Middle East Technical University Department of Computer Engineering

CENG 462

Artificial Intelligence

Fall '2021-2022

Homework 4

Due date: 25 December 2021, Saturday, 23:55

1 Objectives

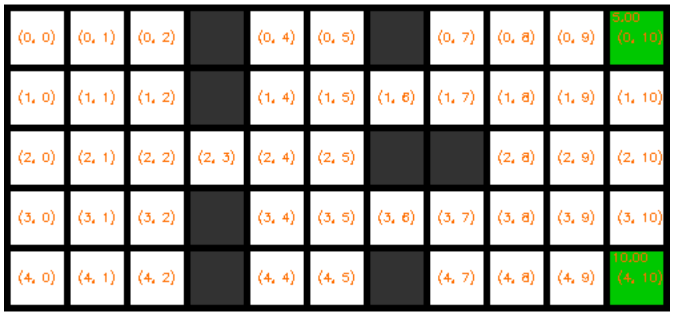
This assignment aims to familiarize you with the Markov Decision Process (MDP) framework at the implementation level and provide hands-on experience with two solution algorithms to solve it, namely Value Iteration and Policy Iteration.

2 Problem Definition

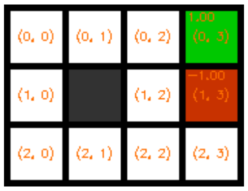
So far, we've been dealing with the problems where the actions have had deterministic consequences and running search methods in order to obtain their optimal solutions (i.e. sequence of actions that attains the problem goal with the least incurred cost). Each applied action has resulted in a particular state with a probability of 1 and all the techniques (we have implemented so far) have depended on this fact for execution. For the problems in which this assumption is violated, it is no use employing the previous techniques.

Markov Decision Process (MDP) is a mathematical framework for modeling sequential decision-making problems, where an action consequence is modeled with probability distributions and aims to extract optimal action sequences. It is a tuple of (S, A, T, R), where S is the set of states of the problem, A is the set of actions which are available in each state, R is the reward function that governs the outcome of an action (reward/cost) in a particular state and T is the state-transitioning probability distribution which models the dynamics of the problem. Interacting with this framework is carried out in discrete time steps by the problem solver entity (agent) and in each time step a new action is fed and its consequences are observed (next state, reward/cost). Solving an MDP corresponds to extracting an action sequence (policy) that maximizes/minimizes the cumulative expected reward/cost: $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R_t]$ (where γ regulates the importance of future reward/cost values in decision-making). When all components (S, A, T, R) of an MDP is known for a problem, we can solve that MDP with the dynamic programming techniques called Value Iteration and Policy Iteration without having to interact with the problem itself (i.e. the agent does not have to spend time on the problem trying different actions and gathering experience for learning).

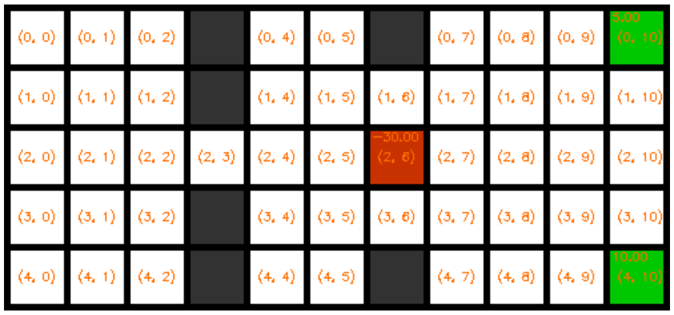
In this assignment, you are expected to implement Value Iteration and Policy Iteration methods in order to solve grid-world (MDP) problems.



(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).

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 Specifications

- You are going to implement Value Iteration and Policy Iteration 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:

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

  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 `["ValueIteration", "PolicyIteration"]`.

  – `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".

  During the evaluation process, this function will be called and returned information will be inspected. The returned information should be as follows for both methods:

```python
U, policy = SolveMDP(method_name, problem_file_name)
```

  – `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).

- 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 ...
[reward]
reward value
[action noise]
forward probability
left probability
right probability
[gamma]
gamma value
[epsilon]
epsilon value
[iteration]
iteration 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 utility score (separated with '|'). The utility values of the goal states are specified after ":".
- `[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.
- `[gamma]` section provides the value of the γ parameter for both methods.
- `[epsilon]` section is for only Value Iteration and specifies the value for ϵ, which indirectly determines the iteration count of Value Iteration.
- `[iteration]` section provides the value for the k parameter of Algorithm 1, which constitudes a part of Policy Iteration.

