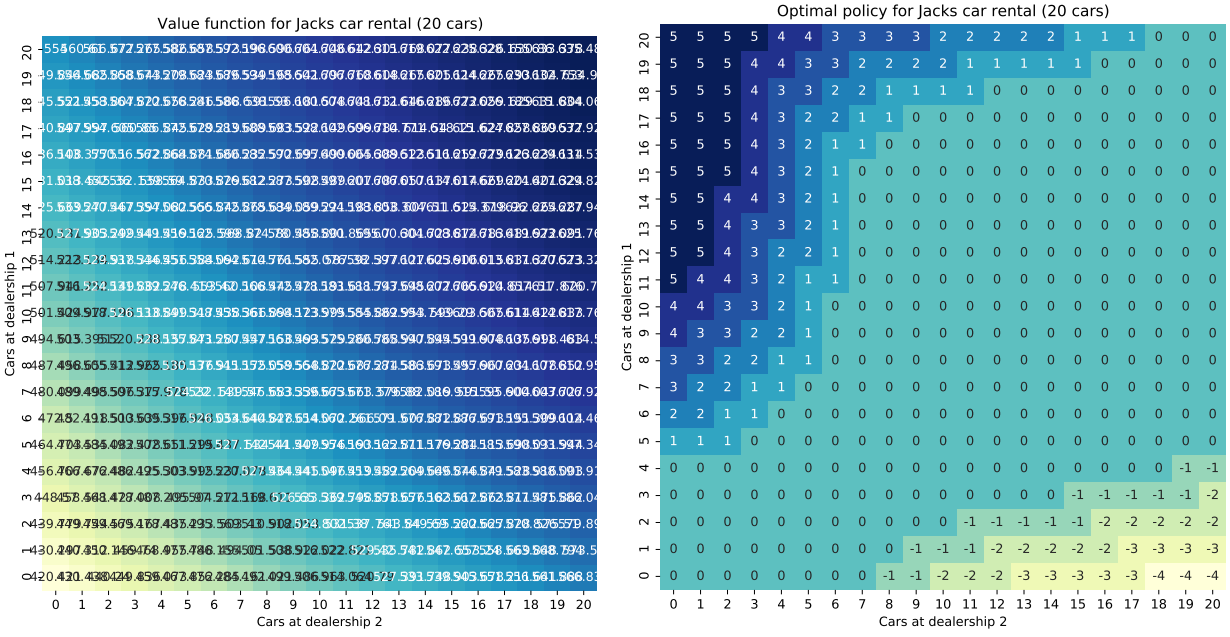
Problem 7 *Jacks's car rental problem*

Implement the Jacks's car rental problem using the above instructions. The script contains code for a (very small) problem for debugging, as well as a larger problem.

Info: For the bonus question, think about γ. My results for the small environment (and I confess there is some room for errors) are as follows for the policy and value functions should look as follows for the small problem:



Meanwhile, for the big problem I get the following results:



References

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (See **sutton2018.pdf**).