Middle East Technical University — Department of Computer Engineering

CENG 462

Artificial Intelligence

Spring 2023-2024

Assignment 4

Regulations

1. The homework is due by **23:55 on May 14th, 2024**. Late submission is not allowed.
2. Submissions are to be made via ODTUClass, do not send your homework via e-mail.
3. Upload your solution as a single **Python** file named *yourStudentId_HW4.py*. **Submissions that violate the naming convention will incur a grade reduction of 10 points.**
4. Send an e-mail to **garipler@metu.edu.tr** if you need to get in contact.
5. **This is an individual homework, which means you have to answer the questions on your own. Any contrary case including but not limited to getting help from automated tools, sharing your answers with each other, extensive collaboration etc. will be considered as cheating and university regulations about cheating will be applied.**

Problem

In this assignment, you are expected to implement Q-Learning-based reinforcement learning to solve grid-world problems.

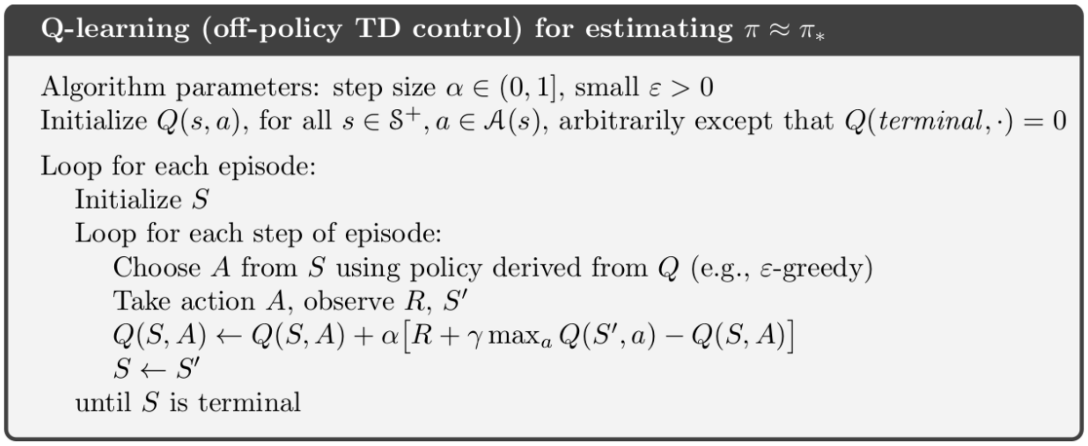
Q-learning is a model-free reinforcement learning method which is based on learning the q-values of states directly, bypassing the need to ever know any values, transition functions, or reward functions. Q-learning uses the following update rule:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

Grid-world problems are environments represented with rectangular cells in a 2D plane and involve tasks such as navigation (the agent has to navigate from a particular location to a target location), item collection (the agent has to gather all items in the environment), etc.

For this assignment, our focus is on the navigation tasks, where the agent has to navigate from a particular location/state to the goal location/state. Each state can be a member of only one of the following types: normal, obstacle, and goal. The agent can navigate between normal states and goal states, whereas it cannot transition to an obstacle state. Attempting to leave the environment from border states (the states that enclose the environment) or to transition an obstacles state results in a failure and the agent remains where it has been before applying the action that has caused the failure. The action set available in every normal state is as follows: right, down, left, up, which are represented with '>', '∨','<', and '∧' characters, respectively. As the names state, each action takes the agent to the next state in the intended direction (i.e. the left action takes the agent to the adjacent state on the left of the current state). The agent is subject to the action noise problem that is the main cause of the non-deterministic behavior (we can think of the action noise as a random glitch occurring in the actuators of the agent that provide the navigation, or the floor is covered with ice and slippery). So the agent is not always guaranteed to take the considered action and transition to the intended state. The action noise is represented with a triple of the form (a, b, c), where b is the probability of taking the intended action, a is the probability of the action that is obtained by rotating the intended action 90 degrees in the counter-clockwise direction, similarly, c is for the one that is obtained by rotating the current action 90 in the clockwise direction.