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** module with which you may perform a deep copy operation on a list/dictionary. 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.

4 Sample I/O

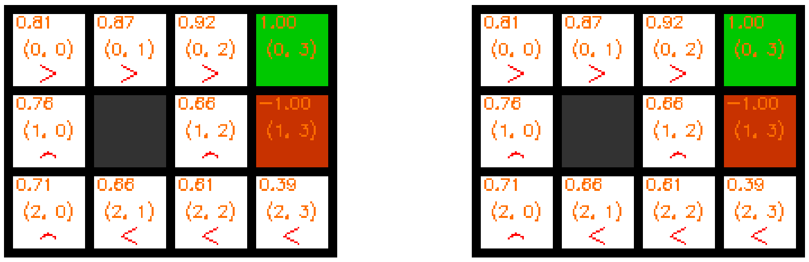
Here, in addition to textual outputs, visual outputs for the solution of the MDP problems are provided. The actions are depicted via their symbol. The utility values are shown at the top left of the states.

mdp1.txt:

```
[environment]
3 4
[obstacle states]
(1,1)
[goal states]
(0,3):1.0|(1,3):-1.0
[reward]
-0.04
[action noise]
0.8
0.1
0.1
[gamma]
0.999999999999999
[epsilon]
0.0001
[iteration]
30
```

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

```
import hw4solutioneXXXXXXX as hw4
>>> hw4.SolveMDP("ValueIteration", "mdp1.txt")
({(0, 0): 0.81, (0, 1): 0.87, (0, 2): 0.92, (0, 3): 1.0, (1, 0): 0.76, (1, 1):
    0.0, (1, 2): 0.66, (1, 3): -1.0, (2, 0): 0.71, (2, 1): 0.66, (2, 2): 0.61,
    (2, 3): 0.39}, {(0, 0): ’>’, (0, 1): ’>’, (0, 2): ’>’, (1, 0): ’^’, (1, 2):
    ’^’, (2, 0): ’^’, (2, 1): ’<’, (2, 2): ’<’, (2, 3): ’<’})
>>> hw4.SolveMDP("ValueIteration", "mdp2.txt")
({(0, 0): 0.88, (0, 1): 0.92, (0, 2): 0.96, (0, 3): 1.0, (1, 0): 0.84, (1, 1):
    0.0, (1, 2): 0.92, (1, 3): -1.0, (2, 0): 0.8, (2, 1): 0.84, (2, 2): 0.88,
    (2, 3): 0.84}, {(0, 0): ’>’, (0, 1): ’>’, (0, 2): ’>’, (1, 0): ’^’, (1, 2):
    ’^’, (2, 0): ’^’, (2, 1): ’>’, (2, 2): ’^’, (2, 3): ’<’})
```



(a) Value Iteration result (b) Policy Iteration result

Figure 2: Visualization of results of both methods on mdp1.txt. Actions are depicted with their character counterparts.

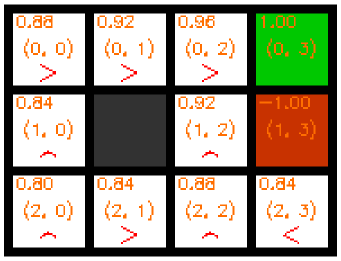
mdp2.txt:

```
[environment]
3 4
[obstacle states]
(1,1)
[goal states]
(0,3):1.0|(1,3):-1.0
[reward]
-0.04
[action noise]
1.0
0.0
0.0
[gamma]
0.999999999999999
[epsilon]
0.0001
[iteration]
30
```

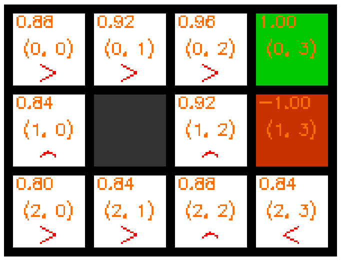
Expected outputs of SolveMDP function with both methods applied on mdp2.txt:

```
>>> hw4.SolveMDP("ValueIteration", "mdp2.txt")
({(0, 0): 0.88, (0, 1): 0.92, (0, 2): 0.96, (0, 3): 1.0, (1, 0): 0.84, (1, 1):
    0.0, (1, 2): 0.92, (1, 3): -1.0, (2, 0): 0.8, (2, 1): 0.84, (2, 2): 0.88,
    (2, 3): 0.84}, {(0, 0): ’>’, (0, 1): ’>’, (0, 2): ’>’, (1, 0): ’^’, (1, 2):
    ’^’, (2, 0): ’^’, (2, 1): ’>’, (2, 2): ’^’, (2, 3): ’<’})
```

```
>>> hw4.SolveMDP("PolicyIteration", "mdp2.txt")
({(0, 0): 0.88, (0, 1): 0.92, (0, 2): 0.96, (0, 3): 1.0, (1, 0): 0.84, (1, 1):
    0.0, (1, 2): 0.92, (1, 3): -1.0, (2, 0): 0.8, (2, 1): 0.84, (2, 2): 0.88,
    (2, 3): 0.84}, {(0, 0): '>', (0, 1): '>', (0, 2): '>', (1, 0): '^', (1, 2):
    '^', (2, 0): '>', (2, 1): '>', (2, 2): '^', (2, 3): '<'})
```



