Middle East Technical University | Department of Computer Engineering

CENG 462

Artificial Intelligence

Fall '2021-2022

Homework 5

Due date: 18 January 2022, Tuesday, 23:55

1 Objectives

This assignment aims to familiarize you with two Temporal Difference (TD) learning methods, namely TD(0) and Q-learning (which are direct approximations for Value Iteration and Policy Iteration methods) at the implementation level and provide hands-on experience with these algorithms.

2 Problem Definition

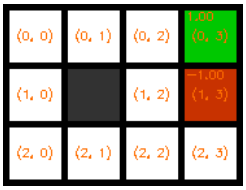
In the previous assignment, we were supposed to implement two dynamic programming methods (Value Iteration and Policy Iteration) in order to solve MDP problems whose transition and reward function components were known in advance. For these types of MDPs, we did not have to simulate the interaction between an MDP and an agent. In this assignment, we shift our focus to more realistic problems where we are devoid of having the knowledge of the transition and reward functions completely and attempt to learn optimal policies. Since we cannot leverage the previous two methods, in this case, we employ the algorithms that approximate them (model-free methods). Temporal difference (TD) learning aims to learn utility scores from trial and error processes. In other words, gathered experience which emerges owing to the interaction between an agent and an MDP is used to build a policy for the agent. TD methods are based on the following update rule:

$$U(s) = U(s) + \alpha(R + \gamma U(s') - U(s)) \tag{1}$$

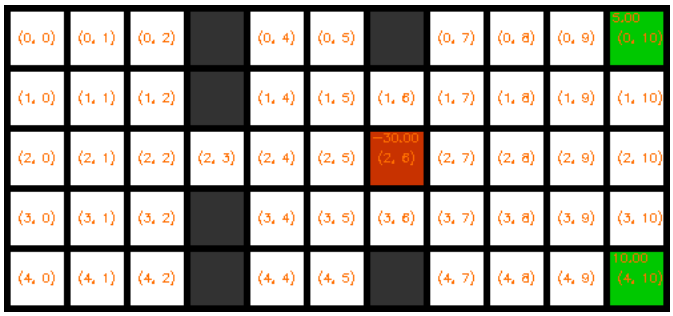
where R is the cost/reward received after applying an action on a state s and s' is the resulting state, γ is the discount factor, α is the learning parameter which governs how much adjustment should be made on the estimated utility score. After gathering a sufficient amount of experience (under certain mathematical conditions), TD learning is guaranteed to obtain the correct utility scores hence the optimal policy. In this sense, TD methods that learn the utility scores can be regarded as an approximation to Value Iteration. In most of the problems, actually we seek an optimal policy, which is the solution to the given problem. Instead of learning only state utility scores, we could learn utility values for state-action pairs, from which the policy can directly be extracted. Q-learning operates on state-action pairs and can be regarded as a direct approximation to Policy Iteration. It learns utility scores without necessarily following its behavior policy hence it is called an off-policy method. The update rule for Q-learning is as follows:

$$Q(s,a) = Q(s,a) + \alpha(R + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{2}$$

(a) Sample 5x11 grid-world MDP



(b) Sample 3x4 grid-world MDP



(c) Sample 5x11 grid-world MDP

Figure 1: Sample MDP grid-world problems. Each state is represented with a rectangular cell and colored depending on its type. Normal, obstacle, goal states are depicted with white, black, green/red colors, respectively. The tuples in the middle of each specify the state label of that cell. The numbers at the top-left indicate a state's utility value. Goals states are colored depending on their reward sign (negative red, positive green).

The parameters are almost the same as that of Equation 1 but while obtaining the utility value of the transitioned (next) state the max operator is used. This is where its "off-policy"ness lies since it does not have to take the action whose utility score is the highest in the next state among all other candidate actions.