For instance, when (0.1, 0.8, 0.1) is given as the action noise and the agent considers the up action, with probability values of 0.1, 0.8, 0.1 it takes left, up, right actions, respectively (If it decides on the down action, it takes right, down, left with probability values of 0.1, 0.8, 0.1 respectively).



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
   Initialize $S$
   Loop for each step of episode:
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Figure1. Q-Learning Pseudocode

Simulation

In order to realize the interaction between an environment and an agent, we need to create a simulation for the problem where the agent starts from a particular state and applies an action on each state till it reaches its goal. Solving the problem only once is not sufficient to calculate correct state values in the problems so the agent has to visit every possible state multiple times over a sufficient period of time in order to learn their correct utility scores. Since our focus is on navigation MDPs we are going to simulate them as episodic tasks. An episode refers to the agent's journey (all state-action-reward sequence) starting from a particular state and ending in one of the goal states. As soon as the agent reaches a goal state, a new episode is started and the agent is re-located in a particular state in the problem. During the episodes, the agent amasses experience (history of states, rewards, actions). Utilizing one of the methods of this assignment it can extract the optimal policy for the solution of the problem after getting exposed to sufficiently many episodes.

Algorithm 1 given below provides a pseudo code of the simulation procedure for an episodic task. Before each episode, the agent is informed of its initial state (the agent is placed to an initial state). Then the agent is asked to deliver an action to take. This action is applied to the MDP and the resulting next state and the reward/cost that occurred is sent to the agent for its calculations (so that the agent knows its new state and its feedback for its behavior). For the goal states, the agent should additionally be informed that it has reached a goal state (The agent may need to perform some calculations before a new episode or it may need to store/calculate some information after an episode). So the agent is informed only with its initial state before each episode, during an episode it gets the next state and reward/cost of each action, and (for goal states) it also knows it's reached a goal state. For this assignment, during the simulation execution, there are two places where randomization should take place: ϵ-greedy policy, action noise. For these purposes, you are going to employ the random package of Python.

Algorithm 1 Agent - MDP Interaction

```
procedure SIMULATE(agent: agent, mdp: problem MDP, k: episode count)
    for i=1 to k do
        Provide agent with its initial state
        while agent has not reached a goal state do
            Ask agent for an action a
            Apply a on mdp, provide agent with the resulting next state $s'$, the reward R
        end while
    end for
end procedure
```

Specifications

- You are going to implement Q-learning in Python 3.
- Each MDP problem is represented with a txt file that will be fed to your implementation.
- You are expected to implement a function with the following function signature:
  def SolveMDP(problem_file_name, random_seed)
  where the parameter ***problem_file_name*** specifies the path of the MDP problem file to be solved (file names are typically "mdp1.txt", "mdp2.txt", "mdp3.txt" etc.);
  and the parameter ***random_seed*** pecifies the value for the random generator package (for the random.seed function). During the evaluation process, this function will be called and returned information will be inspected.
- The returned information should be as follows:
  U , policy = SolveMDP(problem_file_name , random_seed)
  U is the value table that shows the utility score calculated for each state and is in the from of a dictionary that stores (state, utility) key-value pairs (e.g. (0,0): 2.5, (0,1): 1.5...).
  policy represents the policy function learned and is also a dictionary of (state, action) pairs (e.g. (0,0): '∨', (0,1): '>' ...). Actions are represented with one of the following characters: '>', '∨','<', and '∧' (RIGHT, DOWN, LEFT, UP respectively).