(a) Value Iteration result



(b) Policy Iteration result

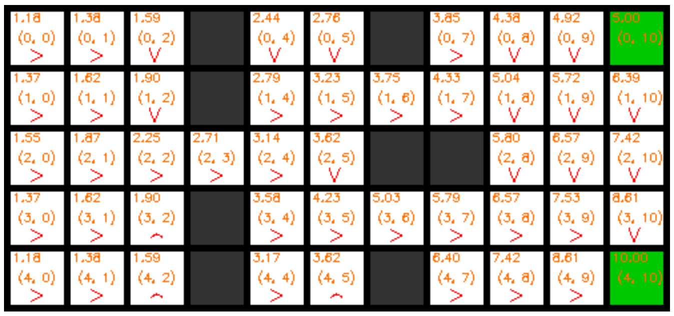
Figure 3: Visualization of results of both methods on mdp2.txt. Actions are depicted with their character counterparts.

mdp3.txt:

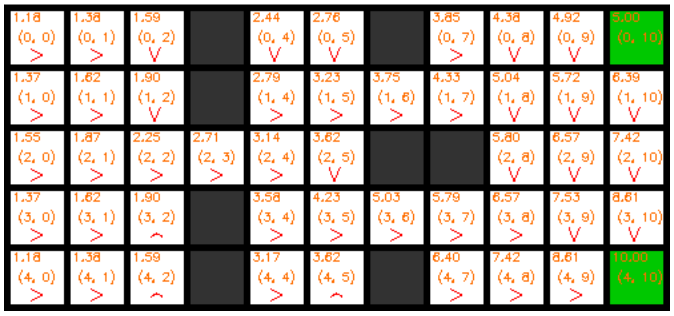
```
[environment]
5 11
[obstacle states]
(0,3)|(1,3)|(3,3)|(4,3)|(2,6)|(2,7)|(0,6)|(4,6)
[goal states]
(0,10):5.0|(4,10):10.0
[reward]
-0.04
[action noise]
0.8
0.1
0.1
[gamma]
0.9
[epsilon]
0.0001
[iteration]
30
```

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

```
>>> hw4.SolveMDP("ValueIteration", "mdp3.txt")
({(0, 0): 1.18, (0, 1): 1.38, (0, 2): 1.59, (0, 3): 0.0, (0, 4): 2.44, (0, 5):
    2.76, (0, 6): 0.0, (0, 7): 3.85, (0, 8): 4.38, (0, 9): 4.92, (0, 10): 5.0,
    (1, 0): 1.37, (1, 1): 1.62, ... }, {(0, 0): '>', (0, 1): '>', (0, 2): 'V',
    (0, 4): 'V', (0, 5): 'V', (0, 7): '>', (0, 8): 'V', (0, 9): 'V', (1, 0):
    '>', (1, 1): '>' ...})
>>> hw4.SolveMDP("PolicyIteration", "mdp3.txt")
({(0, 0): 1.18, (0, 1): 1.38, (0, 2): 1.59, (0, 3): 0.0, (0, 4): 2.44, (0, 5):
    2.76, (0, 6): 0.0, (0, 7): 3.85, (0, 8): 4.38, (0, 9): 4.92, (0, 10): 5.0,
    (1, 0): 1.37, (1, 1): 1.62 ...}, {(0, 0): '>', (0, 1): '>', (0, 2): 'V', (0,
     4): 'V', (0, 5): 'V', (0, 7): '>', (0, 8): 'V', (0, 9): 'V', (1, 0): '>',
    (1, 1): '>' ...})
```



(a) Value Iteration result



(b) Policy Iteration result

Figure 4: Visualization of results of both methods on mdp3.txt. Actions are depicted with their character counterparts.