Due to the lack of knowledge of transition and reward functions, agents have to interact with problems and learn from the experience that they gather. During the interaction, the agent should enrich its knowledge about the problem (environment) with exploratory moves, at the same time it has to act according to the information it has learned so far (exploitation) in order to solve the problem. This confliction (how much the agent should explore and how much it should depend on its gathered experience) is coined as the exploration-exploitation trade-off. In this assignment, we utilize the ϵ-greedy policy (with a probability of ϵ, the agent takes an exploratory action, e.g. random action) as a solution to this trade-off.

As was the case with the previous assignment, in this assignment too, our focus is on grid-world (MDP) problems and you are expected to implement TD(0) and Q-learning in order to solve them.

2.1 Grid-World Problems

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 (Figure 1), 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 `'>'`, `'V'`, `'<'`, `'^'` 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 3 tuple 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).

3 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 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 this purpose, we are going to employ the random package of Python. While sampling depending on the ϵ-greedy policy, the following code piece should be used:

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

```
if random.random() <= epsilon:
    actions = ['<', '^', '>', 'V']
    selected_action = actions[self.random_module.randint(0, 3)]
else:
    # For Q-learning: select the action whose utility score is the highest
    # For TD(0): select the action which leads to the neighbor with the highest
    utility score
```

In the previous assignment, the action noise problem manifested itself on the transition function and the agent was able to consider it explicitly. But for this assignment, the agent is provided only with the initial state, next states, and rewards/costs hence there is no way for the agent to know the presence of this problem. The most convenient time to incorporate the action noise problem into an MDP is when an action is applied to the MDP. For instance, let's suppose the agent has delivered the UP action. As explained in the previous section, the following action distribution emerges (assume the action noise distribution is (0.1, 0.8, 0.1)): {LEFT: 0.1, UP: 0.8, RIGHT: 0.1}. In order to sample action from this distribution, the following code piece could be used:

```
actual_action = random.choices(['<', '^', '>'], weights=[0.1, 0.8, 0.1])[0]
```

Similarly, for the given RIGHT action and the action noise distribution (0.1, 0.75, 0.15):

```
actual_action = random.choices(['^', '>', 'V'], weights=[0.1, 0.75, 0.15])[0]
```

4 Algorithms

Figures 2, 3 show pseudo codes of the methods that incorporate Algorithm 1. Q-learning specifies ϵ-greedy policy explicitly whereas TD(0) learns utility values on a predefined policy. For TD(0), we are going to consider its input policy π as ϵ-greedy as well.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 2: TD(0) pseudo code [1]

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., ε-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

Figure 3: Q-learning pseudo code [1]

5 Specifications

- You are going to implement TD(0) and Q-learning in Python 3.
- Each MDP problem is represented with a file and this file will be fed to your implementation along with one of the methods.
- To be able to run both methods from a single place, you are expected to write the following function:

```
def SolveMDP(method_name, problem_file_name, random_seed):
```

  The parameter specifications are as follows:

  - `method_name`: specifies which method to be run for the given MDP problem. It can be assigned to one of the values from `["TD(0)", "Q-learning"]`.
  - `problem_file_name`: specifies the path of the MDP problem file to be solved. File names take values such as "mdp1.txt", "mdp2.txt", "mdp3.txt".
  - `random_seed`: specifies 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 for all methods:

```
U, policy = SolveMDP(method_name, 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): `'V'`, (0,1): `'>'` ...}). Actions are represented with one of the following characters: `'>'`, `'V'`, `'<'`, `'^'` (RIGHT, DOWN, LEFT, UP respectively).

For Q-learning the following equation holds for state values and state-action values (TD(0) already keeps track of utility scores):

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

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

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

For TD(0), the optimal is extracted by comparing neighbor states as done in the policy improvement step of Policy Iteration.

Note: In TD(0), we should not consider expected utility scores, we were able to calculate this expected value for Policy Iteration since we knew the transition function, but here, the agent has no clue about even the presence of the action noise problem let alone the probability distribution of it. So we need to pick the action that takes the agent to the neighbor with the highest utility score while calculating the optimal policy for a state.

- 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 and width of the problem (the number of cells vertically and horizontally), respectively.

- `[obstacle states]` section specifies the obstacle states (separated with '|') in the problem.
- `[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), you may refer to Section 2.1 for further information). The first numerical value is b, the second is a and the third 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 (Figure 1).

- 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.
- Commenting is crucial for understanding your implementation and decisions made during the process. Your implementation can be manually inspected at random or when it does not satisfy the expected outputs.

6 Sample I/O

Here, in addition to textual outputs, visual outputs for the solution of the MDP problems are provided. The actions are depicted via their symbol. The utility values are shown at the top left of the states.

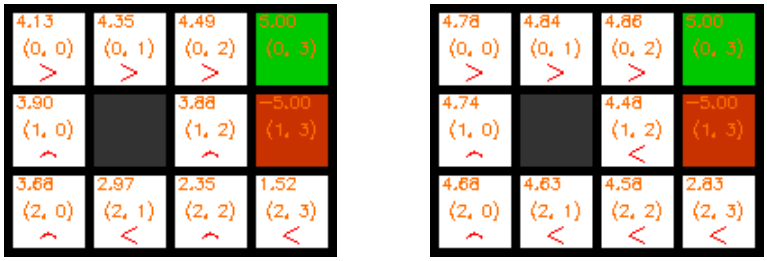
mdp1.txt:

