Assignment 4: Discrete Markov Decision Process
Course: Symbolic AI, Leiden University
Written by: Thomas Moerland

For this assignment you will solve the maze from the previous assignment, but this time using a dynamic programming approach.

Assignment files, Python and Numpy

- First download the assignment zip file from Brightspace. Unzip it and inspect the folder. It should contain three files: `world.py`, `dynamic_programming.py`, and an example environment definition `prison.txt`.
- The coding assignment will be in Python 3. Verify that your operating system has Python 3 installed, otherwise install it.
- We will also need a specific package: NumPy, short for Numerical Python. Numpy is an important Python package, which can be used if you want to do vector and matrix operations. For the assignments, you will mostly need to understand indexing and slicing into value and state-action value arrays.
- You can install a variety of scientific Python packages at once by installing the individual version of the Anaconda package: `https://www.anaconda.com/products/individual`. This will also give you a Python editor: Spyder. However, feel free to manually install Python 3, NumPy, and an IDE of your own choice.
- If your are unfamiliar with Python and Numpy, you may take a quick tutorial for both online. Do not spend too much time here, you will only need a few basic concepts, which you can also search for during the assignment.

Handing in assignment You need to submit three files, where you replace `groupnr` with your group number:

- `dynamic_programming_groupnr.py`, your modified version of `dynamic_programming.py` with the relevant answers. Be sure to check whether your solution runs from the command line.
- `prison_groupnr.txt`, your modified version of `prison.txt` for the relevant exercise.
- `answers_groupnr.pdf`, with your answers to the open questions in the assignment.

Submit these files to Brightspace assignment 3 in a single zip file named:
`assignment3.zip`.

1 Coding Exercises

In these exercises you will implement Value iteration (VI) and Q-value iteration (QI), two variants of Dynamic Programming, to solve a given Markov Decision Process. We will first explain the environment (the MDP definition), and the starting point for your algorithm implementation.

1.1 The environment

The environment is pre-coded for you in `world.py`. It contains the class definition of `World(filename)`, which will initialize the environment specified in the text file `filename`.

- **Definition in txt file**: You can define the environment in a txt file. An example is provided in `prison.txt`. We use the following encoding:

  $\star$ = agent location

  # = wall

  a (lower case) = key

  A (upper case) = door

  1 (numeric) = goal (terminates episode). The reward is equal to 10 times the numeric element at the goal.

```
Initialization map:
#  #  #  #  #  #
#  *     A     #
#  a     #     #
#  #  #  #     #
#              #
#     #  #  B  #
#  b  #  #  3  #
#  #  #  #  #  #
```

Figure 1: Prison example provided in prison.txt.

- **Explanation of MDP**:
  - State space: The state is represented as an index. For the provided example, there are 64 unique states, since there are 16 free locations, and two keys. In each location, we can hold or not hold either key, which gives rise to $16 \cdot 2 \cdot 2$ possible states. These are simply numbered 0-63. If you want to know what situation a particular state actually represents, you can call the method `World.print_state(state))`, see below.
  - Action space: In every state, the agent has four possible actions: {up, down, left, right}.
  - Dynamics: When the agent moves into a wall, it just remains at the same position. The agent automatically picks up a key when stepping on the specific location, and automatically opens the door when stepping on it while holding the specific key.
  - Reward: The reward at every transition is -1, except when we reach a goal: the reward is then equal to 10 times the numeric element at the goal. So, in the example above, the reward at the goal is equal to 30.
  - Gamma: We assume $\gamma = 1.0$ throughout the experiments.
- **Attributes**: An `World` object has a few important attributes:

- `states` returns a list of all states. When you initialize a map, all possible configurations of agent location and key possession are automatically inferred for you, and each possible combination is assigned a unique state index.
- `n_states` returns the total number of states (a scalar).
- `actions` a list of all possible actions.
- `n_actions` returns the total number of actions (a scalar).
- `terminal` indicates whether the agent has reached a goal (task terminates).

• **Methods**: An `World` object has several important methods.

- `transition_function(s,a)` computes, for a given state and action, the next state `s_prime` and reward `r`. It does not affect the agent location!
- `act(a)` executes action `a`, i.e., it calls `transition_function()` and then actually moves the agent. It also checks for termination.
- `reset_agent()` resets the agent to the start location, as given in the initial map. Also sets the `terminal` attribute to False.
- `get_current_state()` returns the current state of the environment.
- `print_state(s)` prints the description what situation of the environment a particular discrete state actually represents.
- `print_map()` prints the current map of the environment.

When you execute the `world.py` script from the command line, which in Python will execute the code below `if __name__ == '__main__':`, found at the bottom of the file. This gives some examples of the above methods. You can play around a little bit to familiarize yourself with the environment.

1.2 The algorithm

For the exercises, you will implement two dynamic programming algorithms in the environment described above. You should use the `dynamic_programming.py` file, which contains the `DynamicProgramming()` class.