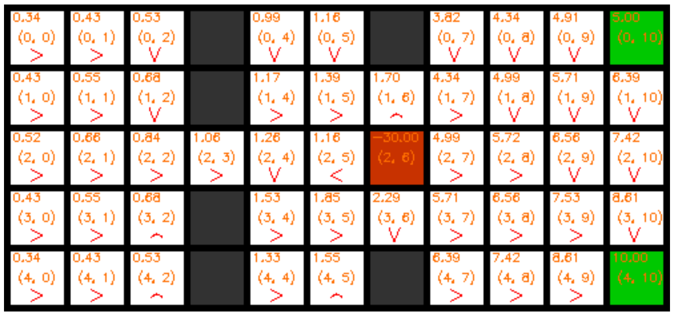
mdp4.txt:

```
[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
[reward]
-0.04
[action noise]
0.8
0.1
0.1
[gamma]
0.9
[epsilon]
0.0001
[iteration]
30
```

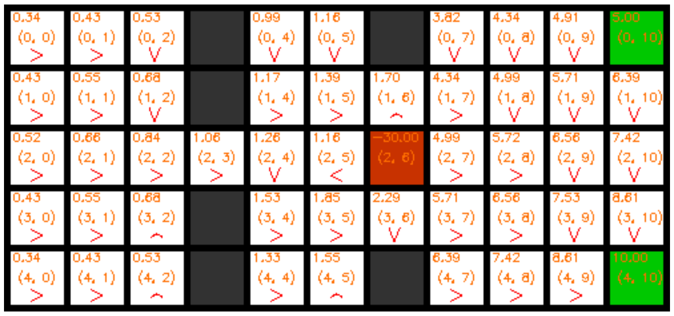
Expected outputs of SolveMDP function with both methods applied on mdp4.txt:

```
>>> hw4.SolveMDP("ValueIteration", "mdp4.txt")
({(0, 0): 0.34, (0, 1): 0.43, (0, 2): 0.53, (0, 3): 0.0, (0, 4): 0.99, (0, 5):
    1.16, (0, 6): 0.0, (0, 7): 3.82, (0, 8): 4.34, (0, 9): 4.91, (0, 10): 5.0,
    (1, 0): 0.43, (1, 1): 0.55, ...}, {(0, 0): '>', (0, 1): '>', (0, 2): 'V',
    (0, 4): 'V', (0, 5): 'V', (0, 7): 'V', (0, 8): 'V', (0, 9): 'V', (1, 0):
    '>', (1, 1): '>' ...})
>>> hw4.SolveMDP("PolicyIteration", "mdp4.txt")
```

```
({(0, 0): 0.34, (0, 1): 0.43, (0, 2): 0.53, (0, 3): 0.0, (0, 4): 0.99, (0, 5):
    1.16, (0, 6): 0.0, (0, 7): 3.82, (0, 8): 4.34, (0, 9): 4.91, (0, 10): 5.0,
    (1, 0): 0.43, (1, 1): 0.55 ...}, {(0, 0): '>', (0, 1): '>', (0, 2): 'V', (0,
     4): 'V', (0, 5): 'V', (0, 7): 'V', (0, 8): 'V', (0, 9): 'V', (1, 0): '>',
    (1, 1): '>' ...})
```



(a) Value Iteration result



(b) Policy Iteration result

Figure 5: Visualization of results of both methods on mdp4.txt. Actions are depicted with their character counterparts.

Note 1: 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 2: The problem files and their outputs here are provided as additional files on ODTUClass.

5 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.

6 Submission

Submission will be done via OdtuClass system. You should upload a **single** python file named in the format **hw4_e<your-student-id>.py** (i.e. hw4_e1234567.py).

7 References

- For `Value Iteration` and `Policy Iteration`, you can refer to the lecture's textbook.
- In the textbook of the lecture, the `POLICY-EVALUATION` function of Policy Iteration is described in the text. In order to eliminate misconceptions we provide a sample pseudo-code for the function.

Algorithm 1 Policy-Iteration (Policy Evaluation Part)

```
procedure POLICY-EVALUATION($\pi$: policy, $U_0$: Utility, $mdp$: (S, A, T, R), $k$: iteration count)
    for i=1 to $k$ do
        for each $s$ in S do
            $U_i = R(s) + \gamma \sum_{s'} T(s, \pi(s), s').U_{i-1}(s')$
        end for
    end for
    return $U_k$
end procedure
```

- Announcements Page
- Discussions Page