```
[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 outputs of the SolveMDP function with both methods applied on mdp1.txt:

```
>>> import hw5solutioneXXXXXXX as hw5
>>> hw5.SolveMDP("TD(0)", "mdp1.txt", 37)
({(0, 0): 4.13, (0, 1): 4.35, (0, 2): 4.49, (0, 3): 5.0, (1, 0): 3.9, (1, 2):
   3.88, (1, 3): -5.0, (2, 0): 3.68, (2, 1): 2.97, (2, 2): 2.35, (2, 3): 1.52},
    {(0, 0): '>', (0, 1): '>', (0, 2): '>', (1, 0): '^', (1, 2): '^', (2, 0):
   '^', (2, 1): '<', (2, 2): '^', (2, 3): '<'})
>>> hw5.SolveMDP("Q-learning", "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): 'V'})
```



(a) TD(0) result (b) Q-learning result

Figure 4: Visualization of results of all methods on mdp1.txt. Actions are depicted with their character counterparts.

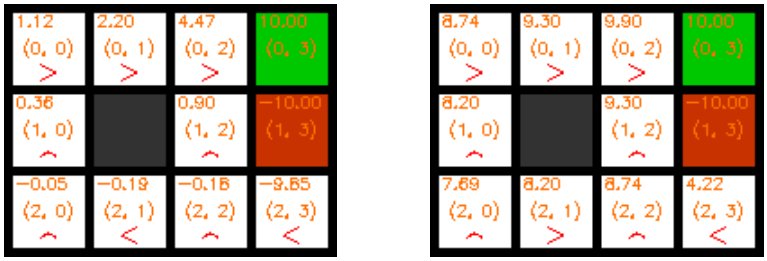
mdp2.txt:

```
[environment]
3 4
[obstacle states]
(1,1)
[goal states]
(0,3):10.0|(1,3):-10.0
[start state]
(2,0)
[reward]
-0.1
[action noise]
1.0
0.0
0.0
[learning rate]
0.1
[gamma]
0.95
```

```
[epsilon]
0.1
[episode count]
10000
```

Expected outputs of SolveMDP function with both methods applied on mdp2.txt:

```
>>> hw5.SolveMDP("TD(0)", "mdp2.txt", 37)
({(0, 0): 1.12, (0, 1): 2.2, (0, 2): 4.47, (0, 3): 10.0, (1, 0): 0.36, (1, 2):
   0.9, (1, 3): -10.0, (2, 0): -0.05, (2, 1): -0.19, (2, 2): -0.18, (2, 3):
   -9.85}, {(0, 0): '>', (0, 1): '>', (0, 2): '>', (1, 0): '^', (1, 2): '^',
   (2, 0): '^', (2, 1): '<', (2, 2): '^', (2, 3): '<'})
>>> hw5.SolveMDP("Q-learning", "mdp2.txt", 462)
({(0, 0): 8.74, (0, 1): 9.3, (0, 2): 9.9, (1, 0): 8.2, (1, 2): 9.3, (2, 0):
   7.69, (2, 1): 8.2, (2, 2): 8.74, (2, 3): 5.16}, {(0, 0): '>', (0, 1): '>',
   (0, 2): '>', (1, 0): '^', (1, 2): '^', (2, 0): '^', (2, 1): '>', (2, 2):
   '^', (2, 3): '<'})
```