- **Attributes**: An `DynamicProgramming()` object has two important attributes:
  - `V_s` a value table. A value table is vector of length `n_states`. Each element in the vector stores the value estimate for the corresponding state index, i.e. $V(s = 4) =$`V_s[4]`. If `V_s = None`, then you have not run any method yet to estimate the optimal value table.
  - `Q_sa` a state-action value table. A state-action value matrix of dimensions `n_states` $\times$ `n_actions`. Actions are indexed according to `World.actions = {up,down,left,right}`. For example, action `up` has index 0. Each element in the `Q_sa` matrix stores the value estimate for the corresponding state-action, i.e., $Q(s = 10, a = 0) =$`Q_sa[10,0]`. If `Q_sa = None`, then you have not run any method yet to estimate the optimal value table.
- **Methods**: An `World` object has several important methods.
  - `value_iteration(self,env,gamma=1.0,theta=0.001)` should run value iteration on the environment `env` (of class `World`). You should implement this function yourself. Gamma is the discount factor, which you can leave at the default value of 1.0. Theta is the threshold for convergence, which you can also leave at the default value of 0.001.
  - `Q_value_iteration(self,env,gamma=1.0,theta=0.001)` should run Q-value iteration on the environment `env` (of class `World`). You should implement this function yourself.
  - `execute_policy(self,env)` executes a policy on environment `env`. This function is partially implemented for you. You should implement estimation of the greedy policy.

1.3 Exercise: Dynamic Programming (coding)

Start by executing `dynamic_programming.py`. This executes the code under `if __name__ == '__main__':` at the bottom of the script. You can manually execute a policy in the environment, and familiarize yourself with the environment.

1. **Value iteration**:

   a Implement value iteration, in the `DynamicProgramming.value_iteration()` method. Do not change the function arguments or return statements. A start value table is already provided for you: `V_s = np.zeros(env.n_states)`. Your function should compute the optimal value function, and at the end of the function store the optimal value table in `self.V_s`. Include a print statement that prints the error in each iteration of your algorithm.

   b Implement `DynamicProgramming.execute_policy()` to execute the greedy policy based on the value table $V(s)$. You only need to implement the code segment below `if table == 'V' and self.V_s is not None:`, which should set the `greedy_action` variable to the greedy action (or one of the greedy actions) in the current state.

   c Check whether your implementation works. Does our agent during execution follow the optimal policy?

2. **Q-value iteration**

   a Implement Q-value iteration in the `DynamicProgramming.Q_value_iteration()` method. Do not change the function arguments or return statements. A start state-action value table is already provided for you: `Q_sa = np.zeros(env.n_states,env.n_actions)`. Your function should compute the optimal state-action value function, and at the end of the function store the optimal state-action value table in `self.Q_sa`.

   b Implement `DynamicProgramming.execute_policy()` to execute the greedy policy based on the state-action value table $Q(s, a)$. You only need to implement the code segment below `elif table == 'Q' and self.Q_sa is not None:`, which should set the `greedy_action` variable to the greedy action (or one of the greedy actions) in the current state.

   c Check whether your implementation works. Does our agent during execution follow the optimal policy?

3. **Multiple goals**

   a Prison.txt only has a single goal. Adapt the prison so that it has two reachable goals. You may also build a new maze, as long as it has two reachable goals. Each goal should be reachable from the start location. Make your maze such that depending on the starting location, the goal that is picked changes under the optimal policy. Note that each goal is a terminal state.

   b Run value iteration or Q-value iteration on your new environment, and describe the observed agent behaviour.

2 Reflection Exercises

4. **Reflection on Dynamic Programming**:
   When you successfully implemented DP, you saw that it solves the problem very fast. The problem to which we applied is was however quite small. Imagine we have a world of size 100×100, which can have 10.000 free agent locations. And imagine this more complex world has 30 keys and doors.

   a How many unique states does this new problem have? (Note: you should count every possible combination of agent location and key possession)

   b Imagine we use 32-bit floating numbers to store the values in the table, i.e., every value estimate takes 32 bits, or 4 bytes, in memory. How much memory would we roughly need to store the value table for this new problem in memory?

   c Roughly how fast would you solve this problem on you laptop? Explain your answer.

   d Explain the *curse of dimensionality*. What aspect of our problem definition causes the exponential growth?

5. **Comparison to search**:
   We may also compare Dynamic Programming to the search approaches you have previously encountered.

   Imagine we apply an **iterative deepening tree search** (i.e., no graph search, so we do not detect whether we already encountered a state, but simply expand the tree in all directions) to the example problem in `prison.txt`.

   a Estimate the time complexity of an iterative deepening tree search on the `prison.txt` problem (hint: first compute the depth of the shortest path towards the goal).

   b Compare the time complexity of iterative deepening tree search to the time complexity you empirically observed for dynamic programming on the prison problem. Which approach is faster?

   c Compare the way Dynamic Programming stores the solution to the way tree/graph search approaches store the solution. What could be a benefit of the DP representation, and what could be a benefit of the tree/graph search representation?