  The following equation holds for state values and state-action values for Q-learning :

  $$U(s) = \max_a Q(s,a)$$

  In order to calculate the optimal policies the following rule should be considered for Q-learning:

  $$\pi^*(s) = arg\max_a Q(s,a)$$

- The grid-world MDPs are described in ".txt" files and have the following structure:

```
[environment]
M N
[obstacle states]
state|state|state ...
[goal states]
state:utility|state:utility ...
[start state]
state
[reward]
reward value
[action noise]
forward probaility
left probability
right probability
[learning rate]
learning rate value
[gamma]
gamma value
[epsilon]
epsilon value
[episode count]
episode count value
```

MDP information and algorithm parameters are provided in separate sections and each section is formed with '[]'. They are as follows:

- **[environment]** section defines the size of the grid-world problem. **M, N** are the height andwidth of the problem (the number of cells vertically and horizontally), respectively.
- [obstacle states] section specifies the obstacle states (multiple obstacle states are separated with '—').
- [goal states] section provides each goal state with its reward value (separated with '—').The reward values of the goal states are specified after ":".
- [start state] section keeps the initial state information that specifies the starting state of the agent for a new episode.
- [reward] section keeps the reward value for each action in every normal state (all actions yield the same reward value in every normal state).
- [action noise] section specifies the action noise tuple (a, b, c). The first numerical value (i.e. forward probability) is b, the second (i.e. left probability) is a and the third (i.e. right probability) is c.
- [learning rate] section specifies the α parameter used in algorithms.
- [gamma] section provides the value of the γ parameter for both methods.
- [epsilon] section specifies the value for the ϵ parameter that is used for ϵ-greedy policy sampling.
- [episode count] section provides the value for the k parameter of Algorithm 1. It specifies how many episodes should be run in the simulation for learning.

Each normal/goal state is represented with a tuple of the form (i, j), where i, j are y-axis value and x-axis values, respectively, when the problem is considered as a grid.

- No import statement is allowed except for **copy** and **random** modules with which you may perform a deep copy operation on a list/dictionary and sample a random number, respectively. All methods should be implemented in a single solution file. You are expected to implement all the necessary operations by yourself.
- Your implementation will be both automatically and manually inspected. Therefore, commenting is crucial for understanding your implementation and decisions made during the process.

Sample I/O

mdp1.txt

```
[environment]
3 4
[obstacle states]
(1 ,1)
[goal states]
(0,3):5.0 | (1,3):-5.0
[start state]
(2,0)
[reward]
-0.04
[action noise]
0.8
0.1
0.1
[learning rate]
0.1
[gamma]
0.9999999999
[epsilon]
0.1
[episode count]
10000
```

Expected output of the SolveMDP function applied on mdp1.txt

```
>>> SolveMDP (”mdp1.txt” , 462) ({(0 , 0) : 4.82 , (0 , 1) : 4.86 , (0 , 2) : 4.92 , (1 , 0) : 4.76 , (1 , 2)
: 4.63 , (2 , 0) : 4.69 , (2 , 1) : 4.62 , (2 , 2) : 4.57 , (2 , 3) : 4.08} , {(0 , 0) : ’>’, (0 , 1) : ’>’, (0 , 2) :
’>’, (1 , 0) : ’∧’ , (1 , 2) : ’<’, (2 , 0) : ’∧’ , (2 , 1) : ’<’, (2 , 2) : ’<’, (2 , 3) : ’∨’})
```

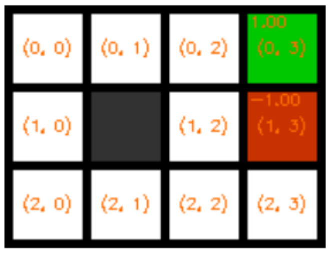


Figure2. Visualization of the Problem Represented by mdp1.txt

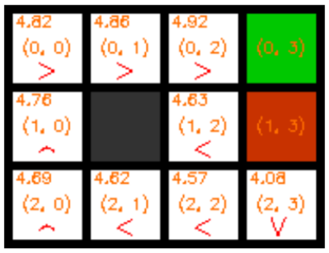


Figure3. Visualization of the Solution for mdp1.txt