(a) TD(0) result (b) Q-learning result

Figure 5: Visualization of results of all methods on mdp2.txt. Actions are depicted with their character counterparts.

mdp3.txt:

```
[environment]
5 11
[obstacle states]
(0,3)|(1,3)|(3,3)|(4,3)|(2,6)|(2,7)|(0,6)|(4,6)
[goal states]
(0,10):5.0|(4,10):20.0
[start state]
(4,0)
[reward]
-0.1
[action noise]
0.8
0.1
0.1
[learning rate]
0.1
[gamma]
0.9999999999
[epsilon]
0.1
[episode count]
10000
```

Expected outputs of SolveMDP function with both methods applied on mdp3.txt:

```
>>> hw5.SolveMDP("TD(0)", "mdp3.txt", 37)
({(0, 0): 9.07, (0, 1): 15.24, (0, 2): 16.49, (0, 4): 14.03, (0, 5): 9.43,
    ...}, {(0, 0): 'V', (0, 1): 'V', (0, 2): 'V', (0, 4): 'V', (0, 5): 'V',
    ...})
>>> hw5.SolveMDP("Q-learning", "mdp3.txt", 462)
({(0, 0): 12.14, (0, 1): 15.31, (0, 2): 17.83, (0, 4): 18.66, (0, 5): 18.77,
    ...}, {(0, 0): 'V', (0, 1): 'V', (0, 2): 'V', (0, 4): 'V', (0, 5): 'V',
    ...})
```

(a) TD(0) result

(b) Q-learning result

Figure 6: Visualization of results of all methods on mdp3.txt. Actions are depicted with their character counterparts.

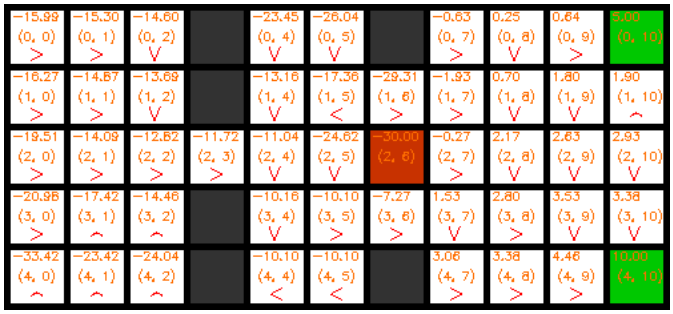
mdp4.txt:

```
[environment]
5 11
[obstacle states]
(0,3)|(1,3)|(3,3)|(4,3)|(0,6)|(4,6)
[goal states]
(0,10):5.0|(4,10):10.0|(2,6):-30.0
[start state]
(0,0)
[reward]
-0.5
[action noise]
0.8
0.1
0.1
[learning rate]
0.1
[gamma]
```

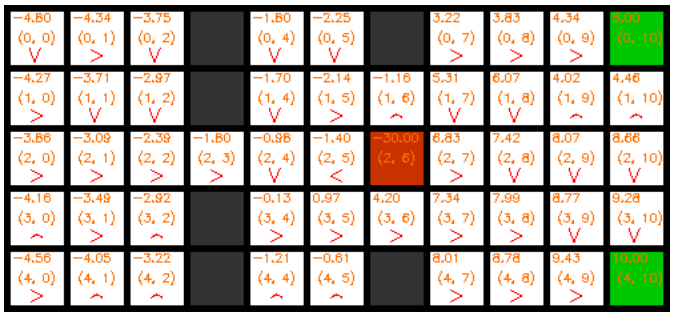
```
0.9999999999
[epsilon]
0.1
[episode count]
10000
```

Expected outputs of SolveMDP function with both methods applied on mdp4.txt:

```
>>> hw5.SolveMDP("TD(0)", "mdp4.txt", 37)
({(0, 0): -15.99, (0, 1): -15.3, (0, 2): -14.6, (0, 4): -23.45, (0, 5): -26.04,
    (0, 7): -0.63, (0, 8): 0.25, (0, 9): 0.64, (0, 10): 5.0, (1, 0): -16.27,
   ...}, {(0, 0): '>', (0, 1): '>', (0, 2): 'V', (0, 4): 'V', (0, 5): 'V', (0,
   7): '>', (0, 8): 'V', (0, 9): '>', (1, 0): '>', ...})
>>> hw5.SolveMDP("Q-learning", "mdp4.txt", 462)
({(0, 0): -4.15, (0, 1): -3.48, (0, 2): -2.88, (0, 4): -2.52, (0, 5): -2.85,
   (0, 7): 2.49, (0, 8): 3.61, (0, 9): 4.3, (1, 0): -3.94, ...}, {(0, 0): '>',
   (0, 1): '>', (0, 2): 'V', (0, 4): 'V', (0, 5): 'V', (0, 7): '>', (0, 8):
   '>', (0, 9): '>', (1, 0): 'V', ...})
```



(a) TD(0) result



(b) Q-learning result

Figure 7: Visualization of results of all methods on mdp4.txt. Actions are depicted with their character counterparts.

Note 1: While comparing action-pair values you may use the following partially pseudo-code piece:

```
max_value = - float('inf')
max_action = None
actions = ['<', '^', '>', 'V']
for action in actions:
    # if Q(current state, action) > max_value:
        # max_value = Q(current_state,action)
        # max_action = action
```

The same action sequence can be used to check utility scores of each neighbor reached by each action (i.e first check then left neighbor, up neighbor, right neighbor, down neighbor).

Note 2: Conceptually we should be lowering the exploration parameter ϵ as the agent gains more experience throughout its interaction, but here we utilize a constant value for it since we are dealing with MDP problems whose state space is relatively small.
Note 3: The learning parameter α should be decaying in order to get technical convergence guarantees, but practically a constant value is sufficient to get optimal policies for the problems presented in this assignment.
Note 4: There could be more one than one action that results in the same utility value, any of them can be returned as the optimal action.).
Note 5: The problem files and their outputs here are provided as additional files on ODTUClass.

7 Regulations

1. **Programming Language:** You must code your program in Python 3. Since there are multiple versions and each new version adds a new feature to the language, in order to concur on a specific version, please make sure that your implementation runs on İnek machines.

2. **Implementation:** You have to code your program by only using the functions in the standard module of python. Namely, you **cannot** import any module in your program (except the **copy** module).

3. **Late Submission:** No late submission is allowed. Since we have a strict policy on submissions of homework in order to be able to attend the final exam, please pay close attention to the deadlines.

4. **Cheating: We have zero-tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from the internet included) will be punished according to the university regulations.

5. **Discussion:** You must follow ODTUClass for discussions and possible updates on a daily basis. If think that your question concerns everyone, please ask them on ODTUClass.

6. **Evaluation:** Your program will be evaluated automatically using the "black-box" technique so make sure to obey the specifications. A reasonable timeout will be applied according to the complexity of test cases. This is not about the code efficiency, its only purpose is avoiding infinite loops due to an erroneous code. In case your implementation does not conform to expected outputs for solutions, it will be inspected manually so commenting will be crucial for grading.

8 Submission

Submission will be done via OdtuClass system. You should upload a **single** python file named in the format **hw5_e<your-student-id>.py** (i.e. hw5_e1234567.py).

9 References

1. For TD(0) and Q-learning methods, you can refer to Reinforcement Learning: An Introduction R. Sutton, and A. Barto. The MIT Press, Second edition, (2018) (Chapter 6)

   (https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf).

2. Announcements Page

3. Discussions Page